Creating a Raspberry Pi-Based Beowulf Cluster

Ellen-Louise Bleeker,
Magnus Reinholdsson

Faculty of Health, Science and Technology

Computer Science

C-level thesis 15hp

Advisor: Kerstin Andersson

Examiner: Stefan Alfredsson

Opposition date: 20170605

Creating a Raspberry Pi-Based Beowulf Cluster

Ellen-Louise Bleeker,
Magnus Reinholdsson

Abstract

This thesis summarizes our project in building and setting up a Beowulf cluster. The idea of the project was brought forward by the company CGI in Karlstad, Sweden. CGI's wish is that the project will serve as a starting point for future research and development of a larger Beowulf cluster. The future work can be made by both employees at CGI and student exam projects from universities.

The projects main purpose was to construct a cluster by using several credit card sized single board computers, in our case the Raspberry Pi 3. The process of installing, compiling and configuring software for the cluster is explained. The MPICH and TensorFlow software platforms are reviewed. A performance evaluation of the cluster with TensorFlow is given.

A single Raspberry Pi 3 can perform neural network training at a rate of seven times slower than an Intel system (i5-5250U at 2.7 GHz and 8 GB RAM at 1600 MHz). The performance degraded significantly when the entire cluster was training. The precise cause of the performance degradation was not found, but is ruled out to be in software, either a programming error or a bug in TensorFlow.

Preface

We thank Kerstin Andersson for providing invaluable feedback and proofreading during the writing of our dissertation. We also want to thank Curt ”Rulle“ Rudolfsson for being our electronics champion, who has guided us through the jungle of electronic equipment.

A big thank you to Jonas Forsman who has been our supervisor on CGI and given us tips and advice during the project. Also thanks to Torbjörn Stolpe who built the fantastic chassis for the cluster. Finally we want to thank CGI, which have given us the opportunity to do our graduate project with them.

Contents

List of Figures

List of Listings

List of Tables

List of Abbreviations

AI Artificial Intelligence
ALARM Arch Linux ARM
API Application Programming Interface
AUR Arch User Repository
COW Cluster of Workstations
CPU Central Processing Unit
DSA Digital Signature Algorithm
ECDSA Elliptic Curve Digital Signature Algorithm
GPIO General Purpose Input/Output
GPU Graphics Processing Unit
gRPC gRPC Remote Procedure Calls
HPC High-Performance Computing
HPL High Performance Linpack
JVM Java Virtual Machine
MMCOTS Mass Market Commodity-Off-The-Shelf
MNIST Modified National Institute of Standards and Technology
MPI Message Passing Interface
MPICH Message Passing Interface Chameleon
NFS Network File system
NN Neural Network
NOW Network of Workstations
OS Operating System
PCB Printed Circuit Board
ps Parameter Server
Protobuf Protocol Buffers
PSU Power Supply Unit
PVM Parallel Virtual Machine

RPi Raspberry Pi
RGB LED Red Green Blue Light Emitting Diode
SVM Support Vector Machine
UUID Universally Unique Identifier
XML Extensible Markup Language,

Chapter 1

Introduction

A computer cluster is a collection of cooperating computers. There are several variations of these, one of them is called Beowulf Cluster. A Beowulf cluster is a uniform collection of Mass Market Commodity-Off-The-Shelf (MMCOTS) computers connected by an Ethernet network. An important distinguishing feature is that only one computer —the head node— is communicating with the outside network. A Beowulf cluster is dedicated only to jobs assigned through its head node, see section 2.1 for a more elaborate definition.

Parallel programming differs from traditional sequential programming. Additional complexity becomes apparent when one must coordinate different concurrent tasks. This project is not about going in depth into parallel programming, but the focus lies more on how to design and build a cluster.

Machine Learning is a sub-field of artificial intelligence. In 2015 Google released a machine learning platform named TensorFlow. TensorFlow is a library that implements many concepts from machine learning and has an emphasis on deep neural network. In this project we run a TensorFlow program in parallel on the cluster.

1.1 Purpose of Project

The principal purpose of the project is to build and study a Beowulf cluster for the company CGI. CGI has 70 000 coworkers in Europe, Asia, North and South America. The company has over 40 years of experience in the IT industry and has the primary goal to help the customers reach their business goals.[9]

The company will continue to develop and expand the cluster after the dissertation's end. CGI is primarily interested in two technologies; Message Passing Interface (MPI) and Machine Learning (ML). MPI is the mainstay in parallel programming and is hence interesting for the company. Machine Learning is presently a growing trend and there is room for a lot of business opportunities and innovation which are interesting to CGI.

Several machine learning development frameworks are available and recently (November 2015) Google released another one as open source; TensorFlow.[52] We are going to investigate the methods for distributed TensorFlow and deploy the methods in a Beowulf cluster.

The value of a Beowulf cluster lies mainly in the economic aspect; one retrieves a significant amount of computing resources for the money. Supercomputers have been around since the early days of computers in the 1960s, but have only been accessible to large companies and state funded agencies. This is rooted in how these systems were developed, a lot of hardware was custom designed. The creation of non-standard hardware is accompanied with high costs.

Beowulf clusters are changing supercomputing by increasing accessibility and price point tremendously, therefore drawing an entirely new audience; small businesses, schools and even private clusters in one's home.

1.2 Disposition and Scope

In Chapter 2 we give background information to areas relevant to our project, including cluster design and the software used in the project. In chapter 3 the physical construction and the power supply of the cluster is demonstrated. In chapter 4 the software installation is clarified. In chapter 5 a description of the software configuration process steps are presented. In chapter 6 we explain the programs that were executed on the cluster in detail. In chapter 7 the result of the program execution is discussed and evaluated. In the last chapter 8 we reflect on the project in general and discuss future directions of development.

Chapter 2

Background

Attention is first focused on the concepts of Beowulf clusters. Next an overview of machine learning is given, followed by a review of the fundamentals of the machine learning framework TensorFlow. TensorFlow's capability to distribute work is investigated more thoroughly as our aim is to perform distributed machine learning with the cluster. The purpose and motivation of the project is given. There exist clusters which have much in common with ours, this similar work is investigated. Finally a summary of the aforementioned sections closes the chapter.

2.1 Beowulf Cluster

The name Beowulf originally comes from an Old English epic poem which was produced between 975 and 1025.[8] The poem tells a story about a big hero named Beowulf, who in young age gets mortally wounded when he slays a dragon. Why name a computer after a big hero? Maybe the name stands for power and strength and so it symbolizes the power of the computer.

The first Beowulf-class PC cluster was created by two professors that worked for NASA in 1994. They were using an early release of the operating system GNU/Linux and ran Parallel Virtual Machine (PVM) on 16 Intel 100 MHz 80486-based computers by connecting them to a dual 10 Mbps Ethernet LAN.[88] The development of the Beowulf project started after the creation of the first Beowulf-class PC cluster. For example some necessary Ethernet driver software for Linux was developed and cluster management tools for low level programming. During the same time the computer community took the first MPI standards under their wings and embraced it. MPI has since become the most dominant parallel computing program.

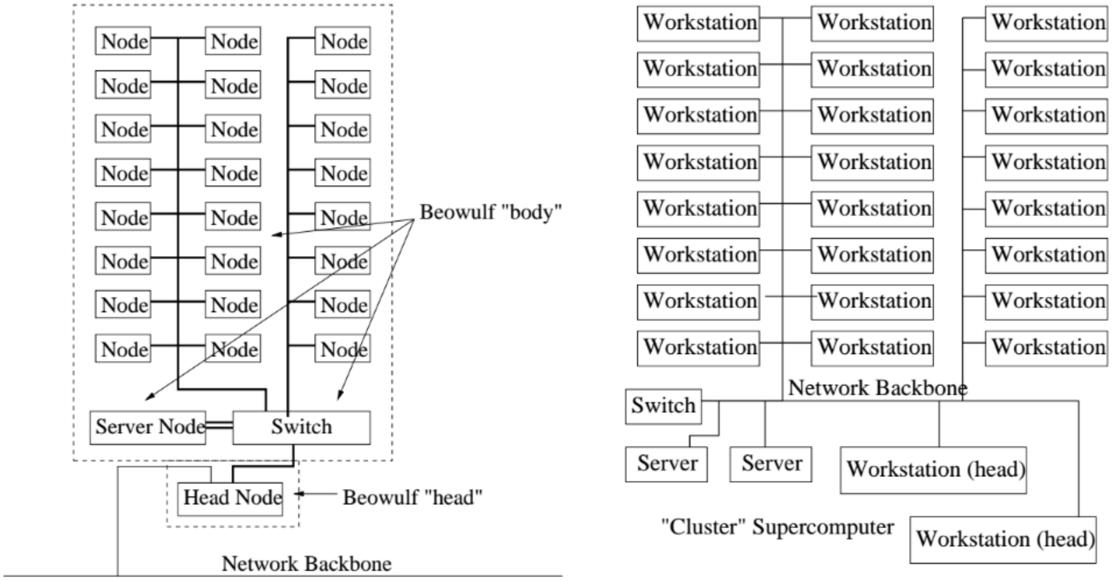


Figure 2.1: The left figure demonstrates a Beowulf cluster while the right demonstrates a mixed cluster, for example COW and NOW.

A Beowulf cluster can be explained as a “virtual parallel supercomputer” that consists of computers connected by a small local area network.[78] All computers in the network have the same programs and libraries installed. This allows the nodes in the cluster to share different processes, data and computation between them.

The definition of a Beowulf cluster is that the components in the cluster interconnect with a network and possess certain characteristics. One of the characteristics is that the nodes sole purpose is to serve the Beowulf in the network. Another characteristic is that all nodes run with open source software. A third characteristic is that the Beowulf cluster is dedicated to High Performance Computing (HPC).[78] If the cluster in some way would deviate from the characteristics it is not a Beowulf cluster. The definition of a Beowulf cluster is exemplified in figure 2.1, where some of the characteristics can be seen.

The right-hard side of figure 2.1 also demonstrates a mixed cluster. For example a COW (cluster of workstations) or a NOW (network of workstations). Clusters as COW and NOW are not technically Beowulf clusters even though they have similarities. Nodes in this type of cluster are not isolated, which means the nodes can be occupied by work that is not HPC. For example letting Alice in the room next door read her email on one node and another node allows Mary to watch a movie on the web. This is not possible in a Beowulf cluster. Nobody from the outside can connect to a working node. This is demonstrated in the left picture where the Beowulf cluster has a dashed line around itself. It can be interpreted as the cluster protects itself from the outside world.[78]

It is very practical to use a Beowulf cluster to construct and build a supercomputer out of cheap electronic equipment. An essential part of the system is the operating system with an open source software environment which provides a completely configurable kernel that can be tweaked for a specific HPC problem. The cluster will perform better than a single node computer but not as fast as a traditional supercomputer. It is way too expensive for a normal paid person to construct and build a supercomputer. By using the architecture of Beowulf people are able to use standard and old machines to build supercomputers, by connecting them with Ethernet and run an open source Unix-like operating system.[78]

One important aspect with Beowulf is that it requires a parallel processing library. There exists a number of libraries and the most commonly used is the MPI or PVM.[78] Then, why parallel computing? Over the last twenty years the demand of supercomputing resources has risen sharply. During these years the parallel computers have become an everyday tool for scientists, instead of just being an experimental tool in a laboratory.[81] The tools are necessary today in order to succeed with certain computationally demanding problems. For this project MPI was chosen. More information about MPI can be found in section 6.1.

2.2 Machine Learning

ML is a sub-field of Artificial Intelligence (AI) that entails a wide plethora of methods to realize computer programs that can learn out of experience. The mathematical underpinnings of ML is based on Computational Learning Theory.[77] The problem of estimation and classification is central. Statistics, probability theory, linear algebra and optimization theory are some of the relevant areas. According to ML pioneer Arthur Samuel in 1959, *"ML is what gives computers the ability to learn without being explicitly programmed"*.[84]

ML algorithms can be classified into three classes of learning. **Supervised** learning learns out of labeled examples. This is currently the most commonly applied way of learning. **Unsupervised** learning learns out of unlabeled examples, looking for structure. In **Reinforcement Learning** there is a software agent which take actions in an environment to maximize some kind of cumulative reward. These different classes of learning are covered by the different methods of ML such as Decision Trees, Support Vector Machines (SVM), Artificial Neural Networks, Clustering and a fair amount of other methods. The field of ML is vast and is an ongoing area of research and is yielding a lot of new applications.[28]

ML has been an area of research since the late 1950's, artificial intelligence itself is older as it has been a subject in philosophy since the ancient Greek philosophers.[86] Arthur Samuel worked on the Perceptron computer which is a primitive predecessor of today's deep neural networks. The 80's was dedicated to knowledge-driven rule-based languages which culminated in expert systems. These systems could provide AI in a narrow problem domain.[45] The SVM (and associated algorithms) has been studied during the second part of the 1950's. In the 90's a form of the SVM very close to today's was introduced.[21] Since 2012 attention to ML has risen significantly. It has been suggested that this is mainly due to better availability to large data sets and computer power. Large companies such as Google, Microsoft and Amazon has invested into making use of ML in several products such as speech and visual recognition.[85]

2.3 TensorFlow Basics

The development of TensorFlow was initiated by the Google Brain team. One of their whitepapers opens with a statement: *"TensorFlow is a machine learning system that operates at large scales and in heterogeneous environments"*.[75] We evaluate this statement as follows based on three articles on TensorFlow.[75][80][74] TensorFlow is an ML system in that it implements several of the ML methods discussed in section 2.2 such as shallow and deep neural networks, stochastic gradient descent, regression and SVM's to name a few. A TensorFlow instance executing in a single machine can utilize multiple devices such as CPU's and Graphic Processing Units (GPU) in parallel, and beyond this several instances of TensorFlow on different machines connected through a network can cooperate in ML tasks. This presents a scalable distributed system and in section 2.4 it is explained how it achieves this in large scales. The machines partaking in a distributed TensorFlow environment need not be identical in neither hardware or software. The CPU architecture, CPU clock, presence/absence of GPU and operating system are some of the variable attributes. Thus, TensorFlow indeed is a machine learning system that can operate at large scale in heterogeneous environments.

A characteristic and fundamental feature of TensorFlow's design is the dataflow graph. In the dataflow programming paradigm a program is structured as a directed graph where the nodes represent operations and the edges represent data flowing between those operations where the data is processed.[82] TensorFlow conforms to this paradigm by representing computation, shared state and operations mutating the shared state in the graph. Tensors (multi dimensional arrays) are the data structures which act as the universal exchange format between nodes in TensorFlow. Functional operators may be mathematical such as matrix multiplication, convolution etc. There are several different kinds of stateful nodes: input/output, variables, variable update rules, constants, etc. The communication is done explicitly with tensors. This makes it simple to partition the graph into sub-computations which can be run on different devices in parallel. It is possible for sub-computations to overlap in the main graph and share individual nodes that hold mutable state. For example, in figure 2.2 a dataflow graph is used to express a simple arithmetic calculation. The TensorFlow graph can express many different machine learning algorithms but can also be used in other areas such as simulating partial differential equations or calculating the Mandelbrot set.[46][68]

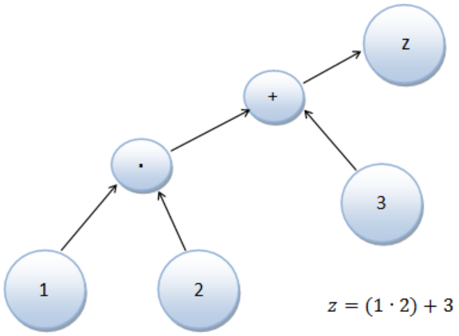


Figure 2.2: An example of a simple dataflow graph.

2.4 Distributed TensorFlow

In April 2016 Google published the distributed version of TensorFlow.[2] All the preceding versions of TensorFlow has operated only in the memory space of one Operating System (OS) with the opportunity to utilize multiple local devices such as CPU's and GPU. The distributed version treats devices on the local (machine) level in the same way. Furthermore, distribution in TensorFlow means that multiple machines can cooperate in executing algorithms such as Neural Network (NN) training. Thus this is a multi-device, multi-machine computing environment.[75]

A TensorFlow cluster is a set of high-level **jobs** that consist of **tasks** which cooperate in the execution of a TensorFlow graph. There are two different types of jobs; Parameter Server (ps) and worker. A ps holds the ”learnable“ variables/parameters. The worker fetches parameters from the parameter server and computes updated parameter values that it sends back to a parameter server. To bring about a distributed computation the host-addresses of the to-be-participating machines need to be noted in the program. Typically each task is bound to a single **TensorFlow server**. This server exports two Remote Procedure Call (RPC) services; **master service** and **worker service**. The master service is positioned as session target. It coordinates work between one or more worker services.[14]

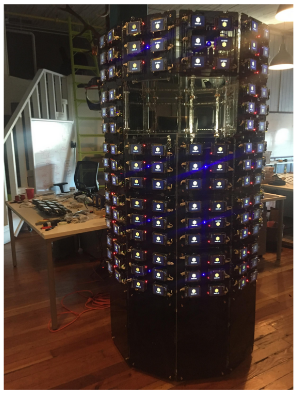
The communication system in Distributed TensorFlow has been implemented with gRPC Remote Procedure Calls (gRPC, recursive acronym). gRPC is an open source, high performance, remote procedures call framework. The development of gRPC was initiated by Google in March 2015.[19][1]

To parallelize the training of an NN one typically employs data parallelism. This kind of parallelism is manifested in that multiple tasks in a worker job can train the same model in small batches of data and update the shared parameters in the tasks of the ps job. Different variants of parallel execution schemes are available and is discussed in section 6.3.

2.5 Related Work

The interest of building Raspberry Pi (RPi) clusters has grown over the years. Why should not the interest be big when it is possible to build a supercomputer with help of cheap electronic components. The interest of the RPi itself has also grown over the years. The possibility to have a fully functional computer as big as a credit card has caught people's attention. A cluster can be built in many different ways. It can consist from a few to hundreds of nodes. The popularity of building clusters today has risen sharply and an important factor seem to be the Internet. The Internet today is an amazing platform where information can be found. For example, this project is inspired by an earlier work by Joshua Kiepert.[83] Kiepert's project and conclusions has been a guide line for our project.

Figure 2.3: The Beast. Published with permission from Alison David from the Resin team.[60]



Apart from our project, many people around the world are doing the same. You can for example today buy a complete 4 node cluster from the store. Thanks to the development of cheep computers as the RPi, people have the opportunity to build small clusters at home and then share the success on the web. For example, there is a team that has shared their success of building a cluster of 144 nodes, “The Beast”. The cluster consists of 144 RPi's, each with a 2,8 Adafruit PiTFT screen.[60] All nodes are attached as a pentagon stack that weigh nearly 150 kg and is 2 m tall, the Beast can been seen in figure 2.3. In each stack there are 24 RPi's. In the back of the panel (inside the pentagon) 20 USB hubs and 10 Ethernet Switches are attached. The project is still running and the development can be followed on their website.[60]

The next example is provided by the University of Southampton, where a Raspberry Pi cluster has been created with help of Lego.[24] The cluster contains 64 RPi's model B, the "'Iridis-Pi". Building the cluster in Lego allows younger users to play with it and the goal is to inspire the next generation of scientists.

There are many different ways of building a RPi cluster. But they all have a few parts in common. No matter how many nodes that are in use, they all look similar. The parts that separate them are their purpose and software. For example, our purpose of the project is to study TensorFlow, while Kiepert's project was focused on developing a novel data sharing system for wireless sensor networks.[83] No matter what the purpose of a project is, they are all based on similar hardware.

2.6 Summary of Chapter

The chapter presents a definition of a Beowulf and a mixed cluster, COW and NOW. A brief look at the subject machine learning is done. Both TensorFlow's basic and distributed versions are explained and discussed. A few similar works are also looked upon in the chapter.

Chapter 3

Hardware

To build the cluster some aspects had to be considered before the Rack could be created. In order to have a cluster size that facilitate the access to the components, the RPi's were stacked on top of each other. The company's wish was to use 33 RPi's in the cluster. To make the accessibility easy eight RPi's were placed in four stacks. By using PCB-to-PCB standoffs between each RPi the cluster became stable. The distance between each standoff made enough room for the air to flow between each RPi. The system was able to get its power supply with help of serially connected PCB cards. Instead of using a micro USB cable to every RPi the amount of cables could be decreased to only one cable per stack.

3.1 Raspberry Pi 3 Model B

For this project the Raspberry Pi 3 Model B was used. It is the third generation of the Raspberry Pi and it replaced the second model in February 2016. The new generation of Raspberry Pi has some new upgrades compared to the Raspberry Pi 2. The Raspberry Pi 3 has both an 802.11n wireless LAN and bluetooth. The construction of Raspberry Pi 3 model B can be seen in figure 3.1.

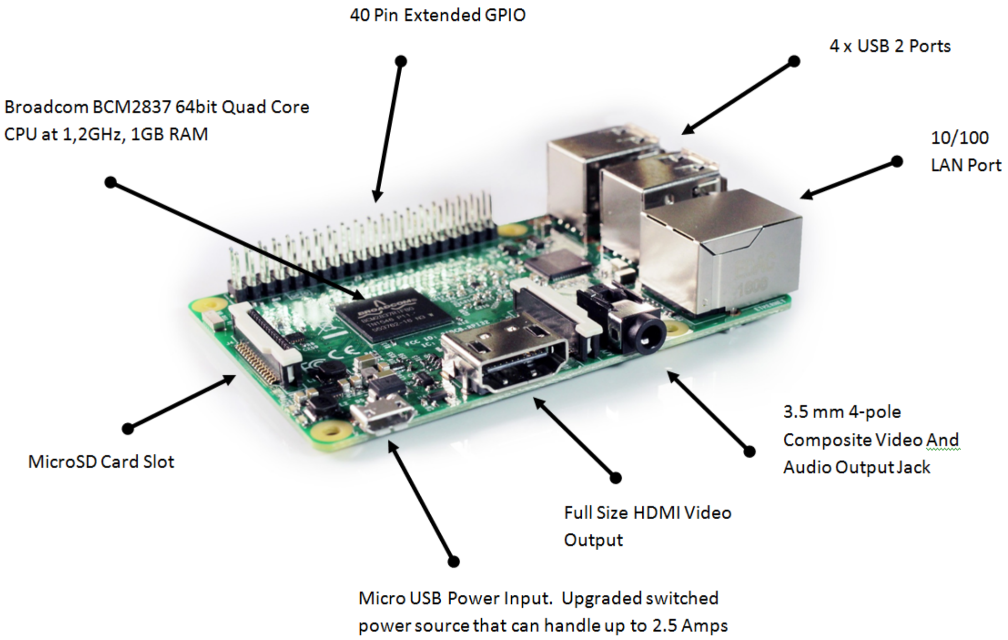

40 Pin Extended GPIO
4 x USB 2 Ports
Broadcom BCM2837 64bit Quad Core CPU at 1,2GHz, 1GB RAM
10/100 LAN Port
MicroSD Card Slot
3.5 mm 4-pole Composite Video And Audio Output Jack
Full Size HDMI Video Output
Micro USB Power Input. Upgraded switched power source that can handle up to 2.5 Amps

Figure 3.1: The Raspberry Pi 3 model B.

Another upgrade compared to the second generation is the CPU. In the third generation it is a 1.2GHz 64-bit quad-core ARMv8 CPU.[39] Meanwhile in the second generation it is a 900MHz quad-core ARM Cortex-A7 CPU.[38] The models shares the same memory setup with 1 GB DDR2 900MHz RAM and a 100 Mbps ethernet port.[40]

3.2 PCB Card

To power the entire system without having an individual micro USB power cable to each node, a special stackable Printed Circuit Board (PCB) was used for the power distribution. A PCB card connects electronic components through conductive tracks that have been printed on the board.[36] For this project a two copper layer card was created. The drawings of the PCB card was created by Joshua Kiepert and the drawings were used for the project. The electronic schema and drawing was downloaded at Kiepert's git repository .[15] Figure 3.2 demonstrates the PCB card's electronic schema and drawing.

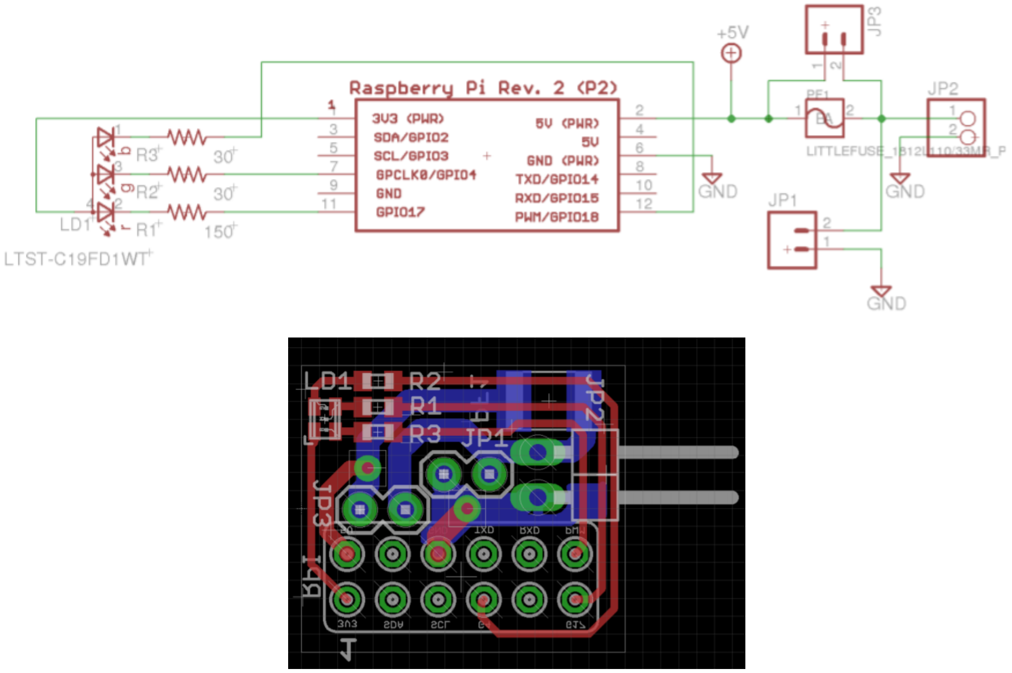


Figure 3.2: The PCB card's electronic schema and drawing in Eagle CAD.[83]

As seen in figure 3.2 a Read Green Blue (RGB) LED was used. For each color of the LED, three different resistances were connected. To the green and blue LED a resistance of 30 Ω was connected to each color. For the red LED a resistance of 150 Ω was connected. The resistances work in the way, that it adjusts the brightness of the lamp. In this case the red LED was significantly stronger. So by adding extra resistance to the red LED, the brightness of the lamp would be the same as in the blue and green LED lamps.

On the back side of each PCB card a polyfuse (PF1) was connected. A fuse is an electrical safety device that provides a safeguard in a circuit in case of short circuit.[18] The RPi itself has a fuse that is connected between the micro USB port and the positive terminal. The GPIO has 5V connection without the fuse. But in this case because the electricity is going through the PCB card and not the micro USB port, a fuse is required on the PCB card instead. By connecting the fuse on the PCB card to incoming current and to the second pin header, JP3 in the drawing in figure 3.2, the same protection will be provided. To make it easier to connect the card to the RPi's General-Purpose input/output (GPIO), a socket header was soldered onto the PCB card. The socket header was connected to the first twelve GPIO's on the RPi. The GPIO's electronic schema is explained in the top picture in figure 3.2.

On every card an angle pin header was soldered. This pin header is where the card receives its power supply to drive the whole stack. The power supply unit (PSU) was chosen by calculating the total energy consumption of all the RPi's. Each RPi of model B draws about 700mA-1000mA, i.e. about 5W. This cluster has 32 RPi's that each draws about 5W, that is a total of 160W. The whole cluster is run with two power supplies. Each PSU had to draw about 80W, that is 16A. We selected the 500W Corsair Builder Series V2 that could provide 20A at 5V. From each PSU two modified serial-ATA connections were established. At the end of the cable the ATA head was cut off and replaced with a pin header. The pin header on the cable could then be connected to the angled pin header on the PCB card. Each ATA connection brought power to a stack of eight RPi's.

By connecting the PSU to one of the middle RPi in each stack the current will be equally divided and get lower power consumption. Would for example the cable be connected to the bottom of the stack the power consumption would be very high. This is because the power has to go a longer way and more power output is required. Because of the high power it increases the possibility of short circuits in the system. The example is demonstrated in the left picture in figure 3.3, where the red arrows in the picture demonstrates the current in the stack. As seen in the left picture there is a large amount of power that goes through the stack. By connecting the cable at the middle of the stack, the current will have the possibility to split into two directions and become more stable. The right picture in figure 3.3 demonstrates a stable and low power consumption in the stack.

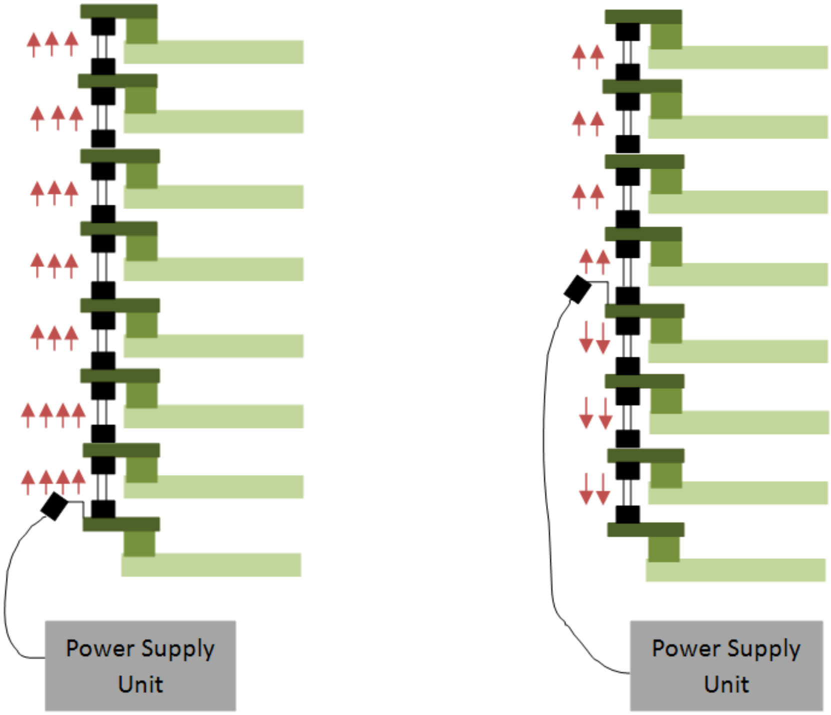
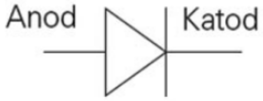


Figure 3.3: The figure demonstrate the current in a stack.

It is very important to connect the ATA cable correctly to the PCB card. If the negative terminal is attached to the positive terminal (5V) the system would possibly break down. This is because the current would flow in the wrong direction and components would risk getting burnt. This problem can be fixed by adding a diode to the system.

The diode stops the negative current and let's only the positive terminal pass, dependent on how it is placed. The drawing for the diode can be seen in figure 3.4. In this system a fuse was connected to the PCB card. The fuse should protect the system against problems like this, because the power goes through the fuse first before it goes in to the RPi. We never tried what would happen if the 5V would be attached to the negative terminal, so we are not sure that the system will be protected. In the picture to the right in figure 3.5 the fuse can be seen as a small black box at the right end of the card.

Figure 3.4: Drawing over a diode.



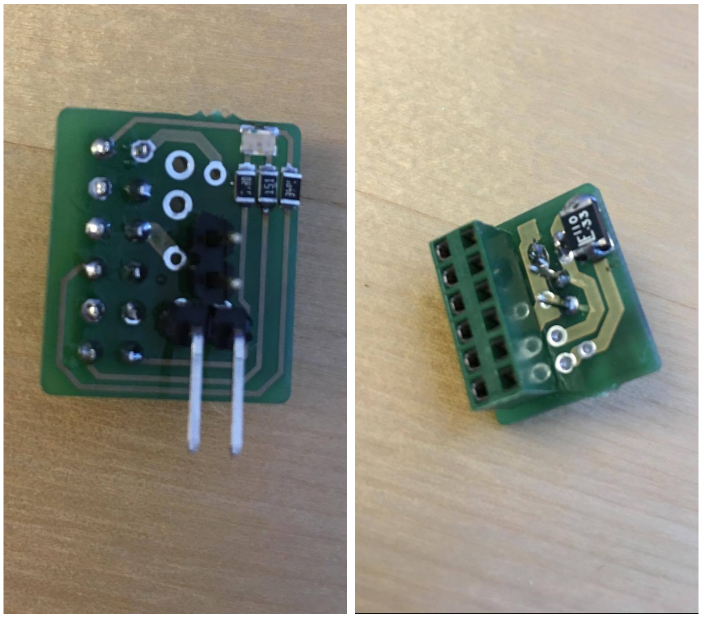


Figure 3.5: The finished soldered PCB card.

Unlike the power schedule in figure 3.2, the second pin header was not attached to the PCB card. This was decided simply because we had no use of it in our setup. The current goes through the hole from the bottom to the top of the card and into the first pin header. This is possible because the holes are covered in tin and are current conductive. So the second pin header would only have been extra work. But as seen in figure 3.5 there is a possibility to connect a second pin header in the two bigger holes.

Before the card was attached to the RPi careful measurements were made. It had to be checked that all the LED lamps were working and that the fuse was properly attached. Then the cards were carefully cleaned with red ethanol. Cleaning the cards prevents the system from short circuit. Short circuits were possible because solder paste was used to attach the resistors. Solder paste can by accident transfer electricity to another resistor if it is not cleaned properly. The components that were used to build the PCB cards can be seen in table 3.1.

Table 3.1: Components to construct a PCB card.

Product	**Model**	**Supplier**	**Art.nr**	**Number**	**Cost (SEK)**	**Total (SEK)**
PCB card		CograPro		35		1800
Poly Fuse	PTC RESETTABLE 33V 1.10A 1812L	Digi-Key	F3486CT-ND	35	5	175
LED Lamp	Tri-Color LED (LED RGB 605/525/470NM DIFF SMD)	Digi-Key	160-2022-1-ND	35	3,186	111,51
Socket Header	Socket headers 2,54mm 1x2p	Electrokit	41001166	66	1,46	96,36
Pin Header	Pin header 2,54mm 1x40p long pins 16mm	Electrokit	41013860	2	12,80	25,60
Solder	Solder 1,00mm 60/40 250g	Electrokit	41000511	1	199	199
Socket Header	Socket header 2,54mm 2x6p	Elfa	143-78-147	35	9,60	336
Resistor 30 Ω	Resistors SMD 39 Ohm 0805±0.1%	Elfa	300-47-355	70	2,47	173,07
Resistor 150 Ω	Resistors SMD 150 Ohm 0805±0.1%	Elfa	160-21-947	35	4,572	160,02
Pin Header	Pin header 40x1 angled	Kjell & Company	87914	2	19,90	39,80
Amount						3116,36

3.3 Raspberry Pi Rack

To be able to connect the RPi's to something, the decision of using two plexiglasses was made. One on the top and one at the bottom of the cluster. On each plexiglass twelve holes were drilled. The holes were placed according to the holes on the RPi. Standoffs could then be attached in the holes.

The standoffs could only be 3 mm wide because of the small holes on the RPi. Each hole on the RPi was sealed with a composite plastic that needed to be removed. By carefully drilling the hole bigger it became more easy to attach the standoffs. The standoffs were attach to the plexiglass using small nuts and screws and made the cluster stable. When the chassis was connected to the cluster it became even more stable. The chassis was built by an employee on the company. All components that were required for the rack can be found in table 3.2.

Table 3.2: Components to the cluster design.

Product	**Model**	**Supplier**	**Art.nr**	**Number**	**Cost SEK**	**Total SEK**
Standoffs	Standoff M/F M3 25mm BB 33mm OAL	Digi-Key	AE10782-ND	111	5,02	557,22
Nuts	HEX NUT 0,217 M3	Digi-Key	H762-ND	20	0,403	8,06
Screws	MACHINE SCREW PAN PHILLIPS M3	Digi-Key	H744-ND	20	0,726	14,52
Washer flats	WASHER FLAT M3 STEEL	Digi-Key	H767-ND	20	0,376	7,52
Computer	Rasberry Pi 3 model B	Dustin	5010909893	33	319kr	10527
Switch	Cisco SF200-48 Switch 48 10/100 ports	Dustin	5010901420	1	2495	2495
Cable	Deltaco STP-611G cat.6 Green 1,5m	Dustin	5010824924	33	69	2277
Cooling	Cooler Master sickleflow 120 2000RPM Green LED	Dustin	5010618893	4	85	340
Power Supply	Corsair Builder Series CX500 V2	Dustin	5010655602	2	559	1118
Mirco-USB	2 Meter, Micro USB cable green/withe	Fyndiq	6638583	1	39	39
Power Adapter	Deltaco power adapter, 2,4A Black	Fyndiq	469250	1	119	119
Connection	Extension Cable Satapower 20cm	Kjell & Company	61587	4	69,90	279,6
Memory	SanDisK MicroSDHC ultra 16 GB 80MB/s UHS	NetonNet	223373	33	79	2614
Plexiglass	120x370x5mm	Glasjouren in Forshaga		2	335	670
Amount						21065,92

The cluster contained 33 RPi's that were distributed into four stacks. One of the RPi's was attached on top of the cluster, the head, see figure 3.6.

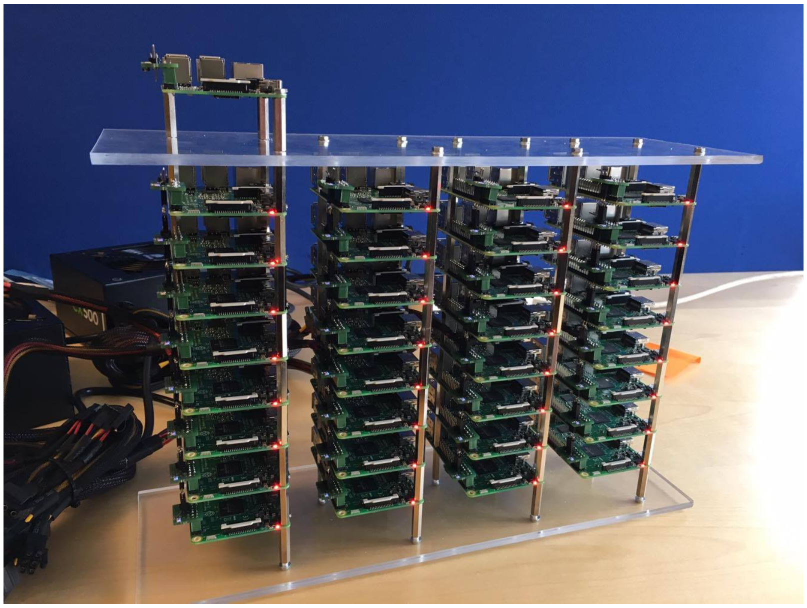


Figure 3.6: The fabricated cluster with the power supply.

As seen in figure 3.6 every RPi's little red lamp is lit. This was possible thanks to the serially connected PCB cards in each stack. The PCB card was connected to each RPi through a small 2x6 socket header, that was connected to the RPi's GPIO. Between every card two pin headers were together soled to create a bridge to the next card. The black components in both figures 3.7 and 3.8 are the bridges.

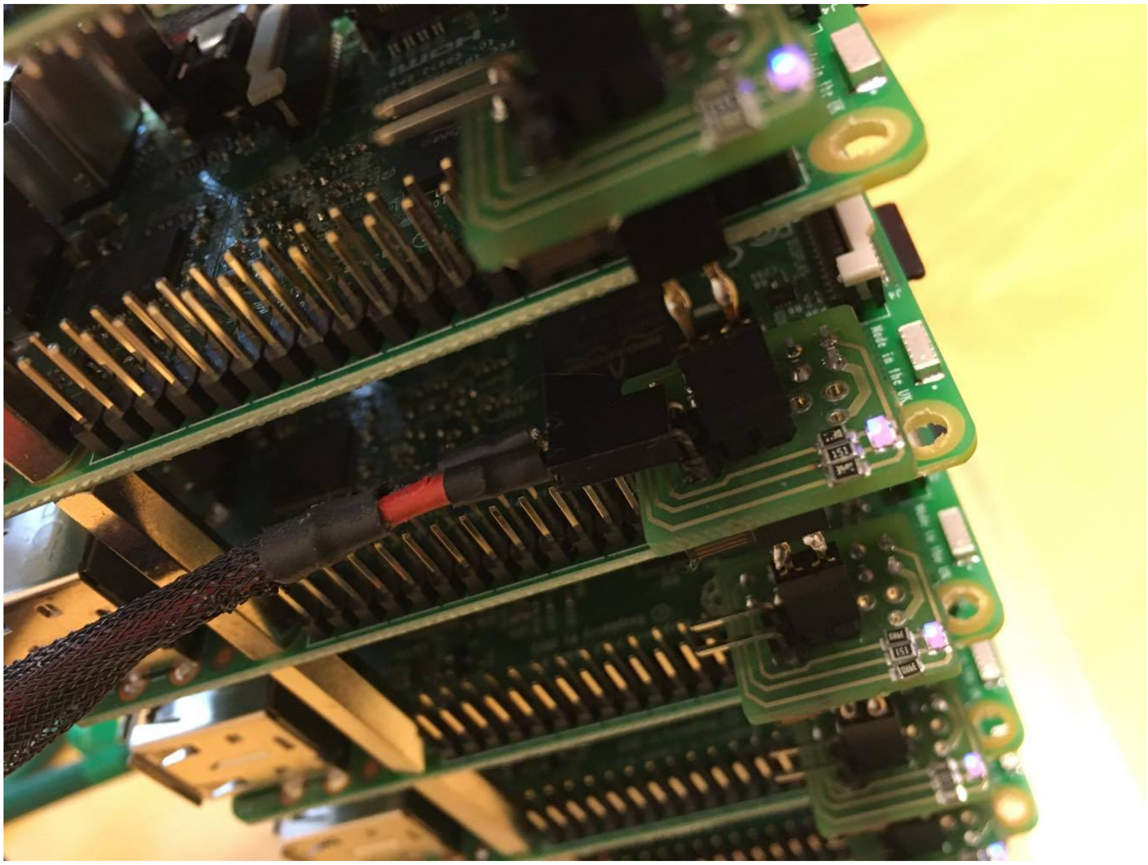


Figure 3.7: Connected unit to Raspberry Pi.

The serially connected PCB cards work with help of the bridge. The current goes from the connected cable through the angle pin header and then through the PCB card and into the fuse. Then it goes through the card and into the straight pin header. From the pin header the current goes into the bridge. The bridge is connected between two PCB cards straight pin headers. The current is then able to flow through the system.

In figure 3.7 the importance of connecting the terminals correctly can also be seen. The red cable, the positive terminal, is connected towards the outside of the card. Read more about the importance of connecting in section 3.2.

An important aspect is that the cable from the PSU was an easy way to provide power to the stack. But maybe it is not the most safe way. Because the contact surface on the PCB card is very small. Would the angle pin header of any cause be greasy, from example fingerprints, the small surface would be over heated and would burn the circuit. For future work a smaller contact would have been a better solution. This would have resulted in a larger contact surface that could handle the current with stability. So for now it is very important to not touch the angle pin header too much and not let the PSU be switched on for too long, because it may result in the system being overheated.

As seen in the right picture in figure 3.8 all blue lamps are lit. When the power is connected to the cluster the PCB card lit the blue LED lamp. Sometimes the lamp was lit and sometimes not. Why the lamp went out is hard to say. At first we thought the PCB card was broken but when the lamps were tested everything worked perfectly. So from our own conclusion the PCB card got some sort of start charge from the power supply that made the lamp lit.

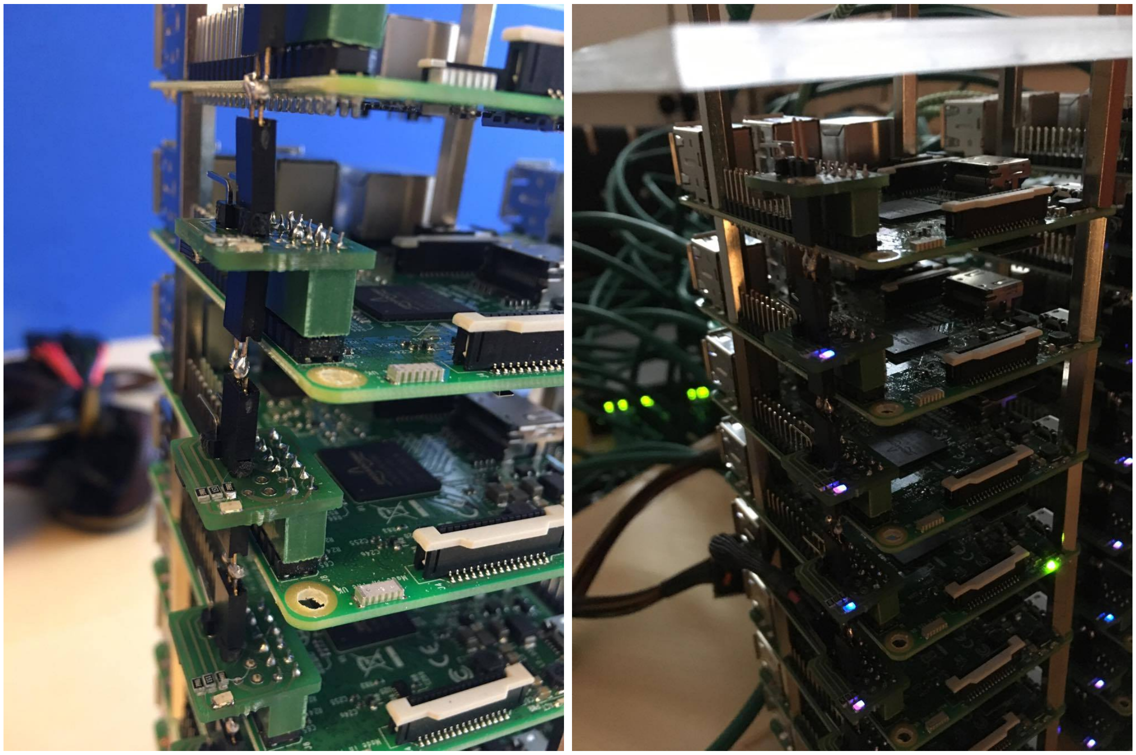


Figure 3.8: The power supply bridge through each Raspberry Pi.

The PCB card was able to bring power to the stack by the unit 500W Corsair Builder Series V2. The power supply did not work at first when it was connected to the stack. This problem established because the 24 pin plug usually is designed to be connected to a motherboard. Usually when the PSU starts, the computer creates a bridge automatically through the motherboard. In our case the 24 pin plug was not attached to anything. By connecting a small cable to the 24 pin plug's grounded and power node, represented by the 15 and 16 hole, the PSU was tricked to produce current at 5V. The shackle and the schedule over the 24 pin contact can be seen in figure 3.9, where the yellow cable is the shackle.

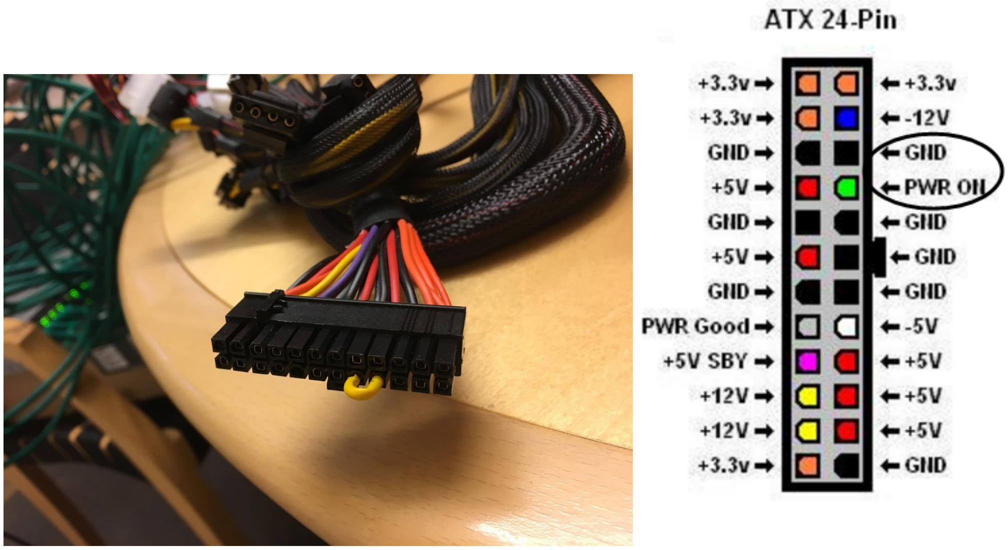


Figure 3.9: The left picture shows the shackle on the 24 pin contact and the right picture shows the schedule over the contact.

3.4 Summary of Chapter

In this chapter the hardware of the system is explained. It gives an overview of the RPi design and how a PCB card works and its importance for the system. The chapter also demonstrates the physical setup and how the components were connected to each other.

Chapter 4

Software Compilation and Installation

Each RPi in our cluster is setup from a common bulk of software. This uniformity in software is a corollary of that the RPi's are identical in hardware. The same OS, hardware drivers, network manager, parallel processing libraries, etc apply to all cluster nodes. This chapter covers the software setup of a single RPi. The storage contents of this RPi could afterwards be replicated onto the storage devices of the other RPi's. The final local configuration changes in the RPi's is covered in chapter 5. The ML framework TensorFlow was compiled from source code by the software build and test automation tool Bazel. Bazel was in turn also required to be compiled from source. Both TensorFlow and Bazel are large pieces of software and more than and RPi's 3 1 GB RAM was needed. To rectify the shortage of internal memory a high-speed 16 GB USB-drive was setup as a swap-drive before performing any compilation.

4.1 Overview of Software Stack

To achieve some goal with a computer system a large collection of software may be required. Typically this collection of software can be partitioned into groups in terms of what hardware/software requirements the software in question has. This collection can be termed as a stack of software or a layered composition of software. At the core of the stack is the physical computer, consisting of CPU, memory, persistent storage, network interface, GPU, etc. In the layer atop the hardware one finds the OS which orchestrates the hardware resources to fulfill two important goals: firstly the OS abstracts hardware details to make the computer easier to program and interact with. Secondly the OS has to provide these abstractions within some performance boundary which is defined by the use case of the complete computer system. The software stack of a RPi in our cluster is shown in figure 4.1. In the center is the RPi itself –the hardware–, in the layer atop is the operating system Arch Linux. Message Passing Interface Chameleon (MPICH) and TensorFlow are the software libraries we want to program with. SSH provides a secure remote shell login which are of great use when managing a cluster. SSH is also one of the underlying protocols of MPICH. NFS is a shared file system, every node in our cluster share home directory to simplify deployment of distributed software. Python is the main programming language when writing client code for TensorFlow.

Protobuf is a protocol which use case is similar to Extensible Markup Language (XML). Protobuf is used in the implementation of TensorFlow. Bazel is a software build and test automation tool which were needed in the compilation of TensorFlow.

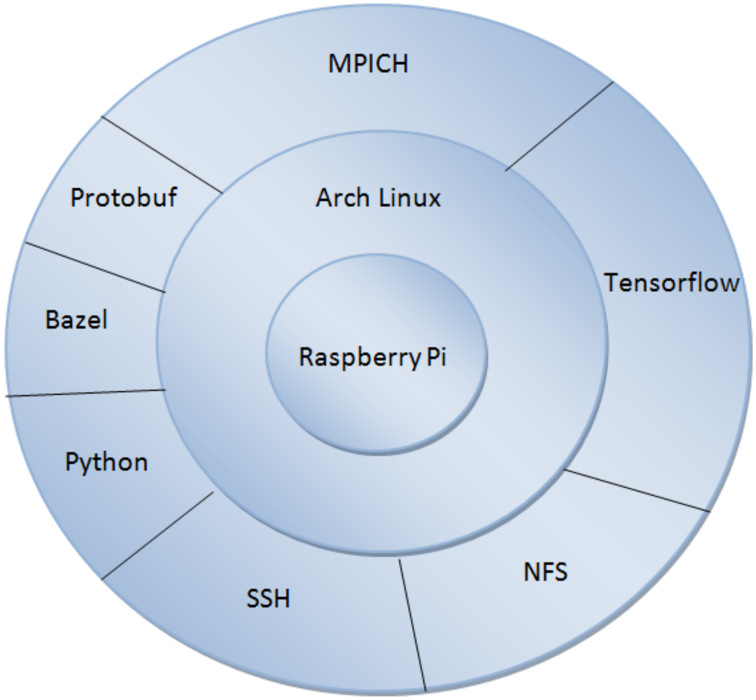


Figure 4.1: Overview of the software stack of one Raspberry Pi.

4.2 Arch Linux

Arch Linux ARM (ALARM) is a Linux distribution that supplies a fairly small set of software from initial installation. This comprises roughly the Linux kernel, GNU user land utilities and the pacman package manager.[34] ALARM is a derivative of Arch Linux. Both these project's goals are to provide a user-centric Linux distribution. A new installation is minimal by default, including the Linux kernel, GNU user land utilities and the project's in-house package manager pacman. The user is assisted by and encouraged to contribute to the Arch Wiki, which provides comprehensive technical documentation of Arch Linux and user land software. This Wiki is highly regarded in the wider Linux community as many articles covers information that is distribution agnostic. Besides the official repositories of pre-compiled software the Arch User Repository (AUR) hosts a large collection of build scripts that has been provided by ordinary users.[59] ALARM has besides the goals mentioned a goal to support a wide range of ARM based architectures.[69] The ALARM project supplies ready to use images for a large number of ARM boards, including the RPi 3. The RPi 3 is based on a Broadcom BCM2837 socket which is an armv8 aarch64 architecture but nevertheless supports the older instruction set armv7. Armv7 is currently the most widely supported platform in terms of drivers and pre-compiled software in ALARM package repositories. Therefore we chose the RPi 2 armv7 image which included a complete GNU/Linux system with drivers, pacman and the users root and alarm pre-configured. A small number of necessary steps to install ALARM to a Secure Digital (SD) memory card was brought out.

1. The SD-card was inserted into a computer running Arch Linux (Most UNIX-like systems can be used) The SD-card should appear as a device file in /dev
2. We partitioned the memory card into two partitions; a FAT file system holding the boot loader and a ext4 file system holding the root file system. The system could be further partitioned to separate /home, /var etc. if desired.

See listing 4.1 for the corresponding shell commands.

Listing 4.1: Partitioning of sd-card and copying of ALARM onto it.

```
# fdisk /dev/sdx
# mkfs.vfat /dev/sdX1
# mkdir boot
# mount/dev/sdX1 boot
# mkfs.ext4 /dev/sdX2
# mkdir root
# mount/dev/sdX2 root
wget http://os.archlinuxarm.org/os/ArchLinuxARM-rpi-2-latest.tar.gz
bsdtar -xpf ArchLinuxARM-rpi-2-latest.tar.gz -C root
sync
```

4.3 Swap Partition on USB-Drive

In order to compile TensorFlow, Bazel, MPICH and Protobuf, more memory than the RPi's 1 GB RAM was required. We utilized a memory drive as a swap drive to overcome this. For this project a drive with 16 GB was used, but anything above 1 GB should work fine.

To setup the swap drive it was inserted into the RPi and the commands of listing 4.3 were executed to initialize the drive with a swap partition. The dev path and Universally Unique Identifier (UUID) of the drive was consulted with the *blkid* utility. To make the swap remain active across boots, an entry to fstab was made. See listing 4.2 below.

Listing 4.2: /etc/fstab.

```
UUID:<UUID> none swap sw,pri=5 0 0
```

Listing 4.3: Creation of swap drive.

```
# mkswap /dev/sda1
# swapon /dev/sda1
```

The compile scripts make use of the swap drive through /tmp. A tmpfs file system is mounted at /tmp and stores its files on the new swap drive, but is by default only assigned half of the system RAM (in our case 1GB). This was expanded to 4GB by executing the command in listing 4.4 below. Note that this is a temporary measure that was used when a large compilation was about to be done.

Listing 4.4: Temporary size increment of /tmp.

```
# mount -o remount,size=4G,noatime /tmp
```

4.4 Protobuf

Protocol Buffers (Protobuf) are used to structure data for efficient binary-encoding format.[79] Protobuf is a mechanism that is both flexible and efficient. It is like a smaller version of XML but much faster and simpler.[37]

Before the installation a few basic packages on which Protobuf depends had to be installed. textitAutoconf is a package that produces shell scripts.[5] *Automake* is a tool that generates Makefile.ins automatically.[6] *Libtool* is a GNU library that supports scripts and supervise *maven.*[27]

Listing 4.5: Installation of the compilation dependencies of Protobuf.

```
# pacman -s autoconf automake libtool maven
```

After the installation of the packages the repository for Protobuf was cloned from GitHubs official web page. It then was configured and installed. The installation took about 30 minutes.

Listing 4.6: Compilation of Protobuf.

```
cd protobuf
git checkout v3.1.0
./autogen.sh
./configure
make -j 4
# make install
# ldconfig
```

When the installation was finished we made sure that the version was correct. The system had now a functioning Protobuf.

4.5 Bazel

Bazel is an open source build and automation tool initiated by Google. Bazel has a built-in set of rules that makes it easier to build software for different languages and platforms.[7] The compilation of Bazel depends on Java and openjdk 8 is currently the recommended Java implementation. The compilation of Bazel require a couple of basic dependencies, see listing 4.7

Listing 4.7: Installation of the compilation dependencies of Bazel.

```
# pacman -s pkg-config zip g++ zlib unzip java-8-jdk
archlinux-java status
```

Bazel required a larger javac heap size than the default to build successfully. At the end of listing *-J-Xmx500M* was appended to allow the Java Virtual Machine (JVM) to allocate more memory if needed when compiling Bazel.

Listing 4.8: -J-Xnx500M was appended to the file script/bootstrap/compile.sh to increase the javac heap size

```
run "${JAVAC}" -classpath
"${classpath}" -sourcepath "${sourcepath}"\
-d "${output}
/classes" -source "$JAVA_VERSION" -target "$JAVA_VERSION"\
-encoding UTF-8 "@${paramfile}"}
```

4.6 TensorFlow

The TensorFlow runtime is a cross-platform library, see figure 4.2 for an overview of its architecture.

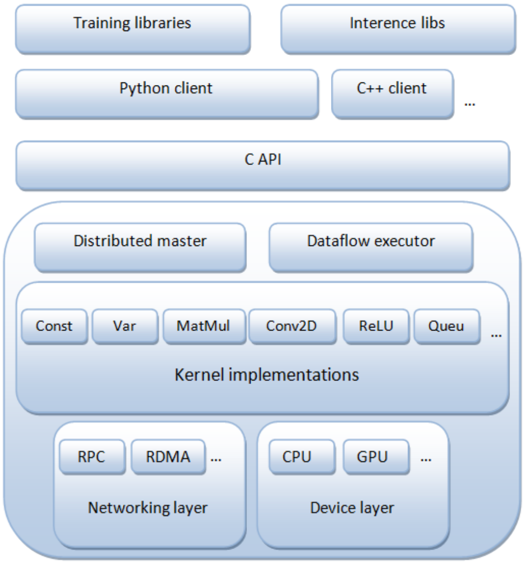


Figure 4.2: Overview of TensorFlow's architecture.[75]

The core of TensorFlow is implemented in C++. All interaction with the core goes through a C Application Programming Interface (API). A TensorFlow program is defined in a client language that has higher-level constructs which ultimately interacts with the C API to execute the client definitions.[65] As of May 2017 Python is the most complete API but language bindings exist for C++, Go, Haskell and Rust. Guidelines on how to implement a new language binding is available in TensorFlow documentation and is encouraged by the TensorFlow authors.[16]

RPi support for TensorFlow is as of May 2017 unofficial and is not merged upstream. Because of this some changes to the source was required. TensorFlow can be setup with either Python 2.7 or 3.3. For this project Python version 3.3 was selected. Python dependencies were installed as shown in listing 4.9.

Listing 4.9: Python dependencies of TensorFlow.

```
# pacman -s python-pip python-numpy swig python
# pip install wheel
```

TensorFlow assumes a 64-bit system and because we installed ALARM in 32-bit all references to 64-bit software implementations needed to be exchanged by 32-bit counterparts. This was accomplished by the command in listing 4.10.

Listing 4.10: References to 64-bit exchanged to 32-bit.

```
grep -Rl ’lib64’ | xargs sed -i ’set/lib64/lib/g’
```

To prevent the RPi from being recognized as a mobile device the line “#define IS_MOBILE_PLATFORM” in the tensorflow/core/platform/platform.h file was removed. Finally the configuration and build was executed as in listing 4.11.

Listing 4.11: Command to initiate the build of TensorFlow.

```
bazel build -c opt --copt="-mfpu=neon-vfpv4"
--copt="-funsafe-math-optimizations" --copt="-ftree-vectorize"
--copt="-fomit-frame-pointer" --local_resources 1024,1.0,1.0
--verbose_failures tensorflow/tools/pip_package:build_pip_package
```

When the build was finished after 3,5 hours, the Python wheel could be built by using the built binary and then installed. The system now had a working machine learning platform, TensorFlow. See listing 4.12 for commands.

Listing 4.12: Installation of the Python wheel containing TensorFlow.

```
bazel-bin/tensorflow/tools/pip_package/build_pip_package/tmp~
/tensorflow_pkg
# pip install /tmp/tensorflow_pkg/tensorflow-1.0.0-cp27-none-
linux_armv7l.whl
```

4.7 MPICH

MPICH can be explained as a *“high performance and widely portable implementation of the Message Passing Interface (MPI) standard”.[41]* MPICH was available in the AUR but two dependencies was missing from the build script, numctl and sowing. numactl could be skipped and sowing was installable from the AUR. Then the build ran for 1.5 hours and was successfully installed.

4.8 Summary of Chapter

This chapter has given some notes regarding the implementations of the software present in our cluster. Some of the software was compiled from source and instructions on how to succeed in this on an RPi have been reviewed.

Chapter 5

Setting Up an MPI Cluster

To build a functioning cluster four major components are required: The computer hardware, Linux software, a parallel processing library and an ethernet switch. Both hardware and software have been explained in previous chapters. Next step is to explain how to set up an MPI library. For the implementation of MPI we chose to work with MPICH, but Open MPI is also an alternative. The two MPI libraries almost entirely works in the same way. MPI is a set of API declarations on message passing, while Open MPI is an API that is all about making it easier to write shared-memory multi-processing programs.[81]

This chapter will explain how to set up a successful parallel computer using MPI and other components. To make the nodes in the system communicate with each other a host file has to be set up, to map the host names to the IP addresses in the network. The nodes will talk over the network using SSH and be able to share data through NFS, which will be explained in this chapter. The installation of the software and MPICH has already been explained in chapter 4.

5.1 Network File System and Master Folder

A host file was first created and transfered to every node. This file included all IP addresses for the cluster. The host file was placed in the /etc/ directory for all the RPi's. Every node, both master and slave, had the same host file in their respective /etc directory.

Listing 5.1: Host file for the network.

```
# IP        Name
-------------------
127.0.0.1   localhost
10.0.0.100  rpi00
10.0.0.101  rpi01
10.0.0.102  rpi02
10.0.0.103  rpi03
10.0.0.104  rpi04
...         ...
```

When all units had access to the host file, the Network File System (NFS) was set up. The NFS is a server/client application that optionally updates and stores files on a remote computer. The mounted NFS file system is mounted on a directory as one does with local file systems, from the users perspective it has the same appearance as a local file system. The installation was made by installing the *nfs-server* on the server and the *nfs-client* on the client machines.

In order to store all data in one common folder a *master folder* was created. By sharing the folder from the master node to the slaves, they could access it using NFS. To be able to export the master folder an entrance had to be set up from the master node. By adding the following two lines to */etc/export* this was achieved, see listing 5.2.

Listing 5.2: Entrance from master node.

```
/srv/nfs 10.0.0.0/24(rw, fsid=root, no_subtree_check)
/srv/nfs/alarm 10.0.0.0/24(rw, no_subtree_check, nohide)
```

By writing the IP address to the subnetwork (10.0.0.0) all nodes in the subnet will get access to the folder. The folder was given read and write privileges. The *no_subtree_check* was necessary to prevent the checking of the subtree. Because the NFS performs scans over every directory that is above it when it is a shared directory. By adding the line it stops the NFS from scanning. In order to let the NFS be able to identify the exports from each filesystem, we had to identify explicitly the filesystem to the NFS. This was done by adding *fsid=root*. By setting the folder to option *nohide* the folder would not be hidden from the clients. Last step for the server was to add a line in */etc/fstab*. This was necessary to make it stick when the client reboots.

In listing 5.3 the bindings between the exported directory /srv/nfs/alarm and the wanted directory is demonstrated.

Listing 5.3: Bindings between directorys.

```
/home/alarm /srv/nfs/alarm none bind 0
```

The same method had to be used from the client's side. In order to make it permanent so the mounting always sticks when the system is rebooting. Otherwise the command has to be written every time the RPi's were started.

5.2 SSH Communication

SSH communication is required to let the server identify itself using public-key cryptography. To let the system use a SSH key it protects the system from outside eavesdropping. This makes it harder for attackers to brute-force the system. The SSH key works by using two different keys, one public key and one private key. By having two keys the system can protect the private key and share the public key with whom it wants to connect with.

First a SSH key had to be created and set up. This was made by the *ssh-keygen* command, see listing 5.4. We used the cryptography function ed25519 as a signature when it has better performance then Elliptic Curve Digital Signature Algorithm (ECDSA) and Digital Signature Algorithm (DSA). Ed25519 can easily be explained as an "*elliptic curve signature scheme*".[48]

Listing 5.4: The cryptography function in use.

```
ssh-keygen -t ed25519
```

When public/private key pair fingerprint had been created the ssh-agent had to be set up. A key's fingerprint is a unique sequence of letters and numbers.[72] Fingerprints are used to identify the key. It works in the same way as two different persons fingerprints, they can never be identical.

When the keys were created the ssh-agent was installed. To be able to set the service file for the SSH key properly the system unit, the service and the install for the ssh-agent had to be added, see listing 5.5.

Listing 5.5: SSH-agent is started with systemd user.

```
[Unit]
    Description= SSH key agent
[Service]
    Type=forking
    Environment=SSH_AUTH_SOCK=%+/ssh-agent.socket
    ExecStar=/usr/bin/ssh-agent -a $SSH_AUTH_SOCKET
[Install]
    WanteBy=default.target
```

The last step to get a functioning SSH key was to export the SSH_AUTH_SOCKET to the .bash_profile, see listing 5.6. The system could then start the key.

Listing 5.6: .The exported socket to the bash profile.

```
export SSH_AUTH_SOCK = "$XDG_RUNTIME-DIR/ssh-agent.socket"
```

All nodes in the cluster got retrieved to the masters public key. When the master node wants to get access to a node, the node will automatically log in using the SSH key.

When the SSH key was placed on all nodes it did not work properly. We could not say why, but after researching more about SSH key it was realized that a Keychain works better. Keychain is designed to easily manage the SSH keys with minimal user interaction.[25] Keychain drives both ssh-agent and ssh-add and is implemented as a shell script. A great feature with Keychain is that it is possible to maintain a single ssh-agent process across multiple login sessions, which makes it possible to only enter the password once when the machine is booted.

Keychain was installed using of pacman. To tell the system where the keys are and to be able to start the ssh-agent automatically the *bashrc* file was edit, see listing 5.7.

Listing 5.7: Keychain setup.

```
if type keychain >/dev/null 2>/dev/null; then
keychain -nogvi -q id_ed25519
[-f ~/.keychain/${HOSTNAME}-sh] &&
. ~/keychain/${HOSTNAME}-sh
[-f ~/.keychain/${HOSTNAME}-sh-gpg] &&
. ~/keychain/${HOSTNAME}-sh-gpg
fi
```

5.3 Summary of Chapter

This chapter describes how a parallel system is constructed. How the host file is created and how NFS works. The chapter also explains and demonstrates how a SSH key is working and how it uses fingerprints. In the chapter the use of Keychain for key management instead of just ssh-agent is discussed.

Chapter 6

Cluster Software Testing

In this chapter we tested the cluster by running different programs. The first MPI program is demonstrated and two different ways of executing the program are shown, mpirun and mpiexec. The chapter also explain the distributed training and different training methods such as: synchronous/asynchronous training and in-graph/between-graph replication.

The chapter ends with a presentation of the MNIST program. The MNIST program consists of handwritten digits images of and calculates the accuracy and the total cost value for the recognition system.

6.1 A First MPI Program

The first parallel program that was written to contact all nodes in the cluster. Was a simple MPI Hello World program. The program was found on Ubuntu’s official documentations website.[20]

Listing 6.1: The first MPI prgram.[20]

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

int main (int argc, char** argv){}
int myrank, nprocs;
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
printf("I’m Alive: %d // %d\n", myrank, nprocs);

MPI_Finalize();
return 0;
}
```

Lets explain the code and explain some parts from listing 6.1. To build an MPI program the first step was to include an MPI header file. In this program mpi.h was chosen, but mpif.h is an alternative header. For more advanced MPI programming, mpi_f08 is more efficient to use, because it provides a newer interface for Fortran.[81] But for now the mpi.h directory is a good start. The directory contains necessary decelerations types for MPI functions.

Next the MPI environment was initialized; MPI_Init. During the initialization all global and internal variables were constructed. After the initialization two main functions were called. The functions MPI_Comm_size and MPI_Comm_rank are called in almost every MPI program, see the functions in listing 6.2.

Listing 6.2: The typical functions MPI_Comm_size and MPI_Comm_rank.

```
MPI_Comm_size(
    MPI_Comm comunicator,
    int* size)
MPI_Comm_rank(
    MPI_Comm comunicator,
    int* rank)
```

MPI_Comm_size determines the size of a communicator.[31] MPI_COMM_WORLD is a communicator group and contains every MPI process that is used in the system.[32] In this program the MPI_COMM_WORLD encloses all processes in the cluster.

All processes in the group has a rank number that has been numbered with consequent integers and begins with 0. To let each process find its own rank in the group that the communicator is associated with, the function MPI_Comm_rank is called.[81] The rank number is primarily used to identify a process when a message is sent or received. Thus, in this program each process will get a number from nprocs.

The last step is MPI_Finalize and the function must be called by every process in the MPI computation. The function terminates the system and cleans up the MPI environment. After MPI_Finalize no more MPI calls can be made. In particular, no more initialization can be done.

6.1.1 Running the First MPI Program

Then the program will be executed after the compilation is done. To run the MPI program the host file had to be used. If the program would have been ran at a single machine or a laptop the additional configurations would not be required.

It may vary from one machine to another how the MPI standards are launched. But several implementations of MPI programs can be used with the syntax in listing 6.3.

Listing 6.3: Executing with mpirun.

```
mpirun -n 32 -f machinefile ./mpi.hello_test
```

It may require different MPI implementation commands to start an MPI program. The mpiexec command is strongly recommended by the MPI standards and it provides a uniform interface of started MPI programs, see listing 6.4.

Listing 6.4: Executing with mpiexec.

```
mpiexec -n 32 -f machinefile ./mpi.hello_test
```

The execution will start 32 MPI processes and set the MPI_COMM_WORLD to a size of 32. As seen in both execution commands a machinefile is used. The machinefile consists of all IP addresses in the system.

The result of the program is that the master node receives a text string from all nodes in the cluster saying; *"I'm Alive: xx //xx"* and prints the string in the terminal.

6.2 MPI Blink Program

The blink program served as a test that confirmed that the MPICH and the cluster in general, worked as planned. The program was found on Kiepert's git repository.[26]

The blink program produces different light patterns by using the LED RGB lamps on the PCB cards. For instance it produced, circle and zig-zag patterns. The program works by letting the nodes synchronize with MPI. By simply looking at the patterns of the lights, one could confirm that the cluster was setup correctly. Incorrect patterns is am immediate indicator that something is wrong.

First we got completely wrong light patterns. We discovered that the program had a if condition, a preprocessor command *#define NETWORK_LAYOUT* which decided between two light pattern definitions. By simply removing this line we got correct light patterns.

6.3 Distributed Training with TensorFlow

Data parallelism is a common training configuration. It involves multiple tasks in a worker job and can train the same model in small batches of data and update the shared parameters in the tasks of a parameter servers job.[14] A network in distributed fashion and can be trained in two different ways, synchronous or asynchronous. Asynchronous is the most typical used training method.

Synchronous training is when all graph replicas read input from the same set of current parameter values. Then, gradients are computed in parallel and finally applied together before the next training cycle begin. Asynchronous training is when every replica of the graph has a training loop. These loops are independent from each other, they execute asynchronously. The two training methods can be seen in figure 6.1.

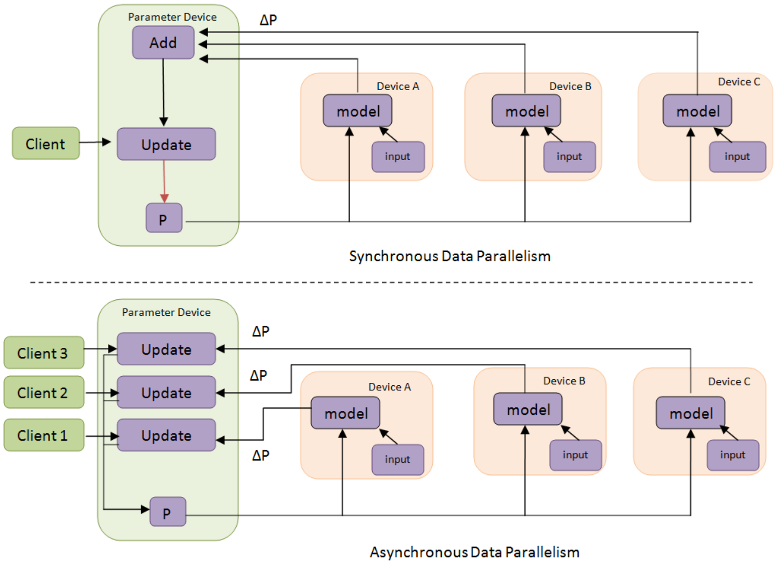


Figure 6.1: Synchronous and asynchronous data parallel training.[14]

TensorFlow can be structured in many different ways. Possible approaches are in-graph replication and between-graph replication.

For in-graph replication only one dataflow graph is built by the client, seen in figure 6.2. The graph consists of one set of parameters and multiple duplicates of the compute-intensive operations. Each of the computer-intensive operations is designated to a different task in the worker job.[43]

For between-graph replication every task in the worker job is setup with a client, seen in figure 6.2. All clients builds a dataflow graph that consists of parameters bound to the parameter server job and a copy of the compute-intensive operations bound to a local task in the worker job.[43] Our training program implements asynchronous training and between-graph replication. This is as of spring 2017 the most common setup found on the internet and was chosen for this reason.

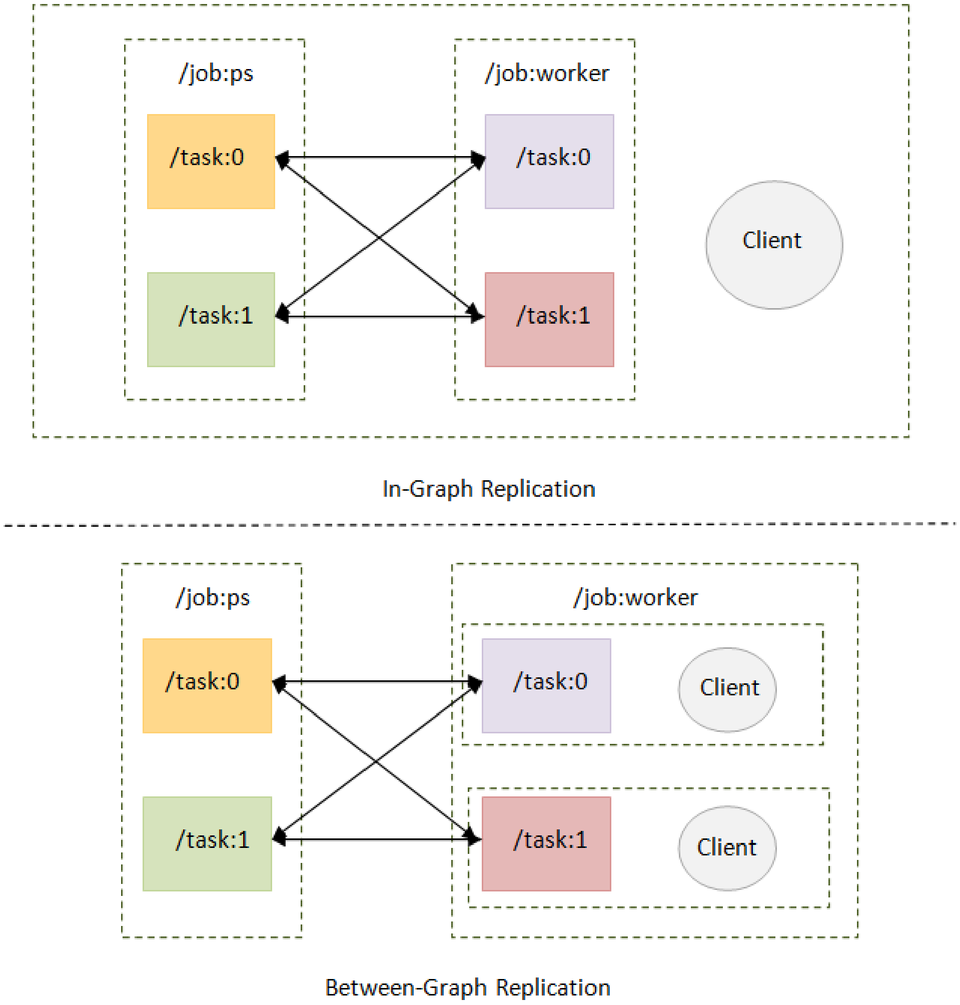


Figure 6.2: In-graph replication and between-graph replication.[44]

6.4 Background MNIST

MNIST stands for Modified National Institute of creator of dataset Standards and Technology. MNIST can be explained as a dataset for simple computer vision.[29] Computer vision is an *"interdisciplinary field that deals with how computers can be made for gaining high-level understanding from digital images or videos"*.[10]

MNIST consists of handwritten digits, for each image the MNIST provides a label. The label tells the system witch digit it is. For example, the labels for the images in figure 6.3 are 5, 0, 4 and 1.

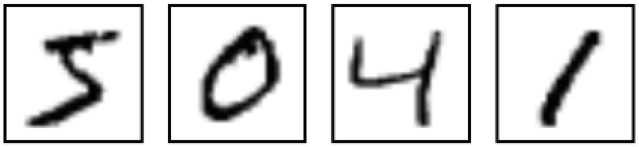


Figure 6.3: For images from the MNIST dataset.[55] License [11].

The MNIST images that were used for the test were hosted on Yann LeCun's website.[62] The images were downloaded and placed in a folder with the program. By downloading the images from the master node and then place them in the home directory, all nodes in the cluster could reach them with NFS.

The data are split into three parts in MNIST. The first part consists of training data; mnist.train. The second consists of test data; mnist.test. And the third part consist of validation data; mnist.validation.[29] These three parts are important, because in machine learning it is an essential part to have separated data, to make sure that what we have actually learned generalize.[29]

Each data point of MNIST has two parts: a handwritten digit image and a corresponding label. The training set contain both a figure and the corresponding labels. Each image consist of 28 pixels by 28 pixels, see example in figure 6.4.

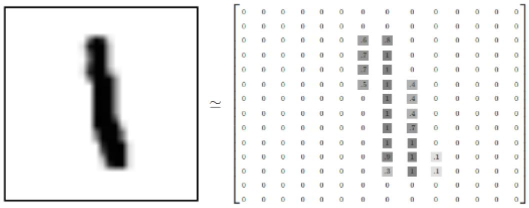


Figure 6.4: The image as a vector of 784 numbers.[55] License [11].

The right picture in figure 6.4 can be seen as a vector of 784 numbers, because 28x28 = 784. The MNIST image can from this perspective look like a vector with a bunch of points in a 784-dimensional vector space.[67]

6.4.1 The MNIST Program

To be able to run a distributed TensorFlow script all nodes in the cluster had to know who am I and who is who. By using a ClusterSpec object it was possible to achieve this. The ClusterSpec object can be seen as a constructor that binds the host-addresses to job names, see listing 6.5. All nodes in the system have the same specifications. The specification is usually made up of parameter servers and workers.

A parameter server can be explained as a shared multi-variable dictionary and can be accessed via API network.[35] The server is used to store parameters for the nodes during runtime. The parameter server's only task is to maintain the shared parameters, meanwhile the workers task is to compute some or all performing for the TensorFlow graph.[73]

After researching which MNIST program to use we choose to work with Imanol Schlag's MNIST program, that was found on his git repository.[30] Every MNIST program that was found looked almost similar, the programs contained the same parts.[57][56][50] The different was that Schalg had a good documentation on how the program worked. It should be noted that Schalg acknowledged that his program was written for improving his own understanding of distributed TensorFlow.[30] Thus, the program may not be perfect.

Listing 6.5: The cluster specifications.

```
parameter_servers = ["rpi01:2222",
                     "rpi02:2222",
                     "rpi03:2222",
                     "rpi04:2222"]

workers = ["rpi05:2223",
           "rpi06:2223",
           "rpi07:2223",
           "rpi08:2223",
           ...
           "rpi32:2223"]

cluster = tf.train.ClusterSpec({"ps":parameter_servers, "worker":workers})
```

The parameter servers and workers are called jobs and contains of basically one or several tasks. A task is unique and a node can have multiple tasks that can be run at the same time. This is possible if e.g. a machine has multiple GPU's.

Listing 6.6: Input flags and a server is started for a specific task.

```
tf.app.flags.DEFINE_strings("job_name", "", "Either ’ps’ or ’worker’")
tf.app.flags.DEFINE_integer("task_index", 0, "Index of task with the job")
FLAGS = tf.app.flags.FLAGS

server = tf.train.Server(cluster,
                         job_name=FLAGS.job_name,
                         task_index=FLAGS.task_index)
```

In listing 6.6 the initialization for a running machine is setup. An important aspect is to run the correct task on the correct node, for example in our program *rpi05* is a worker with task_index 0. During the initialization a server is started for every process in the cluster.

Listing 6.7: The log writer, Summary FileWriter.

```
writer = tf.Summary.FileWriter(logs_path.graph = tf.get.default_graph())
```

Before the training loop the system goes through a log writer, a so-called Summary FileWriter see listing 6.7. The logs contains snapshots of variables. The class Summary FileWriter is a mechanism to add summaries to a given directory by creating an event file.[61] The content in the file updates asynchronously in the class. By using asynchronously updates the data can be added to the file directly from a training program, without disturbing the training session.

With Summary FileWriter TensorBoard is available. TensorBoard is a computation that can be used in TensorFlow for visualizing learning. With TensorBoard a TensorFlow graph can be visualized and additional data can be shown as a image.[51]

Listing 6.8: The start of the training loop.

```
if FLAGS.job_name == "ps":
server.join()
elif FLAGS.job_name == "worker":

    with tf.device(tf.train.replica_device_setter(
                        worker_device="/job:worker/task:%d" % FLAGS.task_index,
                        cluster=cluster)):
```

Before the program goes into the training loop the parameters are checked to determine, which one is a parameter server and which one is a worker. Only the worker node goes into the training loop, see listing 6.8. In the training loop variables are configured and every worker's computation is defined. Inside the training loop the system goes through something that is called a softmax regression.

We know that all handwritten digit images in MNIST are between zero and nine. A given image can then only be one out of ten possible things. The system is looking at the image and calculates the probabilities for being a certain digit. An example, our program looks at an image picturing a shape of a nine and is 80 % sure it is a nine. But gives 5 % probability that it is shape of eight, because of the top circle. This makes the program not 100 % sure, in most cases. In cases like this the softmax regression is a simple model. There are two steps in a softmax regression, first the evidence of the input in certain classes is added, and then the evidences are converted into probabilities.[47]

To be able to calculate a particular class evidence of a given picture a weighted sum is performed on the pixel intensities. If the pixel has a high intensity the weight is negative because it is evidence that tells that the image is not in that class. A positive weight indicate that it is part of the class.[70]

Then a so called *bias* is added as extra evidence. A given class result of evidence for the input can be described as:

$$evidence_i = \sum_j W_{i,j}\, x_j + b_i, \tag{6.1}$$

where W_i are the weights and b_i is the bias for the class i, and the index for the summation over the pixels is j for our input image x. Using the softmax function:

$$y = softmax(evidence), \tag{6.2}$$

the evidence can be converted into predicted probabilities y :

$$softmax(x) = normalize(exp(x)), \tag{6.3}$$

the equations can then be expanded and put together to equation 6.4 to make the calculation easier:

$$softmax(x)_i = \frac{exp(x_i)}{\sum_j exp(x_j)} \tag{6.4}$$

The softmax regression equations can also be demonstrated with pictures. But of course with more x's in reality in comparison to the figure 6.5.

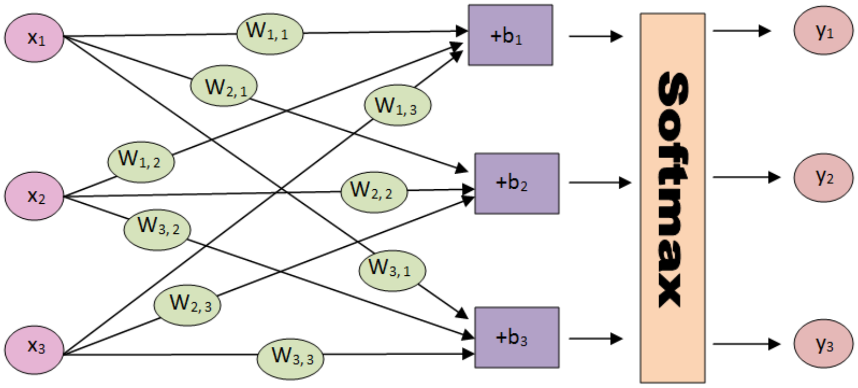


Figure 6.5: The weighted of sum of x's is computed, a bias is added and then the softmax is applied.[47] License [11].

The picture in figure 6.5 can be written as an equation, see figure 6.6.

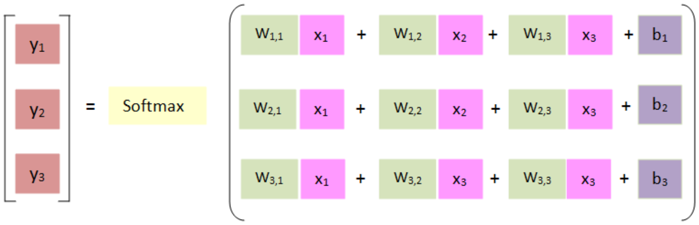


Figure 6.6: Function of the softmax.[47] License [11].

The equation in figure 6.6 can then be vectorized. By turning the equation into a matrix multiplication the computation is more efficient.

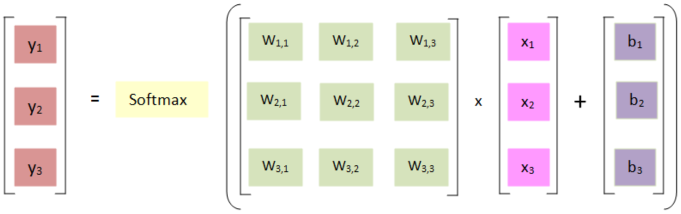


Figure 6.7: The vectorized matrix of the softmax equation.[47] License [11].

After the softmax regression the program goes into a training supervision class, seen in listing 6.9.

Listing 6.9: The training Supervision class.

```
sv = tf.train.Supervison(is_chief=(FLAGS.task_index == 0),
                              global_step=global_step,
                              init_op=init_op)
```

The sessions needs to be in order to be able to run the training cycle. One node was selected as chief to run the distributed setting. In our case the chief was a worker node with task number 0, *rpi05*. The chief's main task is to manage the rest of the cluster. The supervisor object is handled by the chief, see listing 6.9.

In the implementation model the global_setup variable was declared. The variable will after every update be incremented by one. To be able to compare different cluster configurations the random seed was set to 1.

To be able to train our model the definition of a good model has to be setup. In machine learning it is typical to define what it means for a model to be bad. This is called *cost*, the cost represents how far our model is from desired outcome.[66] The model will be better as smaller the error margin is. A function that determines the cost of the model is called *cross-entropy*. The cross-entropy can be defined as:

$$H_{y'}(y) = -\sum_i y'_i log(y_i), \tag{6.5}$$

where the predicated probability distribution is y and the true distribution is y'. The cross-entropy measures how inefficient the predication is for describing the truth. Equation 6.5 can be implemented as listing 6.10.

Listing 6.10: The implemented cross-entropy.

```
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y),
reduction_indices=[1]))
```

Each element of y is computed with the logarithm as seen in listing 6.10. Next the corresponding element of tf.log(y) is multiplied with each element of y_. Then the elements in the second dimension of y is added to the tf.reduce_sum because the reduction_indices = [1]. The last step is that the mean value over the examples in the batch is computed with reduce_mean. [66]

Listing 6.11: Command to start a process.

```
ssh alarm@rpi01 example.py --job_name="ps" --task_index=0
ssh alarm@rpi02 example.py --job_name="ps" --task_index=1
ssh alarm@rpi03 example.py --job_name="ps" --task_index=2
ssh alarm@rpi04 example.py --job_name="ps" --task_index=3

ssh alarm@rpi05 example.py--job_name="worker" --task_index=0
ssh alarm@rpi06 example.py --job_name="worker" --task_index=1
...
ssh alarm@rpi32 example.py --job_name="worker" --task_index=27
```

To start the program each node was assigned with a job name, either "ps" or "worker" and a task_index. In our cluster four processes were started as parameter servers and 28 processes as workers, see listing 6.11.

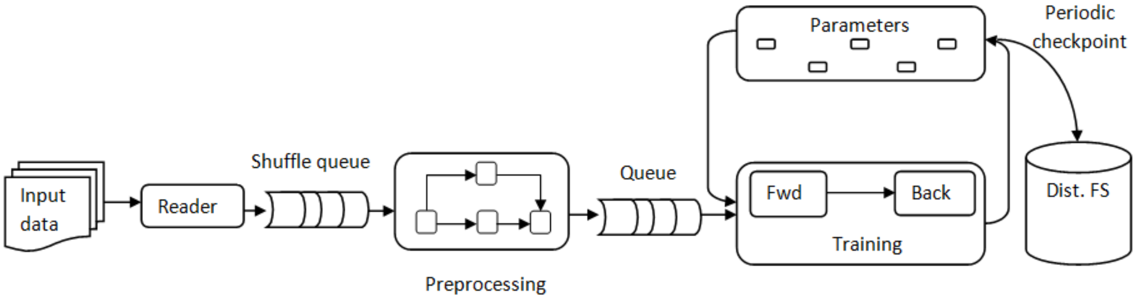


Figure 6.8: A training pipeline.[75]

The training program can be explained as a schematic dataflow graph, see figure 6.8. The figure contains input data, training, input processing, a checkpoint state and the update rules for the parameters. In a machine learning algorithm all computations and states are represented in a single dataflow graph in TensorFlow. The graph includes individual mathematical operations and the preprocessing inputs.[75]

In section 7.2 a number of tests is performed and analyzed. The performance of the cluster is evaluated.

6.5 Summary of Chapter

In this chapter the test programs is discussed and explained. The MPI and MNIST program are the two main parts of the chapter. TensorFlow's visualization learning program TensorBoard is discussed, and two different ways of distributed training are demonstrated.

Chapter 7

Evaluation

In this chapter the design and performance of the cluster is evaluated. The design is made so that additional nodes can be added in the future. The MNIST program in chapter 6 is used to demonstrate the scalability and performance of the design.

7.1 Layout of our Beowulf Cluster

The architecture for our cluster can be seen in figure 7.1. By comparing it to the Beowulf cluster reference layout in figure 2.1, one can confirm that our cluster indeed is a Beowulf cluster. It includes 32 RPi's, a 10/100 switch with 48-ports, Arch Linux ARM, MPICH and TensorFlow.

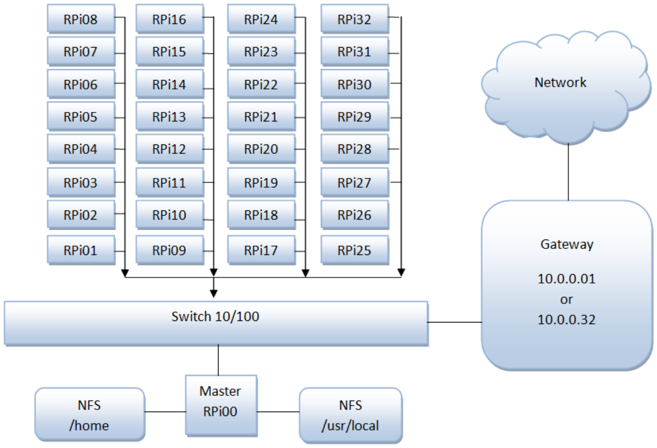


Figure 7.1: The Raspberry Pi Clusters Network Architecture.

The process of adding next RPi's is trivial. Software is copied from one of the RPi's SD-card in the cluster onto a empty SD-card. The only required change is the new nodes IP-address that has to be unique inside the cluster. Of course, one needs to add additional PSU's and switches if required.

7.2 Analysis of Distributed MNIST

This test is based on the MNIST program that is explained in-depth in chapter 6. Parallelism is the aspect we are interested in. The fact that the machine learning algorithm (single layer neural network) used in this MNIST implementation in itself is inefficient and produces good accuracy "slowly", is ignored as it does not interfere with how the parallelism scales.

The test comprises of a number of program runs with a varying number of workers. To keep the test simple the number of ps nodes is kept at four constantly. Another test design could be to vary the number of ps nodes to facilitate a balance between the number of worker and ps tasks. The aim of this balance would be to maximize utilization of the available hardware resources as new nodes are added to the cluster. The cluster has 32 interior nodes. As previously mentioned, throughout all test-runs four of the nodes keep one ps task each. Additionally for each test-run two thus far unassigned nodes are assigned one worker task each. New worker tasks are assigned until all of the 32 nodes in the cluster are doing work. In total this gives 14 test-runs wherein the last test there exists 28 workers.

The program uses between-graph replication and asynchronous updates. The expected behavior when adding more workers is a higher accuracy. This should also happen without any significant increase in test-run duration. It is expected to be some overhead due to communication costs between nodes but should be negligible in comparison to a node's added processing capacity and the considerably larger size of a batch. A small batch size would increase the number of messages sent between nodes and consequently also increase overhead. Every message has startup time that includes preparing the message with various network metadata.[76]

Table 7.1: 14 Test runs of distributed MNIST with an increasing number of workers.

No. of Workers	Final Cost	Accuracy [%]	Total Time [s]	Average Step Time [ms]
2	7.0060	13	59.89	95
4	5.0215	14	88.38	145
6	4.9934	16	123.89	210
8	4.7603	18	170.10	270
10	4.5224	19	212.30	310
12	4.9248	21	236.29	380
14	4.2357	23	273.01	480
16	4.3295	24	314.17	550
18	3.3875	26	345.53	660
20	3.5834	28	385.43	750
22	3.3086	29	407.18	710
24	2.9577	31	467.66	860
26	3.2574	32	489.46	910
28	2.9098	34	546.83	990

The results of the whole test can be seen in table 7.1. The attributes *Final Cost*, *Accuracy*, *Total Time* and *Average Step Time* have been recorded. A single step represents the processing a single example-label pair. *Final Cost* and *Accuracy* both reflects the efficiency of the resulting handwritten-numbers-recognizer. If the ML algorithm is correct then for every training cycle the *Final Cost* should decrease and the *Accuracy* should increase. The plot in figure 7.2 was drawn from the data in table 7.1. Figure 7.2 shows that the *Accuracy* grows at a linear rate of approximately 0.8 $\%/worker$. The *Accuracy* follows the number of training epochs, more epochs results in a higher *Accuracy*. The "*epoch value*" is set to one throughout the whole test. The total number of processed epochs is calculated as "*epoch value*" $*$ "$No.\ of\ Nodes$", every node adds one epoch.

Table 7.2: 14 Test runs of distributed MNIST with one ps and one worker task on one RPi 3 with an increasing number of epochs.

No. of Epochs	**Final Cost**	**Accuracy [%]**	**Total Time [s]**	**Average Step Time [ms]**
2	6.5910	13	30.51	23.2
4	6.4422	14	59.72	24.8
6	5.1001	16	87.44	24.4
8	4.9842	18	116.04	24.8
10	4.5884	19	145.17	25.0
12	4.6271	21	175.53	25.2
14	4.3696	23	204.81	25.4
16	3.9588	24	233.21	25.5
18	3.0994	26	258.99	25.4
20	3.1738	28	291.01	25.5
22	3.3766	29	317.87	25.4
24	3.4285	31	349.97	25.5
26	3.4694	32	374.02	25.3
28	3.3403	34	403.51	25.3

To have a reference point of the parallel performance a similar test was brought out on a single RPi 3, see results in table 7.2 and figure 7.3. The total number of processed epochs was adjusted in accordance with the previous test. Instead of adding new nodes to increase the number of epochs the "*epoch value*" was changed. By comparing figure 7.2 and figure 7.3 it is evident that their accuracies align. In this aspect there is successful parallelization.

On the other side the *Average Step Time* and consequently *Total Time* increases along with the *Accuracy* improvement with a rate of approximately 35 $ms/Worker$ and 19 $s/Worker$ respectively. Figure 7.2 shows the constantly growing lines of the *Average Step Time* and the *Total Time.* This is an unexpected behaviour and completely negates the profits of the increased *Accuracy.* The single node *Average Step Time* stays at approximately 25.5 ms throughout the whole test, which is many times faster even for the two worker node test. This was also tested with a larger number of epochs (560) where the same performance pattern continued, see table 7.3. The *Average Step Time* is expected to increase when adding nodes but not at a rate of 35 $ms/Worker$. Thus, an error is present.

Table 7.3: Comparison between 1 node and 32 nodes when running 28 and 560 epochs.

No. of Nodes	No. of Epochs	Final Cost	Accuracy [%]	Total Time [s]	Average Step Time [ms]
1	28	3.3403	34	403.51	25.3
32	28	2.9098	34	546.83	990
1	560	0.7403	83	8023.73	25.5
32	560	0.5461	83	10825.66	990

The Cisco switch has a total capacity of 13.6 Gbps.[12] A single RPi's ethernet network interface can output 100 Mbps. The sum of all 33 RPi's network output gives 3.3 Gbps which is far from the switch's total capacity at 13.6 Gbps. Hence, the switch is not a bottleneck.

The source of the error seems to be in software, either in the MNIST implementation we used or in the implementation of TensorFlow itself.[30] Debugging the MNIST program has been difficult for two reasons. Firstly, the documentation and program examples of Distributed TensorFlow is currently scarce as of May 2017. Distributed TensorFlow is a recent ML framework (released in April 2016), so this is expected. Secondly our goal has not been to go in-depth into the programming model of TensorFlow. Rather, our goal has been to evaluate the clusters performance when training NN. We compiled TensorFlow from the development branch (master) and it is possible that we have encountered a bug. Stack Overflow is the main forum for ordinary users having technical problems with their TensorFlow programs.[71][54] We found a forum-thread in which the author run a distributed TensorFlow program and just like us get slower performance when adding machines.[17] A bug report is linked in this forum-thread.[53] This bug report describes a bug present in gRPC, the network communication framework on which TensorFlow depends. If this is the actual source of our error is not certain, although possible.

Table 7.4: Comparison of an RPi 3 and an Intel based laptop (i5-5250U, 2.7 GHz, 2 cores, 4 threads) with 8 GB RAM. (Micron 2x4 GB, synchronous DDR3, 1600 MHz) Both have one ps and one worker task.

Machine	No. of Epochs	Final Cost	Accuracy [%]	Total Time [s]	Average Step Time [ms]
Intel laptop	560	0.4759	83	1219.98	3.65
RPi 3	560	0.7403	83	8023.73	25.5

To get a sense of a single RPi's speed a comparison to a Intel processor (i5-5250U, 2.7 GHz, 2 cores, 4 threads) with 8 GB RAM (Micron 2x4GB, synchronous DDR3, 1600 MHz) was done. See table 7.4. The Intel is approximately seven times faster than the RPi 3. This favors the RPi 3 in terms of performance for the money. The laptop (manufactured in 2015) had a price of $700 (6500 sek) and the Raspberry Pi 3 $50 (400 sek).

The cluster has 32 worker nodes, therefore it should in theory be comparable to four modern Intel based laptops (with previously mentioned specifications) as seven nodes is comparable to one laptop.

To exploit the full capacity one need to run algorithms that is parallelizeable. Algorithms may have different levels of parallelization. A program can be split into a serial part and a parallel part. Amhdahl's law demonstrates a simple and important principle:

> **Amdahl's Law (Gene Amdahl, 1967)**
>
> If S is the fraction of a calculation that is serial and $1-S$ the fraction that can be parallelized, then the greatest speedup that can be achieved using P processors is:
>
> $$\frac{1}{(S+(1-S)/P)}$$
>
> which has a limiting value of 1/S for an infinite number of processors.[78]

Certain parts of a program may not be possible to split up in concurrent tasks. Amdahl's law shows that a program with a mix of parallelizeable and serial code, will have the serial codes run time as an upper bound on the speedup of adding additional processors.

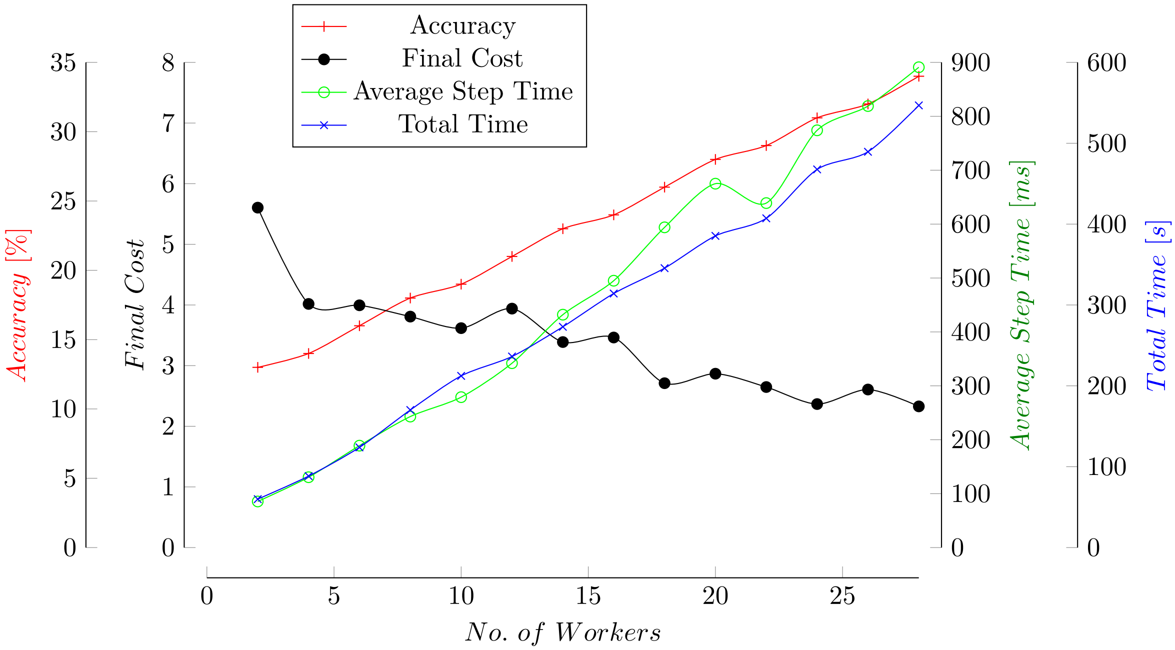


Figure 7.2: Comparison of data from table 7.1.

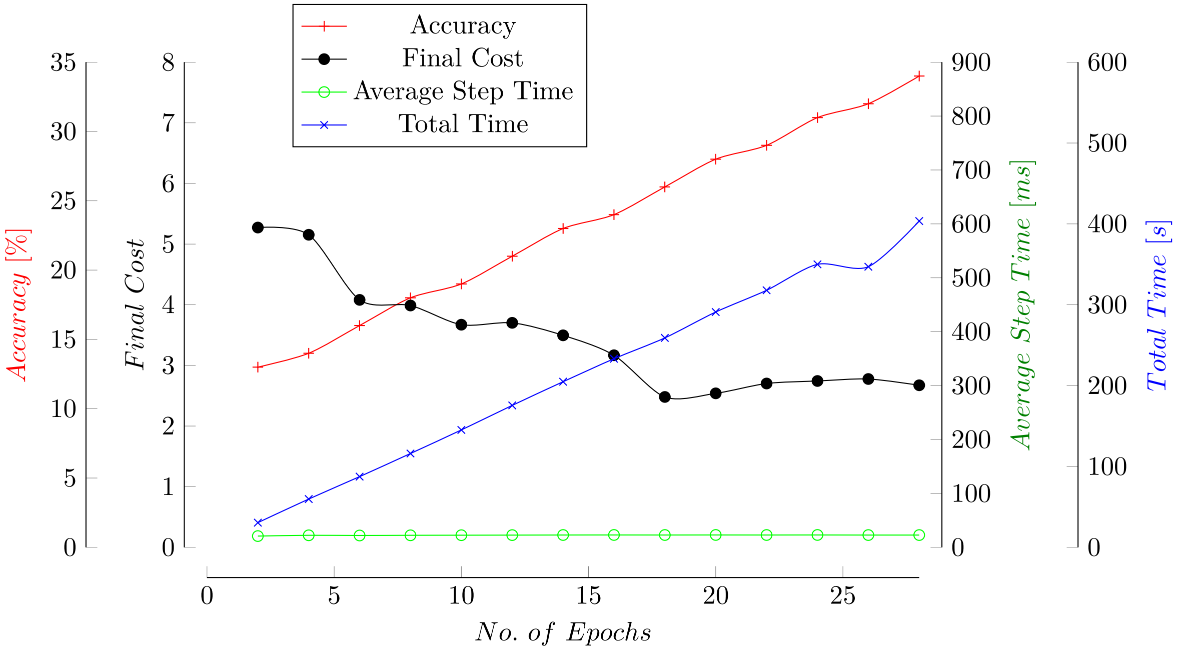


Figure 7.3: Comparison of data from table 7.2.

7.3 Summary of Chapter

First, the final layout of our Beowulf cluster's architecture is presented. Next a performance evaluation of the cluster when running a distributed MNIST program is performed. The cluster did not perform as expected. The run time of the cluster was beaten by a singe RPi running the same software. A discussion on the cause of the bad performance is given. The exact cause is not found but is ruled out to be a problem in the software and not in the hardware. In theory the cluster should roughly have the performance of four Intel based laptops (Intel i5-5250U). Lastly, Amdahl's law is presented to explain the limits of parallel computing.

Chapter 8

Conclusion

In this chapter a general summary of the project is given, from both a hardware and a software perspective. Possible improvements of our system is discussed in future work. We give some concluding remarks on our project and personal reflections.

8.1 General Summary

During 16 weeks we have constructed, configured and executed distributed software on a Raspberry Pi cluster. To construct the cluster 33 RPi's were used. We added a PCB card to every RPi's GPIO to manage the power supply. This PCB card was soldered by hand and the drawings of the card itself was found on Kiepert's online git repository.[15] The PCB cards were interconnected and the power was retrieved from two standard PC PSU's.

The cluster required several different software components. An operating system was installed with the parallel programming frameworks TensorFlow and MPICH. The nodes were assigned with static IP's and the shared filesystem NFS was setup. To be able to access the nodes in the cluster SSH was also configured.

MPI is a communication protocol for parallel computing. To confirm that the installed MPI implementation MPICH worked correctly a simple MPI Hello World program was executed successfully. Next a more complex MPI program was run. This program used the led lamps on the PCB cards to create different light patterns. This MPI program can be seen as a more complete test of MPI and the whole cluster software setup.

The machine learning framework TensorFlow was used in visual recognition of the MNIST data set. The MNIST data set is a collection of handwritten digits. Every image of a handwritten digit has a corresponding label in the data set. The program produced a correct result but executed with poor performance. An analysis was made to find the cause of the poor performance. The precise cause was not found but it could be concluded that it was a software issue and not a hardware issue.

8.2 Future Work

There are several different interesting directions that can be explored in the cluster, both in hardware and in software. The Beowulf architecture is both expandable and flexible.

The PCB card currently has a number of flaws, most importantly a couple of electricity security aspects. The contact surface between the PCB and the PSU was very small and on one occasion it was overheated and burnt. Additionally nothing prevented the user from making this connection with the wrong voltage polarity. To solve these problems a cable with a larger contact and with a shape that restricts the polarity of the connection, could be soldered to the PCB terminals. The PSU would be modified with a suitable connector.

The computer hardware —the Raspberry Pi 3— can be replaced with any computer system with a network interface. In the scope of single board computers there are many different alternatives available. These alternatives may have different advantages in performance, economic aspects and simplicity of electronic setup. A few interesting alternatives are Parallella and ODROID-C2.

The Parallella computer includes an ARM A9 processor and a unique 16-core Epiphany co-processor.[64] The Parallella computer has a number of advantages such as: a very high parallelism is possible with the Epiphany, and the expansion connector makes the power supply easy by connecting them through the expansion connector. This setup would be safer and simpler than our PCB card. It has a couple of disadvantages: more expensive $99 (880 sek), and it is currently more difficult to program.[22][42]

The ODROID-C2 computer has a Cortex-A53 4-core running at 2 GHz, also it has a 2 GB DDR3 memory.[63] The advantages with ODROID-C2: you get faster hardware for only $5 (45 sek) more than a Raspberry Pi 3. A disadvantages is that no one has built a larger cluster with the ODRIOD-C2 as far as we know. Because of this it is uncertain how the power supply would be resolved in a bigger cluster. Both Parallella and ODRIOD-C2 can be seen in figure 8.1.

Figure 8.1: To the left a ODROID-C2 computer and to right a Parallella. [64][63]

The advantage of RPi is that it provides a lot of performance for a low price. An even greater advantage is the large amount of information that is available on the web on how to build a RPi cluster.

The cause of our performance problems with TensorFlow is explored in section 7.2, the precise cause is currently unclear. A more in depth understanding of the TensorFlow programming model is required to analyze the problem. The issue may lie in that the MNIST program we used is programmed incorrectly. Another reason could be a bug in TensorFlow that will be fixed in a later version. Stack Overflow is the recommended forum for questions on TensorFlow. We found a thread where a problem which seemed to be the same as ours.[49] In this thread a bug report was linked which contains a discussion on a bug in gRPC on which TensorFlow depends.[53]

Unfortunately a proper benchmark of the cluster was never run. The initial plan was that the MNIST program would benchmark the cluster, but as explained in chapter 7 it contained an error. Because of lack of time we never had the opportunity to run a benchmark. A well known benchmark is High Performance Linpack (HPL) and would probably be a good fit for our cluster.[23] An advantage for us with HPL is that it runs on MPICH so the additional configuration would be small.

8.3 Concluding Remarks

This project has been so much fun. We have had the opportunity to work with many different areas of computer science, both hardware and software. As the software used in this project is open source we have had access to a large community on the web. Many issues have been solved by consulting online documentation and various online forums.

Bibliography

[1] About gRPC.
http://www.grpc.io/about/.
Accessed: 2017-05-18.

[2] Announcing TensorFlow 0.8 - now with distributed computing support!
https://research.googleblog.com/2016/04/announcing-tensorflow-08-now-with.html.
Accessed: 2017-05-18.

[3] Arch Linux.
https://wiki.archlinux.org/index.php/Arch_Linux#Principles.
Accessed: 2017-02-22.

[4] Arch Linux Arm.
https://archlinuxarm.org/platforms/armv8/broadcom/raspberry-pi-3.
Accessed: 2017-02-22.

[5] Autoconf.
https://www.gnu.org/software/autoconf/manual/autoconf.html.
Accessed: 2017-03-25.

[6] Autoconf.
https://www.gnu.org/software/autoconf/manual/autoconf.html.
Accessed: 2017-03-25.

[7] Bazel.
https://bazel.build/.
Accessed: 2017-02-23.

[8] Beowulf Poem.
https://en.wikipedia.org/wiki/Beowulf.
Accessed: 2017-04-12.

[9] CGI Company Overview.
https://www.cgi.se/company-overview.
Accessed: 2017-05-13.

[10] Computer Vision.
https://en.wikipedia.org/wiki/Computer_vision.
Accessed: 2017-05-05.

[11] Creative Commons Attribution 3.0 Unported.
https://creativecommons.org/licenses/by/3.0/.
Accessed: 2017-05-15.

[12] Data sheet of the switch Cisco SF200-48.
http://www.cisco.com/c/en/us/products/collateral/switches/
small-business-200-series-smart-switches/data_sheet_c78-634369.html.
Accessed: 2017-05-17.

[13] Distributed TensorFlow.
https://www.tensorflow.org/deploy/distributed.
Accessed: 2017-04-19.

[14] Distributed TensorFlow documentation.
https://www.tensorflow.org/deploy/distributed.
Accessed: 2017-05-18.

[15] Electronic schematic and drawings over the PCB card.
https://bitbucket.org/jkiepert/rpicluster/src.
Accessed: 2017-04-27.

[16] Extending TensorFlow with additonal client programming languages.
https://www.tensorflow.org/extend/.
Accessed: 2017-05-15.

[17] Forum discussion on performance issue with distributed TensorFlow.
http://stackoverflow.com/questions/42500739/between-graph-replication-\
version-of-ptb-rnn-model-is-slower-than-single-gpu-ver?rq=1.
Accessed: 2017-05-17.

[18] Fuse(electrical).
https://en.wikipedia.org/wiki/Fuse_(electrical).
Accessed: 2017-03-15.

[19] gRPC Frequently Asked Questions.
http://www.grpc.io/faq/,
Accessed: 2017-05-18.

[20] Hello World MPI.
https://help.ubuntu.com/community/MpichCluster.
Accessed: 2017-05-02.

[21] History of the Support Vector Machine.
http://www.svms.org/history.html.
Accessed: 2017-05-16.

[22] How the @#$% do I program the Parallella?
https://www.parallella.org/2015/05/25/how-the-do-i-program-the-parallella/.
Accessed: 2017-05-18.

[23] HPL (High Performance Linpack): Benchmarking Raspberry Pi's.
https://www.howtoforge.com/tutorial/hpl-high-performance-linpack-benchmark-raspberry-pi/.
Accessed: 2017-05-19.

[24] Iridis-pi: a low-cost, compact demonstration clusters.
http://www.southampton.ac.uk/~sjc/raspberrypi/raspberry_pi_iridis_lego_supercomputer_paper_cox_Jun2013.pdf.
Accessed: 2017-02-27.

[25] Keychain.
https://wiki.archlinux.org/index.php/SSH_keys#Keychain.
Accessed: 2017-04-29.

[26] LED blink progam using MPI.
https://bitbucket.org/jkiepert/rpicluster/src.
Accessed: 2017-04-27.

[27] Libtool.
https://www.gnu.org/software/libtool/.
Accessed: 2017-03-25.

[28] Machine Learning.
https://en.wikipedia.org/wiki/Machine_learning.
Accessed: 2017-05-18.

[29] MNIST For ML Beginners.
https://www.tensorflow.org/get_started/mnist/beginners.
Accessed: 2017-04-19.

[30] MNIST program.
https://github.com/ischlag/distributed-tensorflow-example.
Accessed: 2017-05-07.

[31] Most commonly used MPI functions.
http://www.mathcs.emory.edu/~cheung/Courses/355/Syllabus/92-MPI/intro.html.
Accessed: 2017-05-02.

[32] MPI Communicator Groups.
http://www.mathcs.emory.edu/~cheung/Courses/355/Syllabus/92-MPI/intro.html.
Accessed: 2017-05-02.

[33] Network File System.
http://searchenterprisedesktop.techtarget.com/definition/Network-File-System.
Accessed: 2017-04-03.

[34] Overview of Arch Linux describing what to expect from an Arch Linux system.
`https://wiki.archlinux.org/index.php/Arch_Linux`.
Accessed: 2017-05-15.

[35] Parameter Server.
`http://wiki.ros.org/Parameter%20Server`.
Accessed: 2017-05-08.

[36] Printed Circuit Board.
`https://en.wikipedia.org/wiki/Printed_circuit_board`.
Accessed: 2017-02-13.

[37] Protocol Buffers.
`https://developers.google.com/protocol-buffers/docs/overview`.
Accessed: 2017-02-28.

[38] Raspberry Pi 2 Model B.
`https://www.raspberrypi.org/products/raspberry-pi-2-model-b/`.
Accessed: 2017-05-01.

[39] Raspberry Pi 3 Model B.
`https://www.raspberrypi.org/products/raspberry-pi-3-model-b/`.
Accessed: 2017-05-01.

[40] Raspberry Pi hardware specifications.
`https://www.htpcguides.com/raspberry-pi-vs-pi-2-vs-banana-pi-pro-benchmarks/`.
Accessed: 2017-05-13.

[41] Raspberry Pi vs Pi 2 vs Banana Pi Pro Benchmarks.
`https://www.htpcguides.com/raspberry-pi-vs-pi-2-vs-banana-pi-pro-benchmarks/`.
Accsessed: 2017-05-13.

[42] Recommended Parallella Programming Frameworks.
`https://www.parallella.org/programming/`.
Accessed: 2017-05-18.

[43] Replication.
`https://www.tensorflow.org/deploy/distributed#replicated_training`.
Accessed: 2017-05-13.

[44] Replication Pictures.
`https://www.slideshare.net/cfregly/advanced-spark-and-tensorflow-meetup-may-26-2016`.
Accessed: 2017-05-13.

[45] Review of the history of Artificial Intelligence.
`http://watson.latech.edu/book/intelligence/intelligenceOverview4.html`.
Accessed: 2017-05-15.

[46] Simulating the bahaviour of a partial differential equation with TensorFlow.
`https://www.tensorflow.org/extend/architecture`.
Accessed: 2017-05-15.

[47] Softmax Regression.
https://www.tensorflow.org/get_started/mnist/beginners#softmax_regressions.
Accessed: 2017-05-08.

[48] SSH Keys.
https://wiki.archlinux.org/index.php/SSH_keys.
Accessed: 2017-04-07.

[49] Stack Overflow's discuss TensorFlow bug.
http://stackoverflow.com/questions/42500739/between-graph-replication-\
version-of-ptb-rnn-model-is-slower-than-single-gpu-ver?rq=1.
Accessed: 2017-05-17.

[50] Stack Overflow's version of the MNIST program.
https://stackoverflow.com/questions/37712509/how-to-run-tensorflow-distributed-mnist-example.
Accessed: 2017-05-17.

[51] TensorBoard: Visualizing Learning.
https://www.tensorflow.org/get_started/summaries_and_tensorboard.
Accessed: 2017-05-05.

[52] TensorFlow - Google's latest machine learning system, open sourced for everyone.
https://research.googleblog.com/2015/11/tensorflow-googles-latest-machine_9.html.
Accessed: 2017-05-18.

[53] TensorFlow Bug.
https://github.com/tensorflow/tensorflow/issues/6116.
Accessed: 2017-05-08.

[54] TensorFlow forum on Stack Overflow for technical questions.
http://stackoverflow.com/questions/tagged/tensorflow.
Accessed: 2017-05-17.

[55] TensorFlow images.
https://www.tensorflow.org/get_started/mnist/beginners#top_of_page.
Accessed: 2017-05-12.

[56] TensorFlow's distributed version of the MNIST program.
https://github.com/tensorflow/tensorflow.
Accessed: 2017-05-17.

[57] TensorFlow's MNIST Program.
https://www.tensorflow.org/deploy/distributed.
Accessed: 2017-05-17.

[58] TensorFlow's official webpage.
https://www.tensorflow.org/.
Accessed: 2017-02-23.

[59] The Arch User Repository explained.
https://wiki.archlinux.org/index.php/Arch_User_Repository.
Accessed: 2017-05-15.

[60] The Beast Cluster.
https://resin.io/blog/what-would-you-do-with-a-120-raspberry-pi-cluster/.
Accessed: 2017-02-23.

[61] The class SummaryWriter.
https://www.tensorflow.org/versions/r0.11/api_docs/python/train/adding_summaries_to_event_files#SummaryWriter.
Accessed: 2017-05-05.

[62] The MNIST database.
http://yann.lecun.com/exdb/mnist/.
Accessed: 2017-04-19.

[63] The ODROID-C2 Computer.
http://www.phoronix.com/scan.php?page=article&item=raspberry-pi3-odroid2&num=1.
Accessed: 2017-05-13.

[64] The Parallella Computer.
https://www.parallella.org/.
Accessed: 2017-05-13.

[65] The TensorFlow Architecture.
https://www.tensorflow.org/extend/architecture.
Accessed: 2017-05-15.

[66] Training loop MNIST.
https://www.tensorflow.org/get_started/mnist/beginners#training.
Accessed: 2017-05-10.

[67] Vector space.
https://www.tensorflow.org/get_started/mnist/beginners#top_of_page.
Accessed: 2017-05-09.

[68] Visualizing the Mandelbrot set with TensorFlow.
https://www.tensorflow.org/extend/architecture.
Accessed: 2017-05-15.

[69] Website of the ALARM operating system.
https://archlinuxarm.org/.
Accessed: 2017-05-15.

[70] Weight.
https://www.tensorflow.org/get_started/mnist/beginners#softmax_regressions.
Accessed: 2017-05-09.

[71] Welcome to the TensorFlow Community.
`https://www.tensorflow.org/community/welcome`.
Accessed: 2017-05-17.

[72] What is a key fingerprint?
`https://help.gnome.org/users/seahorse/stable/misc-key-fingerprint.html.en`.
Accessed: 2017-04-29.

[73] Worker.
`https://www.tensorflow.org/deploy/distributed#glossary`.
Accessed: 2017-05-09.

[74] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[75] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI). Savannah, Georgia, USA*, 2016.

[76] George Karypis Ananth Grama, Anshul Gupta and Vipin Kumar. *Introduction to Parallel Computing*. Pearson Education Limited, 2003.

[77] Dana Angluin. Computational Learning Theory: Survey and Selected Bibliography. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Theory of Computing*, STOC '92, pages 351–369, New York, NY, USA, 1992. ACM.

[78] Robert G Brown. Engineering a beowulf-style compute cluster. *Duke University Physics Department*, 2004.

[79] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: a flexible data processing tool. *Communications of the ACM*, 53(1):72–77, 2010.

[80] Peter Goldsborough. A Tour of TensorFlow. *arXiv Computing Research Repository*, abs/1610.01178, 2016.

[81] William Gropp, Anthony Skjellum, and Ewing Lusk. *Using MPI : Portable Parallel Programming with the Message-Passing Interface.*, volume Third edition of *Scientific and Engineering Computation*. The MIT Press, 2014.

[82] Wesley M Johnston, JR Hanna, and Richard J Millar. Advances in dataflow programming languages. *ACM Computing Surveys (CSUR)*, 36(1):1–34, 2004.

[83] Joshua Kiepert. Creating a raspberry pi-based beowulf cluster. *Boise State*, 2013.

[84] Andres Munoz. Machine Learning and Optimization. *URL: https://www. cims. nyu. edu/˜ munoz/files/ml_optimization. pdf [accessed 2016-03-02][WebCite Cache ID 6fiLfZvnG]*, 2014.

[85] Roger Parloff. THE DEEP-LEARNING REVOLUTION. *Fortune*, 174(5):96 – 106, 2016.

[86] Stuart J Russell and Peter Norvig. Artificial Intelligence: A Modern Approach. 1995.

[87] Thomas Lawrence Sterling. *How to Build a Beowulf : A Guide to the Implementation and Application of PC Clusters.* Scientific and Engineering Computation Series. The MIT Press, 1999.

[88] Thomas Lawrence Sterling. *Beowulf Cluster Computing with Linux.* Scientific and Engineering Computation Series. The MIT Press, 2002.