Building a Parallella board cluster

Submitted in partial fulfilment
of the requirements of the degree of

Bachelor of Science (Honours)

of Rhodes University

Michael Johan Kruger

Grahamstown, South Africa
November 23, 2015

Abstract

Currently, high performance computing is largely restricted to well-funded research groups. This project aimed to create a high performance cluster using the cheap and energy-efficient 18-core Parallella boards. Four boards were connected over a network as a cluster and basic test programs were run using MPI. Experimental results show that the Epiphany chip performs very well compared with other energy-efficient chips such as the Cortex A9 ARM with a 11× speedup. Similar performance is achieved by the cluster of four Parallella boards against an Intel i5 3570 running a single thread. The Epiphany however, sees a drop in speed when attempting complex arithmetic operations compared with the other processors owing to the lack of hardware support. It is possible to achieve high performance using low-powered Parallella boards as long as the user is aware of the Epiphany chip's weaknesses and avoids these.

ACM Computing Classification System Classification

•**Computer systems organization** → **Multicore architectures; Heterogeneous (hybrid) systems;** *Interconnection architectures;* •**Networks** → *Network on chip;* •**Hardware** → Impact on the environment;

Acknowledgements

I would like to acknowledge the financial and technical support of Telkom SA, Tellabs, Easttel, Bright Ideas 39, THRIP and NRF SA (UID 75107) through the Telkom Centre of Excellence in the Department of Computer Science at Rhodes University.
I would also like thank for all the support and restraint shown by my supervisor Dr Karen Bradshaw for not paring my skin followed by hanging in a gibbet, that she would have located above her office door or outside her window as a warning to the next honours student whose thesis is not prompt in execution.
Thank you to Anthony J Sullivan for the delicate soldering done on the Parallella boards.

CONTENTS

1 Introduction **1**

1.1 Problem Statement 2

1.2 Objectives 2

1.3 Approach 2

1.4 Thesis Organisation 3

2 Literature Review **4**

2.1 High Performance Computing 4

2.1.1 Concepts/Terminology 5

2.1.2 Clusters Architectures 6

2.1.3 Existing HPC Architectures 6

2.1.4 Benchmarks for HPC 8

2.2 Software For Clusters 10

2.2.1 MPICH 10

2.2.2 Open-MPI 10

2.2.3 OpenMP 10

2.2.4 Rocks Cluster Distribution 11

2.3 Parallella 11

2.3.1 Specifications 12

2.3.2 Existing Parallella Configurations 15

2.4 Summary 15

3 Configuring the Cluster **16**

3.1 Physical Layout of Cluster 16

3.1.1 High Level Overview 16

3.1.2 Components 18

3.2 Cluster Design and Interface 18

3.2.1 Powering the Cluster 19

3.2.2 Communication Between Nodes 21

3.3 Software 21

3.3.1 Linux 22

3.3.2 SSH 23

3.3.3 Open-MPI 24

3.3.4 Network File Share 25

3.4 Problems encountered with Cluster Configuration 27

3.4.1 DHCP issue 27

3.4.2 File Sharing NFS 27

3.5 Summary 27

4 Benchmarking the Cluster **29**

4.1 Creating Programs for Parallella Cluster 29

4.2 Benchmarks 34

4.2.1 Floating Point Multiplication 34

4.3 Ethernet Bandwidth Benchmark 36

4.4 Limitations 37

4.4.1 Complex Arithmetic 38

4.4.2 Hardware Optimisations 38

4.5 Summary 39

5 Conclusion **40**

5.1 Summary 40

5.2 Future Work 41

References **43**

LIST OF FIGURES

2.1 Photo of the Iridis-pi at the University of Southampton [11] 7
2.2 Photo of the Tianhe-2 8
2.3 Architectural layout of the Parallella board (taken from [2]) 12
2.4 Epiphany mesh architecture [6] 14

3.1 Basic overview of the cluster 17
3.2 Soldered bridge allowing power from corner pads 19
3.3 Stacked cluster using Nickel plated spacers 20
3.4 Modified Molex connector 20

4.1 Starting a program using MPI for execution on Epiphany cores 30
4.2 Execution on the cluster without using one of the Parallella boards to set up the MPI environment 30
4.3 Time taken per machine for 100,000,000 iterations 35

4.4 Speed increase relative to a single ARM core 36

4.5 Average point to point bandwidth using MPI 37

4.6 Average point to point bandwidth using MPI with multiple processes per board 38

LISTINGS

3.1 Contents of /etc/hosts 23
3.2 Contents of ~/NFS_share/hello_world/machinefile 24
3.3 Contents of /etc/exports 26
4.1 Typical Epiphany host program 31
4.2 Typical Epiphany core program 32
4.3 Basic MPI setup 33

CHAPTER 1

INTRODUCTION

As processor clock speeds get faster they reach a soft cap, from either excessive power consumption or heat output, which makes it infeasible to increase the speed any higher. To get around this problem chips have started to incorporate multiple processors that can run slower but achieve the same or better performance. Using multiple individual computers over a network, which is known as cluster computing, it is possible for them to work together as one system to solve problems [9]. This has made access to a "supercomputer" simpler for the ordinary user. Obviously, there is still the need for specialised supercomputers, but this is largely restricted to well-funded research groups.

Cluster computing has become more and more popular as it is a low-cost way of creating high performance computers. Using ordinary personal computers (PCs) is one way of setting up a cluster, but the advent of even lower cost "computing" devices, has provided more opportunities than ever before of creating such clusters.

The Parallella is one such device that provides 18 cores per board at a comparatively low cost.

1.1 Problem Statement

The release of the Parallella board provides a cheap and energy efficient computer that requires minimal configuration.

In this research, we investigate whether it is possible to build a high performance computer using a cluster of Parallella boards, thereby providing a very low-cost alternative to an ordinary PC.

Our research question is as follows: What speedup can be obtained executing typical parallel programs on a Parallella cluster, compared with a normal PC? We hypothesise that this is possible, provided that the cluster overhead and limitations of the hardware/-software are not too great.

1.2 Objectives

In order to investigate the research question, the proposed objectives of this project are as follows:

- Build a high performance computer using multiple Parallella boards connected over a network.
- Install applicable operating systems and software to set up the cluster.
- Compare performance of the Parallella cluster with other similarly priced systems and/or a desktop PC.
- Discover the limitations of the Parallella cluster.

1.3 Approach

To achieve the research objectives, the first step is the physical building of the Parallella cluster. This involves setting up cooling on all the Parallella boards using a fan, providing power to all the boards via the mounting hole pads, setting up a router or switch to facilitate communication between the boards, and mounting and connecting all the boards to power and the switch.

After setting up the components, an operating system and software need to be installed on each board or alternatively, network booting needs to be set up on each. The software must be configured so that the boards are aware of each other and work together.

Once the Parallella cluster is operational, comparisons against other systems can be made using software to benchmark each system's performance. Once comparisons and benchmarks are taken ways in which to optimise the Parallella cluster can be explored and tested; this would involve changing the cluster configuration and any other optimisations discovered after further research. Results and any limitations of the Parallella boards will be recorded.

1.4 Thesis Organisation

The remainder of this thesis is organised as follows:

- Chapter 2 introduces the topic of high performance computing, discusses some of the attempts to produce both low-cost and high-cost supercomputers, and goes over some of the benchmarks used in high performance computing.
- Chapter 3 explains how the Parallella cluster was set up and configured to allow MPI and Epiphany programs to be created and run.
- Chapter 4 discusses how programs are created for the Epiphany co-processor and presents the results of the benchmarks.
- Chapter 5 summarises the research, giving the conclusions, and suggests possible extensions to the research.

CHAPTER

2

LITERATURE REVIEW

The aim of this project is to create and benchmark a cluster of Parallella boards. In preparation for this, information relevant to the project is reviewed. The topics discussed include high performance computing (HPC), HPC benchmarks, and the Parallella boards themselves.

2.1 High Performance Computing

HPC is the term for very fast systems aimed at processing large volumes of information quickly. High performance computers are made up of multiple processors as the speed of a single processor has reached its limits due to physics [16]. HPC is most cost effectively obtained using cluster computing, as most places needing large amounts of processing power have multiple computers readily available [9]. HPC is used to run high cost simulations that would be too expensive or difficult to do physically; these require large amounts of computational power to be completed in a timely manner [30; 28], and therefore, more powerful machines are continuously being built. HPC has evolved over time with the number of cores in a single computer approaching the millions and performance reaching multiple petaFLOPS (10^{15} floating-point operations per second) [4].

Owing to current size and speed constraints on single core processors, it has been found that running multiple slower cores is both more efficient and faster. Algorithms need to take into account ways in which to split the workload evenly between multiple processors if they want to obtain faster execution speeds using this type of architecture [16]. Many compilers have special flags so that they can optimise programs for parallel computation [24]; this, however, only achieves a minor boost in speed when compared with an efficient algorithm that splits the work into multiple pieces that can be distributed among multiple processors. According to David Geer [16], to take advantage of multiple cores, programs need to be rewritten so that they can run on multiple threads, with each thread assigned to a separate processor.

2.1.1 Concepts/Terminology

Throughput The rate at which data can be successfully transferred over a channel.

Shared Memory Memory, over which multiple processes have control and which is shared between them.

Distributed memory This is the term used in a multi-core system when a processor has its own private memory that it can access and use; however, when it needs information from another process, it has to communicate with the other process and request the particular data.

Bottleneck A bottleneck occurs when the effectiveness of a system is restricted by a single or small number of resources.

Latency This refers to the amount of time required for an instruction to travel from its source to its location and be acted upon. A large amount of latency is detrimental as the time to pass information around a system will become a bottleneck and the system will not be able to make use of all its computational power.

FLOPS Floating Point Operations Per Second is the usual measurement of a high performance computer. It refers to the number of instructions using floats that a system can compute per second. It is usually referred to using a prefix such as giga for 10^9 or peta for 10^{15}.

2.1.2 Clusters Architectures

With the demand for large amounts of processing power, various ways of creating supercomputers cheaply have appeared. Clusters of computers connected on a network can be purposed to work together as a supercomputer. With the increased speed and decreased latency of the Internet, it is possible to create a cluster using computers from all over the world; this has led to programs and applications that allow a computer to connect to a pool of other computers and add its processing power to the computation. There are, however, some factors limiting the effectiveness of cluster computing. These include building a switch to keep up with the speed of a single core processor and creating compilers that make good use of multiple processors. There are two generally used methods for controlling communication within a cluster:

MPI The individual nodes of the cluster can communicate with each other using a message passing interface (MPI), which provides a thread safe application programming interface (API) that allows the work to be effectively delegated to multiple nodes on the network [27; 19] and information passed between each node so that it can be worked on. More information on MPI is provided in Section 2.2.1 with the overview of MPICH.

Parallel Virtual Machine uses a parallel virtual machine (PVM) approach, which combines all the nodes and allows them to appear as a single PVM. This PVM handles all the message passing, task scheduling, and data conversions. To set this up, each node of the cluster needs the same PVM image installed and must be marked as a PVM node. Parallel virtual machines are popular due to the ease with which the cluster can be managed[17]. Some of the available PVMs are reviewed in Section 2.2.

2.1.3 Existing HPC Architectures

This subsection gives an overview of some existing high performance computing systems.

Iridis-pi The Iridis-pi[11] is a cluster constructed from 64 Raspberry Pi model B nodes and held together with Lego. The Iridis-Pi was created by a team at the University of Southampton. Each Raspberry Pi has a 700 MHz ARM processor, 256 MB of RAM, a 16GB SD card, and a 100Mb/s network interface. The benefits of the cluster are its cheap price, compact size, and low power consumption. Using the HPL benchmark

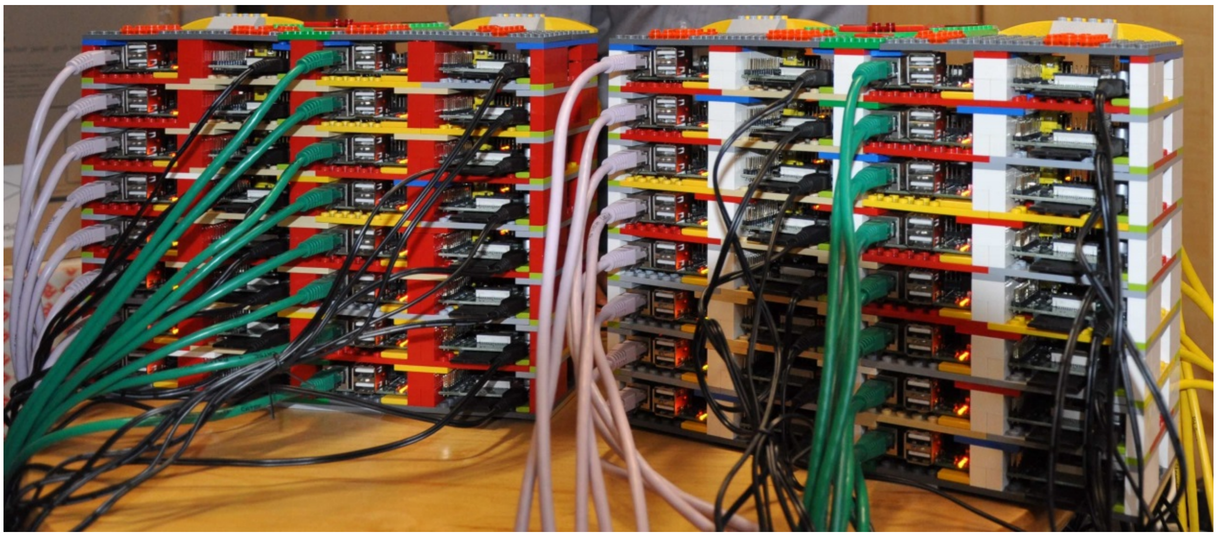

Figure 2.1: Photo of the Iridis-pi at the University of Southampton [11]

and all 64 nodes the Iridis-pi achieves a throughput of 1.14 gigaFLOPS[11]. The price of a Raspberry Pi model B at time of writing is 450 South African Rand[1] bringing the cost of the 64 Iridis-pi nodes to 28800 South African Rand. This price excludes the cost of the switch, Lego and cableing.

Tianhe-2 (Milkyway-2) The most powerful supercomputer according to the Top500 list of the world's supercomputers in November 2015 is the Tianhe-2 (Milkyway-2)[4]. On the LINPACK benchmark, it achieved a performance of 33,862.7 teraFLOPS with a theoretical peak of 54,902.4 teraFLOPS[4]. The Tianhe-2 is sixth according to the graph500 November 2015 results. The graph500 uses different benchmarks that are focused on data intensive supercomputer applications. The Tianhe-2 performed at 2061.48 GTEPS (10^9 traversed edges per second)[2]. Developed by China's National University of Defence Technology (NUDT) in collaboration with the Chinese IT firm Inspur, the Tianhe has managed to maintain the top spot since June 2013. The Tianhe is composed of 16,000 computer nodes, each consisting of two Intel Xeon IvyBridge processors and three Xeon Phi processors with access to 88 GB of RAM; this equates to 3.12 million cores and 1.404 petabytes of RAM[3]. Running NUDT's own operating system Kylin Linux, which is optimised for high-performance parallel computing as well as having support for power management and high-performance virtual computing zone, the Tianhe-2 uses MPICH2 with a customised GLEX channel[4] for executing HPC programs. A picture of a

[1]`http://pifactory.dedicated.co.za/product/raspberry-pi-1-model-b/`

[2]`http://www.graph500.org/`

[3]http://www.extremetech.com/computing/159465-chinas-tianhe-2-supercomputer-twice-as-fast-as-does-titan-shocks-the-world-by-arriving-two-years-early

Figure 2.2: Photo of the Tianhe-2

portion of the supercomputer can be seen in Figure 4.3[4]

2.1.4 Benchmarks for HPC

The HPC Challenge Benchmark is a set of seven benchmarks for testing an HPC system's ability to cope with different scenarios, thereby giving an indication of real-world performance[21]. These benchmarks include:

HPL "A Portable Implementation of the High-Performance LINPACK Benchmark for Distributed-Memory Computers" The computers in the Top500 list are ranked by the HPL NxN benchmark results[4]. The reason for choosing LINPACK as a benchmark is that it is widely used, and the benchmarked performance of a large number of systems is available. LINPACK is a collection of Fortran subroutines for solving systems of linear equations. Owing to the distributed nature of both the memory and computing nodes of HPC, the highly-parallel LINPACK (HPL) benchmark was

[4] Picture taken from : `http://chinadaily.com.cn/business/tech/img/attachement/jpg/site1/20140626/eca86bd9e2eb1516011b02.jpg`

created to compare results on these systems. In the Top500 list, the results taken into account are:

- R_{max} – the maximum performance achieved by LINPACK
- N_{max} – how large the problem was to get the result in R_{max}
- $N_{1/2}$ – the size of the problem to get half of R_{max}
- R_{peak} – theoretical peak performance.

all of which are taken from the HPL benchmark [13; 14; 5; 20; 4].

DGEMM is a method that calculates the product of double precision matrices and measuring the rate of execution, provides insight into the performance of the HPC device.

STREAM provides a measurement of the “sustainable memory bandwidth and the corresponding computation rate for simple vector kernels”[22].

PTRANS (parallel matrix transpose) forces pairs of processors to communicate simultaneously, testing the communication capacity of the network[5].

Random Access By providing a measurement in GUPS (giga updates per second), this benchmark measures random memory access[6].

FFT (Fast Fourier Transform) This benchmark measures “the floating point rate of execution of double precision complex one-dimensional Discrete Fourier Transform (DFT)”[29; 21].

Communication bandwidth and latency Using the effective bandwidth benchmark[7], latency and bandwidth are measured for a number of simultaneous communication patterns[21].

All these benchmarks measure different aspects that are useful in the real world and provide an idea of what should be tested.

[5] `http://www.netlib.org/parkbench/html/matrix-kernels.html`
[6] http://icl.cs.utk.edu/projectsfiles/hpcc/RandomAccess/
[7] `https://fs.hlrs.de/projects/par/mpi//b_eff/`

2.2 Software For Clusters

Since we are deploying a cluster, in this section we only discuss software relevant to cluster computers.

2.2.1 MPICH

MPICH is a high performance, portable and widely used implementation of the Message Passing Interface (MPI) standard. MPICH was created for distributed memory systems, with the idea of portability and high performance in mind. Excellent results have been achieved with MPICH, which is the most used implementation of MPI in the world, and its derivatives. MPICH is able to work in many different environments and take advantage of what is available to increase performance while maintaining portability, for example, using shared memory to pass messages between processors faster. MPICH is distributed as source code and has an open-source freely available licence [8; 19; 18].

2.2.2 Open-MPI

Open-MPI is an open-source implementation of the Message Passing Interface [1]. It has multiple partners from across the HPC community, maintaining its libraries[15]. These partners include ARM, which provided the Zynq-chip for use on the Parallella board [1]. Open-MPI conforms fully to MPI-3.1 standards, supports multiple operating systems[15], and is provided by default on the Parallella board Ubuntu distribution.

2.2.3 OpenMP

OpenMP is “an industry standard API for shared-memory programming” [12]. A shared-memory parallel system describes a multi-processor system where individual processors share one memory location [10]. Each processor can still have its own personal cache memory to work with as the speed difference between main memory and processor memory would cripple the speed if the processor needed to pick up everything from the shared memory space. OpenMP was introduced to fix the inability of compilers to make good decisions on how to split up a program to take advantage of multiple processors; although this is possible for simpler programs, a user would need to cast a more discerning eye for more complex problems [10]. OpenMP provides an alternative to message passing in parallel programming. OpenMP is a set of routines and compiler directives to manage shared-memory parallelism. The OpenMP standard is made up of four parts, namely, control structure, data environment, synchronisation, and runtime library [12], which can be added to a sequential program written in C, C++ or Fortran [10].

2.2.4 Rocks Cluster Distribution

Rocks is a Linux distribution, with its latest version Sidewinder based on Cent-OS 6.6, but only available for 64-bit machines [8]. It is unlikely that Rocks will be used in the construction on the Parallella cluster as there does not seem to be an official release for the ARM processor, which is used by the Parallella. Rocks is an open-source distribution that was built to provide a simple environment in which to build computational clusters, grid endpoints and visualisation tiled-display walls.

2.3 Parallella

The Parallella board is an “affordable, energy efficient, high performance, credit card sized computer”[2] that aims to provide a platform for developing and implementing high performance parallel processing. The 66-core version (64-Epiphany cores and two ARM cores) of the Parallella board achieves over 90 gigaFLOPS (10^9 floating point operations per second), while the 18-core (16-Epiphany and 2 ARM cores) version can reach 32 gigaFLOPS using only about 5 Watts. The Parallella has a 1-Gbps Ethernet port allowing a large amount of information to be passed quickly over the network. This increases its ability to work in a cluster as it can pass information to its peers rapidly, provided that the switch is capable of handling the 1Gbps bandwidth.

The aim of creating the Parallella board was to make parallel computing more accessible by creating an affordable, open-source, and open-access platform.

The price of a Parallella board starts at $99 (at the time of writing) for the 16-core board and uses a customised ARM implementation of Linux (Ubuntu 14.04). The Parallella is three years old and software that takes advantage of this is still being developed[9][25].

[8]http://www.rocksclusters.org/

[9]`https://www.kickstarter.com/projects/adapteva/parallella-a-supercomputer-for-everyone`

Programming for the Epiphany chip (the Parallella boards co-processor) is done in C and the Parallella team have provided some basic primitives with the SDK (Software Development Kit). Memory addressing, barriers, and communication between eCores are a few examples of what is provided by the SDK.
To run programs on the Epiphany chip, a workgroup of cores needs to be set up. This can be done using the provided SDK to give a starting node and the number of columns and rows in the matrix of cores [6; 23; 31; 27].

2.3.1 Specifications

As the proposed cluster will make use of the P1601 (Parallella board with a 16-core Epiphany chip co-processor) instead of the 64-core version, the technical specifications of this board are given below [25].

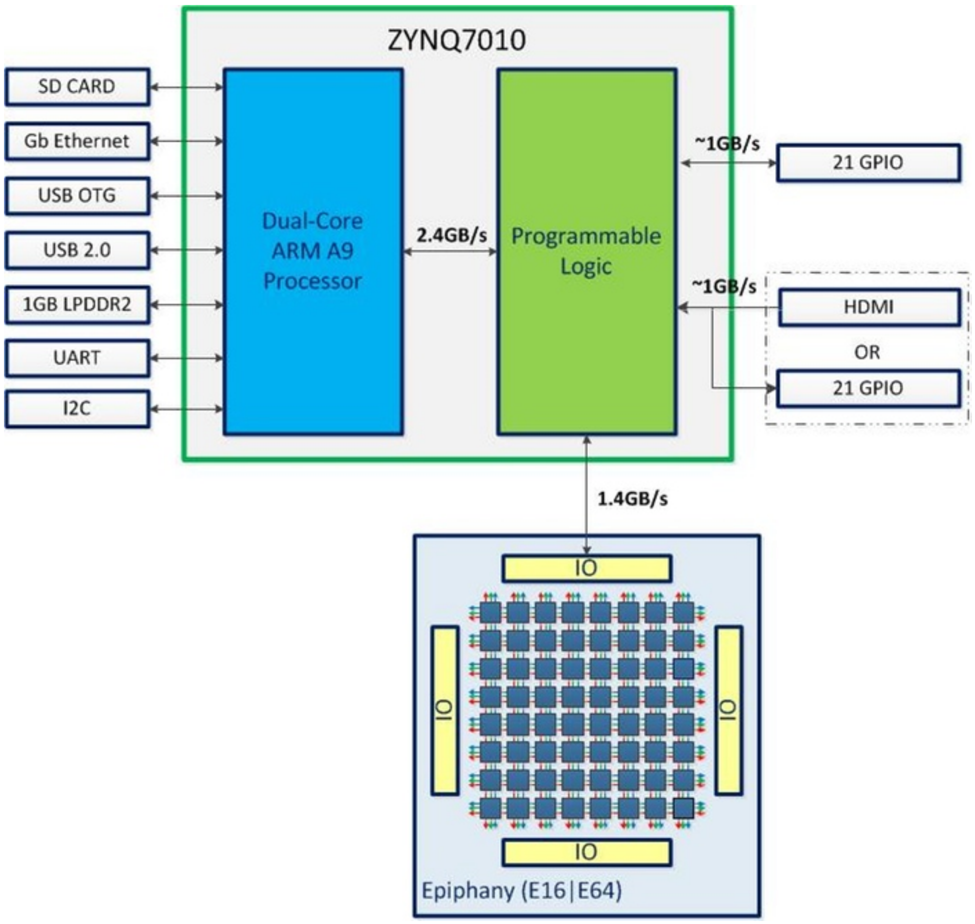

Figure 2.3: Architectural layout of the Parallella board (taken from [2])

Figure 2.3 shows that the interfaces and operating system are run with the dual-core ARM processors; the programs running on the ARM processors can use the Epiphany libraries provided with the SDK to set up and run programs on the individual cores [25; 26; 6; 31].

The Parallella makes use of a Zynq-Z7010 dual-core ARM A9 CPU to run the operating system and programs not designed to run on the Epiphany chip. A rundown of the Parallella components is given below[25]:

- Zynq-Z7010 Dual-core ARM A9 CPU
- 16-core Epiphany Co-processor
- 1GB DDR3 RAM
- MicroSD Card: Allows storage of local files.
- USB 2.0
- Up to 48 GPIO signal
- Gigabit Ethernet: The high-speed Gigabit Ethernet allows for rapid transfer of data across a network allowing the cluster to communicate with lower latency.
- HDMI port
- Linux Operating System: The Linux operating system is well supported by multiple MPI libraries [8; 19; 18; 15; 1].
- 54mm x 87mm form factor: The small form factor of each of the boards makes it highly portable even if using multiple boards.

Figure 2.4 illustrates the 2D array of cores, which Adapteva calls eCores. Each eCore has a 1GHz RISC CPU, 32 KB of local memory, a network interface, and a direct memory access (DMA) engine. This matrix is connected to the rest of the chip via a router. The router communicates with the rest of the chip via three connections: the blue connector is the on-chip write network, green is the off-chip write network, and red is the read request network. The eCore CPU is super-scalar and can execute two floating-point operations and a 64-bit memory load/store operation in every clock cycle. The local memory can provide up to 32 bytes per clock cycle of bandwidth [25; 26; 6; 31].

The Epiphany processor on this board is the Epiphany III (E16G301), the feature summary of which is given below [6]:

- 16 high performance RISC CPU cores:

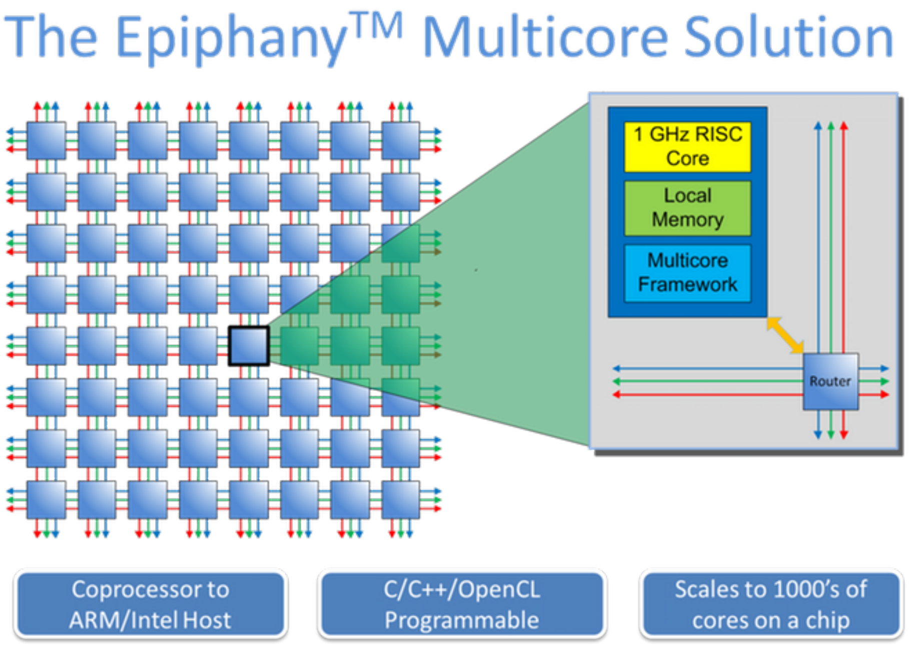
The Epiphany™ Multicore Solution
1 GHz RISC Core
Local Memory
Multicore Framework
Router
Coprocessor to ARM/Intel Host
C/C++/OpenCL Programmable
Scales to 1000's of cores on a chip

Figure 2.4: Epiphany mesh architecture [6]

- C/C++ and OpenCL programmable
- 32-bit IEEE floating point support
- 512KB on-chip distributed shared memory: This can be used to pass messages and share data across the chip.
- 32 independent DMA channels
- Up to 1GHz operating frequency
- 32 gigaFLOPS peak performance
- 512 GB/s local memory bandwidth: The access speed of each RISC core's local RAM.
- 64 GB/s Network-On-Chip bisection bandwidth: The speed at which message passing takes place on the chip.
- 8 GB/s off-chip bandwidth: The performance of communicating off chip.
- 1.5ns network per-hop latency
- <2 Watt maximum chip power consumption: The power consumption of the Epiphany processor; combined with the rest of the board, the total is 5 W.

2.3.2 Existing Parallella Configurations

Below we discuss a few of the HPC configurations that have been implemented using Parallella boards.

Parallac The Parallac is a cluster consisting of eight Parallella boards for the main computation and two intel NUCs (next unit of computing) that are each equipped with an Intel i3 processor, 16 GB of RAM and 120 GB of SSD storage. The design of the configuration was inspired by the Cray 1 supercomputer. The Parallellas are powered using their mounting holes connected to a power supply unit (PSU) via a copper plate. The configuration is compact and cables are well managed[10]. Unfortunately, the Parallac website does not show any results on how well this system performs on any benchmarks.

supercomputer.io Supercomputer.io is a project to create a community-hosted supercomputer created by connecting the Parallellas of anyone who signs up and sets up their Parallella with the required software and hardware. These Parallellas are connected to each other and the supercomputer.io network over the Internet. Scientists can then submit requests to use the cluster for their research[3]. The supercomputer.io project intends supporting other boards in time but at the time of writing it only provides support for Parallella boards. To connect a Parallella board to the supercomputer.io network, the Parallella must have at least a 4GB SD card and a connection to the Internet. The supercomputer.io operating system is copied onto the SD card, and using an active Internet connection, the Parallella connects to the supercomputer.io network and waits for work to be allocated to it [3].

2.4 Summary

The three main topics researched in this chapter were HPC, software options for clusters and an overview of the Parallella board. Aspects of HPC that we discussed included how the performance of an HPC system is measured and the different types of high performance configurations. The cluster software looked at included message passing implementations, shared memory and a cluster suite Rocks. Finally, the Parallella board and information on existing Parallella cluster implementations were investigated. Of the cluster configurations found, no benchmark results were obtained.

[10]`http://www.parallac.org/`

CHAPTER

3

CONFIGURING THE CLUSTER

This chapter discusses the physical construction of the cluster (which we refer to as the Parallella stack), including powering, networking, and cooling. On the software side, we look at configuring Ubuntu 14.04, NFS, SSH, Open-MPI and the network layout as needed for the Parallella stack to work as a cluster. At finally, we mention some of the problems encountered.

3.1 Physical Layout of Cluster

In this section we cover how the physical components of the Parallella stack were put together to create the completed project.

3.1.1 High Level Overview

Here we provide a summary of the entities required to manage and enable the Parallella stack. The following figure shows an overview of how each entity is connected to the cluster:

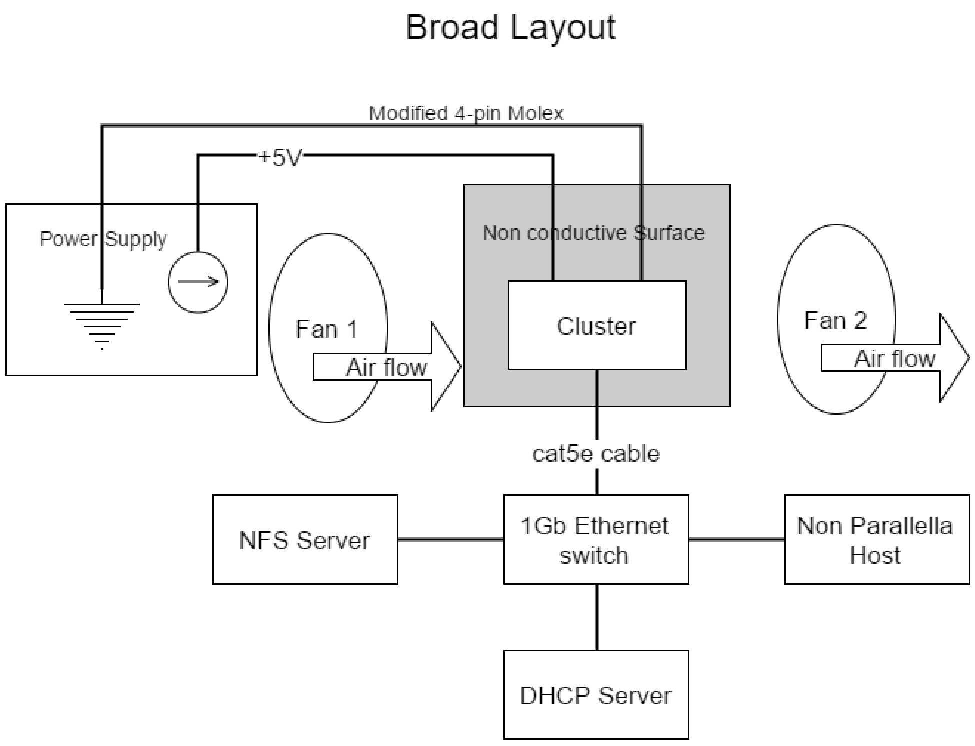

Figure 3.1: Basic overview of the cluster

The four parallella boards are connected to a gigabit switch with each having the hostname “parallella” followed by a unique number, for example, “parallella1”. Parallella1 was chosen to launch programs to the cluster and can be referred to as the head node. The parallella boards are stacked on top of each other using spacers to separate them and two fans on either side. The fans are arranged to force air through the stack, with one pushing and the other pulling air through to cool them. This setup increases airflow allowing for heat to be carried away faster. The stack of parallella boards are placed on a non conductive surface and power is transfered through the spacers connecting each board. Access to the head node is done via ssh from a non Parallella host which will be referred to as the Controller, which is also connected to the gigabit switch. Giving commands to the cluster is typically done by connecting to the head node of the parallella stack via the Controller. Connected to the switch other than the cluster and external machine are a NFS server and a DHCP server. These two entities do not need to be on separate machines but in this case they are, They may also be setup to be on the Controller but when connecting a new computer to control and interact with the cluster it will have to be setup to fulfil the NFS and DHCP roles making the cluster less portable from machine to machine.

3.1.2 Components

The following are the components needed to build the cluster:

- 20 Nickel plated male-female M3 hex spacers. 15mm body and 6mm stud
- 2x M3 nuts and 4x M3 washers
- 2x M3 crimp/solder tags
- 4 small 6mm long wires
- A power-supply that can output 5V 8A (2A per board)
- 3x 4-Pin Molex to 3-Pin ATX adaptors
- 2x 120mm case fans
- 4x P1601 Parallella boards [2]
- 4x SD cards
- A non-conductive surface
- 1GB Ethernet switch
- Cat5e Ethernet cables
- A non parallella for NFS: For this project a Raspberry pi was used.
- A DHCP server

3.2 Cluster Design and Interface

In this section we provide more details on the design of the cluster and its external interface.

3.2.1 Powering the Cluster

It was decided to power the Parallella boards using the padded mounting holes provided in each corner of the board. To allow power to be received from the mounting holes a conductive bridge was needed in the corner of the board closest to the barrel connector [2] as shown in Figure 3.2. Using solder and a short length of wire a jumper was placed connecting the two provided holes together; this location may be seen in more detail in [2].

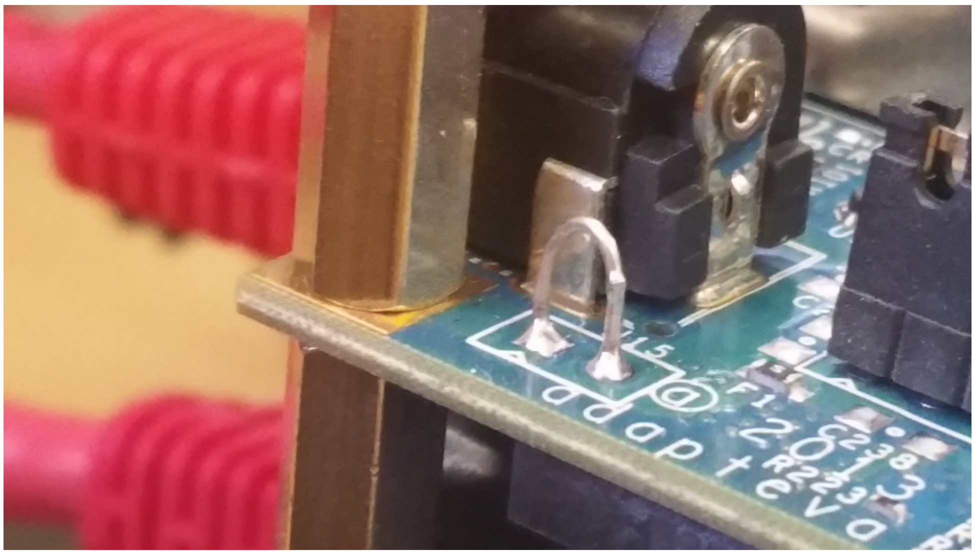

Figure 3.2: Soldered bridge allowing power from corner pads

Once the bridge was in place, power could be provided to multiple boards through the conductive material used to stack the boards. In this case, the provided Nickel plated male-female M3 hex spacers were used to connect all the boards to each other and carry a charge to each board as seen in Figure 3.3. Since the nickel stands are conductive care must be taken when powering the boards that the occupied surface is non-conductive;This is achieved by using the black material seen at the bottom of Figure 3.3.

To operate, each board requires 5 V at 2 A. Since four Parallella boards are being used, a power supply was needed that could output at least 8 A. This cluster uses a Corsair 450W which can provide a DC output of +5 V with a maximum load of 16 A. This meets the required power output requirements.

To connect the rails to the power supply, one of the 4-Pin Molex to 3-Pin ATX adaptors was modified by removing the 3-Pin ATX connector and one ground cable. Then, the cable providing power needed to be swapped from the 12V side of the Molex to the 5V

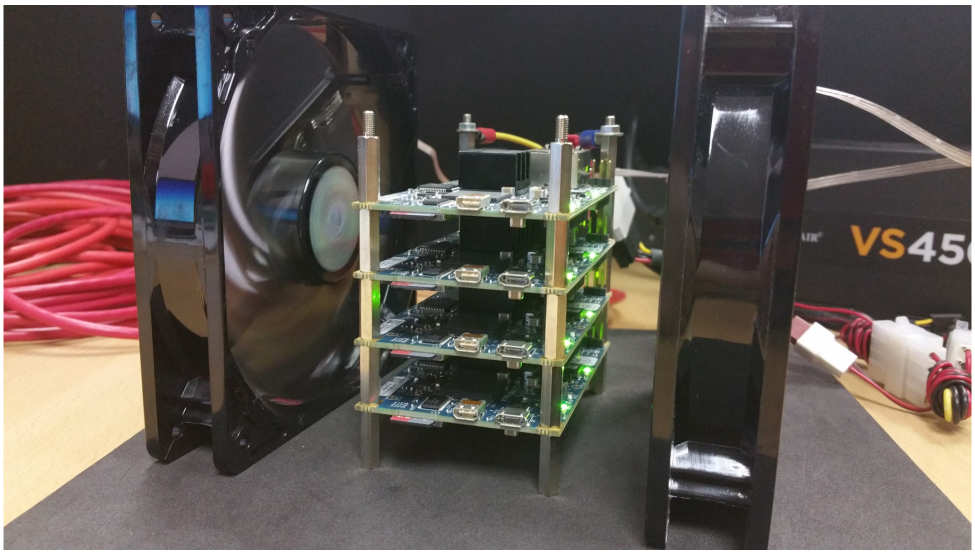

Figure 3.3: Stacked cluster using Nickel plated spacers

side. The loose ends of the cable were then affixed with solder tags so that the cable could easily and neatly be attached to the spacers connecting the individual Parallella boards. The yellow wire with a red solder tag is the 5 V connector and the black wire with the dark blue solder tag is ground. These are affixed in the corners with the 5 V wire being attached to the corner closest to the soldered bridge in Figure 3.2; the second ground connection may be connected to any of the other corners on the Parallella board.

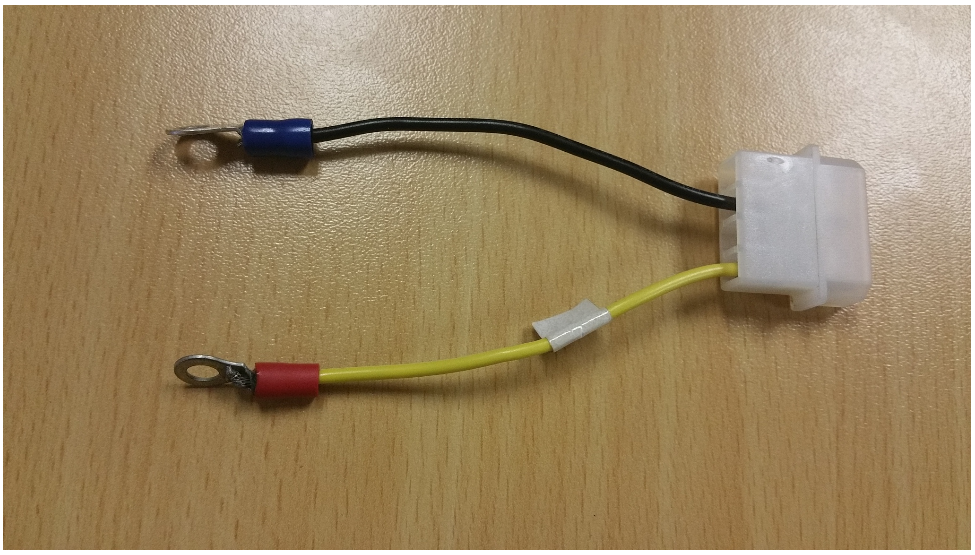

Figure 3.4: Modified Molex connector

This configuration meant that the standard power-supply did not need to be modified and the cluster could easily be detached . This opens up the power-supply to be used for other things while not powering the Parallella cluster. Having a standardised connector increases the cluster's ease of use by making it more obvious to future users how to provide the cluster with power.

Due to passive cooling not being enough to keep the Parallella board from over heating when under heavy use two fans have been set up on either side. One fan pushes air onto the boards while the other pulls air from the boards; this may be seen in Figure 3.1 at the start of the chapter. The fans are connected to the power supply by 4-Pin Molex to 3-Pin ATX adaptors thereby directly connecting the fans to the power supply and allowing the clusters cooling to also be easily detached. A heat sink was attached to the Zynq processors of each Parallella board with the fins directed parallel to the direction of the air pushed by the fans. This lets the air pass over the largest area and offers more heat dissipation compaired to if it was placed perpendicular to the air flow.

3.2.2 Communication Between Nodes

To allow communication between the individual parallella boards, a network needed to be set up. we used a 1-gigabit switch with sufficient ports to handle at least six connections and multiple category-5 enhanced or better Ethernet cables. Connecting each board to the switch using an Ethernet cable allows network traffic to ensue.

To allow communication with an individual board, an IP address was needed. To provide these addresses, a DHCP server was connected via an Ethernet cable to the switch. The DHCP server could then provide each of the boards with an individual IP address and a connection to the Internet via nat. To allow communication to the cluster from the outside world, a PC was connected to the switch and an IP address assigned to it from the DHCP server.

3.3 Software

In this section, we give a break down of the software and required configuration needed to allow programs to be executed using MPI on the Parallella cluster.

3.3.1 Linux

The Parallella boards were supplied with a raw disk image of Ubuntu 14.04, downloaded from the Parallella website[1]. There are three different versions of the Parallella board for which Ubuntu images are provided: the z7010 image applies to the P1600 and P1501, while the z7020 image is for the P1602 [25]. The z7010 is specific to Parallella boards with the Xilinx Zynq Dual-core ARM A9 XC7Z010 host processor, which matches those used in the cluster. The other image is for the Xilinx Zynq Dual-core ARM A9 XC7Z020, which is used on the embedded version of the board.

There are also two different images per host processor: the first is a headless one, which is what we have used as it is the most stripped down of the images, while the other includes a desktop environment and HDMI drivers. The latter version is useful for displaying generated graphics and writing code in a GUI environment. The latest version of Ubuntu 14.04 headless for the z7010 was downloaded[2] and extracted.

Each Parallella board has a microSD card that acts as its main hard drive and into which the board is booted. All software that will ultimately be executed on a Parallella board must be copied to the microSD card. The capacity of the cards used in the cluster is 16 GB and the speed class is UHS mark 1 which makes the Minimum Serial Data Writing Speed 10 MB/s[3]. Using a Windows 7 machine, Win32 Disk Imager[4] and an SD to USB adaptor, the downloaded image was copied to each of the microSD cards.

After installing the microSD cards, each Parallella was powered up; the default username was“parallella” and password“parallella” and each had the same hostname “parallella”. Having the same hostname is not ideal as it makes it difficult to tell one machine from the other. Running a network scan discovered the assigned IP addresses of the four Parallella boards given by the DHCP server. Using SSH each of the hostnames was changed to “parallella” followed by a number from 1 to 4 so that each could be uniquely identified.

Next the hostnames of each Parallella board were added to the *hosts* folder in the root directory of each Parallella, which allowed the name of the device to be used, instead of remembering its IP address when wishing to interact with it. The lines of code added to the hosts file are given in Listing 3.1.

[1] `ftp://ftp.parallella.org/ubuntu/dists/trusty/image/`

[2] At the time of writing, the appropriate file was Ubuntu-14.04-headless-z7010-20150130.1.img.gz

[3] `https://www.sdcard.org/developers/overview/speed_class/`

[4] Downloaded from `http://sourceforge.net/projects/win32diskimager/`

Listing 3.1: Contents of /etc/hosts

```
...
10.42.0.21   parallella1
10.42.0.22   parallella2
10.42.0.23   parallella3
10.42.0.24   parallella4
```

To use the Epiphany co-processor, the Epiphany SDK was installed on the system and its location added to the PATH variable. In the provided Ubuntu image, the SDK was compiled and the path added in the *.bashrc* file. The associated environment variables are “EDIR” and “EPIPHANY_HOME”; these are important when trying to run and compile Epiphany programs.

3.3.2 SSH

SSH or Secure Shell is a network protocol that allows one computer to securely access another over a network. In the case of the Parallella stack the Open-SSH implementation will be used to facilitate the passing of messages between nodes by OpenMPI. Secure shell by standard uses port 22 and TCP to create connections between nodes so it is vital that this port is available and open. For the master node to command the slave nodes using Open-SSH the ssh client needs to be installed using “apt-get install openssh-client” on the Parallella board Ubuntu distribution. On each of the slave nodes “openssh-server” needs to be installed for the openssh client to connect to. For convenience, passwordless SSH was set up between the nodes so that OpenMPI is able to run programs on the nodes without needing a password to be provided for each run. To do this RSA public/private key pairs are used to authenticate the master with the slave nodes. To set up the keys ssh-keygen is used to create the key pairs and then the public key needs to be added to the slave nodes’ authorised users. Using “ssh-copy-id [host]” where the host is replaced with the host to be accessed without a password.

The ssh-copy-id adds the public key of the node running the command to the authorised list of public keys on the host node, the user is prompted to input the host nodes password and the list is updated. Once the public key is added the trusted node may easily access the remote host using the “ssh” followed by the IP or hostname of a node.

3.3.3 Open-MPI

OpenMPI was preinstalled and the environment correctly set up on the Ubuntu image provided. To set up OpenMPI, a user account was created on each parallella with the same directories and file locations for consistency when running anything through mpiexe or mpirun. To synchronise the files so that each instance of a program on each node had access to its local directory, NFS was used, the configuration of which is described in Section 3.3.4.

If SSH has been configured correctly, the password of each node being used does not need to be entered when running an MPI program.

To inform MPI what nodes to run on using their address and how many cores must be used on each node, a machine/host file is used. The format of this file is given in Listing 3.2; here, the node address is followed by the number of cores to run on, which are referred to as slots. The hostfile is chosen when invoking mpiexe or mpirun using any of these flags "-h, -hostfile, -machinefile" followed by the hostfile to be used. In Listing 3.2, the number of slots is specified one per node even though the Zynq processor has two ARM cores; if the program used both cores and both threads tried to interact with the Epiphany core it would be undesirable. An explanation on how to avoid this is given in Section 4.1. The hostfile in Listing 3.2 is for interacting with only one core per board which accesses the 16 cores of the co-processor.

To compile programs the command "mpicc" is used together with the flags needed by gcc to link the epiphany libraries, is used.

Listing 3.2: Contents of ~/NFS_share/hello_world/machinefile

```
parallella1 slots=1
parallella2 slots=1
parallella3 slots=1
parallella4 slots=1
```

When using MPI over SSH, the hosts environment is not carried over to each node, and when OpenMPI connects to another node it does not run .bashrc where the Epiphany variables and environment are set up. Instead, these variables have to be exported using the -x flag when invoking mpirun or mpiexe; otherwise, the programs executing on the slave nodes will not be able to find the shared libraries for the Epiphany co-processor and will not know what sort of Epiphany chip is being used. (This may be a problem when executing on Parallella boards with different sized Epiphany chips.)

The environment variables that have to be passed across are LD_LIBRARY_PATH which contains the location of the Epiphany's shared library's (eg e-hal.so) and EPIPHANY_HDF which contains whether it is a 64-core Epiphany chip or a 16-core Epiphany chip. (Scalability is one of the strengths of the Epiphany architecture so more chip sizes may become available. At time of writing there are only 64 core and 16 core chips.)

3.3.4 Network File Share

For OpenMPI to work correctly, the file structure and locations of used files must be the same on each node. For this to happen, one of the nodes or a different machine was chosen to host all the program files and have all the nodes connect to that shared drive so that each node has the exact same version of the files as every other node. For our cluster, a different machine was used as problems occurred when running nfs-kernel-server on the master node, which was originally set as one of the Parallella boards; for details, refer to Section 3.4.2. To remedy this, a Raspberry Pi was set up with an installation of nfs-kernel-server and its dependences through a repository. The upside of using a Raspberry Pi is it means that the cluster continues to use very little power, the downside is that the Ethernet adaptor used by the Raspberry Pi is only 100Mb/s and will not take advantage of the 1Gb/s Ethernet that is used by everything else that is part of the cluster. So the shared storage is accessed slower and if the cluster tries to access and copy large amounts of data there will be a large performance drop. To counter this when using large files and data they should be copied to each Parallella board's local hard drive for faster access.

NFS manages what is shared using an *exports* file that lists the permissions and locations of what is shared and allows clients to sync with the folder.

A folder called cloud was created on the NFS server to be the shared folder for the cluster. This was then added to the nfs exports file, as illustrated in Listing 3.3. The asterisk allows any host to connect to this share; however, this can be restricted if desired to increase security and limit unwanted access. The words following the asterisk are the options for that shared folder

- rw allows the nfs client to read and write files to the shared directory
- sync forces any changes that are made to files to be immediately flushed to disk

- all_squash All users including root when creating a file or folder using NFS will be creating it as a user named nobody

The rw option allows any of the Parallella boards or machine connected to the NFS server to read and write to the directory, this is vital as it allows the compiled files to be created and shared by any machine connected and then read and executed by other connected machines. The sync option forces a change to be written to the servers disk before informing the client that it is safe to remove its cached data. If async is chosen the server replies before it has properly written the data and this may lead to corruption of the data if something happens to the server before it has properly copied the data. More on sync vs async behaviour may be found at[5]. The all_squash option means every client connected to the server when creating and writing files they are treated as a user "nobody". The user nobody has been setup to create files with read write and execute allowed so that clients do not have to chmod as root when wanting to run a program. It also prevents files being created on the server as root which may pose a security risk.

Listing 3.3: Contents of /etc/exports

```
/cloud   *(rw,sync,no_subtree_check ,all_squash)
```

After the export file had been set up, the nfs-kernel-server was restarted to refresh its settings. Once the server configuration had been completed, all the nodes in the cluster were connected to the server. To do this, on each node a new folder was created in which to mount the share. This folder was located in the same place on all the nodes. Using mount only mounts the share until the cluster is next restarted; to prevent having to remount the shared folder every time the cluster was restarted, the following line was added to the file */etc/fstab*: "10.42.0.21:/cloud /home/parallella/cloud nfs". fstab is looked at by the operating system on boot and each line is the configuration for automatically mounting a folder. The line is split into: location of folder to mount followed by where to mount it and then the type of file system the folder uses, There are more options but for the purposes of this project just these are needed.

On completion of the configuration, when a change is made on one node, it is saved and reflected on every other node.

[5]http://nfs.sourceforge.net/nfs-howto/ar01s05.html

3.4 Problems encountered with Cluster Configuration

This section discusses some of the problems encountered when setting up the cluster and how these were mitigated.

3.4.1 DHCP issue

After several alternative attempts, the way an Internet connection was provided to each of the boards was to connect a DHCP server to the switch which provided internet via nat. With each of the Parallella nodes having an IP address, once the server has access to the Internet, it can provide a connection to each of the nodes. Using a DHCP server to provide IP addresses to each of the boards simplifies the set up required to communicate with the boards and other devices connected to the switch.

3.4.2 File Sharing NFS

Originally one of the Parallella boards was designated as the main node; however, due to NFS-kernel-server package provided by the Ubuntu repository not working with the Parallella kernel, it was decided to move the NFS server to a different machine.

This problem arose after installing the NFS-kernel-server package available from the Ubuntu repository on the designated Parallella main node. When attempting to set up the exports, the user was notified of an incompatibility with the kernel. It may be possible to compile the NFS-kernel from source on the Parallella, but this may still not work as NFS requires the kernel to be set up to support it[6].

3.5 Summary

The first section of this chapter provided an overview on how the Parallella stack was going to be built and the entities needed for operation. In the following sections the process

[6] Linux NFS-HOWTO `http://nfs.sourceforge.net/nfs-howto/ar01s02.html#software_prereqs`

of building the physical Parallella stack and its requirements are described leading into the required software and software configurations so that programs may be run on the cluster. To end the chapter the issues discovered whilst building the cluster and how they were overcome.

CHAPTER

4 BENCHMARKING THE CLUSTER

In this chapter we discuss how programs are created for execution on the cluster and compare the execution speeds of these programs running on different machines.

4.1 Creating Programs for Parallella Cluster

As the Parallella stack is a heterogeneous cluster, to be able to utilise fully the computational power of each board, MPI is used in conjunction with the Epiphany libraries.

Creating a program for both MPI and the Epiphany co-processor requires splitting of the work on two occasions: once at the MPI level and again for distribution to the co-processor. Figure 4.1 illustrates the process of running a program that uses MPI to spilt work between the four Parallella boards in the cluster, which then load the required srec files onto their respective co-processors. Figure 4.1 displays the way that MPI is run over SSH using parallella1 as the master node; this may slow down parallella1 owing to the overhead of setting up the MPI program on each Parallella board. If the cluster has a large number of nodes, the delay to parallella1 may be large, causing an unbalanced workload and extended execution time.

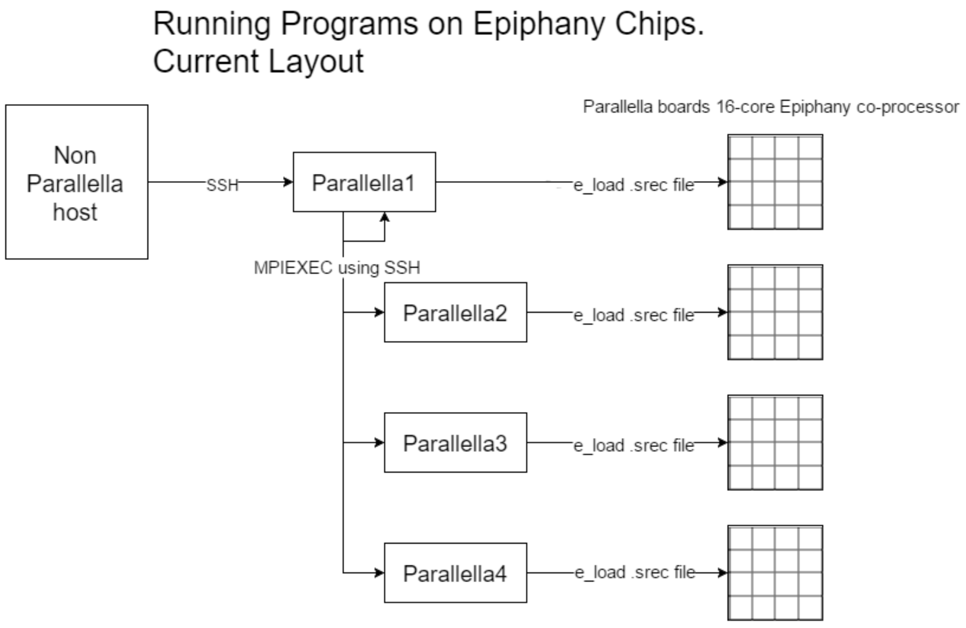

Figure 4.1: Starting a program using MPI for execution on Epiphany cores

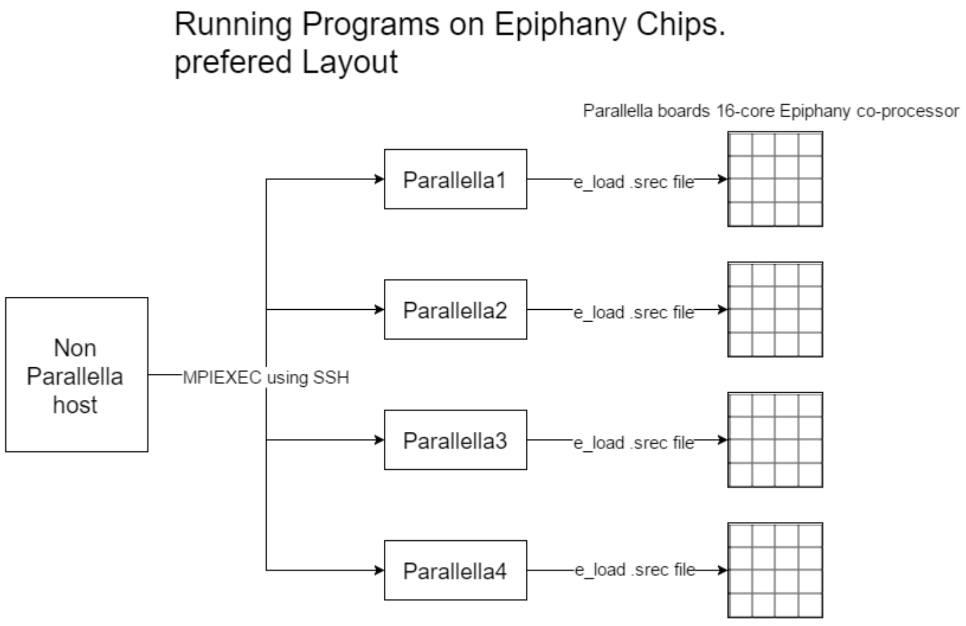

Figure 4.2: Execution on the cluster without using one of the Parallella boards to set up the MPI environment

A configuration favouring performance can be seen in Figure 4.2 where a separate computer is used as the master node. For ease of use, the master node should be equipped with an ARM compiler and e-gcc so that programs can be created and run without the need to SSH into the cluster manually. The master node should also be able to run an NFS server and each node should have the file system mounted so that they can retrieve the compiled srec files. In Figure 4.2, the master node sets up the MPI environment and organises each of the Parallella boards in the cluster. This removes the strain of setting up the MPI environment from one of the Parallella boards so that its load is similar to that of every other node in the cluster. Running a program on a Parallella board while using the Epiphany co-processor requires multiple files. Each program/kernel that needs to be run on an Epiphany core is compiled using the e-gcc compiler provided by Adapteva[1] and then converted to an srec file using e-objcopy. An srec file is a representation of a binary file using ASCII hex text. The srec file is loaded onto a core by a separate program that is compiled by gcc using the e-hal library in the Epiphany SDK to provide the necessary functions. These libraries must be provided at compile time using the -l argument. A program on a core can use "e-lib" to get its core id or coordinates in the 16×16 matrix of cores. The core's identity can be used to split work when a single program is loaded onto multiple cores instead of creating one program for each core. When the host program loads an srec onto the Epiphany co-processor, the host program needs to know onto which cores the program must be loaded and have access to the srec file. The host program can do this one core at a time using e_load(), or by choosing a grid of cores on which to load the srec using e_load_group.

Listing 4.1: Typical Epiphany host program

```
#include <e-hal.h>
...
e_init(NULL);
e_reset_system();
e_get_platform_info(&platform);
e_open(&dev, 0, 0, platform.rows, platform.cols);
...
e_load_group("e_hello_world.srec", &dev, 0, 0,
                          platform.rows, platform.cols, E_TRUE);
...
e_close(&dev);
e_finalize();
...
}
```

In Listing 4.1, we first initialise the core and reset it to remove any garbage that may have

[1]ftp://ftp.parallella.org/esdk

remained from a previous program. The next step is for the program to work out the size of the core. This information is provided by the path variable EPIPHANY_HDF and is stored in the platform. Once the type of Epiphany processor is known, the dimensions of the Epiphany core can be used to split programs equally over the processor. The example given in Listing 4.1 loads one program as its srec file onto all the cores. It is possible to load data into the local memory of an eCore or into the shared memory of the Epiphany processor before and after loading the srec onto any individual core. This allows the required data to be set up depending on conditions external to the Epiphany. Once an srec is loaded onto an eCore, the program starts executing; the host program continues executing immediately after initialising the Epiphany co-processor. The result of the computation can be retrieved from the eCore's memory but care must be taken to ensure that the core has completed its computation. When the Epiphany processor terminates, the host program closes the opened workgroup or core and finalises the use of the processor. Closing the Epiphany processor is similar to how the MPI libraries exit cleanly.

Listing 4.2 shows some of the basic commands to obtain the current core's ID and its position in the matrix of cores, and then to idle the core. To get to the point where this code is loaded as an srec onto the co-processor by the host program, it needs to be compiled using e-gcc. Then, using e-objcopy, the ebinary is converted to an srec file.

Listing 4.2: Typical Epiphany core program

```
#include "e_lib.h"
...
e_coreid_t        coreid;
e_memseg_t        emem;
e_shm_attach(&emem, "Name_Shared_Memory")

coreid = e_get_coreid();
e_coords_from_coreid(coreid, &my_row, &my_col);
...
__asm__ __volatile__ ("idle");
}
```

The first line in Listing 4.2 includes the e_lib library that contains definitions and functions for dealing with the eCore; for a complete list of functions, the reader is referred to [7]. All the Epiphany functions are prepended with e_ to differentiate them from normal C functions. The emem variable in Listing 4.2 is used to access shared memory. It is attached to the memory allocated by the host using e_shm_alloc, and which can be accessed with e_read and e_write. At the end of the example, the core is forced into its idle state; it is also possible to just return with EXIT_SUCCESS or EXIT_FAILURE.

The MPI setup portion of the code is very similar to the Epiphany code, given in Listing 4.3. When compiling a host program that uses MPI, the MPI compiler mpicc must be used. The same compiler flags and libraries are used when compiling.

Listing 4.3: Basic MPI setup

```
#include <mpi.h>
...
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Get_processor_name(processor_name, &name);
...
MPI_Finalize();
return 0;
}
```

The first line in Listing 4.3 serves to include the MPI library, so that the program can use Open-MPI's definitions and functions. Before MPI commands can be used to pass messages carrying data, MPI_Init must be called by the main thread. It is recommended that this is done as little as possible before the call and especially after finalisation to prevent anything changing the external program state[2]. Once MPI has been initialised, the number of threads to be activated is obtained as well as the current process' rank, which is that process' unique identifier. It is also possible to retrieve the hostname of the system, making it easier to identify the processor node. In the case of the Parallella stack,"parallella" 1, 2, 3 or 4 is placed into the variable name. After initial setup, the bulk of the processing takes place and when this terminates, the main thread executes the Finalize command to ensure that all the other threads have returned correctly without error.

By using the code provided in Listings 4.1, 4.2, and 4.3, all the cores in the Parallella stack can be accessed and used.

[2]http://www.mpich.org/static/docs/v3.1/www3/MPI_Init.html

4.2 Benchmarks

In this section, we discuss the two benchmarks executed. The first benchmark determines the Parallella stack's ability to do a large number of floating point multiplications while the second tests the cluster's communication speed.

4.2.1 Floating Point Multiplication

This benchmark, which is embarrassingly parallel, tries to emulate ideal conditions for the Parallella stack by creating and splitting up work for each core to do. This work requires no synchronisation with any other processes. This benchmark merely multiplies two floating point numbers for the desired number of iterations. However, the two floats are modified on every iteration to prevent the processor from possibly using a cached answer. This benchmark has a sequential version, which was executed on the i5 CPU and on a single ARM processor, a Parallella version for running on a single Parallella, and a modified version of the Parallella implementation with added MPI compatibility for execution on the cluster.

For splitting work among the eCores, the number of iterations allocated to each core is calculated on the ARM processor by dividing them amoungst the eCores and using e_write copied to all the eCores. Another integer-size piece of memory is also initialised to zero using e_write, for use as a flag to notify when the eCore has finished processing. The srec of the program for multiplying floats is then loaded onto the Epiphany, which first retrieves how many iterations to perform and then starts looping through the work. Once the eCore has finished its required work, it sets the section of memory set aside as a flag to one and goes into an idle state. The host program after starting all the eCores, begins a while loop where it polls the flag section of memory until it retrieves sixteen ones, which signifies that all the eCores have finished their calculations and it may continue to finalize and close the Epiphany processor.

The MPI cluster version does the same as the single Parallella version, but it splits the number of iterations into four chunks, each of which is further divided by sixteen. The program then uses MPI_Barrier to make sure all the nodes are at the same point. It then starts a timer and goes through the single Parallella version and afterwards is stopped by another barrier. The time is measured and reported by processor 0 using the rank variable to ensure only one process is printing to screen. The processors all run MPI_Finalize and exit. The version that uses the Epiphany and the ARM cores is the same but using the processor rank variable, only even numbered processors load programs onto the Epiphany cores and then run through the same number of iterations themselves before checking to see if the eCores have completed. The odd numbered processors immediately start with their assigned number of iterations and synchronise with the even numbered processors at the last barrier before calculating the total time taken.

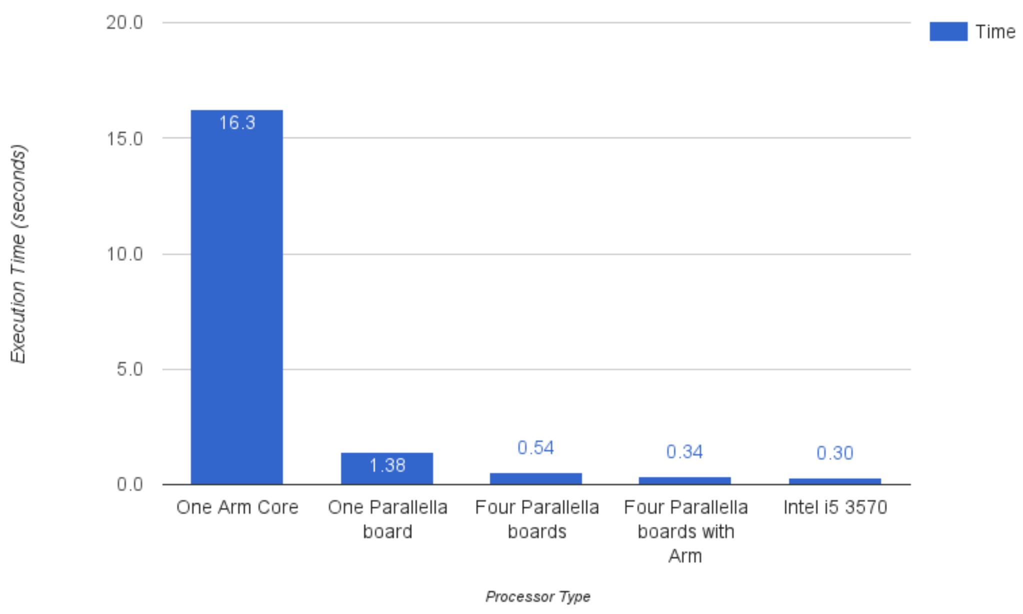

Figure 4.3: Time taken per machine for 100,000,000 iterations

Figure 4.3 shows that the time difference between one Parallella board and four Parallella boards is closer to a 2.5× speedup instead of the expected 4× increase. Before adding the MPI_Barriers and having each process started by MPI being completely separate after launch, the speedup was between 3.9× and 4.1×. This situation we felt was unrealistic as even the most parallel application would need to amalgamate its results at the end of the computation. This adds a constant time to the total computation, which, if the computation time were sufficiently large, would not effect the time in a significant way as it is not reliant apon the amount of iterations.

The difference in the timings of the Intel i5 3570 and the Parallella are very close and while the i5 is faster, it is worth noting that at 100% load, the Parallella stack consumes approximately 20 Watts of power, which is three times less than that used by the i5 when idle[3].

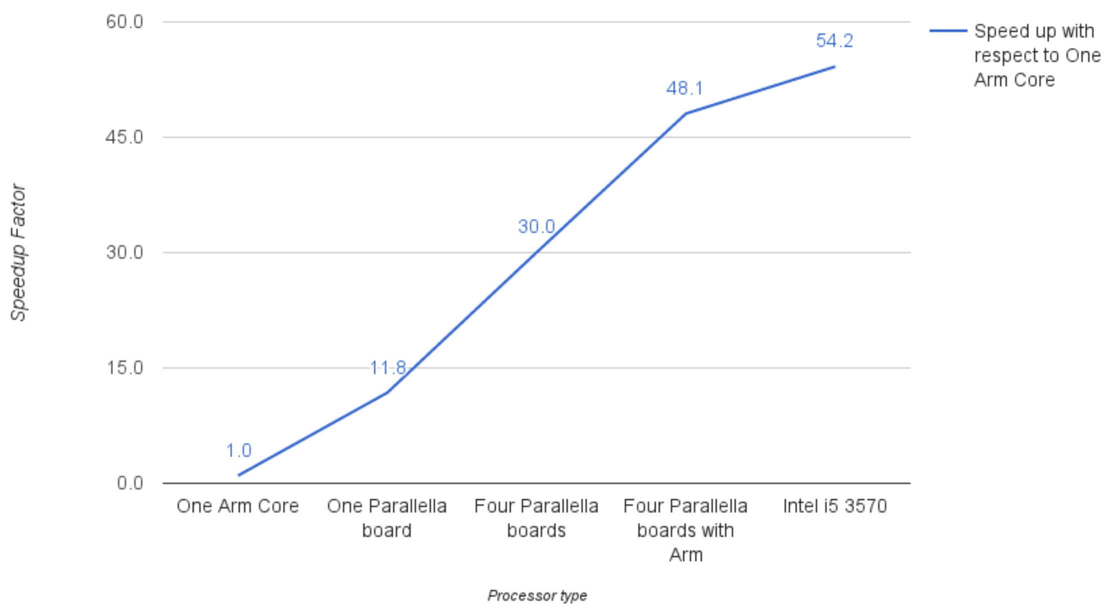

Figure 4.4: Speed increase relative to a single ARM core

The performance increase using one Epiphany chip for low powered computation is 11× faster than the ARM core provided with the Parallella board. The downside is that using the Epiphany core requires more code and specialised algorithms to achieve the speed increase. On the other end of the spectrum using all the ARM cores and the Epiphany co-processors is nearly enough to achieve equal performance gains with the sequential i5.

4.3 Ethernet Bandwidth Benchmark

This benchmark was created by Blaise Barney[4] to measure point-to-point communications between MPI processors.

The benchmark pairs processes together and sends incrementally larger and larger messages between them and notes the speeds. As seen in Figure 4.5, the bandwidth appears to get faster the larger the messages get, excluding some outliers. The larger message

[3]`http://www.tomshardware.com/reviews/core-i5-3570-low-power,3204-13.html`
[4]available from `https://computing.llnl.gov/tutorials/mpi/samples/C/mpi_bandwidth.c`

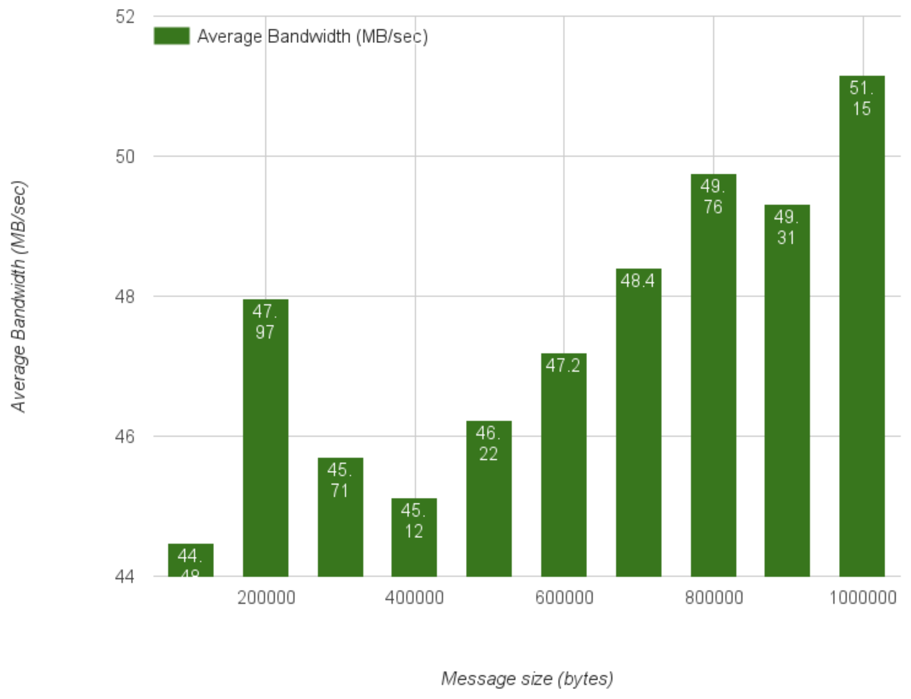

Figure 4.5: Average point to point bandwidth using MPI

size increases the time in which a message is in transmission while the time to set up the connection between nodes stays the same as the number of messages sent is constant. The larger message size causes the MPI overhead per byte of data to be less.

The average bandwidth in Figure 4.6 is lower than the values recorded in Figure 4.5, this is due to the two processes per Parallella board competing to use the Ethernet interface. These benchmarks show that it may be better for one process to handle the boards communications and retrieve data for both local processes.

4.4 Limitations

This section will cover the limitations and weaknesses of the Parallella board that were discovered over the course of this research.

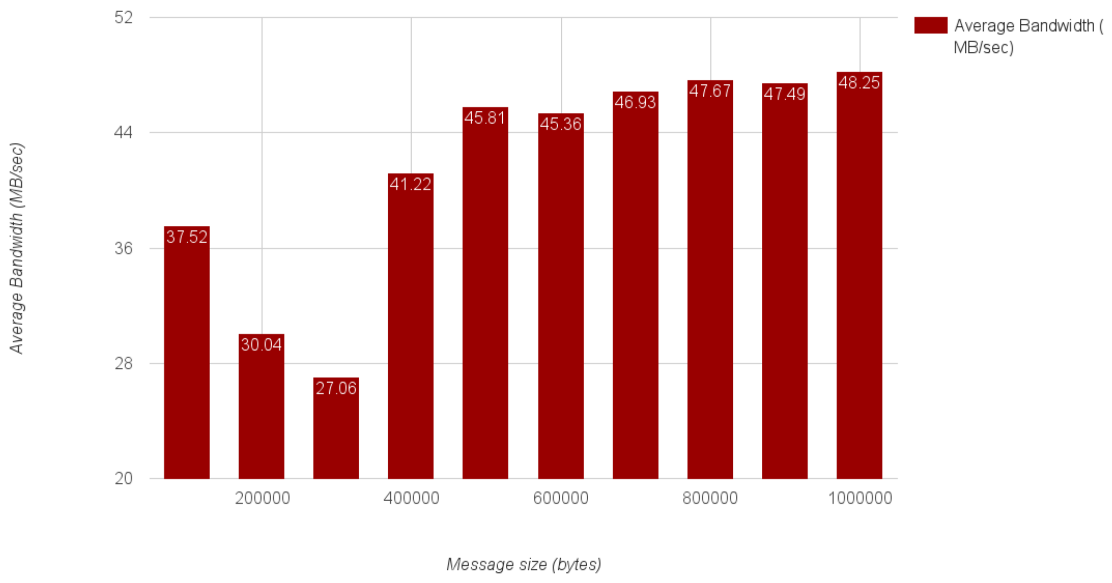

Figure 4.6: Average point to point bandwidth using MPI with multiple processes per board

4.4.1 Complex Arithmetic

When running the benchmark in Section 4.2.1 instead of multiplying we attempted to divide the two floating numbers. The performance difference in this case was huge, with the ARM 667MHz core 16.74× faster than the Parallella stack just using the Epiphany co-processor and 66.17× faster than a single 16-core Epiphany chip. This is due to the eCores having no hardware support for higher-complexity arithmetic. This translates to doing long division and requiring more CPU cycles per instruction. The slow division has far reaching consequences as many useful functions rely upon it such as modula and rand(). To try mitigate the performance drop if a large amount of division or complex arithmetic needs to be done it is better to pass the work to the ARM processor which can do the calculations and pass the data back to the co-processor.

4.4.2 Hardware Optimisations

All optimisations for Epiphany programs must be done by the compiler instead of clever hardware. The simple RISC cores provided on the Epiphany chip do not attempt to predict or change the ordering of instructions being executed[5].

4.5 Summary

Within this chapter we provided the basic commands and information needed to create and run an MPI Epiphany program on the Parallella stack. The results, and method of two benchmarks and a breakdown of what may be concluded from the results. The last section of the chapter goes over the discovered limitations and if possible how the author thinks they may be mitigated.

[5]`http://www.bdti.com/InsideDSP/2012/09/05/Adapteva`

CHAPTER

5 CONCLUSION

This chapter summarises the research, gives our conclusions on the Parallella cluster, and suggests possible extensions to the research.

5.1 Summary

Our objectives for this project were to build a cluster using four Parallella boards and to benchmark the cluster against similar low-cost systems.

In this thesis, we discussed how a cluster of four Parallella boards was constructed and powered using a single power supply. The Parallella stack is actively cooled by two fans and connected to a 1Gb/s switch to facilitate communication between boards.

Software for programming and running distributed programs on the cluster was set up and configured and an NFS server for storing files and a DHCP server to provide Internet and IP addresses were set up and added to the network. This means that the only requirement for interaction with the Parallella stack is a working SSH-client, making the system easily accessible with minimum effort and easy to set up on multiple machines. This satisfies the first two main objectives.

For benchmarking, some simple programs were created and executed sequentially and in parallel on the Parallella stack. The Parallella stack performed slightly worse when compared with an Intel i5 3.40GHz processor, but used at least three times less power [1]. In terms of cost, both the four Parallella boards and the Intel i5-3570 cost approximately \$400 [2], but to make use of both devices more components are needed adding to the total cost. Depending what hardware is bought to use with these devices the costs could be similar.

We found that the lack of hardware support for complex arithmetic such as square roots and division can be costly to performance if the instruction is frequent enough and steps are not taken to delegate complex arithmetic to the local ARM core, which does have the hardware support to process quickly. It is was also discovered and noted that the RISC cores on the Epiphany do not do any runtime optimisations and the only automated optimisations are provided by the compiler.

Unfortunately, owing to issues in rewriting the standard parallel benchmark programs to allow the ARM processor to partition the work among the Epiphany cores, we were unable to compare the performance of the cluster against results available for other similar low-cost, low-power clusters and systems.

5.2 Future Work

As this research only set out to create a basic cluster, there is plenty of scope for further development and optimisation.

More benchmarks and programs should be executed to further test the Parallella stack. Additionally, research into using the Brown Deer COPRTHR (co-processing threads) library[3] to help create Epiphany MPI programs is needed. The Parallella's Zynq processor has a field-programmable gate array that may be used to re-program the boards to better suit different purposes.

To facilitate use of the cluster, a web front end could be created to receive and run programs or to monitor the cluster's current work load. This could be extended to process

[1]`http://www.tomshardware.com/reviews/core-i5-3570-low-power,3204-13.html`

[2]`http://www.amazon.com/Intel-i5-3570-Quad-Core-Processor-Cache/dp/B0083U94D8/ref=sr_1_1?s=pc&ie=UTF8&qid=1446331124&sr=1-1&keywords=Intel+Core+i5-3570`

[3]`http://www.browndeertechnology.com/coprthr.htm`

requests and schedule work so that the cluster can be used simultaneously by multiple people.

Other extensions could be creating a library for languages other than C, C++ and OpenCL to be able to use the Parallella stack.

REFERENCES

[1] Open MPI: Open Source High Performance Computing. Online. Accessed: 2015.11.17. Available from: `http://www.open-mpi.org/`.

[2] The Parallella Board. Online. Accessed: 2015-03-01. Available from: `https://www.parallella.org/`.

[3] Supercomputer.io. Online. Accessed: 2015.05.27. Available from: `http://supercomputer.io/`.

[4] Top 500 super computers. Online. Accessed: 2015.02.25. Available from: `http://www.top500.org/`.

[5] A. Petitet, R. C. Whaley, J. D., and Cleary, A. HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers. Online. Accessed: 2015.05.16. Available from: `http://www.netlib.org/benchmark/hpl/`.

[6] Adapteva. Epiphany Datasheet. Online. Accessed: 2015.05.05. Available from: `http://adapteva.com/docs/e16g301_datasheet.pdf`.

[7] Adapteva. Epiphany SDK Reference. Online. Accessed: 2015.11.01. Available from: `http://adapteva.com/docs/epiphany_sdk_ref.pdf`.

[8] Bridges, P., Doss, N., Gropp, W., Karrels, E., Lusk, E., and Skjellum, A. User's guide to MPICH, a portable implementation of MPI. *Argonne National Laboratory 9700* (1995), 60439–4801.

[9] Buyya, R. High performance cluster computing. *New Jersey: Prentice Hall* (1999).

[10] Chapman, B., Jost, G., and Van Der Pas, R. *Using OpenMP: portable shared memory parallel programming*, vol. 10. MIT press, 2008.

[11] Cox, S., Cox, J., Boardman, R., Johnston, S., Scott, M., and O'Brien, N. Iridis-pi: a low-cost, compact demonstration cluster. *Cluster Computing 17*, 2 (2014), 349–358.

[12] Dagum, L., and Menon, R. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE 5*, 1 (1998), 46–55.

[13] Davies, T., Karlsson, C., Liu, H., Ding, C., and Chen, Z. High performance linpack benchmark: a fault tolerant implementation without checkpointing. In *Proceedings of the international conference on Supercomputing* (2011), ACM, pp. 162–171.

[14] Dongarra, J., and Luszczek, P. Linpack benchmark. In *Encyclopedia of Parallel Computing*. Springer, 2011, pp. 1033–1036.

[15] Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., et al. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer, 2004, pp. 97–104.

[16] Geer, D. Chip makers turn to multi-core processors. *Computer 38*, 5 (2005), 11–13.

[17] Geist, A. *PVM: Parallel virtual machine: a users' guide and tutorial for networked parallel computing*. MIT press, 1994.

[18] Gropp, W., and Lusk, E. Installation guide for mpich, a portable implementation of MPI. Tech. rep., Technical Report ANL-96/5, Argonne National Laboratory, 1996.

[19] Gropp, W., Lusk, E., Doss, N., and Skjellum, A. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel computing 22*, 6 (1996), 789–828.

[20] Jack J. Dongarray, Piotr Luszczeky, A. P. The LINPACK Benchmark: Past, Present, and Future. Online, 12 2001. Accessed: 2015.05.28. Available from: `http://www.netlib.org/utk/people/JackDongarra/PAPERS/hpl.pdf`.

[21] Laboratory, I. C. HPC Challenge Benchmark. Online, May 2015. Accessed: 2015.05.29. Available from: `http://icl.cs.utk.edu/hpcc/`.

[22] McCalpin, J. D. STREAM: Sustainable Memory Bandwidth in High Performance Computers. Online. Accessed: 2015.05.29. Available from: `http://www.cs.virginia.edu/stream/`.

[23] Olofsson, A., Nordström, T., and Ul-Abdin, Z. Kickstarting high-performance energy-efficient manycore architectures with Epiphany. *arXiv preprint arXiv:1412.5538* (2014).

[24] Padua, D. A., and Wolfe, M. J. Advanced compiler optimizations for supercomputers. *Communications of the ACM 29*, 12 (1986), 1184–1201.

[25] Parallella. Parallella Reference Manual. Online. Accessed: 2015.05.05. Available from: `http://www.parallella.org/docs/parallella_manual.pdf`.

[26] Richie, D. Threaded MPI for the Epiphany Architecture. Online. Accessed: 2015.05.13. Available from: `https://www.parallella.org/2015/04/09/threaded-mpi-for-the-epiphany-architecture/`.

[27] Richie, D., Ross, J., Park, S., and Shires, D. Threaded MPI programming model for the Epiphany RISC array processor. *Journal of Computational Science* (2015).

[28] Sanbonmatsu, K., and Tung, C.-S. High performance computing in biology: multimillion atom simulations of nanoscale systems. *Journal of structural biology 157*, 3 (2007), 470–480.

[29] Takahashi, D. FFTE: A Fast Fourier Transform Package. Online. Accessed: 2015.05.29. Available from: `http://www.ffte.jp/`.

[30] Tezduyar, T., Aliabadi, S., Behr, M., Johnson, A., Kalro, V., and Litke, M. Flow simulation and high performance computing. *Computational Mechanics 18*, 6 (1996), 397–412. test.

[31] Varghese, A., Edwards, B., Mitra, G., and Rendell, A. P. Programming the Adapteva Epiphany 64-core network-on-chip coprocessor. In *Parallel &*

Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International (2014), IEEE, pp. 984–992.