MSc Scientific Computing Dissertation
Benchmarking a Raspberry Pi 4 Cluster

John Duffy

September 2020

Abstract

Dedication

Declaration

I declare that..

Acknowledgements

I want to thank...

Contents

Chapter 1

Introduction

1.1 Arm

Since the release of the Acorn Computers Arm1 in 1985, as a second coprocessor for the BBC Micro, through to powering today's fastest supercomputer, the 7,630,848 core *Fugaku* supercomputer [1], Arm has steadily grown to become a dominant force in the microprocessor industry, with more than 170+ billion Arm-based microprocessors shipped to date [2].

Famed for power efficiency, which directly equates to battery life, Arm-based microprocessors dominate the mobile device market for phones and tablets. And market segments which have almost exclusively been based upon x86 microprocessors from Intel or AMD are also increasingly turning to Arm. Microsoft's current flagship laptop, the Surface Pro X, released in October 2019, is based on a Microsoft designed Arm-based microprocessor. And Apple announced in June 2020 a roadmap to transition all Apple devices to Apple designed Arm-based microprocessors within 2 years.

When Acorn engineers designed the Arm1, and subsequently the Arm2 for the Acorn Archimedes personal computer, low power consumptions was not the primary design criteria. Their focus was on simplicity of design. Influenced by research projects [3] at Stanford University and the University of California, Berkeley, their focus was on producing a RISC (Reduced Instruction Set Computer) design. In comparison to contemporary CISC (Complicated Instruction Set Computer) designs, the simplicity of RISC required fewer transistors, which directly translated to lower power consumption. The RISC design permitted the Arm2 to outperform the Intel 80286, a contemporary CISC microprocessor, whilst using less power.

Figure 1.1: **The Raspberry Pi 4 Model B**.

1.2 Raspberry Pi

The Raspberry Pi Foundation, founded in 2009, is a UK based charity whose aim is to "promote the study of computer science and related topics, especially at school level, and to put the fun back into learning computing". Through it's subsidiary, Raspberry Pi (Trading) Ltd, it provides low-cost, high-performance single-board computers called Raspberry Pi's, and free software.

At the heart of every Raspberry Pi is a Broadcom "System on a Chip" (SoC). The SoC integrates Arm microprocessor cores with video, audio and Input/Output (IO). The IO includes USB, Ethernet, and General Purpose IO (GPIO) pins for interfacing with devices such as sensors and motors. The SoC is mounted on small form factor circuit board which hosts the memory chip, and video, audio, and IO connectors. A MicroSD card is used to boot the operating system and for permanent storage.

Initially released in 2012 as the Raspberry Pi 1, each subsequent model has seen improvements in SoC microprocessor core count or performance, clock speed, connectivity and available memory.

The Raspberry Pi 1 has a single-core 32-bit ARM1176JZF-S based SoC clocked at 700 MHz and 256 MB of RAM. The RAM was increased to 512 MB in 2016.

The Raspberry Pi 2, released in 2015, introduced a quad-core 32-bit Arm Cortex-A7 based SoC clocked at 900 MHz and 1 GB of RAM.

Figure 1.2: **The Raspberry Pi Zero**.

In 2016, the Raspberry Pi 3 was released with a quad-core 64-bit Arm Cortex-A53 based SoC clocked at 1.2 GHz, together with 1 GB of RAM.

The most recent addition to the range, in 2019, is the Raspberry Pi 4, sporting a quad-core 64-bit Cortex-A-72 based SoC clocked at 1.5 GHz. This model is available with 1, 2, 4 and 8 GB of RAM. This model with 4 GB of RAM was used for this project.

Since 2012 the official operating system for all Raspberry Pi models has been Raspbian, a Linux operating system based on Debian. Raspbian has recently been renamed Raspberry Pi OS. To support the aims of the Foundation, a number of educational software packages are bundled with Raspberry Pi OS. These include *Wolfram Mathematica*, and a graphical programming environment aimed at young children called *Scratch*.

Python is the official programming language, due to its popularity and ease of use, and the inclusion of an easy to use Python IDE has been a Foundation priority. This is currently *Thonny*.

Even though the Raspberry Pi 3 introduced a 64-bit microprocessor, Raspberry Pi OS has remained a 32-bit operating system. However, to complement the introduction of the Raspberry Pi 4 with 8 GB of RAM, a 64-bit version is currently in public beta testing.

Raspberry Pi OS is not the only operating system available for the Raspberry Pi. The Raspberry Pi website provides downloads for Raspberry Pi OS, and

Figure 1.3: **The Raspberry Pi Compute Module 3+**.

also NOOBS (New Out of the Box Software), together with a MicroSD card OS image writing tool called Raspberry Pi Imager. NOOBS and Raspberry Pi Imager make it easy to install operating systems such as Ubuntu, RISC OS (the original Acorn Archimedes OS), Windows 10 IoT Core, and more. Ubuntu 20.04 LTS 64-bit, the operating system used for this project, is available for download from the Ubuntu website, and is also available as an install option within Raspberry Pi Imager.

Since the release of the Raspberry Pi 1, the Raspberry Pi has been available in a number of model variants and circuit board formats. The Model B of each release is the most powerful variant, and is intended for desktop use. The Model A is a simpler and cheaper variant intended for embedded projects. The models B+ and A+ designate an improvement to the current release hardware. The Raspberry Pi Zero is a tiny, inexpensive variant without most of the external connectors, designed for low power, possibly battery powered, embedded projects. The Raspberry Pi Compute Module is a stripped down version of the Raspberry Pi without any external connectors. This model is aimed at industrial applications and fits into a standard DDR2 SODIMM connector.

1.3   Aims

1.3.1   Benchmark Performance

The main aim of this project is to benchmark the performance of an 8 node Raspberry Pi 4 Model B cluster using standard HPC benchmarks. These benchmarks include High Performance Linpack (HPL), HPC Challenge (HPCC) and High Performance Conjugate Gradients (HPCG).

A pure OpenMPI topology was benchmarked, together with a hybrid OpenMPI/OpenMP topology.

1.3.2   Performance Optimisations

Having determined a Baseline performance benchmark, opportunities for performance optimisations were investigated for a single core, single node and the whole cluster. Network optimisation was also investigated, and proved to be significant factor in overall cluster performance.

1.3.3   Investigate Gflops/Watt

The Green500 List ranks computer systems by energy efficiency, Gflops/Watt. In June 2020, ranking Number 1, the most energy-efficient system was the MN-3 by Preferred Networks in Japan, which achieved a record 21.1 Gigaflops/Watt [4]. Ranking 200 was Archer at the University of Edinburgh, which achieved 0.497 Gflops/Watt [4].

The final aim of this project was to investigate where the Aerin cluster might fare in relation to the Green500 List.

1.4   Project GitHub Repositories

The project code and benchmark results are hosted in the following GitHub project repository:

`https://github.com/johnduffymsc/picluster`

Detailed instructions for building the Aerin Cluster and running the benchmarks are included in the project repository wiki:

`https://github.com/johnduffymsc/picluster/wiki`

This dissertation LATEX and PDF files, and the Jupyter Notebook used to generate the plots, are hosted in the following GitHub repository:

`https://github.com/johnduffymsc/dissertation`

Chapter 2

Computer Architecture and HPC Benchmarks

2.1 Introduction

In his 1937 seminal paper "On Computable Numbers, with an Application to the Entscheidungsproblem" Alan Turing imagined a *univeral computing machine* capable of performing any conceivable mathematical operation. Turing proved that by formulating a mathematical problem as an algorithm, consisting of a sequence of numbers and operations on these numbers, on an infinitely long tape, and with operations to move the tape left and right, it was possible to mechanise the computation of any problem. These machines became known as Turing Machines.

Today's computers are Turing Machines. Turing's original sequence of numbers and operations are now referred to as the data and instructions contained within a computer program. The infinitely long tape is now referred to as a computer's memory. And the set of instructions which manipulate program data, and which also permit access to the full range of available memory (move the tape left and right), are referred to as a computer's *instruction set*.

High Performance Computing (HPC) is the solving of numerical problems which are beyond the capabilities of desktop and laptop computers in terms of the amount of data to be processed and the speed of computation required. For example, numerical weather forecasting (NWF) uses a grid of 3D positions to model a section of the Earth's atmosphere, and then solves partial differential equations at each of these points to produce forecasts. The processing performance and memory required to model such systems far exceeds that of even a

high-end desktop.

The UK Met Office use a number of grids to model global and and UK weather. The finest UK grid being a 1.5 km spaced 622 x 810 point inner grid, with a 4 km spaced 950 x 1025 outer grid, both with 70 vertical levels. To model the atmosphere on these grids the UK Met Office currently uses three Cray XC40 supercomputers, capable of 14 Petaflops (10^{15} Floating Point Operations per Second), and which contain 460,000 computer cores, 2 Petabytes of memory and 24 Petabytes of storage.

Clearly a single Cray XC40 used for NWF is a somewhat different beast than a single imaginary Turing Machine. Some of the differences obviously relate to the imaginary nature of the Turing Machine, with its infinitely long tape, and some to what it is possible to build within the limits of today's technology. The Cray XC40's 2 Petabytes of memory is large, but not infinite. But possibly the most important differences are architectural. Each Cray XC40 is a massively parallel supercomputer, made up of a large number of individual processing nodes. Each node has a large but finite amount of processing capacity and memory. The problem data and program instructions must be divided up and distributed amongst the nodes. The nodes must be able to communicate in an efficient manner. And opportunities for *parallel* and *concurrent* processing should be exploited to minimise processing time. Each of these differences is a requirement to map HPC workloads onto a real-world machine. And each of these difference introduces a degree of complexity.

Since the birth of electronic computing, there has always been a need to know long it will take for a computer to perform a particular task. This may be solely related to allocating computer time efficiently, or simply just wanting to know how long a program will take to run. Or, it may be commercially related; even a moderately sized single computer can be a large investment requiring the maximum performance possible for the purchase price. And more recently, the need to know how much processing power per unit of electricity a computer can achieve has become an important metric. This need for information is addressed by using a benchmark.

A benchmark is a standardised measure of performance. In computing terms this is a piece of software which performs a known task, and which tests a particular aspect(s) of computer performance. One aspect may be raw processing performance. High Performance Linpack (HPL) is one such benchmark, which produces a single measure of Floating Point Operations per Second (Flops) for a single, or more commonly, a cluster of computers. To address the complexity of design of modern supercomputers, as discussed above, a number of complementary benchmarks have been introduced, namely HPC Challenge (HPCC) and HP Conjugate Gradient (HPCG). HPC Challenge is a suite of benchmarks which measure processing performance, memory bandwidth, and network latency and bandwidth, to give a broad view of likely real-world application performance.

HP Conjugate Gradient is intended to measure the performance of modern HPC workloads.

To put benchmark results into context, and to extrapolate from the results where performance gains might be realised, it is necessary to have an understanding of the main components of a computer and the network connecting a cluster of computers. The following sections of this chapter describe these components and the network in more detail.

2.2 Computer Architecture

2.2.1 CPU

The CPU (Central Processing Unit) is the hardware that executes program instructions. Program data is loaded from main memory into CPU registers, the program instructions operate on the registers, and then the results are stored back in main memory. Registers may be general purpose registers, or have a specific use, such as floating point registers for fast floating point operations. A special purpose register called the Instruction Pointer points to the next instruction to be executed. A modern CPU will typically have multiple processing cores, each with its own set of registers.

2.2.2 Processes

A *process* is a running program executing on a CPU, or on a core of multi-core CPU. At any time each process has a *state*. This state includes the current contents of the registers and the Instruction Pointer. A multi-core CPU can run multiple processes simultaneously. A process may be a user program, such as a benchmark, or an operating system process.

The single-threaded benchmarks used in this project run as a single process, one process per CPU core.

2.2.3 Threads

A *thread*, sometimes referred to as a *lightweight process*, is the minimum amount of work that can be *scheduled* by a CPU. Scheduling is discussed shortly. A process may consist of multiple threads, each of which shares the address space of the process. Starting and stopping a thread is less expensive than starting and stopping a process. And because a process address space is shared between

threads, data sharing between threads can be less expensive and easier than other mechanisms of inter-process communication. It is for these reasons that multi-threaded programs are used. However, multi-threaded programs can be difficult to write, and data sharing between threads must be considered carefully to avoid *data races*, *livelocks* and *deadlocks.*

The multi-threaded benchmarks used in this project use OpenMP to parallelise computation. A single process is run on each node, with the benchmark work being distributed across cores by OpenMP using threads.

2.2.4   Context Switch

A *context switch* is the suspension of a running process and the starting, or resuming, of a different process. Context switching is implemented for a number of reasons; to share access to the CPU across multiple processes (*concurrency*), whenever a program issues a *system call* to request a service from the *kernel*, whenever the *kernel* receives an *interrupt* from a timer or peripheral device and requires access to the CPU, and to avoid wasting processor time when waiting for slow Input/Output (IO) operations.

Whenever a context switch takes place, the current *state* of the running process is saved to main memory, and the *state* of the new process is retrieved from main memory. This take time and wastes valuable clock cycles which could be used for computation. For this reason context switching should be minimised whenever possible through software design and process scheduling policies.

Linux uses a mechanism called vDSO (Virtual Dynamic Shared Object) to avoid a context switch whenever a *system call* is made which does not require an elevation of system privileges. However, on the Arm64 architecture this only includes the *clock_gettime()* and *gettimeofday()* system calls, so effectively all system calls involve a context switch.

As discussed in Chapter 5, each packet of network data sent between cluster nodes generates a network card *interrupt* on the receiving node. This *interrupt* requires *servicing* by the kernel, which involves a context switch from the benchmark process to a kernel process. And this takes a considerable amount of time and drastically affects benchmark performance. The Networking section of this chapter discusses measures to mitigate this performance penalty.

2.2.5   Concurrency and Parallelism

Concurrency and *Parallelism* are similar concepts and refer to a computer running multiple processes at the same time, or the illusion of this. A single core

CPU can only run a single process at a time. However, if the context switching between processes is fast enough, then this may result in the illusion of *concurrency*. This is sometimes referred to to as *time-slicing* or *time-sharing*. But this is not *parallelism*. *Parallelism* is the simultaneous running of multiple processes, which requires multiple cores or multiple computers.

A benchmark will typically be running the same process in parallel on each node, which each node operating on a different portion of the benchmark data.

2.2.6   Interrupts

Modern operating systems, such as Linux, are *event driven*. This means that instead of continuously looping over a list of actions, the operating system performs actions when events occur. Events generate *interrupts* which cause the operating system to pause the running process and *service* the interrupt. Interrupts may be hardware interrupts, such as the interrupt generated by a network card upon receipt of a data packet, or may be generated by software, *software interrupts*.

There are a number of sources of interrupts, but the main source of interrupts for the operating system is the system clock. The clock of the BCM2711 ticks at a frequency of 1.5 GHz, and at predetermined counts of the system clock the operating system performs predetermined actions. Other sources of interrupts include user input from the keyboard, hard disk IO and environmental sensors.

Interrupts generated by network activity, and the effects of this on benchmark performance, are discussed shortly.

2.2.7   Kernel Preemption Model

The *kernel preemption model* is related to process and thread scheduling. Scheduling can either be *preemptive* or *non-preemptive*. A pre-emptive scheduler can interrupt a running thread or process, based upon a *scheduling policy*, to enable a different thread or process to run. Scheduling policies include *First-Come First-Served*, *Round Robin*, and *Priority-Driven Scheduling*. Non-preemptive scheduling does not interrupt running threads or processes. The kernel preemption model is a kernel configuration option set during kernel compilation.

Linux supports three kernel preemption models, *preemptive*, *voluntary preemption* and *non-preemptive*. The *preemptive* model is used where *low latency* is the primary requirement, such as for audio recording. This model prioritises latency over processing throughput. The *non-preemptive* model priorities processing throughput over latency, and it used where maximum processing power

is required. The *voluntary preemption* model is a compromise between the other two models, and is typically used for desktop systems where the user requires responsive mouse and keyboard input and also no excessive reduction in performance.

As previously stated, the kernel preemption model is a kernel configuration option. Quoting the *help* associated with the `CONFIG_PREEMPT_NONE` kernel configuration option:

"This is the traditional Linux preemption model, geared towards throughput. It will still provide good latencies most of the time, but there are no guarantees and occasional longer delays are possible. Select this option if you are building a kernel for a server or scientific/computation system, or if you want to maximise the raw processing power of the kernel, irrespective of scheduling latencies."

The preemption model of default kernel installed with Ubuntu 20.04 LTS 64-bit is *voluntary preemption*. The recompilation of the Linux kernel as a *non-preemptive* kernel to maximise raw benchmark processing power is discussed in Chapter 5.

2.2.8   Main Memory

Main memory is the largest component of the memory system of a computer. On desktop, laptop and larger computers, the memory chips usually reside on small circuit boards that fit into sockets on the computer mainboard. These can be upgraded in size by the user. On some smaller computers, such as the Raspberry Pi 4, the memory chip is soldered onto the computer circuit board and is not upgradable.

Each memory location contains a byte of data, where a byte is 8 binary bits. Bytes are stored sequentially at an *addresses*, which is a binary number in the range 0 up to the maximum address supported by the system. The maximum address typically aligns with the register size. For example, 64-bit computer has 64-bit registers which can hold an address in the range 0 to 64^2. This requires a 64-bit physical *address bus* to address each byte of memory. Practical considerations sometimes limit the size of the address bus. The Raspberry Pi 4 is a 64-bit computer but has a 48-bit physical address bus.

Computers systems without an operating system, such as embedded systems, permit direct access to main memory from software. In this case there is a direct mapping between the memory address within a computer program and the physical address in main memory. Most operating systems present an abstracted view of main memory to each program running on the system. This is called *virtual memory*.
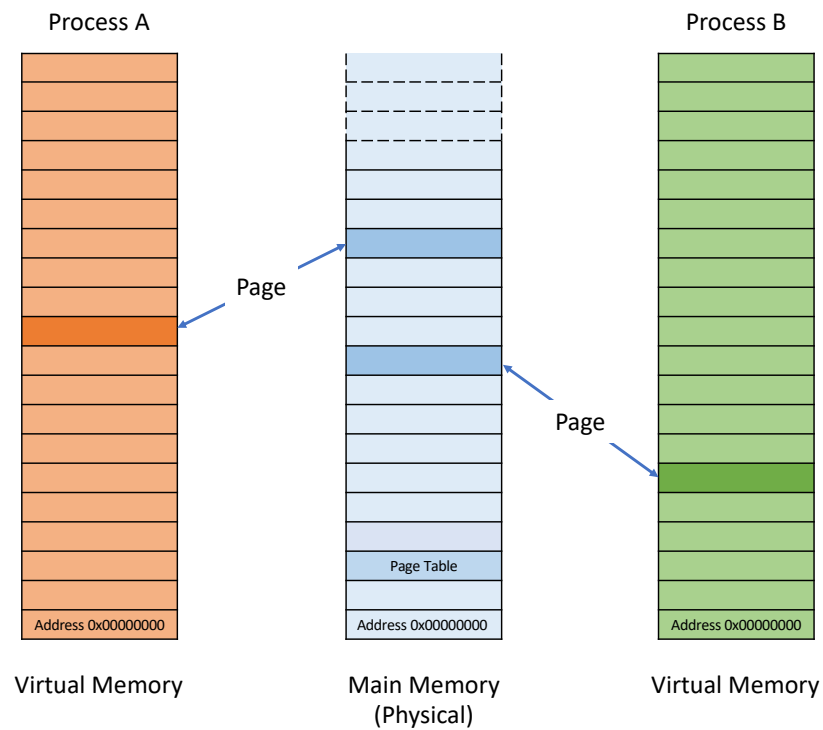
Figure 2.1: **Virtual Memory**. Each process has a *virtual address* space mapped to main memory in *pages* by a *page table* which resides in main memory. A smaller page table called the *Translation Lookaside Buffer* (TLB) is a *cache* in close proximity to each core. The TLB enables fast lookup of physical page addresses without resorting to a slower lookup in the main memory page table.

2.2.9   Virtual Memory

Virtual memory is the abstracted view of main memory presented to a running program by the operating system. Virtual memory requires both hardware support, through the Memory Management Unit (MMU), and software support by the operating system. Contiguous regions of virtual memory are organised into *pages*, typically 4 KB in size. Each page of virtual memory maps to a page of physical memory through a *page table* which resides in main memory. A smaller page table called the *Translation Lookaside Buffer* (TLB), which is a *cache* in close proximity to each processing core, is discussed later.

There are a number of benefits of implementing virtual memory. One is to permit the use of a smaller amount of physical memory than is actually addressable. In this case, pages currently in use reside in main memory, and pages no longer required are *swapped* to permanent storage to make space for new pages. This illusion of a full amount of addressable main memory is transparent to the user. But the *paging* between main memory and permanent storage is slow, and is therefore not used in HPC applications.

Possibly the most important benefit of using virtual memory is to implement a protection mechanism called *process isolation*. Each running program, or *process*, executes in its own private, virtual address space. This means that it is not possible for a process to overwrite memory in the address space of another process, possibly due a bug in a program. This process isolation in managed by the operating system using virtual memory. It is possible for multiple processes to communicate through *shared memory*, where each process can read and write to the same block of memory, but this requires programs to be specifically written to make use of this mechanism.

2.2.10   Caches

If we imagine Turing's infinitely long tape and the inertia that must be overcome to move such a tape left and right, it would not be too much of a leap of the imagination to propose copying some sequential part of the tape onto a finite, lighter tape which could be moved left and right faster. Then if the data required for the current part of our computation was contained within this faster tape, the computation would be conducted faster. The contents of the finite tape would be refreshed with data from the infinite tape as required, which may be expensive in terms of time. And if the speed at which we can perform operations on the lighter tape began to outpace the speed of movement of the tape, we might propose copying some of the data onto an even shorter, even faster tape.

If we replace speed of tape movement with speed of memory access, then this

imaginary situation is analogous to the layering of memory within a real computer system. Main memory access is slow compared to processor computing speed, so main memory is copied into smaller, faster *caches* colocated on the same silicon die as the processing cores. Each level of cache closer to a processing core is smaller but faster than the previous, with the cache closest to the processing core being referred to as Level 1 (L1) cache. A processor may have L1, L2 and L3 caches, the outer cache possibly being shared between a number of processing cores. As we shall discuss later in this chapter, the speed at which program data flows from main memory through the caches to the processing cores is critical for application performance, and considerable care is taken to minimise *cache misses* which require a *cache refresh* from main memory.

There are typically three types of cache, the *instruction cache*, the *data cache* and the *Translation Lookaside Buffer*, each of which may be in L1, L2 or L3 proximity to a processing core, or in the case of the TLB proximity to the Memory Management Unit (MMU) . A *unified* cache is a combined instruction and data cache. The *instruction cache* holds program instructions laid out in memory in close proximity to the instruction currently being executed. Similarly, the *data cache* holds program data laid out in memory in close proximity to the data currently in use. In both cases, if the next instruction or next piece of data required is found in a cache, a *cache hit*, this information can be accessed very quickly. If the information in not in a cache, a *cache miss*, then an expensive *cache refresh* from main memory is required. The *Translation Lookaside Buffer* (TLB) is a small page table which enables fast lookup of required pages in main memory. If the address of required page is not in a TLB then an expensive main memory page table lookup is required.

Data (program instructions, program data, or page addresses) in a cache are a copy of data in main memory. In addition to the data itself, the location of the data in main memory must also be stored in the cache. This additional information increases the physical size of the cache and is expensive in terms transistor count and silicon surface area. To address this problem, rather than storing the location of each element of data, the data in the cache is organised into *cache lines*. This reduces the number of locations to store, and also has the additional benefit that a request for a new piece of data will also bring into the cache data in close proximity which fit into a cache line.

A cache may be *fully-associative* or *set-associative*. In a *fully-associative* cache every memory address can be stored in all cache entries, i.e. a cache entry can point to any memory address. A *set-associative* cache restricts memory addresses to a set of entries in the cache, i.e. certain cache entries can only point to certain memory addresses. The use of a *fully-associative cache* versus a *set-associative* cache is a compromise between a high probability of a cache hit but slower lookup, the *fully-associative cache*, and a lower probability of a *cache hit* but faster lookup, the *set-associative cache*.

The BCM2711 contains the following caches, each with 64 byte cache lines:

- 48 KB 3-way set-associative L1 instruction cache per core

- 32 KB 2-way set-associative L1 data cache per core

- 1 MB 16-way set-associative shared L2 unified cache

The BCM2711 MMU contains the following caches:

- 48-entry fully-associative L1 instruction TLB

- 32-entry fully-associative L1 data TLB

- 4-way set-associative 1024-entry L2 TLB in each processor

The appropriate layout of data in memory, in either *row-major* or *column-major* ordering, and subsequent program access pattern, via the *data cache*, has a major impact on program performance. If the data layout matches the access pattern, then each *cache refresh* will fill a *cache line* with the data required and also the data likely to be required next in sequential order. A *cache refresh* will only be required once the entire *cache line* has been used. If the data layout does not match the access pattern, then data will be moved in and out of the *data cache* without being used, and this will result in an unnecessarily high number of expensive *cache misses*.

2.3 Networking

Most modern computer network technologies are based on the idea of *packet switching*. In a *packet switched* network data is split up into *payload* chunks which are encapsulated with *headers* into network *packets*. The *headers* include addressing and network protocol information. Packets from many sources may be transmitted simultaneously over the network and *routed* to their destination based upon the information in the packet headers. Compared to a *circuit switched* network, where network resources are dedicated to a single connection (circuit) for a period of time, *packet switched* networks provide improved network efficiency, redundancy and load balancing.

The speed and efficiency of the network connecting a cluster of computers has a major effect on both HPC application and benchmark performance. The most common HPC network technology is called InfiniBand. InfiniBand provides a high throughput and low latency interconnect between cluster nodes. But InfiniBand requires dedicated hardware which in not normally found in

commodity computers and network switches. Ethernet is the de-facto standard for most computer networks, and Ethernet ports can usually be found on most commodity computers. The Raspberry Pi 4 used for this project has a Gigabit Ethernet port which can transmit/receive data at 1 Gigabit/second. InfiniBand and Ethernet are both *packet switched* technologies.

It is not only the speed at which a network transmits that affects benchmark performance. How the data is processed at the receiving node also plays a significant role, especially in a multi-core node where each core may be running a benchmark process which consumes data.

The following sections describe methodologies for improving network efficiency, and network packet processing at the receiving node which improves data locality. Both of these improve benchmark performance.

2.3.1   MTU

The Maximum Transmission Unit (MTU) is the maximum size of a network packet payload. Data to be transmitted which is larger than the MTU is *fragmented* into multiple packets. The default MTU size for Ethernet, and therefore most local area networks (LANs), is 1500 bytes. This is based upon the maximum frame size of a standard Ethernet connection which is 1518 bytes, the additional 18 bytes being the Ethernet header. Obviously, most data is much larger than 1500 bytes, so fragmenting data into 1500 byte chunks is quite normal.

Smaller packet sizes improve network latency as seen by multiple connections. This is because each node receives data regularly, and large packets are not blocking the network. However, there is overhead associated with this. Multiple packets destined for the same computer require the same header information to be included with each packet. A larger packet size improves network efficiency by reducing overhead, but potentially at the cost of reduced latency.

A Jumbo Frame is any Ethernet MTU greater than 1500 bytes. The is no standardised maximum Jumbo Frame size, but the norm is 9000 bytes. In Chapter 5, an investigation is conducted in to the effect on benchmark performance of increasing the Aerin Cluster network MTU from 1500 to 9000 bytes.

2.3.2   Interrupt Coalescing

Whenever a data packet is received by a Network Interface Card (NIC), the packet is placed in a buffer to be processed by the kernel. The NIC informs the kernel of the receipt of the packet via a hardware interrupt, which results

in a context switch into a kernel process. The greater the number of packets, the greater the number of context switches, and the less CPU time that is utilised performing computational operations. This effect was observed during experiments, as discussed in Chapter 5, and has a negative effect on benchmark performance.

Interrupt Coalescing is the delaying of notifying the kernel of a packet receipt until a number of packets have been received. Received packets are placed in a queue and a hardware interrupt is subsequently raised to notify the kernel. The reduction of in the number of interrupts results in a reduced number of context switches, which potentially has a positive effect on benchmark performance. *Interrupt Coalescing* requires NIC hardware support and also network driver support, and is configured using the Linux `ethtool` command.

2.3.3   Receive Side Scaling

By default, the interrupts generated by network packets arriving at a network interface are serviced by a single core of a multi-core CPU. This creates an unbalanced workload across the CPU cores, and may result in the benchmark process stalling on the affected core. This may have a detrimental effect on the processing performance of the CPU as a whole.

With a NIC that supports multiple receive queues, it is possible to assign a queue to multiple cores and to configure the NIC to spread interrupt handling across these cores. This is called *Receive Side Scaling* (RSS). The number of interrupts generated is not reduced, but it does prevent a single core from being overloaded. This has been shown to have a positive effect on network packet processing and improve overall CPU performance.

On Linux, RSS is configured through the `/proc` and `/sys` filesystems for NICs which support multiple receive queues.

RSS is a building block for *Receive Flow Steering* which is discussed shortly.

2.3.4   Receive Packet Steering

Not all network interface cards support multiple receive queues, or the network driver may not have implemented this functionality. *Receive Packet Steering* (RPS) is a software implementation of RSS, which can work with a single receive queue.

RPS is the software equivalent of RSS as a building block for *Receive Flow Steering*.

2.3.5   Receive Flow Steering

Receive Flow Steering (RFS) has the potential to improve benchmark performance by improving data locality.

Building upon the distribution of network interrupt servicing across multiple cores, RFS add a layer of packet destination inspection, and routes packets directly to the core requiring the packet. For example, a 4-core CPU may be running 4 HPL `xhpl` processes, one on each core. Each core consumes data for its particular `xhpl` process. By implementing RSS/RPS, each core may receive each packet, and may then need to pass it on to the core that requires it. RFS *remembers* the destination of previous packets matching certain criteria, and then forwards subsequent packets matching the same criteria directly to the same destination core. This improves data locality.

On Linux, RFS is configured through `/proc` and `/sys` file systems.

2.4   ARM Architecture

Ever since the introduction of the ARMv8-A architecture in 2011, Arm processors have become an increasingly popular choice for High Performance Computing (HPC) due to improvements in the ARMv8 instruction set specifically targeting HPC combined with low power requirements. The 415 Petaflops Fugaku supercomputer, based on the Fujitsu AX64 Arm-based processor, topped the June 2020 Top500 List and also the November 2019 Green500 List.

Arm processors are based upon RISC (Reduced Instruction Set Computer) principles with a simple Load/Store architecture. Data is loaded from main memory into processor registers, computations are performed using fixed-length instructions, and the results are then stored back in main memory. This compares with CISC (Complicated Instruction Set Computer) architectures with complicated memory addressing and computation modes with variable length instructions. The simplicity of design, which directly translates to a low transistor count, results in high performance combined with low power requirements.

The ARMv8-A in introduced the first Arm 64-bit architecture.

armv8.1...

armv8.2...

SVE...

Fugaku chip...

Fujitsu chip...

2.5 Benchmarks

2.5.1 Landscape

High Performance Linpack (HPL) is the industry standard HPC benchmark and has been for since 1993. It is used by the Top500 and Green500 lists to rank supercomputers in terms of raw performance and performance per Watt, respectively. However, it has been criticised for producing a single number, and not being a true measure of real-world application performance. This has led to the creation of complementary benchmarks, namely HPC Challenge (HPCC) and High Performance Conjugate Gradients (HPCG). These benchmarks measure whole system performance, including processing power, memory bandwidth, and network speed and latency, in relation to standard HPC algorithms such as FFT and CG.

2.5.2 High Performance Linpack (HPL)

HPL did not begin life as a supercomputer benchmark. LINPACK is a software package for solving Linear Algebra problems. And in 1979 the "LINPACK Report" appeared as an appendix to the LINPACK User Manual. It listed the performance of 23 commonly used computers of the time when solving a matrix problem of size 100. The intention was that users could use this data to extrapolate the execution time of their matrix problems.

As technology progressed, LINPACK evolved through LINPACK 100, LINPACK 1000 to HPLinpack, developed for use on parallel computers. High Performance Linpack (HPL) is an implementation of HPLinpack.

In 1993 the Top500 List was created to rank the performance of supercomputers and HPL was used, and still is used, to measure performance and create the rankings.

HPL solves a dense system of equations of the form:

$$A\mathbf{x} = \mathbf{b}$$

HPL generates random data for a problem size N. It then solves the problem using LU decomposition and partial row pivoting.

HPL requires an implementation of MPI (Message Passing Interface) and a BLAS (Basic Linear Algebra Subroutines) library to be installed. For this project, OpenMPI was the MPI implementation used, and OpenBLAS and BLIS were the BLAS libraries used. Both BLAS libraries were used in the single-threaded serial version and also the multi-threaded OpenMP version.

In HPL terminology, R_{peak} is the theoretical maximum performance. And R_{max} is the maximum achieved performance, which will normally be observed using the maximum problem size N_{max}.

Determining Input Parameters

The main parameters which affect benchmark results are the block size NB, the problem size N, and the processor grid dimensions P and Q.

The block size NB is used for two purposes. Firstly, to "block" the problem size matrix of dimension N x N into sub-matrices of dimension NB x NB. This is described in more detail in the Section ??. And secondly, as the message size (or multiples of) for distributing data between cluster nodes.

The optimum size for NB is related to the BLAS library `dgemm` *kernel* block size, which is related to CPU register and L1, L2, and L3 (when available) cache sizes. But this is not easily determined as a simple multiple of the *kernel* block size. Some experimentation is required to determine the optimum size for NB.

HPL Frequently Asked Questions suggests NB should be in the range 32 to 256. A smaller size is better for data distribution latency, but may result in data not being available in sufficiently large chunks to be processed efficiently. Too high a value may result in data starvation while nodes wait for data due to network latency.

For this project, HPL benchmarks were run with NB in the range 32 to 256 in order to determine the optimum size for the Aerin Cluster, and for each BLAS library in serial and OpenMP versions.

For maximum data processing efficiency, and therefore optimum benchmark performance, the problem size N should be as large as possible. This optimises the cluster processing/communications ratio. Optimum efficiency is achieved when the problem size utilises 100% of memory. But this is never actually achievable, since the operating system and benchmark software require memory to run. HPL Frequently Asked Questions suggests 80% of total available memory as a good starting point, and this value was used for this project.

For optimum benchmark performance the problem size N needs to be an integer

multiple of the block size NB. This ensures every NB x NB sub-matrix is a full sub-matrix of the N x N problem size, i.e. there are no partially full NB x NB sub-matrices at N x N matrix boundaries.

For each value of NB, the following formula is used to determine the problem size N, taking into account 80% memory usage:

$$N = \left[\left(0.8 \sqrt{\frac{\text{Memory in GB} \times 1024^3}{8}} \right) \div NB \right] \times NB$$

The division by 8 in the inner parenthesis is the size in bytes of a double precision floating point number.

The online tool HPL Calculator by Mohammad Sindi automates the process of calculating the problem size N for block sizes NB in the range 96 to 256, and for memory usage 80% to 100%.

The values of N determined using HPL Calculator were cross-checked with the formula above.

The processor grid dimensions P and Q represent a P x Q grid of processor cores. For example, the Aerin cluster has 8 nodes, each with 4 cores, giving a total of 32 processor cores. These core can be organised in compute grids of 1 x 32, 2 x 16 and 4 x 8.

The HPL algorithm favours P x Q grids as square as possible, i.e. with P almost equal to Q, but with P smaller than Q. So, for a single node with 4 cores, a processor grid of 1 x 4 gives better benchmark performance than 2 x 2.

If the Aerin Cluster used a high speed interconnect between nodes, such as InfiniBand, as used on large HPC clusters, maximum performance would be expected to be achieved using a processor grid of 4 x 8. This is the "squarest" possible P x Q grid using 32 cores whilst maintaining P less than Q. However, as noted in HPL Frequently Asked Questions, Ethernet is not a high speed interconnect. An Ethernet network is simplistically a single wire connecting the nodes, with each node competing (using random transmission times) for access to the wire to transmit data. This physical limitation reduces potential maximum cluster performance, and the maximum achievable performance is seen using a flatter P x Q grid. This proved to be the case, and maximum cluster performance was observed using a processor grid of 2 x 16 for all 8 nodes. This phenomena was also observed using when using less than 8 nodes.

2.5.3   HPC Challenge (HPCC)

HPCC is a suite of benchmarks which test different aspects of cluster performance. These benchmarks include tests for processing performance, memory bandwidth, and network bandwidth and latency. HPCC is intended to give a broader view of cluster performance than HPL alone, which should reflect real-world application performance more closely. HPCC includes HPL as one of the suite of benchmarks.

The HPCC suite consists of the following 7 benchmarks, where *single* indicates the benchmark is run a single randomly selected node, *star* indicates the benchmark in run independently on all nodes, and *global* indicates the benchmark is run using all nodes in a coordinated manner.

HPL

HPL is a *global* benchmark which solves a dense system of linear equations.

DGEMM

The DGEMM benchmark tests double precision matrix-matrix multiplication performance in both *single* and *star* modes.

STREAM

The STREAM benchmark tests memory bandwidth, to and from memory, in both *single* and *star* modes.

PTRANS

PTRANS, Parallel Matrix Transpose, is a *global* benchmark which tests system performance in transposing a large matrix.

RandomAccess

The RandomAccess benchmark tests the performance of random updates to a large table in memory, in *single*, *star*, and *global* modes.

FFT

FFT tests the Fast Fourier Transform performance of a large vector, in *single*, *star*, and *global* modes.

Network Bandwidth and Latency

This benchmark measures network/communications bandwidth and latency in *global* mode.

2.5.4   High Performance Conjugate Gradients (HPCG)

HPCG is intended to be complementary to HPL, and to incentivise hardware manufacturers to improve computer architectures for modern HPC workloads.

Quoting the Super Computing 2019 HPCG Handout:

"The HPC Conjugate Gradient (HPCG) benchmark uses a preconditioned conjugate gradient (PCG) algorithm to measure the performance of HPC platforms with respect to frequently observed, yet challenging, patterns of execution, memory access, and global communication."

"The PCG implementation uses a regular 27-point stencil discretixation in 3 dimensions of an elliptic partial differential equation (PDE) with zero Dirichlet boundary condition. The 3-D domain is scaled to fill a 3-D virtual process grid of all available MPI process ranks. The CG iteration includes a local and symmetric Gauss-Seidel preconditioner, which computes a forward and a back solve with a triangular matrix. All of these features combined allow HPCG to deliver a more accurate performance metric for modern HPC architectures."

Chapter 3

Mathematical Background of HPC Benchmarks

This chapter initially describes the *matrix-matrix multiplication* operation, before describing the mathematical background to the High Performance Linpack (HPL) and High Performance Conjugate Gradients (HPCG) benchmarks.

The HPC Challenge (HPCC) benchmark suite is not described because the suite includes HPL and DGEMM which are already covered in this chapter.

3.1 Matrix-Matrix Multiplication

The *matrix-matrix multiplication* operation is of fundamental importance in High Performance Computing. As indicated in Figure 3.1, when running HPL on all 8 nodes of the Aerin Cluster, 87.26% of the run time is spent in the `HPL_dgemm` function which calls the BLAS `dgemm` function. The `dgemm` function name is derived from **d**ouble precision **ge**neral **m**atrix **m**ultiplication. Running on single node, without any networking overhead, this increases to 96+%.

The naive computational complexity of $\mathbf{C} = \mathbf{AB}$, where $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C} \in \mathbf{R}^{n \times n}$, is $\mathcal{O}(n^3)$. Each *row-column* dot product is $\mathcal{O}(n)$, requiring n multiplications and $n - 1$ additions. There are n of these dot products per row of $\mathbf{A}$ and each column of $\mathbf{B}$, and $\mathbf{A}$ has n rows. In the more general case of $\mathbf{A} \in \mathbf{R}^{m \times k}$ and $\mathbf{B} \in \mathbf{R}^{k \times n}$, the complexity is $\mathcal{O}(mkn)$.

In 1969, Volker Strassen [5] proved the computational complexity can be reduced to $\mathcal{O}(N^{\log_2 7 + o(1)}) \approx \mathcal{O}(N^{2.8074})$. This is achieved by subdividing each matrix

Figure 3.1: Output of `sudo perf report` generated from recording the events of an `xhpl` process using `sudo perf record -p 9020 -g -- sleep 30`, where 9020 is the `xhpl` process identifier. As indicated, 87.26% of the process time is spent in the `HPL_dgemm` function, which calls the BLAS `dgemm` function.

into sub-matrices of dimension 2×2. By doing so, and introducing intermediate matrices, it is possible to reduce the number of multiplications required from 8 to 7 for each sub-matrix multiplication. This is called the *Strassen Algorithm*.

In 1990, Don Coppersmith and Shmuel Winograd [6] further reduced the complexity to $\mathcal{O}(n^{2.375477})$. The *Coppersmith-Winograd Algorithm* was further improved by Virginia Vassilevska Williams [7] in 2012, and most recently in 2014 by François Le Gallf [8], which reduced the complexity to $\mathcal{O}(n^{2.3728639})$.

However, the reductions in complexity beyond the *Strassen Algorithm* also add implementation complexity. The benefits may not outweigh the computational overhead unless the matrices are very large. For this reason the naive and *Strassen Algorithm* are usually implemented in practice.

The $\mathcal{O}(n^3)$ complexity of *matrix-matrix multiplication* plays a central role in HPC benchmarks because, compared to other matrix operations, it has the

highest $\frac{\text{computation}}{\text{memory movement}}$ ratio. This tests data movement from main memory, through the L1/L2/L3 caches, to the processor registers for computational operations. In a computer cluster it also tests network connectivity. It is for this reason that BLAS libraries usually indicate performance, and comparison with alternative libraries, with plots of `dgemm` Gflops versus matrix size.

3.2 High Performance Linpack (HPL)

The High Performance Linpack (HPL) Benchmark [9] measures the time taken to solve a dense system of linear of equations, and from this timing derives a performance metric stated in Gflops.

The system of equations is of the form:

$$\mathbf{Ax} = \mathbf{b}, \text{ where } \mathbf{A} \in \mathbf{R}^{n \times n} \text{ and } \mathbf{x}, \mathbf{b} \in \mathbf{R}^n \tag{3.1}$$

Random data is generated to populate $\mathbf{A}$ and $\mathbf{b}$ for each benchmark run.

The system of equation is solved using *LU factorisation with row partial pivoting*.

Before discussing the *LU factorisation* in more detail it is worth noting that if the HPL algorithm is amended (as is permitted for a submission to the TOP500 List), then the *Strassen* and *Coppersmith-Winograd* algorithms discussed in the previous section for *matrix-matrix multiplication* are excluded from being used. The amended algorithm must adhere to the floating point operation count for *LU factorisation with row partial pivoting*, $2/3n^3 + \mathcal{O}(n^2)$ [9]. This is to ensure uniformity across submissions.

3.2.1 LU Factorisation

LU Factorisation is the splitting of a matrix $\mathbf{A}$ into an upper-triangular matrix $\mathbf{U}$, and a lower-triangular matrix $\mathbf{L}$, such that:

$$\mathbf{A} = \mathbf{LU} \tag{3.2}$$

The original system of equations (3.1) can then be restated as:

$$\mathbf{LUx} = \mathbf{b} \tag{3.3}$$

This can be reformulated as two equations:

$$\mathbf{U}\mathbf{x} = \mathbf{y} \tag{3.4}$$

$$\mathbf{L}\mathbf{y} = \mathbf{b} \tag{3.5}$$

Because $\mathbf{L}$ is in lower-triangular form, (3.5) is is easily solved using forward substitution to yield $\mathbf{y}$. Subsequently, because $\mathbf{U}$ is in upper-triangular form, (3.4) is easily solved using backward substitution to yield $\mathbf{x}$. This solves the original system (3.1).

It remains to show how $\mathbf{A}$ is factored into $\mathbf{L}$ and $\mathbf{U}$.

The matrix $\mathbf{A}$ can be factored into an upper-diagonal matrix $\mathbf{U}$ in a similar manner to *Gaussian Elimination*. Multiples of each row are subtracted from subsequent rows to produce *zeros* below the matrix diagonal. Each of these subtraction operations can be considered a matrix operation $\mathbf{L}_i$ on $\mathbf{A}$. For example:

$$
\begin{aligned}
\mathbf{A} &= \begin{pmatrix} x & x & x & x \\ x & x & x & x \\ x & x & x & x \\ x & x & x & x \end{pmatrix} \\
\mathbf{L}_1\mathbf{A} &= \begin{pmatrix} x & x & x & x \\ 0 & x & x & x \\ 0 & x & x & x \\ 0 & x & x & x \end{pmatrix} \\
\mathbf{L}_2\mathbf{L}_1\mathbf{A} &= \begin{pmatrix} x & x & x & x \\ 0 & x & x & x \\ 0 & 0 & x & x \\ 0 & 0 & x & x \end{pmatrix} \\
\mathbf{L}_3\mathbf{L}_2\mathbf{L}_1\mathbf{A} &= \begin{pmatrix} x & x & x & x \\ 0 & x & x & x \\ 0 & 0 & x & x \\ 0 & 0 & 0 & x \end{pmatrix} \\
\mathbf{L}_3\mathbf{L}_2\mathbf{L}_1\mathbf{A} &= \mathbf{U}
\end{aligned}
\tag{3.6}
$$

And so,

$$\mathbf{A} = \mathbf{L}_1^{-1}\mathbf{L}_2^{-1}\mathbf{L}_3^{-1}\mathbf{U} = \mathbf{L}\mathbf{U} \tag{3.7}$$

37

The product $\mathbf{L} = \mathbf{L}_1^{-1}\mathbf{L}_2^{-1}\mathbf{L}_3^{-1}$ is, surprisingly, easily determined because each inverse $\mathbf{L}_i^{-1}$ is equal to $\mathbf{L}_i$ with its values below the diagonal negated, and the columns of the inverses then map directly into the same locations in $\mathbf{L}$. "Trefethen and Bau" [10] refers to these as two *Stokes of Luck*, which are explained by the sparsity nature of the matrices $\mathbf{L}_i$.

3.2.2   Row Partial Pivoting

Row partial pivoting is the swapping of rows during the factorisation of $\mathbf{A}$ into the upper-diagonal matrix $\mathbf{U}$. At each step of the factorisation, rows are swapped such that the *pivot* row has the highest numerical value at the *pivot point*. The reason for doing this is twofold. Firstly, if the value of the pivot point is *zero*, then the factorisation fails. And secondly, to improve numerical stability [10].

For example, initially a_{11} is the *pivot point*. However a_{41} has the highest numerical value in the *pivot column*. So, row_1 and row_4 are swapped (pivoted) by the application of matrix P_1:

$$\mathbf{A} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 5 & 8 & 11 \\ 4 & 10 & 14 & 16 \\ 4 & 8 & 12 & 20 \end{pmatrix}$$
$$\mathbf{P}_1\mathbf{A} = \begin{pmatrix} 4 & 8 & 12 & 20 \\ 2 & 5 & 8 & 11 \\ 4 & 10 & 14 & 16 \\ 1 & 2 & 3 & 4 \end{pmatrix}$$

(3.8)

The factorisation matrix $\mathbf{L}_1$ is then applied as before:

$$\mathbf{L}_1\mathbf{P}_1\mathbf{A} = \begin{pmatrix} 4 & 8 & 12 & 14 \\ 0 & 1 & 2 & 1 \\ 0 & 2 & 2 & -4 \\ 0 & 0 & 0 & -1 \end{pmatrix}$$

(3.9)

The final factorisation is of the form:

$$\mathbf{L}_3\mathbf{P}_3\mathbf{L}_2\mathbf{P}_2\mathbf{L}_1\mathbf{P}_1\mathbf{A} = \mathbf{U}$$

(3.10)

By what "Trefethen and Bau" [10] refer to as a third *Stroke of Luck*, (3.10) can be restated, and easily determined, as:

$$\mathbf{PA} = \mathbf{LU} \tag{3.11}$$

where

$$\mathbf{L} = (\mathbf{L}_3^{'}\mathbf{L}_2^{'}\mathbf{L}_1^{'})^{-1}, \text{ and } \mathbf{P} = \mathbf{P}_3\mathbf{P}_2\mathbf{P}_1 \tag{3.12}$$

and where $\mathbf{L}_i^{'} = \mathbf{L}_i$ with the entries below the diagonal permuted.

3.2.3   The HPL Algorithm

As previously stated, HPL solves a system of equations of the form:

$$\mathbf{Ax} = \mathbf{b}, \text{ where } \mathbf{A} \in \mathbf{R}^{n \times n} \text{ and } \mathbf{x}, \mathbf{b} \in \mathbf{R}^n \tag{3.13}$$

The system is solved by distributing the data over a two dimensional grid of processors, $P \times Q$, on a cluster of *distributed memory* computers [9]. HPL requires an implementation of the Message Passing Interface (MPI) for the data distribution. The Aerin Cluster uses the OpenMPI implementation. In a pure MPI topology, each grid element represents a processor core of a CPU. In a hybrid OpenMPI/OpenMP topology, each grid element represents a cluster node, and OpenMP distributes data between the cores of the node.

HPL partitions the data into *blocks* of dimension $NB \times NB$, where NB is referred to as the *block size*. The data blocks are distributed cyclically between the elements of the processor grid for load balancing and scalability.

For optimum performance the problem dimension n should be selected to utilise all available memory, and be integer multiple of the block size NB. Equation (3.14) can be used to determine the problem dimension N for a given block size NB and a required total memory usage.

$$N = \left[\left(memory\_percent \sqrt{\frac{memory\_in\_gb \times 1024^3}{8}} \right) \div NB \right] \times NB \tag{3.14}$$

The division by 8 in the square root is the size in bytes of a double precision floating point number.

The LU factorisation is logically partitioned using the same block size NB used for data distribution.

3.3 High Performance Conjugate Gradients (HPCG)

Like HPL, the HPCG benchmark, developed by Dongarra, Heroux and Luszczek [11], measures the time taken to solve a system of linear equations. And from this timing derives a performance metric stated in Gflops. The system of equations is also of the form:

$$\mathbf{Ax} = \mathbf{b}, \text{ where } \mathbf{A} \in \mathbf{R}^{n \times n} \text{ and } \mathbf{x}, \mathbf{b} \in \mathbf{R}^n \qquad (3.15)$$

This is where the similarity with HPL ends.

The HPCG system of equations is a sparse system resulting from a discretised three dimensional partial differential equation model. The benchmark computes *preconditioned Conjugate Gradient* iterations for the sparse system.

The motivation for developing HPCG is that HPL no longer accurately indicates the likely performance of computer systems running modern HPC workloads. Historically, solving dense linear systems was the mainstay of HPC computing. This workload was predominantly the *matrix-matrix multiplication* of locally aligned data, which was closely correlated with HPL benchmark preformance. This data access pattern, which represents a high computation to memory access ratio, is termed *Type 2* by Dongarra, Heroux and Luszczek. HPCG address the need to measure the performance of *distributed memory* computer systems when running a workload with a low computation to memory access pattern, which Dongarra, Heroux and Luszczek refer to as *Type 1*. This is typified by the solving of sparse systems resulting from partial differential equations.

HPCG uses the Conjugate Gradient method with a symmetric Gauss-Seidel preconditioner to solve the Poisson differential equation on a three dimensional grid discretised with a 27-point stencil [11].

The Conjugate Gradient Method is a method for solving a system of equations with a *symmetric positive-definite* matrix. When implemented as an iterative scheme, it solves a $n \times n$ system in at most n iterations. The method will be introduced through a discussion of *Quadratic Forms*, the *Steepest Descent Method* and the *Method of Conjugate Directions*. This broadly follows the description of the method by Shewchuck [12].

And finally... discuss the blocking/implementation of HPCG... and why CG is used.

3.3.1 Quadratic Forms and the Relationship to $\mathbf{Ax} = \mathbf{b}$

A *quadratic form* is a quadratic function of a vector $\mathbf{x}$, whose return value is a scalar. It is of the form:

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^{\mathbf{T}}\mathbf{Ax} - \mathbf{x}^{\mathbf{T}}\mathbf{b} + c \tag{3.16}$$

where $\mathbf{A}$ is a matrix, $\mathbf{b}$ is a vector and c is a scalar.

Consider the derivatives of (3.16):

$$\nabla f(\mathbf{x}) = \mathbf{Ax} - \mathbf{b}$$
$$\nabla^2 f(\mathbf{x}) = \mathbf{A} \tag{3.17}$$

Equation (3.16) is minimised when $\nabla f(\mathbf{x}) = 0$, i.e. $\mathbf{Ax} = \mathbf{b}$, and $\mathbf{A}$ is symmetric positive-definite.

So, finding the minimum of the quadratic form by an iterative method is equivalent to finding the solution of $\mathbf{Ax} = \mathbf{b}$ by a direct method. And, if a sparse system occupies all of the computer system memory, it will not be possible to solve the system by a direct method.

3.3.2 Steepest Descent

The *Steepest Descent* method is an iterative method for determining the minimum $\mathbf{x}$ of an *objective function*. The name of the method is derived from the fact that at each iteration, the next step is taken in the direction of steepest descent.

For example, starting at an arbitrary point $\mathbf{x}_0$, the first iteration generates the first approximation $\mathbf{x}_1$ of the exact (unknown) solution $\mathbf{x}$ as:

$$\mathbf{x}_1 = \mathbf{x}_0 + \alpha_0 \mathbf{r}_0 \tag{3.18}$$

where $\mathbf{r}_0$ is the step direction and α_0 is the step length.

From (3.17) it can be seen that at each iteration the direction of steepest descent is:

$$\mathbf{r}_i = -\nabla f(\mathbf{x_i}) = \mathbf{b} - \mathbf{A}\mathbf{x_i} \qquad (3.19)$$

The step direction $\mathbf{r}_i$ is also termed the *residual* and measures the difference between $\mathbf{b}$ the current value of $\mathbf{A}\mathbf{x}_i$.

The step length α_i is chosen such that the directional derivative along the direction of steepest descent is *zero*. This occurs when $\nabla f(\mathbf{x_{i+1}})^{\mathbf{T}}\mathbf{r_i} = 0$, i.e. when they are orthogonal. The result of this is that *steepest descent* potentially takes many steps in the same direction as the iteration progresses towards the exact solution. This leads on to the *conjugate directions* method which attempts to address this.

3.3.3   Conjugate Directions

3.3.4   Conjugate Gradients

3.3.5   Why use Conjugate

Chapter 4

The Aerin Cluster

This chapter describes the hardware and software components of the Aerin Cluster. A description of the BLAS (Basic Linear Algebra Subroutines) Libraries installed on the cluster is also included, together with a description of the pure OpenMPI and hybrid OpenMPI/OpenMP topologies. Finally, a description of Pi Cluster Tools, a suite of scripts which make command line management of the cluster easier, is also included.

Detailed build instructions for the Aerin Cluster, and the implementation details of Pi Cluster Tools, and included in the `picluster` repository wiki.

Photo...

4.1   Hardware

- 8 x Raspberry Pi 4 Model B compute nodes, `node1` to `node8`

- 1 x Raspberry Pi 4 Model B build node, `node9`

- 9 x Official Raspberry Pi 4 power supplies

- 9 x Class 10 A1 MicroSD cards

- 9 x Heatsinks with integrated fans

- 1 x Netgear FVS318G 8 Port Gigabit Router/Firewall

- 1 x Netgear GS316 16 Port Gigabit Switch (with Jumbo Frame Support)

- Cat 7 cabling

4.1.1 Raspberry Pi's

The 9 x Raspberry Pi 4 used in the cluster are the 4GB RAM version. Recently, an 8GB RAM version became available. This which would be the preferred version for a future cluster.

The compute nodes of the cluster are `node1` to `node8`. These are used to run the benchmarks.

Some benchmarks require a substantial amount of time to run, so it is helpful to have a dedicated build node for compiling software, developing scripts, etc, while benchmarks run on compute nodes. This build node is `node9`.

It is convenient to have one of the compute nodes designated the "master" compute node, this is `node1`. If any software needs to be compiled locally to the compute nodes, and not on the build node, then the "master" node is used to do this. This node is also used to mirror the GitHub repository and to run the various Pi Cluster Tools.

4.1.2 Power Supplies

The Raspberry Pi 4 is sensitive to voltage drops, especially whilst booting. So it was decided to purchase 9 Official Raspberry Pi 4 power supplies, rather than a USB hub with multiple power outlets which may not have been able to maintain output voltage whilst booting 9 nodes. The 9 power supplies do take up some space, so a future development would be to investigate a suitably rated USB hub.

4.1.3 MicroSD Cards

MicroSD cards are available in a number of speed classes and "use" categories. The recommended minimum specification for the Raspberry Pi 4 is Class 10 A1. The "10" refers to a 10 MB/s write speed. The "A" refers to the "Application" category, which supports at least 1500 read operations and 500 write operations per second.

4.1.4 Heatsinks

Cooling is a major consideration when building any cluster, even an 8 node Raspberry Pi cluster. The Raspberry Pi 4 throttles back the clock speed at approximately 85°C, which would not only have had a negative impact on bench-

mark results, but also on repeatability. So, it was very important to select suitable cooling. After some investigation, it was decided to purchase heatsinks with integrated fans. These proved to be very successful, with no greater than 65°C observed at any time, even with 100% CPU utilisation for many hours.

4.1.5 Network Considerations

The MTU is the network packet payload size in bytes, i.e. the size of your data that is transmitted in a single network packet. It is actually 28 bytes less than this due to network protocol overhead. We shall see later how a larger MTU can improve network efficiency and improve benchmark performance.

A Jumbo Frame is any MTU greater than 1500 bytes. There is no standard maximum size for a Jumbo Frame, but the norm seems to be 9000 bytes. Not all network devices support Jumbo Frames, and a change of MTU size from the default 1500 bytes has to be supported by all devices on the network (although some devices are smart enough to accommodate multiple MTU's).

As we shall see, the Raspberry Pi 4 has very good Ethernet networking capabilities. The theoretical maximum bandwidth of a Gigabit Ethernet connection is 1 Gbit/s. With the default MTU (Maximum Transmission Unit) of 1500 bytes, the Raspberry Pi 4 can achieve 930 Mbit/s. This is 93% of the theoretical maximum bandwidth. Increasing the MTU to 9000 bytes increases the achievable, and measurable, bandwidth to 980 Mbit/s. This is, effectively, full Gigabit speed. It is important we make full use of this with adequate network equipment.

It would be tempting to use any old router/firewall, switch, and cabling found lurking around in some dusty cupboard. This would be a mistake, and potentially cripple the cluster network. Courtesy of Ebay, I acquired a professional grade router/firewall and switch for less than £30 each. And the switch supports Jumbo Frames up to 9000 bytes, which we will be making use of.

4.1.6 Router/Firewall

The router/firewall acts a cluster interface to the outside world. The firewall wall only permits certain network packets access to the cluster through holes in the wall, in our case only `ssh` packets. One side of the firewall is the cluster LAN (Local Area Network). The other side of the firewall is the WAN (Wide Area Network).

In my home environment the WAN is connected to my ADSL router via an ethernet cable. This permits the compute nodes on the LAN to connect to the

internet and download updates. When relocated to UCL, the WAN would be connected to the internal UCL network.

The router exposes a single IP address for the cluster to the WAN. Access from the outside world to the cluster is through this single IP address via `ssh`, which is routed to `node1`.

The router also acts as DHCP (Dynamic Host Configuration Protocol) server for the compute node LAN. Compute node hostnames, such as `node1` etc, are configured by a boot script which determines the node hostname from the last octet of the node IP address, served by the DHCP server based on the MAC address. This ensures that each compute node is always assigned the same LAN IP address and hostname across reboots.

It sounds more complicated than it actually is, and is easily configured through the router/firewall web-based setup. More details are in Part II Chapter 7.

4.1.7   Network Switch

The switch acts as an extension to the number of ports on the compute node LAN. And because it supports Jumbo Frames it can accommodate an MTU increase to 9000 bytes localised to the compute nodes.

My initial build of the Aerin Cluster only used the 8 port router/firewall. Only having 8 ports quickly became tiresome, so a 5 port switch was added so that I could directly connect `node9` and my `macbook` to the compute nodes without having to `ssh` through the firewall. Later, when it became apparent that the 5 port switch didn't support Jumbo Frames, this was replaced with the current 16 port switch. I did anticipate having to replace the router/firewall because it doesn't support Jumbo Frames, but the switch is sufficiently smart to route 9000 byte packets between the compute nodes, and fragment any packets to/from the outside world, through the router/firewall, into 1500 byte packets.

4.1.8   Cabling

Cat 5 network cables only support 100 Mbit/s, and without any electrical shielding. Cat 5e supports 1 Gbit/s without shielding. Cat 6 supports 1 Gbit/s, possibly with shielding depending on the cable. Cat 6a and Cat 7 support 10 Gbit/s with electrical shielding. Therefore, to ensure maximum use of the network capabilities of the Raspberry Pi 4, a minimum of Cat 5e cabling must be used. Because the Aerin Cluster cable lengths are relatively short, and therefore inexpensive, I opted to use CAT 7 cabling. My advice would be to do the same for any future clusters. Any performance limiting factor is then not the cabling.

If you need to use old cable, check the labelling!

4.2   Software

4.2.1   Operating System

The operating system used for the Aerin Cluster is Ubuntu 20.04 LTS 64-bit Pre-Installed Server for the Raspberry Pi 4. This can be downloaded from the Ubuntu website or installed via Raspberry Pi Imager.

4.2.2   `cloud-init`

The `cloud-init` system was originally developed by Ubuntu to simplify the instantiation of operating system images in cloud computing environments, such as Amazon's AWS and Microsoft's Azure. It is now an industry standard.

It is also very useful for automating the installation of the same operating system on a number of computers using a single installation image. This dramatically simplifies building a cluster.

The idea is that a `user-data` file is added to the `boot` directory of an installation image. When a node boots using the image, this file is read and the configuration/actions specified in this file are automatically applied/run as the operating system is installed.

For the Aerin Cluster the following configuration/actions were applied to each node:

- Add the user `john` to the system and set the initial password
- Add `john's` public key
- Update the `apt` data cache
- Upgrade the system
- Install specified software packages
- Create a `/etc/hosts` file
- Set the hostname based on the IP address

All of this is done from a single image and `user-data` file. The time invested in getting the `user-data` file right pays off handsomely, especially when the cluster may need to be rebuild from scratch a number of times.

The main software packages used for benchmarking installed by `cloud-init` are:

- build-essential

- openmpi-bin

- libopenblas0-serial

- libopenblas0-openmp

- libblis3-serial

- libblis3-openmp

This installs essential software build tools, such as C/C++ compilers, `make`, etc, OpenMPI binary and development files, and the OpenBLAS and BLIS libraries in both serial and OpenMP versions.

The package names for the OpenBLAS and BLIS libraries are somewhat cryptic and can cause confusion. For example, the BLIS packages libblis64-serial and libblis64-openmp are not the 64-bit packages we would expect to install on a 64-bit operating system. The "64" refers to the integer size for the BLAS library. The packages we need are the libblis3-serial and libblis3-openmp versions, which are still 64-bit packages.

4.2.3   Benchmark Software

The HPL, HPCC and HPCG benchmark software was all compiled locally from source. The instructions for how to do this are in the project repository wiki.

4.3   BLAS Libraries

If we use the Linux `perf` command to sample and record CPU stack traces (via frame pointers) for an `xhpl` process (with Process ID 6595) for 30 seconds:

```
$ sudo perf record -p 6595 -g -- sleep 30
```

And then look at the stack trace report:

```
$ sudo perf report
```

```
+ 100.00%  0.00%  xhpl  xhpl              [.]  _start
+ 100.00%  0.00%  xhpl  libc-2.31.so      [.]  __libc_start_main
+ 100.00%  0.00%  xhpl  xhpl              [.]  main
+ 100.00%  0.00%  xhpl  xhpl              [.]  HPL_pdtest
+ 100.00%  0.00%  xhpl  xhpl              [.]  HPL_pdgesv
+ 100.00%  0.00%  xhpl  xhpl              [.]  HPL_pdgesv0
+  98.03%  0.00%  xhpl  xhpl              [.]  HPL_pdupdateTT
+  97.71%  0.00%  xhpl  libblas.so.3      [.]  0x0000ffffaa839ff0
+  97.71%  0.00%  xhpl  libgomp.so.1.0.0  [.]  GOMP_parallel
+  97.70%  0.00%  xhpl  libblas.so.3      [.]  0x0000ffffaa839e80
-  96.56%  0.00%  xhpl  xhpl              [.]  HPL_dgemm
    HPL_dgemm
    dgemm_
```

It can be seen that 96.56% of the time within the xhpl process is spent in the HPL_dgemm function, which subsequently calls the BLAS dgemm_ function (the _ appended to dgemm function name is the Fortran function name decoration).

It is for this reason that the efficiency of the BLAS library is critical for both benchmark and real-world application performance. The efficiency of the dgemm (**d**ouble precision **ge**neral **m**atrix **m**ultiplication) function is particularly important for the dense matrix HPL benchmark.

The mathematical operation implemented by dgemm is:

$$C := \alpha \times A \times B + \beta \times C$$

where A is a $M \times K$ matrix, B is a $K \times N$ matrix, C is a $M \times N$ matrix, and α and β are scalars.

In the case of the HPL benchmark, $M = N = K$.

Efficient BLAS libraries "block" matrix multiplication into smaller sub-matrix multiplications. The "block" sizes of these sub-matrices multiplications are carefully chosen to make optimum use of CPU registers, and L1, L2, and L3 (when available) cache sizes. These sub-matrix multiplications are referred to as *kernels*, or sometimes *micro-kernels*.

Maximum performance is achieved when the matrix multiplication problem size in an integer multiple of the dgemm "block" size. In the case of the HPL benchmark, maximum performance is achieved when the the problem size N is an integer multiple of the HPL NB block size, which in turn is an integer multiple of the BLAS "block" size.

Figure 4.1: `dgemm` *kernel* Matrix-Matrix Multiplication.

The above is depicted in Figure ??.

4.3.1 GotoBLAS

GotoBLAS is a high performance BLAS library developed by Kazushige Goto at the Texas Advanced Computing Center (TACC), a department of the University of Texas at Austin.

GotoBLAS achieves high performance through the use of hand-crafted assembly language *kernels*. Higher level BLAS routines are decomposed in *kernels*, which stream data from the L1 and L2 CPU caches. These kernels typically reflect the size of the CPU registers, and L1 and L2 caches. For example, a CPU architecture may have a 4 x 4 *dgemm kernel* and a 4 x 8 *dgemm kernel* which conduct a double precision matrix-matrix multiplication on 4 x 4 and 4 x 8 matrices, respectively, and which have been sized for a specific architecture.

The source code for GotoBLAS and GotoBLAS2 is still available as Open Source software, but the library is no longer in active development.

4.3.2 OpenBLAS

OpenBLAS is an Open Source fork of the original GotoBLAS2 library, and is in active development by volunteers led by Zhang Xianyi at the Lab of Parallel Software and Computational Science, Institute of Software, Chinese Academy of Sciences (ISCAS).

OpenBLAS is used by many of the Top500 supercomputers, including the Fugaku supercomputer which tops the June 2020 TOP500 List.

For the Arm64 architecture, OpenBLAS implements the following `dgemm` *kernels*, where `.S` indicates an assembly language file:

- dgemm_kernel_4x4.S

- dgemm_kernel_4x8.S

- dgemm_kernel_8x4.S

4.3.3 BLIS

The "BLAS-like Library Instantiation Software" (BLIS) is a BLAS library implementation for many CPU architectures, and also a framework for implementing new BLAS libraries for new architectures. Using the BLIS framework, by solely implementing an optimised `dgemm` *kernel* in assembly language or compiler intrinsics, BLAS library functionality can be realised which achieves 60% - 90% of theoretical maximum performance.

BLIS is developed by the Science of High-Performance Computing (SHPC) group of the Oden Institute for Computational Engineering and Sciences, at The University of Texas at Austin.

For the Arm64 architecture, BLIS implements the following `dgemm` assembly language *kernel*:

- gemm_armv8a_asm_6x8

4.3.4 BLAS Library Management

Two different BLAS libraries are installed on the Aerin Cluster, OpenBLAS and BLIS, both in serial and OpenMP versions. For benchmark consistency it is essential to ensure the same BLAS library in use on each node at any one time.

The Debian/Ubuntu *alternatives* mechanism is used to set the BLAS library in use from a selection of *alternatives*. Each *alternative* is a *symbolic link* in the file system which points to the appropriate shared library, the `libblas.so.3` library installed with each BLAS package. The *alternatives* mechanism is a command line interface to update the symbolic links.

Using the *alternatives* mechanism command line interface on 8 nodes is error prone. So, two `bash` script wrapper tools were written and included in Pi Cluster Tools, discussed in the next section.

4.4 Cluster Topologies

A pure OpenMPI distribution of `xhpl` work processes on a single node with 4 cores/slots is depicted in Figure ?? (a). Each `xhpl` process calls the functions of a single-threaded BLAS serial library.

A hybrid OpenMPI/OpenMP distribution of work is depicted in Figure ?? (b). A single `xhpl` process calls functions from a multi-threaded BLAS library which run as threads on the processor cores.

4.4.1 Pure OpenMPI

In a pure OpenMPI topology, work is distributed across the cluster nodes, and the processor cores on each node, by OpenMPI. Processor cores are referred to as *slots*. The number of nodes in the cluster, and the number of slots on each node, are specified using either the `-host` or `-hostfile` parameter of the `mpirun` command. Each processor slot is a target for a work process.

The `-host` parameter is used to specify nodes and slots on the command line.

The `-hostfile` parameter is used to specify a file which contains the nodes and slots information.

In the following example, the `-host` parameter is used to specify 2 nodes, each with 4 slots, on which to run 8 `xhpl` work processes:

```
$ mpirun --bind-to core -host node1:4,node2:4 -np 8 xhpl
```

The same number of nodes, slots and work processes is specified using the `-hostfile` parameter as follows:

```
$ mpirun --bind-to core -hostfile nodes -np 8 xhpl
```

(a) Pure OpenMPI

(b) Hybrid OpenMPI/OpenMP

Figure 4.2: Single Node Toplologies.

Where `nodes` is a file containing the following:

Listing 4.1: nodes

```
node1 slots=4
node2 slots=4
```

The `--bind-to core` parameter instructs `mpirun` to not migrate a work processes from the core on which it was started. Once started on a specific core, a work process will remain *bound* to that core. This is an optimisation which reduces cache refreshes when a work process is interrupted, by a kernel system call for example, and is then restarted.

4.4.2 Hybrid OpenMPI/OpenMP

In a hybrid OpenMPI/OpenMP topology, OpenMPI is used to distribute work between the nodes of the cluster, and each node runs a single work process. OpenMP is then used to distribute the work of this single process between the cores of the node.

The `-host`, `-hostfile` and `-np` parameters of the `mpirun` command are used in a same manner as the pure OpenMPI case, noting that each node now has 1 slot on which to run a work process.

The additional parameter `-x` is required now required. This parameter distributes and sets the *environmental variable* `OMP_NUM_THREADS` on each node prior to a work process being started on each node. This variable is queried by the multi-threaded BLAS library, and the appropriate number of threads are utilized.

Using 2 nodes, and 4 cores per node, as per the pure OpenMPI example, 2 `xhpl` processes are run, one on each node, with the multi-threaded BLAS library utilising 4 cores, as follows:

```
$ mpirun --bind-to socket -host node1:1,node2:1 -np 2 -x
    ↪ OMP_NUM_THREADS=4 xhpl
```

The `--bind-to socket` parameter indicates to `mpirun` that the `xhpl` process is not associated with a particular core, it is not to be *bound* to a specific core. The OpenMP runtime will determine which core(s) are used to run the `xhpl` process.

Note, without the `--bind-to socket` parameter only a single thread will be utilised for a multi-threaded BLAS library, even if the `OMP_NUM_THREADS` environmental variable is set correctly.

4.5   Pi Cluster Tools

Using `ssh` to manually log into each node and perform command line administration tasks is prone to error. To get around this problem, a number of `bash` wrapper scripts were written as Pi Cluster Tools. Each script loops over a list of node names and uses `ssh` to run a remote command on each node in turn.

The following scripts are included in Pi Cluster Tools.

- upgrade
- reboot
- shutdown
- do
- libblas-query
- libblas-set
- linpack-profiler
- interrupt-coalescing
- receive-packet-steering
- receive-flow-steering

Script listings are sample usage are included in the `picluster` repository wiki.

Chapter 5

Benchmark Parameter Tuning and Optimisation

5.1 Theoretical Maximum Performance (Gflops)

The Raspberry Pi 4 Model B is based on the Broadcom BCM2711 System on a Chip (SoC). The BCM2711 contains 4 Arm Cortex-A72 cores clocked at 1.5 GHz.

Each core implements the 64-bit Armv8-A Instruction Set Architecture (ISA). This instruction set includes Advanced SIMD instructions which operate on a single 128-bit SIMD pipeline. This 128-bit pipeline can conduct two 64-bit double precision floating point operations (Flops) per clock cycle.

A *fused multiply-add* (FMA) instruction implements a multiplication followed by an add in a single instruction. The main purpose of FMA instructions is to improve result accuracy by conducting a single rounding operation on completion of both the multiplication and the add operations. A single FMA instruction counts as two Flops.

The theoretical maximum performance of a single Aerin Cluster node, R_{peak}, is therefore:

$$R_{peak} = 4 \text{ cores} \times 1.5 \text{ GHz} \times 2 \text{ doubles} \times 2 \text{ FMA} \qquad (5.1)$$
$$= 24 \text{ Gflops} \qquad (5.2)$$

This is only achievable continuously if every instruction in a program is an FMA instruction, which obviously cannot be the case, since data has to be loaded from memory and stored back into memory. Nevertheless, this is the standard measure of theoretical maximum performance.

The theoretical maximum performance of the Aerin Cluster as a whole is therefore:

$$R_{peak} = 8 \text{ nodes} \times 24 \text{ Gflops} \tag{5.3}$$
$$= 192 \text{ Gflops} \tag{5.4}$$

For the High Performance Linpack benchmark, to achieve 100% performance requires a problem size that utilises 100% of memory. Because the operating system requires memory, is it not possible to use 100% for benchmarks.

The Linux `dmesg` command prints out the kernel boot messages, which can be searched using `grep` to determine how memory is utilised on the system:

```
$ dmesg | grep Memory
```

```
[    0.000000] Memory: 3783876K/4050944K available (11772K
    ↪ kernel code, 1236K rwdata, 4244K rodata, 6144K init,
    ↪ 1072K bss, 201532K reserved, 65536K cma-reserved)
```

As can be seen, 37838776k of memory is available, which equates to 90% of the 4 GB (4194304k) on each node. It would be optimistic to expect to use every byte of this 90%, and using any more than this would result in swap space being used which would negatively impact benchmark results.

So, for the HPL baseline benchmarks, 80% of memory was chosen for the problem size. This is the amount suggested as an initial "good guess" in the HPL Frequently Asked Questions.

The above necessarily results in the baseline benchmarks only being able to achieve 80% of R_{peak} at best, 4.8 Gflops for a single core, 19.2 Gflops for a single node, and 153.6 Gflops for the 8 node cluster. These values are indicated on the HPL baseline result plots.

5.2   HPL Baseline

The HPL benchmark software was compiled locally from source. Detailed instructions on how to do this are in Part II Chapter??

Ubuntu 20.04 LTS 64-bit packages, without any tweaks...

80% of memory

Methodology...

1 core... to investigate single core performance... caveats... use 1GB of memory...

1 node... to investigate inter-core performance...

2 nodes... to investigate inter-core and inter-node performance...

1..8 nodes ... to investigate over scaling of performance with node count... with optimal N, NB, P and Q parameters determined from 2 node investigation... caveats...

5.2.1 HPL 1 Core Baseline

The purpose of this baseline is to determine the performance of a single core running a single `xhpl` process, with the single core having exclusive access to the shared L2 cache.

As discussed in the previous section, the HPL problem size is restricted to 80% of available memory. In the case of this baseline, this is 80% of a single node's 4 GB.

Using values of block size NB from 32 to 256, in increments of 8, and using formula ?? to ensure the problem size N is an integer multiple of NB, results in the table below of NB and N combinations.

NB	N	NB	N	NB	N	NB	N	NB	N
32	18528	80	18480	128	18432	176	18480	224	18368
40	18520	88	18480	136	18496	184	18400	232	18328
48	18528	96	18528	144	18432	192	18432	240	18480
56	18536	104	18512	152	18392	200	18400	248	18352
64	18496	112	18480	160	18400	208	18512	256	18432
72	18504	120	18480	168	18480	216	18360	-	-

Table 5.1: 1 Core NB and N Combinations using 80% of 4 GB Memory.

The results are plotted in Figure ??.
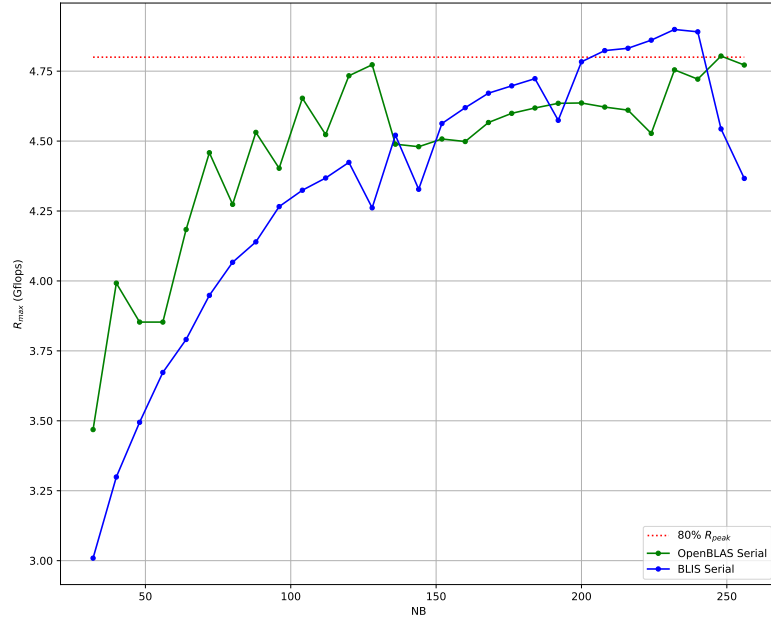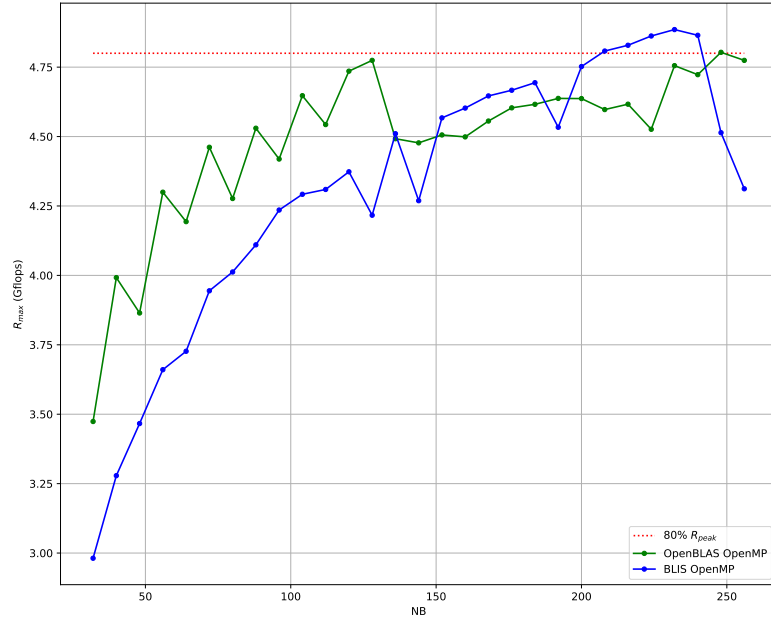
(a) Pure OpenMPI

(b) Hybrid OpenMPI/OpenMP

Figure 5.1: 1 Core R_{max} vs NB using 80% memory.

Observations

As expected, there is no noticeable performance difference between a pure Open-MPI and hybrid OpenMPI/OpenMP topology for a single `xhpl` process running on a single core.

Both topologies attain 80% R_{peak} for a single core.

Discussion about OpenBLAS and BLIS internal kernel blocking...

5.2.2   HPL 1 Node Baseline

The purpose of this baseline is to determine the performance of the 4 cores of a single node. In this case each core has to share access to the L2 cache, which will result in more main memory accesses. It is therefore anticipated that this will result in a performance reduction, per core, compared to the single core case.

As per the single core benchmark, the HPL problem size is restricted to 80% of available memory. Again, in the case, this is 80% of a single node's 4 GB. This results in the same table of NB and N combinations as the single core benchmark.

NB	N	NB	N	NB	N	NB	N	NB	N
32	18528	80	18480	128	18432	176	18480	224	18368
40	18520	88	18480	136	18496	184	18400	232	18328
48	18528	96	18528	144	18432	192	18432	240	18480
56	18536	104	18512	152	18392	200	18400	248	18352
64	18496	112	18480	160	18400	208	18512	256	18432
72	18504	120	18480	168	18480	216	18360	-	-

Table 5.2: 1 Node NB and N Combinations using 80% of 4 GB Memory.

The results are plotted in Figure ??
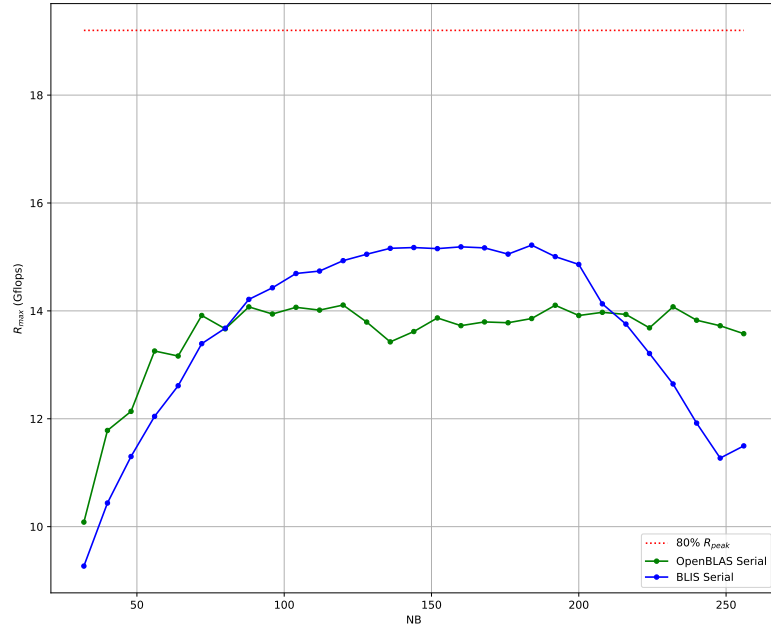
(a) Pure OpenMPI

(b) Hybrid OpenMPI/OpenMP

Figure 5.2: 1 Node R_{max} vs NB using 80% memory.

Observations

As anticipated, there is indeed a reduction in performance per core, 80% R_{peak} in no longer attained.

Pure OpenMPI topology attains a R_{max} of ?? with an NB of ??.

The hybrid OpenMPI/OpenMP topology attains a R_{max} of ?? with an NB of ??.

5.2.3   HPL 2 Node Baseline

The purpose of this baseline is to determine the performance of 2 nodes. Now, each core not only has to share access to the L2 cache, but the cache may be refreshed with data less frequently due to network delays and competition between the nodes for access to network. It is therefore anticipated that this will result in a performance reduction, per node, compared to the single node case.

For this baseline the HPL problem size is restricted to 80% of 2 nodes combined memory, 80% of 8 GB. This results in NB and N combinations as tabulated below:

NB	N	NB	N	NB	N	NB	N	NB	N
32	26208	80	26160	128	26112	176	26048	224	26208
40	26200	88	26136	136	26112	184	26128	232	25984
48	26208	96	26208	144	26208	192	26112	240	26160
56	26208	104	26208	152	26144	200	26200	248	26040
64	26176	112	26208	160	26080	208	26208	256	26112
72	26208	120	26160	168	26208	216	26136	-	-

Table 5.3: 2 Node NB and N Combinations using 80% of 8 GB Memory.

The results are plotted in Figure ??

(a) Pure OpenMPI

(b) Hybrid OpenMPI/OpenMP

Figure 5.3: 2 Node R_{max} vs NB using 80% memory.

Observations

5.2.4   HPL Cluster Baseline

This cluster baseline uses the optimum values of NB from the 2 Node Baseline. For each of the 4 BLAS library combinations, OpenBLAS serial, OpenBLAS OpenMP, BLIS serial, and BLIS OpenMP, with the corresponding value of N for 80% of memory for the particular node count is used, as tabulated below.

BLAS	NB	Nodes					
		3	4	5	6	7	8
OpenBLAS Serial	000	32000	36000	41000	45000	48000	52000
OpenBLAS OpenMP	000	32000	36000	41000	45000	48000	52000
BLIS Serial	000	32000	36000	41000	45000	48000	52000
BLIS OpenMP	000	32000	36000	41000	45000	48000	52000

The baseline results are presented in Figure ??.

	Nodes	N	NB	P	Q	R_{max} (Gflops)
OpenBLAS Serial	3	32040	120	1	12	3.3720e+01
	3	32040	120	2	6	3.1946e+01
	3	32040	120	3	4	3.3844e+01
	4	36960	120	1	16	4.7742e+01
	4	36960	120	2	8	4.9390e+01
	5	41400	120	1	20	5.6513e+01
	5	41400	120	2	10	5.6038e+01
	5	41400	120	4	5	5.5649e+01
	6	45360	120	1	24	6.8392e+01
	6	45360	120	2	12	7.3856e+01
	6	45360	120	3	8	6.9952e+01
	7	48960	120	1	28	7.8248e+01
	7	48960	120	2	14	8.1017e+01
	7	48960	120	4	7	8.1433e+01
	8	52320	120	1	32	8.6787e+01
	8	52320	120	2	16	9.5517e+01
	8	52320	120	4	8	9.5525e+01
OpenBLAS OpenMP	3	32032	88	1	3	3.7842e+01
	4	37048	88	1	4	4.8657e+01
	5	41448	88	1	5	6.0428e+01
	6	45320	88	1	6	6.8713e+01
	6	45320	88	2	3	7.3722e+01
	7	49016	88	1	7	7.8712e+01
	8	52360	88	1	8	9.4245e+01
	8	52360	88	2	4	9.6630e+01

Table 5.4: HPL Cluster Baseline using OpenBLAS.

	Nodes	N	NB	P	Q	R_{max} (Gflops)
BLIS Serial	3	32088	168	1	12	3.9005e+01
	3	32088	168	2	6	3.9050e+01
	3	32088	168	3	4	3.8958e+01
	4	36960	168	1	16	4.9694e+01
	4	36960	168	2	8	5.4268e+01
	5	41328	168	1	20	5.5398e+01
	5	41328	168	2	10	6.5226e+01
	5	41328	168	4	5	6.2356e+01
	6	45360	168	1	24	7.0278e+01
	6	45360	168	2	12	7.9685e+01
	6	45360	168	3	8	7.5475e+01
	7	48888	168	1	28	8.0168e+01
	7	48888	168	2	14	8.7571e+01
	7	48888	168	4	7	8.6035e+01
	8	52416	168	1	32	9.1148e+01
	8	52416	168	2	16	1.0341e+02
	8	52416	168	4	8	1.0190e+02
BLIS OpenMP	3	32000	200	1	3	3.5132e+01
	4	37000	200	1	4	4.6953e+01
	5	41400	200	1	5	6.2550e+01
	6	45400	200	1	6	6.7204e+01
	6	45400	200	2	3	7.2585e+01
	7	49000	200	1	7	8.1255e+01
	8	52400	200	1	8	9.1180e+01
	8	52400	200	2	4	1.0041e+02

Table 5.5: HPL Cluster Baseline using BLIS.
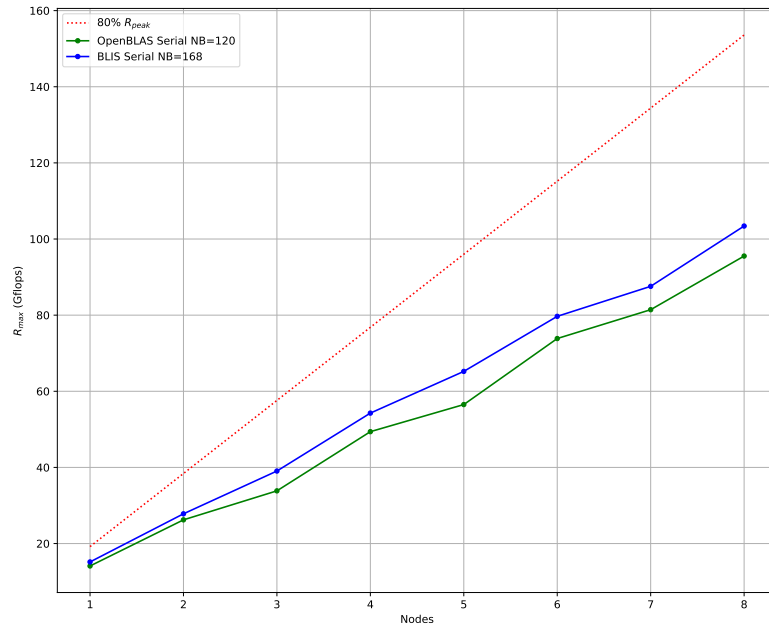
(a) Pure OpenMPI
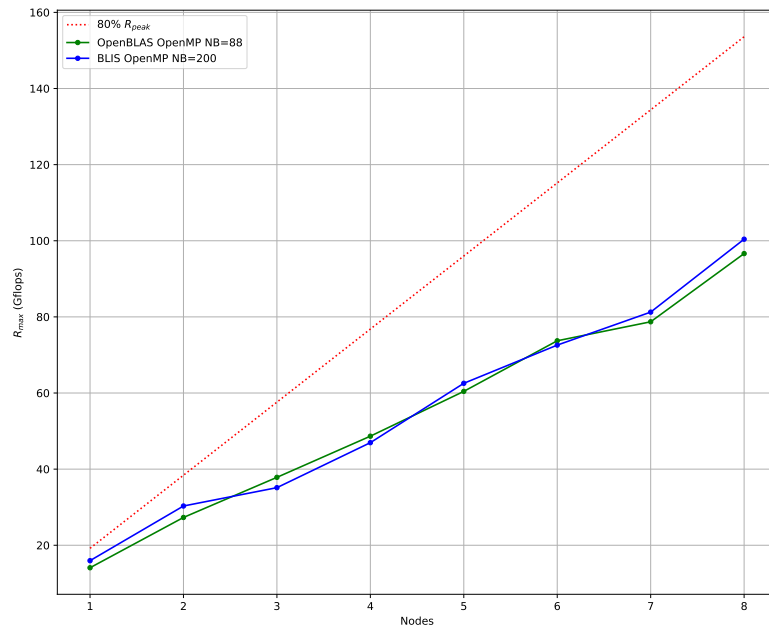
(b) Hybrid OpenMPI/OpenMP

Figure 5.4: R_{max} vs Nodes using 80% memory.

5.2.5 Observations

Best NB...

PxQ discussion... 1x8 vs 2x4... ethernet comment...

Iperf...

htop...

top...

perf...

cache misses...

software interrupts...

Suggests... improve network efficiency?

5.3 HPCC Baseline

The HPCC baseline benchmarks were run using all 8 nodes of the Aerin Cluster. The results for each benchmark are presented below.

5.3.1 HPL

	Nodes	N	NB	P	Q	R_{max} (Gflops)
OpenBLAS Serial	8	52320	120	1	32	
	8	52320	120	2	16	
	8	52320	120	4	8	
OpenBLAS OpenMP	8	52360	88	1	8	8.685e+01
	8	52360	88	2	4	9.497e+01
BLIS Serial	8	52416	168	1	32	8.866e+01
	8	52416	168	2	16	1.014e+02
	8	52416	168	4	8	1.005e+02
BLIS OpenMP	8	52400	200	1	8	8.163e+01
	8	52400	200	2	4	8.841e+01

Table 5.6: HPCC HPL.

5.3.2 DGEMM

	Results
OpenBLAS Serial	DGEMM_N=5339
	StarDGEMM_Gflops=3.59743
	SingleDGEMM_Gflops=4.91086
OpenBLAS OpenMP	DGEMM_N=10687
	StarDGEMM_Gflops=14.4261
	SingleDGEMM_Gflops=14.426
BLIS Serial	DGEMM_N=5349
	StarDGEMM_Gflops=3.02439
	SingleDGEMM_Gflops=4.95418
BLIS OpenMP	DGEMM_N=10695
	StarDGEMM_Gflops=16.3355
	SingleDGEMM_Gflops=15.2042

Table 5.7: HPCC DGEMM.

Table ?? requires some interpretation. For the single-threaded serial versions of the OpenBLAS and BLIS libraries, the cluster consists of 32 processing cores, so the `SingleDGEMM_Gflops` are per core. For the multi-threaded OpenMP versions of the libraries, the cluster consists of 8 processing nodes, so the `SingleDGEMM_Gflops` are per node.

The results are consistent with the HPL benchmarks, which spend 96%+ of the benchmark time in the BLAS `dgemm` subroutine.

5.3.3 STREAM

	Results
OpenBLAS Serial	STREAM_VectorSize=28514400
	STREAM_Threads=1
	StarSTREAM_Copy=0.92926
	StarSTREAM_Scale=0.979969
	StarSTREAM_Add=0.902324
	StarSTREAM_Triad=0.899619
	SingleSTREAM_Copy=5.36868
	SingleSTREAM_Scale=5.41684
	SingleSTREAM_Add=4.75638
	SingleSTREAM_Triad=4.75692
OpenBLAS OpenMP	STREAM_VectorSize=114232066
	STREAM_Threads=1
	StarSTREAM_Copy=4.76068
	StarSTREAM_Scale=5.44287
	StarSTREAM_Add=4.51713
	StarSTREAM_Triad=4.53621
	SingleSTREAM_Copy=5.47035
	SingleSTREAM_Scale=5.46963
	SingleSTREAM_Add=4.87128
	SingleSTREAM_Triad=4.89569
BLIS Serial	STREAM_VectorSize=28619136
	STREAM_Threads=1
	StarSTREAM_Copy=0.943137
	StarSTREAM_Scale=0.989024
	StarSTREAM_Add=0.910843
	StarSTREAM_Triad=0.909211
	SingleSTREAM_Copy=4.72341
	SingleSTREAM_Scale=4.21768
	SingleSTREAM_Add=3.90016
	SingleSTREAM_Triad=3.94385
BLIS OpenMP	STREAM_VectorSize=114406666
	STREAM_Threads=1
	StarSTREAM_Copy=5.05861
	StarSTREAM_Scale=5.39591
	StarSTREAM_Add=4.66044
	StarSTREAM_Triad=4.6751
	SingleSTREAM_Copy=5.41884
	SingleSTREAM_Scale=5.45544
	SingleSTREAM_Add=4.80613
	SingleSTREAM_Triad=4.81397

Table 5.8: HPCC STREAM.

5.3.4 PTRANS

	Results
OpenBLAS Serial	PTRANS_GBs=0.465891 PTRANS_n=26160 PTRANS_nb=120 PTRANS_nprow=1 PTRANS_npcol=32
OpenBLAS OpenMP	PTRANS_GBs=0.616885 PTRANS_n=26180 PTRANS_nb=88 PTRANS_nprow=2 PTRANS_npcol=4
BLIS Serial	PTRANS_GBs=0.483766 PTRANS_n=26208 PTRANS_nb=168 PTRANS_nprow=1 PTRANS_npcol=32
BLIS OpenMP	PTRANS_GBs=0.637484 PTRANS_n=26200 PTRANS_nb=200 PTRANS_nprow=2 PTRANS_npcol=4

Table 5.9: HPCC PTRANS.

5.3.5 Random Access

	Results
OpenBLAS Serial	MPIRandomAccess_LCG_N=2147483648
	MPIRandomAccess_LCG_GUPs=0.000642364
	MPIRandomAccess_N=2147483648
	MPIRandomAccess_GUPs=0.000645338
	RandomAccess_LCG_N=67108864
	StarRandomAccess_LCG_GUPs=0.00373175
	SingleRandomAccess_LCG_GUPs=0.00815537
	RandomAccess_N=67108864
	StarRandomAccess_GUPs=0.00373372
	SingleRandomAccess_GUPs=0.00837312
OpenBLAS OpenMP	MPIRandomAccess_LCG_N=2147483648
	MPIRandomAccess_LCG_GUPs=0.000473649
	MPIRandomAccess_N=2147483648
	MPIRandomAccess_GUPs=0.000477404
	RandomAccess_LCG_N=268435456
	StarRandomAccess_LCG_GUPs=0.00580416
	SingleRandomAccess_LCG_GUPs=0.00582959
	RandomAccess_N=268435456
	StarRandomAccess_GUPs=0.00614337
	SingleRandomAccess_GUPs=0.00613214
BLIS Serial	MPIRandomAccess_LCG_N=2147483648
	MPIRandomAccess_LCG_GUPs=0.000644523
	MPIRandomAccess_N=2147483648
	MPIRandomAccess_GUPs=0.00064675
	RandomAccess_LCG_N=67108864
	StarRandomAccess_LCG_GUPs=0.00374527
	SingleRandomAccess_LCG_GUPs=0.00835127
	RandomAccess_N=67108864
	StarRandomAccess_GUPs=0.00374741
	SingleRandomAccess_GUPs=0.00820883
BLIS OpenMP	MPIRandomAccess_LCG_N=2147483648
	MPIRandomAccess_LCG_GUPs=0.000475485
	MPIRandomAccess_N=2147483648
	MPIRandomAccess_GUPs=0.000476047
	RandomAccess_LCG_N=268435456
	StarRandomAccess_LCG_GUPs=0.00580705
	SingleRandomAccess_LCG_GUPs=0.00578222
	RandomAccess_N=268435456
	StarRandomAccess_GUPs=0.00614596
	SingleRandomAccess_GUPs=0.00613275

Table 5.10: HPCC Random Access.

5.3.6 FFT

5.3.7 Network Bandwidth and Latency

5.4 HPCG Baseline

The June 2020 HPCG List ranks 169 computer in order of conjugate gradient performance. Ranking number 1 in the list is the Fugaku supercomputer. And ranking 169 is the Spaceborne Computer, which is onboard the International Space Station. The Spaceborne Computer is a 32 core system based on the Intel Xeon E5-2620 v4 8 Core CPU clocked at 2.1GHz, with an Infiniband interconnect. The HPCG List performance results of these two computers are in Table ??.

HPCG Rank	Name	Cores	HPL R_{max} Pflops	TOP500 Rank	HPCG Pflops	Fraction of Peak
1	Fugaku	6,635,520	415.530	1	13.366	2.6%
169	Spaceborne Computer	32	0.001	-	0.000034	2.9%

Table 5.11: Extract from June 2020 HPCG List.

For computers to be officially ranked in the HPCG List, the HPCG benchmark must be run for in excess of 30 minutes using at least 25% of available memory.

The following results were obtained for the Aerin Cluster running the HPCG benchmark for 60 minutes using 75% of available memory and using all 8 nodes.

HPCG Rank	Name	Cores	HPL R_{max} Pflops	TOP500 Rank	HPCG Pflops	Fraction of Peak
-	OpenBLAS Serial	32		-		%
-	OpenBLAS OpenMP	32		-		%
-	BLIS Serial	32		-		%
-	BLIS OpenMP	32		-		%

Table 5.12: The Aerin Cluster HPCG Benchmark.

5.5 Optimisations

5.5.1 Interrupt Coalescing

As discussed in Chapter 2, each packet received by a network interface generates a hardware interrupt. This results in a context switch to the kernel to process the packet. *Interrupt coalescing* delays the raising of a hardware interrupt until a specified number of packets have been received, or a specified period of time has elapsed, thereby reducing the number of context switches. This potentially improves throughput and benchmark performance.

The Pi Cluster Tools `interrupt-coalescing` tool enables *Adaptive RX* interrupt coalescing. In *Adaptive RX* interrupt coalescing the kernel actively manages the number of packets received, and the time elapsed, before raising a hardware interrupt on the network interface receive queue.

The results of running the `linpack-profiler` tool with *Adaptive RX* interrupt coalescing enabled are plotted in Figure ??.
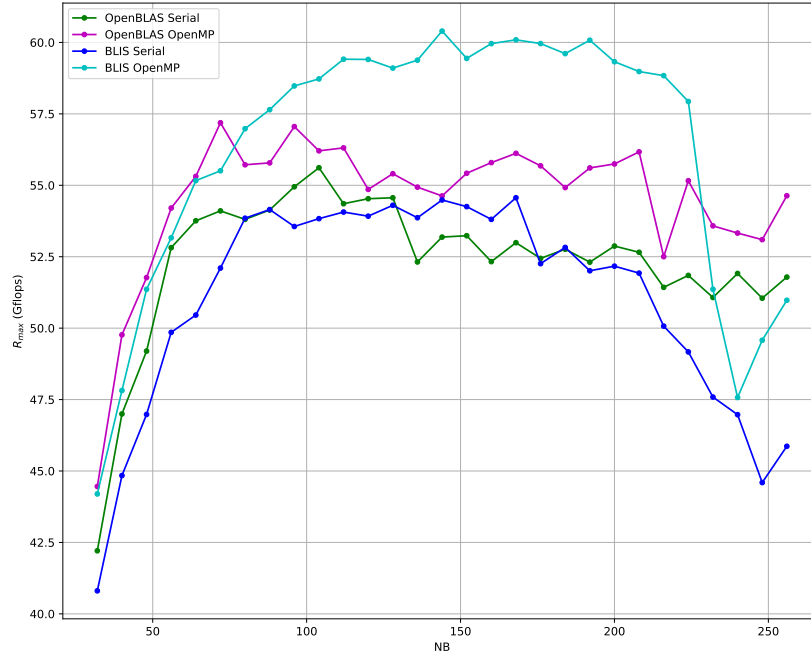

Figure 5.5: Pi Cluster Tools `linpack-profiler` R_{max} with *Adaptive RX* interrupt coalescing enabled.

Observations and discussion...

Using the HPL parameters which achieved optimum performance running the `linpack-profiler` tool, the HPL benchmark results using 80% of available memory are presented in Table ??.

					R_{max} (Gflops)	
BLAS	**N**	**NB**	**P**	**Q**	**Baseline**	**Adaptive RX Coalescing**
OpenBLAS Serial						
OpenBLAS OpenMP						
BLIS Serial						
BLIS OpenMP						

Table 5.13: Comparison of Baseline HPL R_{max} and that achieved with *Adaptive RX* interrupt coalescing enabled, using all 8 cluster nodes and 80% of total memory.

Observations and discussion...

5.5.2 Receive Packet Steering and Receive Flow Steering

HPL was run using all 8 nodes with Receive Packet Steering (RPS) and Receive Flow Steering (RFS) enabled. The results are tabulated and compared with the default RPS and RFS disabled in Table ??.

					R_{max}	
BLAS	**N**	**NB**	**P**	**Q**	**RPS & RFS Disabled**	**RPS & RFS Enabled**
OpenBLAS Serial	52320	120	1	32	8.6787e+01	
	52320	120	2	16	9.5517e+01	
	52320	120	4	8	9.5525e+01	
OpenBLAS OpenMP	52360	88	1	8	8.8626e+01	
	52360	88	2	4	9.6630e+01	
BLIS Serial	52416	168	1	32	9.1148e+01	
	52416	168	2	16	1.0341e+02	
	52416	168	4	8	1.0190e+02	
BLIS OpenMP	52400	200	1	8	9.1180e+01	
	52400	200	2	4	1.0041e+02	

Table 5.14: Comparison of HPL R_{max} with Receive Packet Steering (RPS) and Receive Flow Steering (RFS) enabled and the default RPS and RFS disabled.

Observations and discussion...

5.5.3 Kernel Preemption Model

The Linux kernel has 3 Preemption Models, as discussed in Chapter 2:

- Preemptive

- Voluntary Preemption

- No Forced Preemption

For scientific computing workloads the *No Forced Preemption* model should be used, as suggested by the *help* accompanying the Linux kernel configuration utility:

"This is the traditional Linux preemption model, geared towards throughput. It will still provide good latencies most of the time, but there are no guarantees and occasional longer delays are possible. Select this option if you are building a kernel for a server or scientific/computation system, or if you want to maximise the raw processing power of the kernel, irrespective of scheduling latencies."

The kernel installed by Ubuntu 20.04 LTS 64-bit uses the *Voluntary Preemption* model. To use the *No Forced Preemption* model the kernel needs to be recompiled. Detailed instructions on how to do this are included in the *Kernel Build With No Forced Preemption* `picluster` repository wiki page.

The Pi Cluster Tools `linpack-profiler` tool was run with a *No Forced Preemption* kernel. The results are plotted in Figure ??.

Using the parameters which achieved optimum performance when running the `linpack-profiler` tool, the HPL benchmark results using 80% of available memory are presented in Table ??.

					R_{max} (Gflops)	
BLAS	**N**	**NB**	**P**	**Q**	**Baseline**	**No Forced Preemption**
OpenBLAS Serial	52416	112	2	16		9.5723e+01
OpenBLAS OpenMP	52416	72	2	4		9.3790e+01
BLIS Serial	52320	120	2	16		1.0021e+02
BLIS OpenMP	52320	160	2	4		

Table 5.15: Comparison of the Baseline HPL R_{max} with a *No Forced Preemption* kernel.
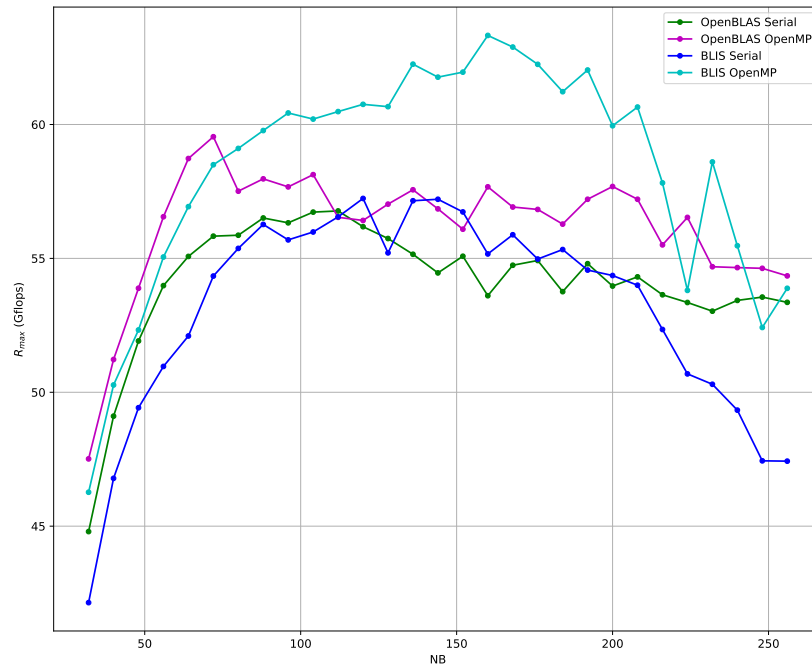
Figure 5.6: Pi Cluster Tools `linpack-profiler` R_{max} with a *No Forced Preemption* kernel.

Observations and discussion...

?

```
$ sudo perf record mpirun -allow-run-as-root -np 4 xhpl
```

Running xhpl on 8 nodes using OpenBLAS...

```
$ mpirun -host node1:4 ... node8:4 -np 32 xhpl
```

SHORTLY AFTER PROGRAM START...

On node1,... where we initiated...

top...

```
top - 20:33:15 up 8 days,  6:02,  1 user,  load average:
    ↪ 4.02, 4.03, 4.00
Tasks: 140 total,   5 running, 135 sleeping,   0 stopped,
    ↪ 0 zombie
```

```
%Cpu(s): 72.5 us, 21.7 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0
    ↪ hi,  5.8 si,  0.0 st
MiB Mem :   3793.3 total,    330.1 free,   3034.9 used,
    ↪ 428.3 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.
    ↪ 698.7 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM
       ↪        TIME+ COMMAND
  34884 john      20   0  932964 732156    7980 R 100.3  18.8
    ↪   106:40.29 xhpl
  34881 john      20   0  933692 732272    7916 R 100.0  18.9
    ↪   107:29.75 xhpl
  34883 john      20   0  932932 731720    8136 R  99.3  18.8
    ↪   107:33.25 xhpl
  34882 john      20   0  932932 731784    8208 R  97.7  18.8
    ↪   107:33.64 xhpl
```

SOFTIRQS...

NODE 2 - 2 NODES ONLY TO SEE EFFECT...

IPERF!!!

On node8, running the top command...

```
$ top
```

We can see...

```
top - 18:58:44 up 8 days,  4:29,  1 user,  load average:
    ↪ 4.00, 3.75, 2.35
Tasks: 133 total,   5 running, 128 sleeping,   0 stopped,
    ↪ 0 zombie
%Cpu(s): 50.7 us, 47.8 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0
    ↪ hi,  1.4 si,  0.0 st
MiB Mem :   3793.3 total,    392.7 free,   2832.6 used,
    ↪ 568.0 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.
    ↪ 901.1 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM
       ↪        TIME+ COMMAND
  23928 john      20   0  883880 682456    8200 R 100.0  17.6
    ↪   13:14.17 xhpl
  23927 john      20   0  883988 682432    7932 R  99.7  17.6
    ↪   13:12.58 xhpl
  23930 john      20   0  883912 682664    7832 R  99.7  17.6
    ↪   13:17.01 xhpl
```

```
  23929 john       20   0  883880 682640    8376 R  99.3  17.6
     ↪     13:16.25 xhpl
```

Indicates that only 50.7% of CPU time is being utilised by user programs (us), Linpack/OpenMPI...

I hypothesise that the 1.4% of software interrupts (si) is responsible 47.8% of CPU time in the kernel (sy) servicing these interupts...

Lets have a look at the software interrupts on the system...

```
$ watch -n 1 cat /proc/softirqs
```

```
Every 1.0s: cat /proc/softirqs

                   CPU0         CPU1         CPU2         CPU3
        HI:           0            1            0            1
     TIMER:   122234556     86872295     85904119     85646345
    NET_TX:   222717797       228381       147690       144396
    NET_RX:  1505715680         1132         1294         1048
     BLOCK:       63160        11906        13148        11223
  IRQ_POLL:           0            0            0            0
   TASKLET:    58902273           33            2            6
     SCHED:     3239933      3988327      2243001      2084571
   HRTIMER:        8116           55           53           50
       RCU:     6277982      4069531      4080009      3994395
```

As can be seen...

1. the majority of software interrupts are being generated by network receive (NET_RX) activity, followed by network transmit activity (NET_TX)...

2. these interrupts are being almost exclusively handled by CPU0...

What is there to be done?...

1. Reduce the numbers of interrupts...

1.1 Each packet produces an interrupt - interrupt coalesing...

1.2 Reduce the number of packets - increase MTU...

2.1 Share the interrupt servicing activity evenly across the CPUs...

5.5.4 Network Optimisation

On node2 start the Iperf server...

```
$ iperf -s
```

On node1 start the Iperf client...

```
$ iperf -c
```

ping tests of MTU...

iperf network speed...

Jumbo Frames

Requires a network switch capable of Jumbo frames...

On `node2` create the `Iperf` server...

```
$ iperf -s
```

On `node1` create and run the `Iperf` client...

```
$ iperf -i 1 -c node2
```

```
------------------------------------------------------------
Client connecting to node2, TCP port 5001
TCP window size:  682 KByte (default)
------------------------------------------------------------
[  3] local 192.168.0.1 port 46216 connected with
    ↪ 192.168.0.2 port 5001
[ ID] Interval        Transfer      Bandwidth
[  3]  0.0-10.0 sec  1.15 GBytes   991 Mbits/sec
```
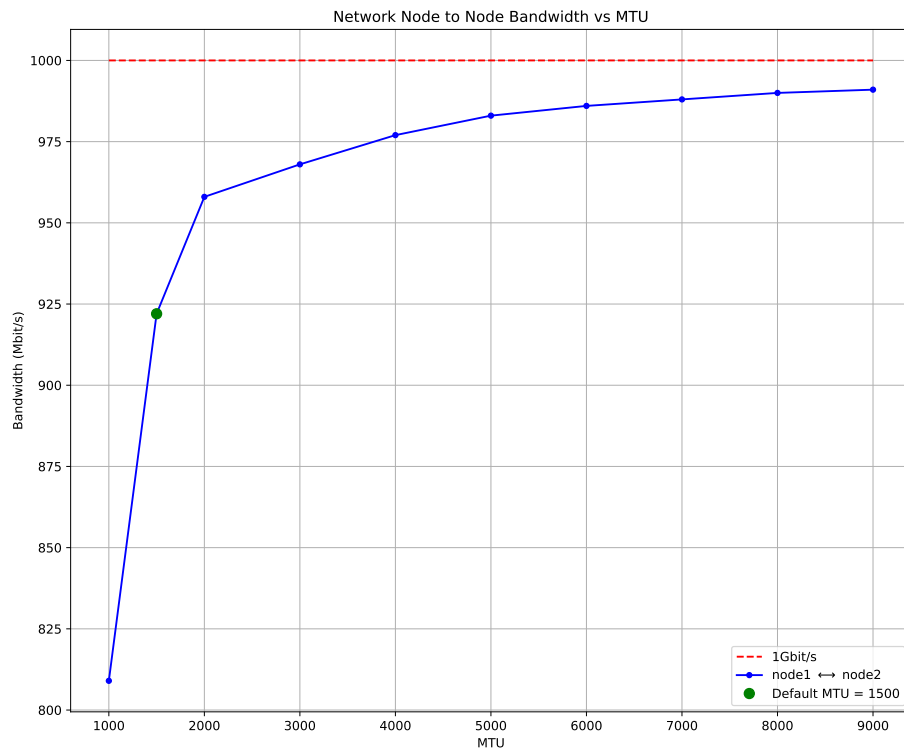
Figure 5.7: Network Node to Node Bandwidth vs MTU.

Chapter 6

Summary

References

[1] Fujitsu. *Fugaku Specification*. 2020. URL: https://www.fujitsu.com/global/about/innovation/fugaku/specifications.

[2] Arm Holdings. *Arm Powering the Fastest Supercomputer*. 2020. URL: https://www.arm.com/company/news/2020/06/powering-the-fastest-supercomputer.

[3] David A. Patterson and Carlo H. Sequin. "RISC I: A Reduced Instruction Set VLSI Computer". In: *Proceedings of the 8th Annual Symposium on Computer Architecture*. ISCA '81. Minneapolis, Minnesota, USA: IEEE Computer Society Press, 1981, pp. 443–457.

[4] Green500 List Editors. *Green500 List*. 2020. URL: https://www.top500.org/lists/green500/list/2020/06/.

[5] Volker Strassen. "Gaussian elimination is not optimal". In: *Numerische Mathematik* 13.4 (1969), pp. 354–356. DOI: 10.1007/BF02165411. URL: https://doi.org/10.1007/BF02165411.

[6] Don Coppersmith and Shmuel Winograd. "Matrix multiplication via arithmetic progressions". In: *Journal of Symbolic Computation* 9.3 (1990). Computational algebraic complexity editorial, pp. 251–280. ISSN: 0747-7171. DOI: https://doi.org/10.1016/S0747-7171(08)80013-2. URL: http://www.sciencedirect.com/science/article/pii/S0747717108800132.

[7] Virginia Vassilevska Williams. "Multiplying Matrices Faster than Coppersmith-Winograd". In: *Proceedings of the Forty-Fourth Annual ACM Symposium on Theory of Computing*. STOC '12. New York, New York, USA: Association for Computing Machinery, 2012, pp. 887–898. ISBN: 9781450312455. DOI: 10.1145/2213977.2214056. URL: https://doi.org/10.1145/2213977.2214056.

[8] François Le Gall. "Powers of Tensors and Fast Matrix Multiplication". In: *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation*. ISSAC '14. Kobe, Japan: Association for Computing Machinery, 2014, pp. 296–303. ISBN: 9781450325011. DOI: 10.1145/2608628.2608664. URL: https://doi.org/10.1145/2608628.2608664.

[9]   Jack Dongarra, Piotr Luszczek, and Antoine Petitet. "The LINPACK Benchmark: past, present and future". In: (2003).

[10]   Lloyd Nicholas Trefethen and David Bau III. *Numerical Linear Algebra*. Philadelphia: Society for Industrial and Applied Mathematics, 1997.

[11]   Jack Dongarra, Michael Heroux, and Piotr Luszczek. "HPCG Benchmark: a New Metric for Ranking High Performance Computer Systems". In: (2015).

[12]   Jonathan Richard Shewchuk. "An Introduction to the Conjugate Gradient Method Without the Agonizing Pain". In: (1994).