Benchmarking a Raspberry Pi 4 Cluster
MSc Scientific Computing
UCL

John Duffy

September 2020

Declaration

I, John Duffy, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the thesis.

Abstract

Dedication

To my wife Arlene-Marie and my daughter Aerin, without whose love and support my MSc would not have been possible.

Acknowledgements

I want to thank my supervisor, Professor Timo Betcke, for initiating this project, and for his guidance throughout all stages of the project.

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 consumption 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.

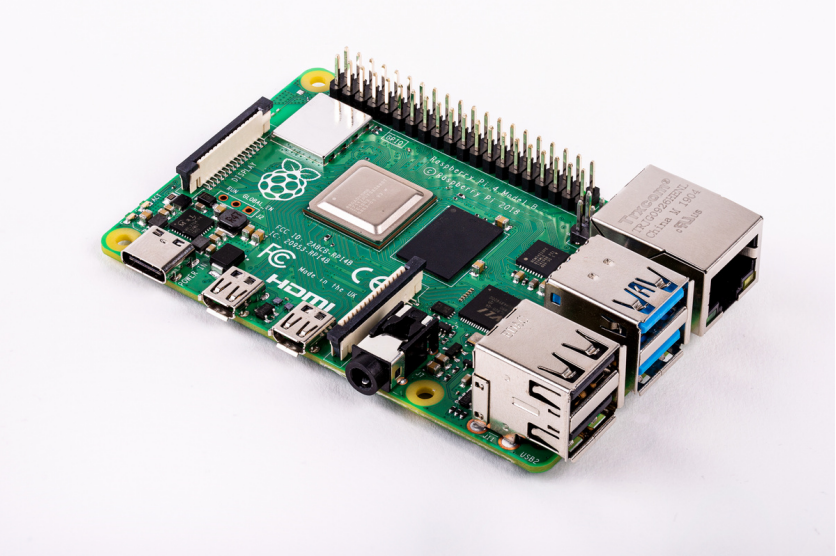


Figure 1.1: **The Raspberry Pi 4 Model B** [**4**].

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.

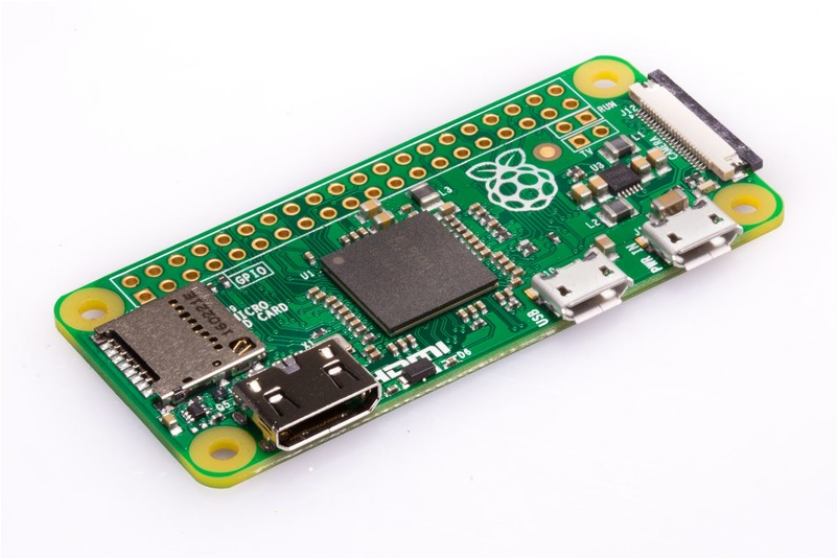


Figure 1.2: **The Raspberry Pi Zero** [**5**].

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*. Raspberry Pi OS is also available in a *Lite* version, without a desktop environment, for use in servers and embedded systems.

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.

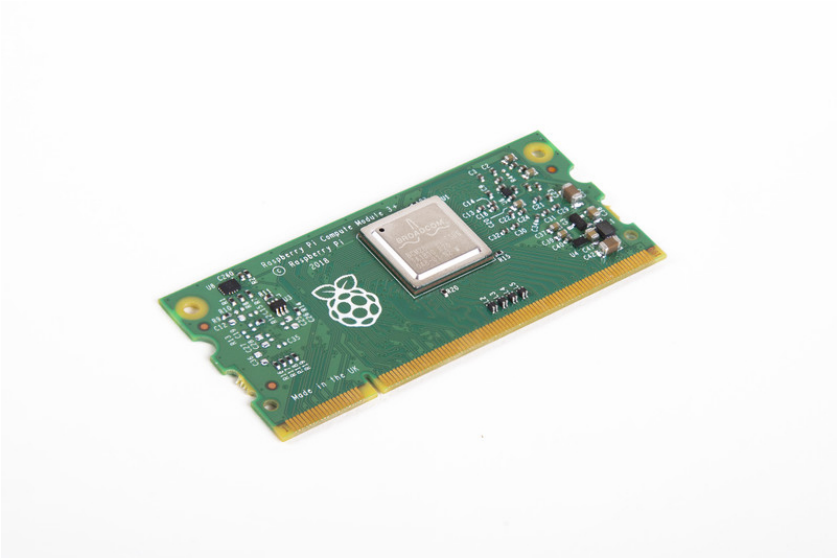


Figure 1.3: **The Raspberry Pi Compute Module 3+ [6]**.

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 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

- **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.
- **Investigate performance optimisation.** Having determined a *baseline* performance, opportunities for performance optimisations were investigated for single core, single node, two node and whole cluster configurations. Network optimisation was also investigated.
- **Investigate energy efficiency.** The Green500 List ranks computer systems by energy efficiency. In June 2020, ranking number 1, was the *MN-3* by Preferred Networks in Japan, which achieved a record 21.1 Gigaflops/Watt [7]. Ranking 200 was *Archer* at the University of Edinburgh, which achieved 0.497 Gflops/Watt [7]. The final aim of this project was to investigate the energy efficiency of the Aerin Cluster in comparison to computer systems included in 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" [8] 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 far exceeds that of even a high-end desktop.

The UK Met Office uses a number of grids to model global 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 point outer grid, both with 70 vertical levels [9]. To model the atmosphere on these grids the UK Met Office currently uses three Cray XC40 supercomputers [10], capable of 14 Petaflops (10^{15} *floating point operations per second*), and which contain 460,000 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, processing power per unit of electricity 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 High Performance Conjugate Gradients (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. High Performance Conjugate Gradients is intended to measure the performance of a computer system when solving large sparse linear system systems, which is typical of modern HPC workloads.

To put benchmark results into context, and to extrapolate from benchmark 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 data in 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, i.e. in parallel, one on each core. 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 is 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 interface *hardware interrupt* on the receiving node. This *interrupt* requires *servicing* by the kernel, which involves a context switch from the benchmark process to the kernel. This can take a considerable amount of time, and may 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 interface upon receipt of a data packet, or may be generated by software, *software interrupts*.

To maintain system responsiveness, and to ensure subsequent interrupts are not missed whilst processing the current interrupt, *interrupt service routines* are kept as short as possible. Linux, and other operating systems, use a *top-half/botton-half* mechanism to service interrupts. The *top-half* responds to the interrupt, but only carries out the essential minimum processing to service the interrupt, and then schedules the *bottom-half*, which is less time sensitive, to conduct the remaining processing required to service the interrupt.

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

The effect of interrupts generated by network activity on benchmark performance is 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 *no forced preemption*. 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 *no forced preemption* 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 default preemption model of the kernel installed with Ubuntu 20.04 LTS 64-bit is *voluntary preemption*. The recompilation of the Linux kernel with *no forced preemption* 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, 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 *address*, 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 2^{64}. 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

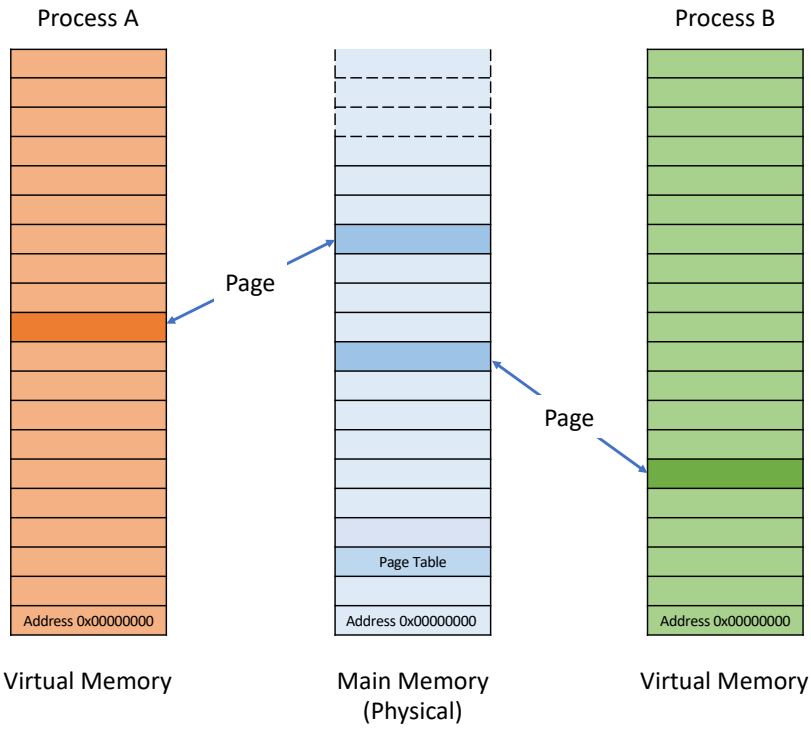


Figure 2.1: **Virtual Memory**. Each process has a *virtual address* space mapped to main memory in *pages* by a *page table* which resides in main memory. A smaller page table called the *Translation Lookaside Buffer* (TLB) is a *cache* in close proximity to each core. The TLB enables fast lookup of physical page addresses without resorting to a slower lookup in the main memory page table.

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*.

2.2.9 Virtual Memory

Virtual memory is the abstracted view of main memory presented to a process 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 using the finite 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 data packets 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. The affect on benchmark performance of these methodologies was investigated as part of the project.

2.3.1 MTU

The MTU (Maximum Transmission Unit) is the maximum size of a network packet (sometimes referred to as a frame). 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 (LAN), 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 do not block 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 packet overhead, but potentially at the cost of increased 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, the packet is placed in a buffer to be processed by the kernel. The interface informs the kernel of the receipt of the packet via a hardware interrupt, which results in a context switch to the kernel. 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 raising a hardware interrupt until a specified number of packets has been received, or a specified time has elapsed. Re-

ceived packets are placed in a queue until a hardware interrupt is subsequently raised. 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 network interface hardware support and also network driver support, and is configured using the Linux `ethtool` command.

2.3.3 Receive Side Scaling

By default, the hardware 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 network interface which supports multiple receive queues, it is possible to assign a receive queue to each CPU core, and to configure the interface to spread interrupt handling across the cores. This is called *Receive Side Scaling* (RSS) [11]. 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 network interfaces 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 interfaces support multiple receive queues, or the driver may not have implemented this functionality. *Receive Packet Steering* (RPS) [11] is a software implementation of RSS, which works with a single receive queue. The Raspberry Pi 4 Model B currently only supports a single receive queue. The network interface may support multiple receive queues, but this is not enabled in the open source driver.

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) [11] has the potential to improve benchmark performance by improving data locality.

Building upon the distribution of network interrupt servicing across multiple cores by RSS/RPS, 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 any packet, and then may need to pass it on to the core that requires it. This spreads the interrupt processing workload but does not improve data locality. By enabling RFS on top of RSS/RPS, 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 microprocessors have become an increasingly popular choice for High Performance Computing (HPC). This is 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 microprocessor, topped the June 2020 TOP500 List and also the November 2019 Green500 List.

Arm microprocessors are based on 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 and variable length instructions. The simplicity of design, which directly translates to a lower transistor count, results in high performance combined with low power requirements.

The ARMv8-A instruction set has included *Advanced SIMD* instructions since initial release. Each subsequent revision to the instruction set has seen improvements related to HPC. In 2014, ARMv8.1-A introduced enhancements to the Advanced SIMD instructions. In 2016, ARMv8.2-A introduced *Scalable Vector Instructions*, permitting variable length vectors of size 128 to 2048 bits to be used. ARMv8.6-A announced in 2019 introduces General Matrix Multiply instructions, and also SIMD matrix instructions.

2.5 HPC 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, using 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 computational complexity of $\mathbf{C} = \mathbf{AB}$, where $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C} \in \mathbf{R}^{n \times n}$, using a naive algorithm 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 [12] 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

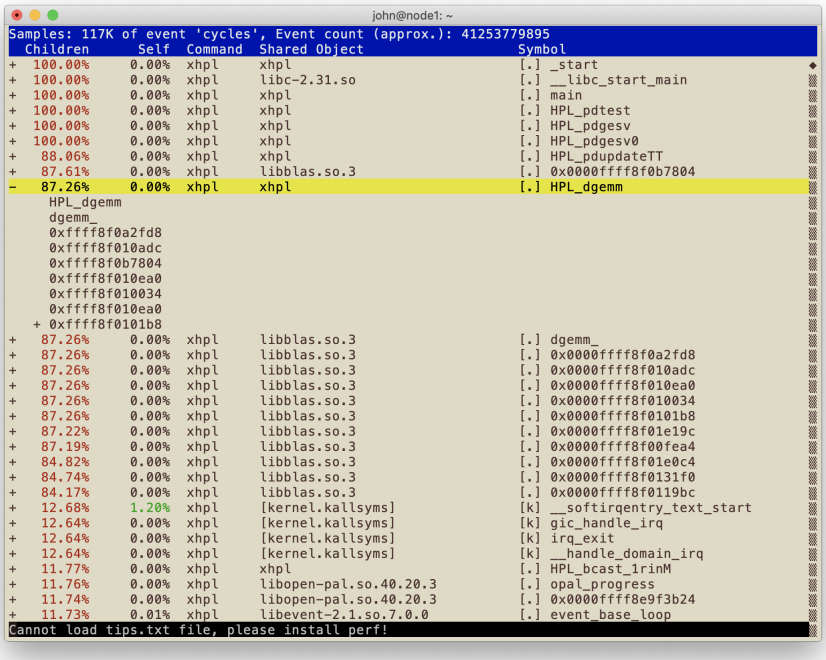


Figure 3.1: Output of `sudo perf report` generated from recording the events of an `xhpl` process using `sudo perf record -p 9020 -g -- sleep 30`, where `9020` is the `xhpl` process identifier. As indicated, 87.26% of the process time is spent in the `HPL_dgemm` function, which calls the BLAS `dgemm`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

matrix 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 [13] further reduced the complexity to $\mathcal{O}(n^{2.375477})$. The *Coppersmith-Winograd Algorithm* was further improved by Virginia Vassilevska Williams [14] in 2012, and most recently in 2014 by François Le Gall [15], which reduced the complexity to $\mathcal{O}(n^{2.3728639})$.

However, the reductions in computational complexity beyond the naive algorithm also add implementation complexity, and potential numerical instability. The *Strassen Algorithm*, for example, introduces *subtraction* operations which are a source of rounding errors. And the benefits may not outweigh the implementation complexity unless the matrices are very large. For this reason the naive algorithm is 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.

Matrix-matrix multiplication is a BLAS (Basic Linear Algebra Subroutines) Level 3 operation. BLAS libraries such as GotoBLAS (no longer under active development), and it's derivatives OpenBLAS and BLIS, achieve high performance *matrix-matrix multiplication* through effective use of the L1/L2/L3 caches. The inherent looping over *dot product* operations is carefully structured so that data is retained in the caches for maximum reuse.

3.2 High Performance Linpack (HPL)

The High Performance Linpack (HPL) Benchmark [16] 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:

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

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*, $2/3n^3 + \mathcal{O}(n^2)$ [16]. 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:

$$\mathbf{Ux} = \mathbf{y} \tag{3.4}$$

$$\mathbf{Ly} = \mathbf{b} \tag{3.5}$$

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).

3.2.2 Block LU Factorisation

A *block algorithm* is an algorithm which operates on *blocks* of data instead of scalar values [17]. In the case of *block LU factorisation*, a matrix $\mathbf{A} \in \mathbf{R}^{n \times n}$ is partitioned into *blocks* $\mathbf{A}_{ij} \in \mathbf{R}^{nb \times nb}$ and factorised into $\mathbf{L}_{ij}, \mathbf{U}_{ij} \in \mathbf{R}^{nb \times nb}$.

$$\mathbf{A} = \begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} \tag{3.6}$$

$$= \begin{pmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & L_{33} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{pmatrix} \tag{3.7}$$

Dense matrices are easily partitioned into *blocks*, and the subsequent LU decomposition, which uses optimised BLAS Level 3 *matrix-matrix multiplication*, can be distributed across multiple processes using MPI for parallel processing.

3.2.3 The HPL Algorithm

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

$$\mathbf{Ax} = \mathbf{b}, \text{ where } \mathbf{A} \in \mathbf{R}^{n \times n} \text{ and } \mathbf{x}, \mathbf{b} \in \mathbf{R}^{n} \tag{3.8}$$

The HPL algorithm [18] perform LU factorisation of $\mathbf{A}$ with partial row pivoting. The $\mathbf{Ly} = \mathbf{b}$ forward substitution is conducted during the factorisation process. The $\mathbf{Ux} = \mathbf{y}$ backward subsitution is the final step.

The matrix $\mathbf{A}$ is partitioned into *blocks* of dimension $nb \times nb$, where nb is the *block size*.

For optimum performance the problem dimension n should be selected to utilise all available memory, and be an integer multiple of nb. Equation (3.9) can be used to determine the problem dimension n for a block size nb and a required total memory usage.

$$n = \left[\left(memory\_percent \sqrt{\frac{memory\_in\_gb \times 1024^3}{8}} \right) \div nb \right] \times nb \tag{3.9}$$

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

The system is solved by distributing the *blocks* over a two dimensional $P \times Q$ grid of processors on a cluster of *distributed memory* computers [16]. MPI is used for *block* distribution.

In a pure MPI topology, each $P \times Q$ grid element represents a processor core. In a hybrid OpenMPI/OpenMP topology, each grid element represents a cluster node, and OpenMP is used to distributes data between the cores.

The LU factorisation progresses in columns of processes, of width nb.

3.3 High Performance Conjugate Gradients (HPCG)

Like HPL, the HPCG benchmark, developed by Dongarra, Heroux and Luszczek [19], 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:

$$\mathbf{Ax} = \mathbf{b}, \text{ where } \mathbf{A} \in \mathbf{R}^{n \times n} \text{ and } \mathbf{x}, \mathbf{b} \in \mathbf{R}^{n} \tag{3.10}$$

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 [19].

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 [20].

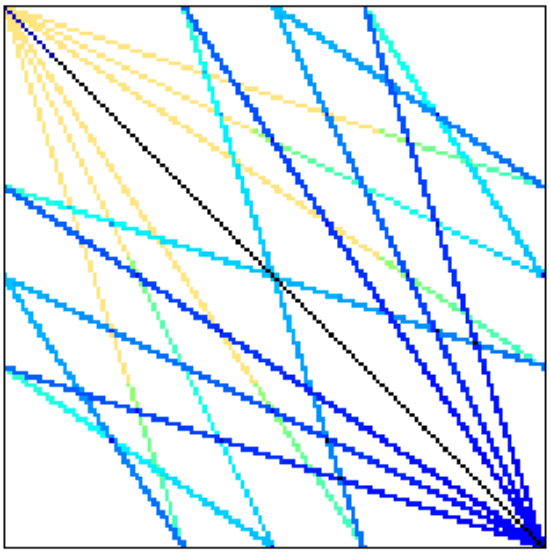
And finally... discuss the blocking/implementation of HPCG... and why CG is used.

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

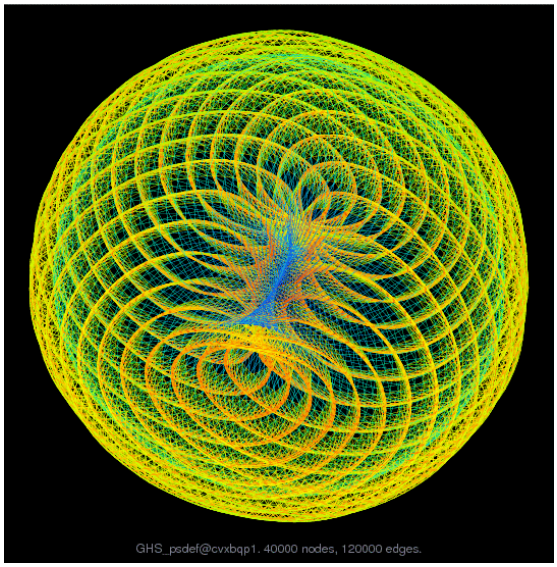
A *quadratic form* is a quadratic function of a vector $\mathbf{x}$, whose return value is a scalar. It is of the form:

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x^T Ax} - \mathbf{x^T b} + c \tag{3.11}$$

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



(a) **Sparse Matrix.** Sparse matrix `cvxbqp1` generated from a convex... quadratic programming proble.



(b) **Graph of Sparse Matrix cvxbqp1.**

Figure 3.2: **Sparse Matrix Collection...**

Consider the derivatives of (3.16):

$$\begin{aligned} \nabla f(\mathbf{x}) &= \mathbf{A}\mathbf{x} - \mathbf{b} \\ \nabla^2 f(\mathbf{x}) &= \mathbf{A} \end{aligned} \tag{3.12}$$

Equation (3.16) is minimised when $\nabla f(\mathbf{x}) = 0$, i.e. $\mathbf{A}\mathbf{x} = \mathbf{b}$, 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 $\mathbf{A}\mathbf{x} = \mathbf{b}$ 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:

$$\mathbf{x}_1 = \mathbf{x}_0 + \alpha_0 \mathbf{r}_0 \tag{3.13}$$

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:

$$\mathbf{r}_i = -\nabla f(\mathbf{x_i}) = \mathbf{b} - \mathbf{A}\mathbf{x_i} \tag{3.14}$$

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 pure OpenMPI and hybrid OpenMPI/OpenMP cluster topologies. Finally, a description of Pi Cluster Tools, a suite of scripts which make command line management of the cluster easier and less prone to error is also included.

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

4.1 Hardware

The Aerin Cluster consists of the following hardware components:

- 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

Figure 4.1: **The Aerin Cluster**.

4.1.1 Raspberry Pi 4 Model B

The 9 x Raspberry Pi 4 Model B's 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 cluster compute nodes are `node1` to `node8`. These nodes are used to run benchmarks. Some benchmarks take a substantial amount of time to run, so it is convenient to have a dedicated build node for compiling software, etc, while benchmarks are running. This build node is `node9`.

It is also convenient to have one of the compute nodes designated the "master" compute node. This is `node1`. Software which needs to be compiled locally to the compute nodes, and not on the build node, is compiled on the "master" node. This node is also used to mirror the GitHub `picluster` repository and to run the tools from the Pi Cluster Tools suite.

4.1.2 Power Supplies

The Raspberry Pi 4 is sensitive to voltage drops, especially whilst booting. So it was decided to use 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 occupy some space, so a future development would be to investigate a suitably rated USB power 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. And the "A1" 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. The Raspberry Pi 4 Model B throttles back the clock speed at approximately 85°C, which would not only have had a negative impact on benchmark results, but also on repeatability. So, it was very important to select suitable cooling. After some investigation, it was decided to use 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 Raspberry Pi 4 Model B has a single Gigabit Ethernet interface. The theoretical maximum bandwidth of this interface is *1 Gigabits per second*. As observed during benchmarking, the Raspberry Pi 4 is capable of utilising almost all of this bandwidth. It is therefore important that all networking equipment and cabling supports Gigabit Ethernet, otherwise the network performance of the cluster would be unnecessarily degraded.

4.1.6 Router/Firewall

The router/firewall acts as the Aerin Cluster interface to the outside world. One side of the firewall is the cluster LAN (Local Area Network), on which the compute nodes and `node9` are connected. The other side of the firewall is the WAN (Wide Area Network). The firewall only permits specifically configured network packets from the WAN through the firewall to the LAN. The Aerin Cluster is configured to only permit `ssh` packets through the firewall, which are then routed to `node1`.

The router exposes a single IP address to the WAN. Access to the cluster is through this single IP address. In my home environment the WAN is connected to my ADSL router via an Ethernet cable. This permits the compute nodes to connect to the internet and download updates. When relocated to UCL the WAN would be connected to the internal UCL network.

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.

The router/firewall is easily configured through a web-based setup. Details on how to do this are included in the `picluster` repository wiki.

4.1.7 Network Switch

The network switch acts as an extension to the number of Ethernet 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.

4.1.8 Cabling

Cat 5 network cabling only support 100 Mbit/s. Cat 5e and Cat 6 supports 1 Gbit/s, but not necessarily with electrical shielding. 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.

The Aerin Cluster uses Cat 7 cabling for optimum network performance.

4.2 Software

The Aerin Cluster consists of the following software components.

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. Detailed instructions for installing the operating system are included in the `picluster` repository wiki.

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 can also be used for automating the installation of the same operating system on a cluster of computers using a single installation image.

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 the above is done from a single installation image and `user-data` file.

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

- build-essential
- openmpi-bin
- libopenblas0-serial
- libopenblas0-openmp
- libblis3-serial
- libblis3-openmp

The `build-essential` package installs essential software build tools, such as C/C++ compilers and `make`. The `openmpi-bin` package installs the OpenMPI binary and development files. And the OpenBLAS and BLIS libraries install both the serial and OpenMP versions of the respective libraries.

4.2.3 Benchmark Software

The HPL, HPCC and HPCG benchmark software is compiled locally from source. The instructions for how to do this are included in the `picluster` repository wiki.

4.3 BLAS Libraries

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/L2/L3 CPU caches. These kernels typically reflect the size of the CPU registers, and L1/L2/L3 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 Aerin Cluster BLAS Libraries

To enable comparison between BLAS library implementations, OpenBLAS and BLIS, in both serial and OpenMP versions, are installed on the Aerin Cluster. For benchmark consistency, and repeatability, it is essential that the same

BLAS library in use on each node at the same time. Two tools from the Pi Cluster Tools suite, `libblas-query` and `libblas-set`, simplify BLAS library management.

4.4 Cluster Topologies

4.4.1 Pure OpenMPI

In a pure OpenMPI topology, work is distributed across the processor cores of each node by OpenMPI. Each core runs a single MPI work process. This is depicted in Figure 4.2(a).

A work process can be *bound* to a specific core. This is an optimisation which reduces *cache refreshes* when a work process is interrupted and is then subsequently re-scheduled.

Each core of the Aerin Cluster supports a single thread of execution. Therefore, in this topology, work processes are linked against the serial, single-threaded, versions of the BLAS libraries.

4.4.2 Hybrid OpenMPI/OpenMP

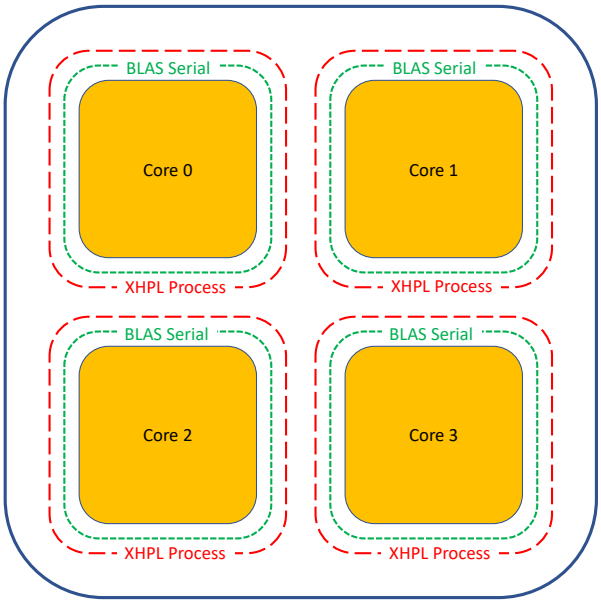
In a hybrid OpenMPI/OpenMP topology, OpenMPI is used to distribute work between cluster nodes. Each node runs a single MPI work process. OpenMP is then used to distribute the work of this single process between node cores. This is depicted in Figure 4.2(b).

Each node of the Aerin Cluster supports a multiple threads of execution, one on each core. Therefore, in this topology, work processes are linked against the OpenMP, multi-threaded, versions of the BLAS libraries.

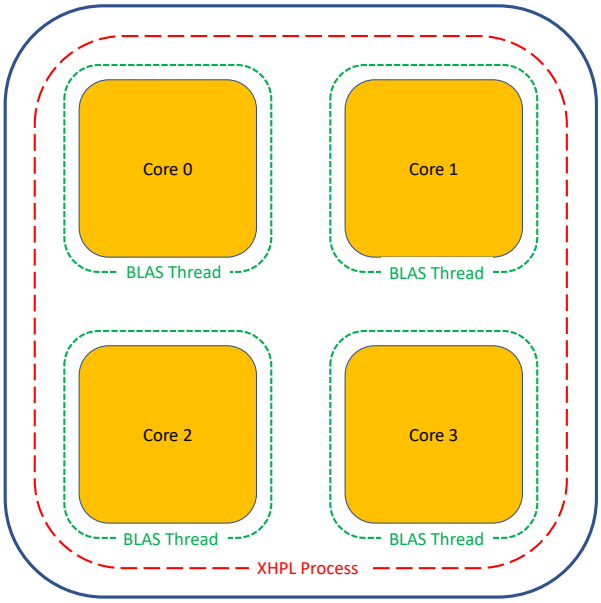
4.5 Pi Cluster Tools

Command line administration of a cluster of computers is repetitive and prone to error. To work around this problem, and to make administration of the Aerin Cluster easier, a selection of `bash` scripts were written and called *Pi Cluster Tools*. Each tool loops over a list of nodes and uses `ssh` to invoke a remote action on each node in turn.

The Pi Cluster Tools scripts should be invoked from `node1`.



(a) Pure OpenMPI



(b) Hybrid OpenMPI/OpenMP

Figure 4.2: **Single Node Toplologies.**

Pi Cluster Tools consists of the following 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 Results and Optimisations

5.1 Theoretical Maximum Performance

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

Each core of the Arm Cortex-A72 implements the 64-bit Armv8-A ISA (Instruction Set Architecture). This instruction set includes Advanced SIMD instructions which operate on a single 128-bit SIMD pipeline. This pipeline can conduct two 64-bit double precision *floating point operations* 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 *add* operations. A single FMA instruction conducts two *floating point operations* per clock cycle.

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

$$R_{peak} = 4 \text{ cores} \times 1.5 \text{ GHz} \times 2 \text{ doubles} \times 2 \text{ FMA} \tag{5.1}$$

$$= 24 \text{ Gflops} \tag{5.2}$$

This R_{peak} of 24 Gflops is only achievable, continuously, if every instruction in a program is an FMA instruction, and the program data is aligned in memory appropriately for efficient access. This obviously cannot be the case, since a program will consist of at least some instructions to load data from memory and store results back in memory, and these are not FMA instructions. Therefore, R_{peak} is very much a 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 maximum performance, the HPL benchmark requires a problem size which utilises 100% of memory. But, because the operating system requires memory, this is never fully achievable.

Figure 5.1: **Available Memory**. Output from `dmesg | grep Memory` indicates the memory usage by the Linux kernel, and the memory available to applications and benchmarks

As can be seen in Figure 5.1, 3.6 GB of memory (3,783,864 KB) is available to applications and benchmarks per node. This equates to 90% of the total 4 GB (4,194,304 KB) per node. Any transient use of more than 90% of memory will result in memory pages being swapped to permanent storage, which will negatively impact benchmark performance.

Therefore, for the HPL baseline benchmarks, 80% of available memory was chosen for the problem size. This is also the amount suggested as an initial *good guess* in HPL Frequently Asked Questions [21].

The above necessarily results in the baseline performance only being able to achieve 80% of R_{peak}, at best. This is 4.8 Gflops for a single core, 19.2 Gflops

for a single node, 38.4 Gflops for two nodes, and 153.6 Gflops for the 8 node cluster. These values are indicated on the HPL baseline performance plots.

5.2 HPL Baseline

Detailed instructions on how to install the HPL benchmark software, and how to run the HPL benchmark are included in the project repository wiki.

To establish *baseline* performance, the HPL benchmark was run using the default Ubuntu 20.04 LTS 64-bit packages, and without any system or network tuning.

Baseline performance was investigated for the single core, single node, two node and whole cluster configurations.

5.2.1 HPL 1 Core Baseline

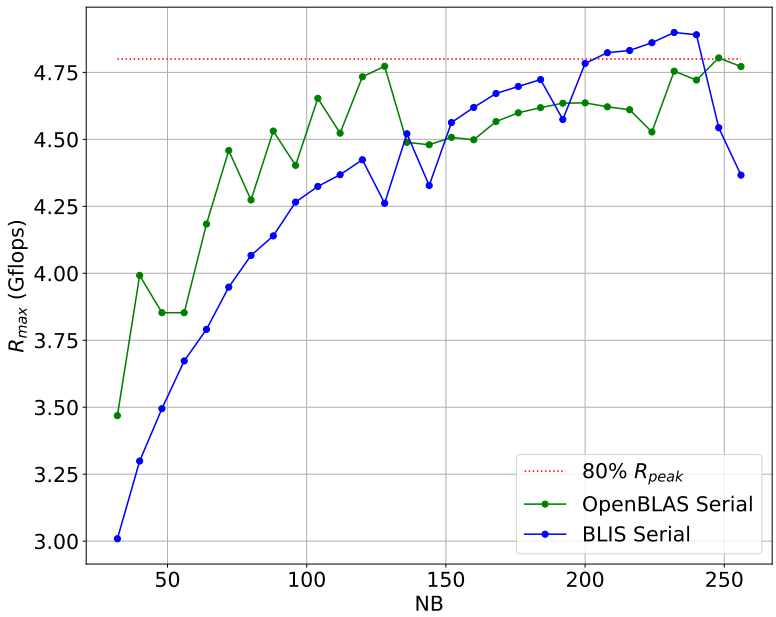
The purpose of this investigation is to determine the performance of a single core running a single `xhpl` process, with the single core having exclusive access to the shared L2 cache.

As discussed in the previous section, the HPL problem size is restricted to 80% of available memory. In the case, 80% of a single node's 4 GB. Using values of block size NB from 32 to 256, as suggested by HPL Frequently Asked Questions [21], and using equation 3.9 to ensure the problem size N is an integer multiple of NB, results in Table 5.1 of NB and N combinations.

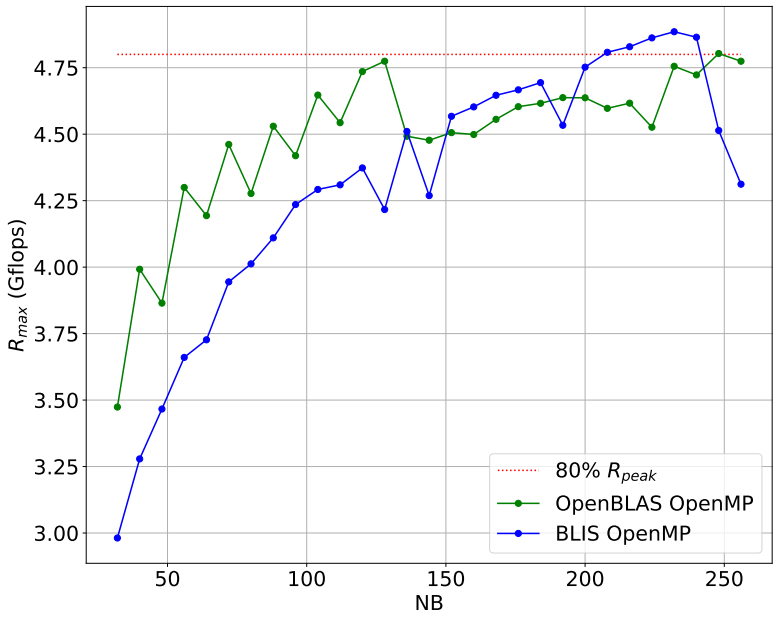
NB	**N**	**NB**	**N**	**NB**	**N**	**NB**	**N**	**NB**	**N**
32	18528	80	18480	128	18432	176	18480	224	18368
40	18520	88	18480	136	18496	184	18400	232	18328
48	18528	96	18528	144	18432	192	18432	240	18480
56	18536	104	18512	152	18392	200	18400	248	18352
64	18496	112	18480	160	18400	208	18512	256	18432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

Table 5.1: **1 Core NB and N Combinations.** Block size NB and problem size N combinations for NB between 32 and 256 using 80% of 4 GB of memory.

The benchmark results are plotted in Figure 5.2.



(a) Pure OpenMPI



(b) Hybrid OpenMPI/OpenMP

Figure 5.2: **HPL 1 Core R_{max} versus NB.**

Note: There is no actual benefit in using a hybrid OpenMPI/OpenMP topology for a single core running a single `xhpl` process, as only a single thread is used. However, to ensure similar results were achieved, both pure OpenMPI and hybrid OpenMPI/OpenMP topologies were benchmarked.

Observations

As expected, there is no significant performance difference between the two topologies for both OpenBLAS and BLIS.

OpenBLAS and BLIS both attain 80% R_{peak}. Without competition from additional cores for access to the L2 cache, both libraries are able to efficiently stream data from main memory, through the L1 and L2 caches, to the core registers.

5.2.2 HPL 1 Node Baseline

The purpose of this investigation is to determine the performance of the 4 cores of a single node. In this case each core shares the L2 cache with the other cores, so less L2 data will be available per core. This should result in more L2 *cache misses* requiring a *cache load* from main memory. 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, this is 80% of a single node's 4 GB. This results in the same NB and N combinations as the single core benchmark of Table 5.1.

The benchmark results are plotted in Figure 5.3.

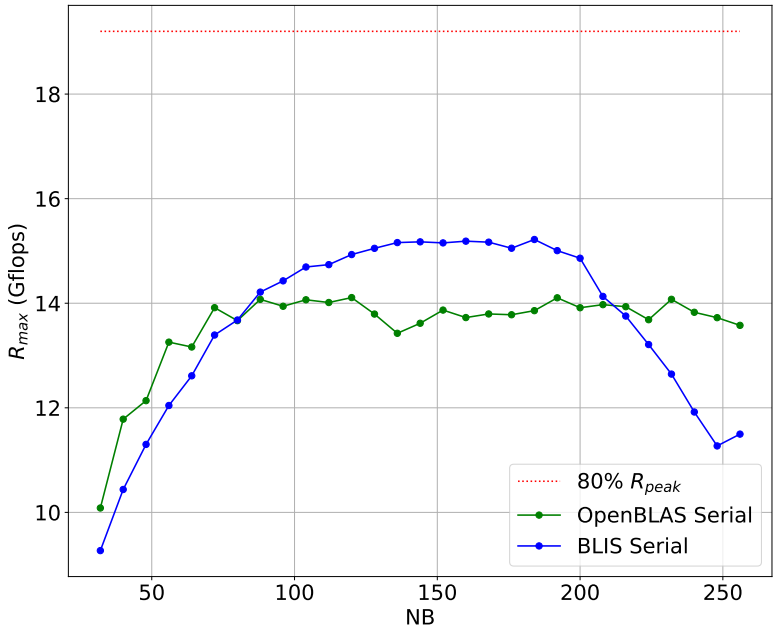
Observations

As anticipated, there is indeed a reduction in performance per core. The 80% of R_{peak}, 19.2 Gflops for the combined 4 cores, in no longer attained.

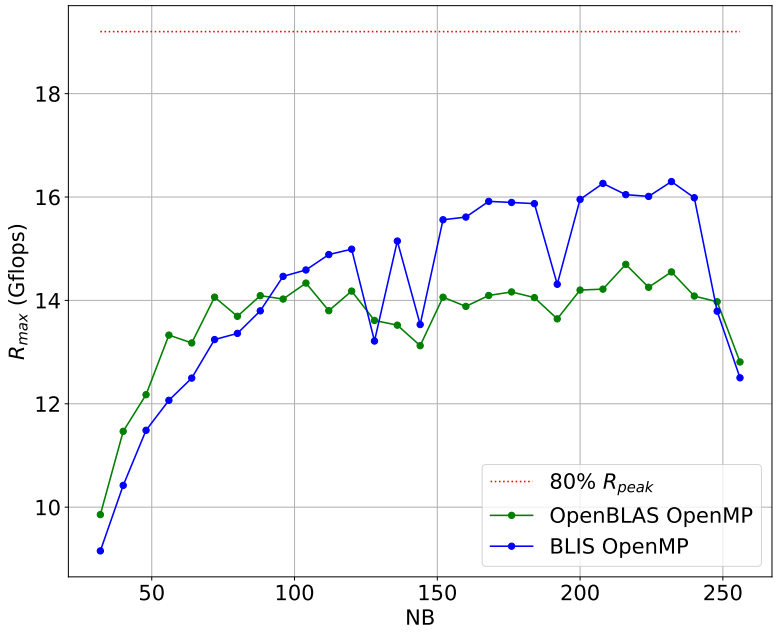
The pure OpenMPI topology attains an R_{max} of 15.219 Gflops using the BLIS library with an NB of 184.

The hybrid OpenMPI/OpenMP topology attains an R_{max} of 16.299 Gflops using the BLIS library with an NB of 232.

The above represent 79.27% and 84.89% of 80% R_{peak}, respectively.



(a) Pure OpenMPI



(b) Hybrid OpenMPI/OpenMP

Figure 5.3: **HPL 1 Node R_{max} versus NB.**

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 will be loaded 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 the NB and N combinations in Table 5.2.

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 5.2: **2 Node NB and N Combinations.** Block size NB and problem size N combinations for NB between 32 and 256 using 80% of 8 GB of memory

The benchmark results are plotted in Figure 5.4

Observations

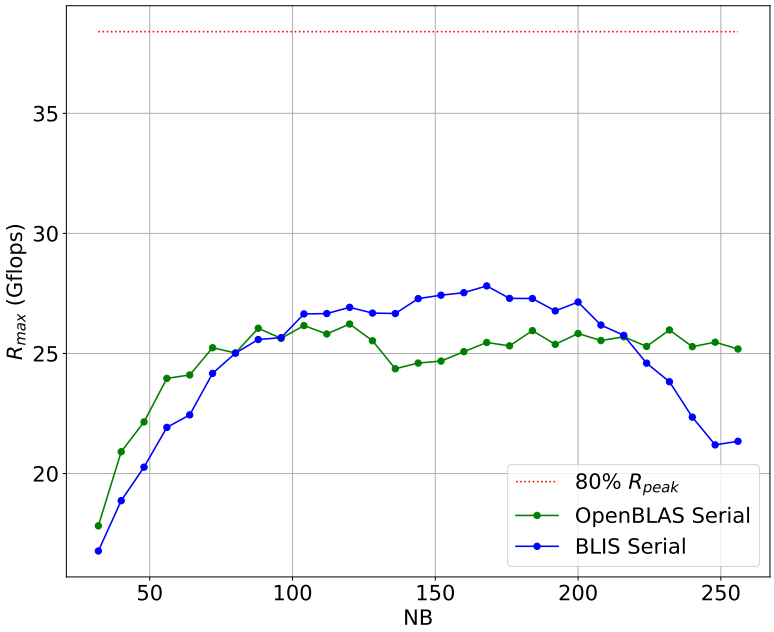
For 2 nodes 80% R_{peak} is 38.4 Gflops.

The pure OpenMPI topology attains an R_{max} of 27.814 Gflops using the BLIS library with an NB of 168.

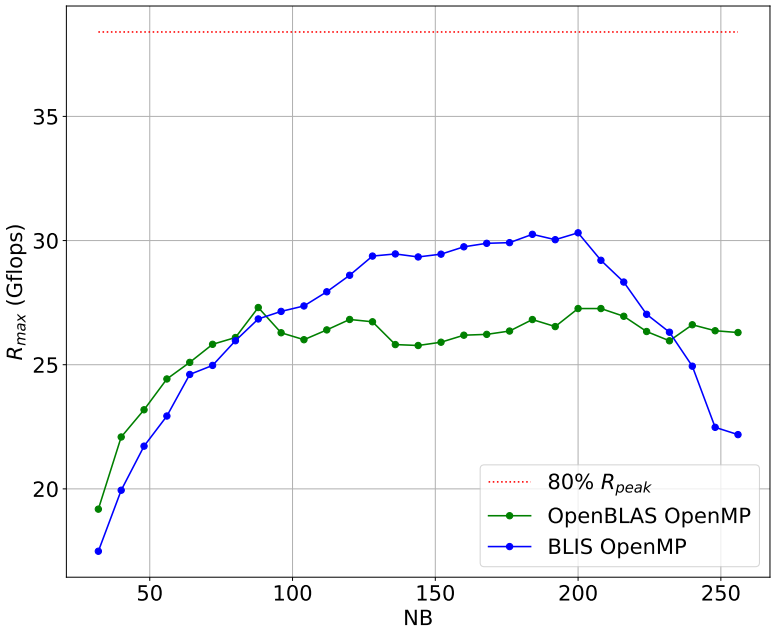
The hybrid OpenMPI/OpenMP topology attains an R_{max} of 30.312 Gflops using the BLIS library with an NB of 200.

The above represent 72.43% and 78.94% of 80% R_{peak}, respectively.

Compared to the single node case, 2 node performance is reduced by 17.8% for the pure OpenMPI topology, and 15.5% for the hybrid OpenMPI/OpenMP topology, due to inter-node MPI process communication and network overhead.



(a) Pure OpenMPI



(b) Hybrid OpenMPI/OpenMP

Figure 5.4: **HPL 2 Node R_{max} versus NB.**

5.2.4 HPL Whole Cluster Baseline

This whole cluster baseline uses the values of NB from the 2 node baseline which attained optimum performance.

The benchmark results are presented in Tables 5.3 and 5.4. In these tables, for each node count, there is also a value of R_{max} for each processor grid shape. This is because within each N, NB, P and Q parameter set, there are also HPL parameters which have a lesser effect on performance. R_{max} is the maximum observed performance for each set of N, NB, P and Q parameters.

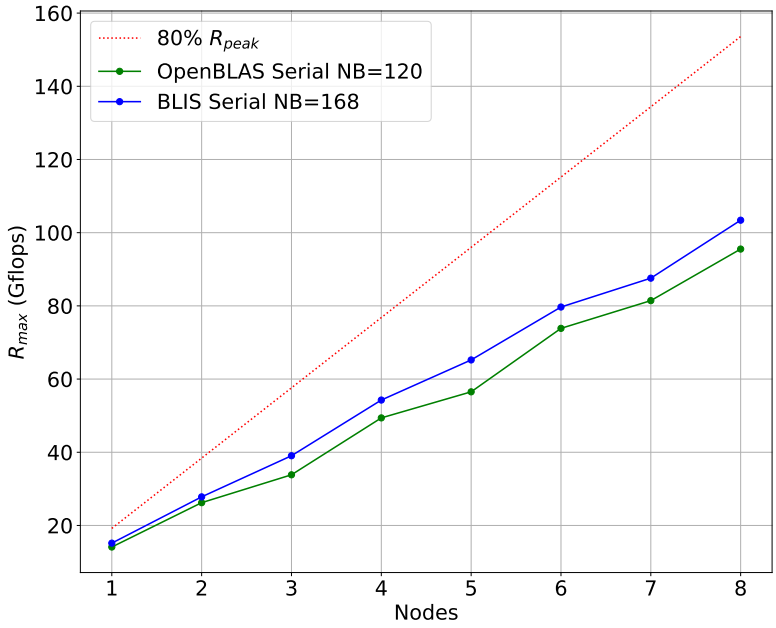
The highest value of R_{max} for each node count is then plotting in Figure 5.5.

BLAS Library	**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

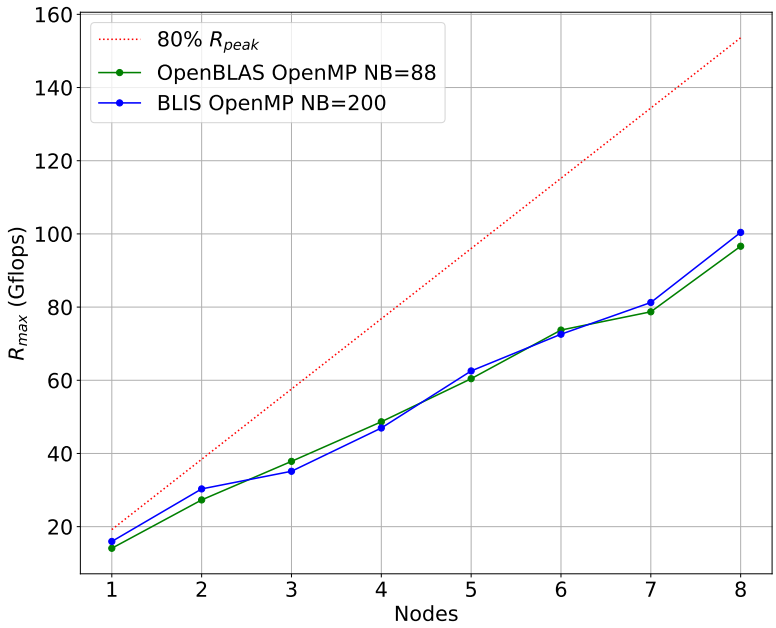
Table 5.3: **HPL Whole Cluster OpenBLAS Baseline.**

BLAS Library	**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 5.4: **HPL Whole Cluster BLIS Baseline.**



(a) Pure OpenMPI



(b) Hybrid OpenMPI/OpenMP

Figure 5.5: **HPL R_{max} versus Node Count.**

5.2.5 Observations

Performance scales linearly with node count.

Where there are multiple P x Q processor grid combinations available, optimum performance is achieved with a P x Q grid which is not *flat*. For example, 2 x 16 and 4 x 8 grids performs better than a 1 x 32 grid. This is consistent with HPL Frequency Asked Questions [21], and is related to the computation to communication ratio of the cluster.

For a high performance network such as Infiniband, optimum performance is expected to be achieved with the most *square* P x Q grid. In the previous example, this would be the 4 x 8 grid. Ethernet is not a high performance network, in HPC terms, and optimum performance is expected to be achieved from less *square*, but not *flat* processor, grid.

A combination of *square* and *less square* processor grids produced optimum performance for the Aerin Cluster. This is attributable to the good network performance of the Raspberry Pi 4, which can utilise almost the full 1 Gigabit of available bandwidth.

For 8 nodes, 80% R_{peak} is 153.6 Gflops.

For 8 nodes, the highest R_{max} of 103.41 Gflops was observed using the BLIS Serial library with an NB of 168, and a P x Q grid of 2 x 16. This is 67.32% of the theoretical maximum performance using 80% of memory.

5.3 HPCC Baseline

The HPCC benchmark suite was run using all 8 nodes of the Aerin Cluster to obtain a whole cluster baseline.

Recall from Chapter 2, *single* indicates the benchmark is run on 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.

The results for each benchmark are presented below.

5.3.1 HPL

HPL is included in HPCC. The performance results when running HPL as part of HPCC were similar to those when running HPL as a standalone benchmark.

5.3.2 DGEMM

BLAS Library	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 5.5: **HPCC DGEMM.**

The DGEMM benchmark measures the performance of double precision real *matrix-matrix multiplication*. The benchmark results are presented in Table 5.5.

For the single-threaded serial versions of the OpenBLAS and BLIS libraries the cluster consists of 32 processing cores. The `SingleDGEMM_Gflops` results are per core.

For the multi-threaded OpenMP versions of the libraries, the cluster consists of 8 processing nodes. The `SingleDGEMM_Gflops` results are per node.

Observations

The results are consistent with the HPL benchmarks, which spends approximately 87% of the benchmark run time in the BLAS `dgemm` subroutine.

Of note is the *jitter* between the *single* and *star* results, particularly the OpenBLAS Serial and BLIS Serial results. This is explained by the fact that a *single* randomly selected core running the benchmark has exclusive access to the L2 cache. The *single* result is consistent with 1 Core Baseline results. In the *star* case, all cores on each node are running the benchmark and have to share access to the L2 cache. The *star* result is consistent with the 1 Node Baseline results.

5.3.3 STREAM

The STREAM benchmark measures sustained memory bandwidth by performing four vector operations, *Copy*, *Scale*, *Sum* and *Triad*, on vectors which are at least 4 times the size of the L2 cache. This ensures the benchmark is measuring main memory access performance.

For the purposes of measuring pure memory bandwidth the *Copy* operation is the most appropriate. This measures the copying of a vector from one memory location to another, without any computation on the vector data.

The STREAM benchmark results are presented in table 5.6.

Observations

As noted previously, the *single* OpenBLAS Serial and BLIS Serial benchmarks relate to single core. And as indicated in the results, the maximum observed single core memory bandwidth is approximately 5.4 MB/s. The *star* serial results of approximately 0.94 GB/s need to be factored by the node count to measure the whole node main memory bandwidth, which is then the same order of magnitude but reduced due to L2 cache collisions.

For bandwidth measurement, the STREAM benchmark counts both the memory read and the memory write as a memory movement. This differs from most memory bandwidth benchmarks in which the read and write from one memory location to another count as a single memory movement. Therefore, for the *Copy* operation, the benchmark results needs to be factored by 0.5 to align with other benchmarks and memory specifications.

For three out of the 4 BLAS library combinations, the observed *Copy* vector operation bandwidth is approximately 5.4 GB/s.

The Raspberry Pi 4 Model B is equipped with LPDDR4–3200 SDRAM (Low-Power Double Data Rate Static DRAM), with a maximum data transfer rate of 3200 MB/s (3.2 GB/s).

The observed benchmark *Copy* performance of 5.4 GB/s, when factored by 0.5 is 2.7 GB/s. This is 80% of the maximum data transfer rate. This suggests that the Raspberry Pi 4 Model B is making good use of the available memory bandwidth.

BLAS Library	**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 5.6: **HPCC STREAM.**

BLAS Library	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 5.7: **HPCC PTRANS.**

5.3.4 PTRANS

PTRANS is a *global* benchmark which implements the transpose of a large matrix in memory using the cluster nodes operating in parallel. The main purpose of this benchmark is to test inter-node communication performance.

The value of NB was selected to be optimum value from the 2 Node Baseline, with the corresponding value of N equating to 40% of total memory. Since this not a computation benchmark, 40% memory usage is sufficient to test inter-node communication without excessive run time.

Observations

Both the OpenBLAS and BLIS serial benchmark runs produce similar results. Similarly, both OpenMP benchmark runs produce similar results. This is to be expected, since this benchmark is not testing BLAS library performance. The difference between the serial and OpenMP performance is related to the topology of the cluster.

With an average pure OpenMPI result of 0.475 GB/s, and a hybrid OpenMPI/OpenMP result of 0.627 GB/s, the inter-node communication performance is 32% faster using the hybrid topology.

This result is to be expected, since the node count is 32 in the pure topology, but only 8 in the hybrid topology. Even though the amount of matrix data to be transposed remains the same, the inter-node communication is reduced. The inter-node messages may be larger, but there are fewer messages with less node addressing overhead, making the cluster network more efficient.

5.3.5 Random Access

The Random Access benchmark tests the integer update performance of a large array in memory. Random numbers generated from a normal distribution and the Linear Congruential Generator algorithm are used. The unit of the results is GUPs (*GigaUpdates per Second*).

Observations

For the *global* benchmarks, the array size $N = 2,147,483,648$ represents 25% of the cluster's total 32 GB of memory.

For the *single* pure OpenMPI benchmarks, the array size $N = 67,108,864$ represents 25% of 1 GB of memory. Each of the 4 cores of each node is allocated 1 GB of the total 4 GB of memory, and the array is selected to be 25% of this.

And, for the *single* hybrid OpenMPI/OpenMP benchmarks, the array size $N = 268,435,456$ represents 25% of a single nodes's 4 GB of memory.

The above array sizes are determined by the benchmark and not selected by the user.

As expected, because this benchmark is not a test of computational performance, the pure OpenMPI topology performance is almost identical for both BLAS libraries. Similarly, the hybrid OpenMPI/OpenMP topology performance is almost identical for both BLAS libraries.

However, the cluster topology does make a difference between the *global* benchmarks. For the pure topology the update rate 0.00064 GUP/s for both random number algorithms. And, the hybrid topology update rate is 0.00047 GUP/s for both algorithms. The pure topology is 43% faster for the same array size. This is likely to be due to the pure topology having more MPI processes to update the array.

BLAS Library	**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 5.8: **HPCC Random Access.**

5.3.6 FFT

The FFT benchmark results are presented in Table 5.9.

BLAS Library	**Results**
OpenBLAS Serial	Global Vector size: 268435456 Global Gflops: 1.569
	Star Vector size: 16777216 Star Min Gflops: 0.251138 Star Avg Gflops: 0.314311 Star Min Gflops: 0.335671
	Single Gflops: 0.494388
OpenBLAS OpenMP	Global Vector size: 268435456 Global Gflops: 1.347
	Star Vector size: 67108864 Star Min Gflops: 0.314714 Star Avg Gflops: 0.344635 Star Min Gflops: 0.375736
	Single Gflops: 0.532823
BLIS Serial	Global Vector size: 268435456 Global Gflops: 1.563
	Star Vector size: 16777216 Star Min Gflops: 0.252763 Star Avg Gflops: 0.312753 Star Min Gflops: 0.344300
	Single Gflops: 0.515476
BLIS OpenMP	Global Vector size: 268435456 Global Gflops: 1.313
	Star Vector size: 67108864 Star Min Gflops: 0.315112 Star Avg Gflops: 0.329216 Star Min Gflops: 0.344110
	Single Gflops: 0.439117

Table 5.9: **HPCC FFT.**

Observations

The...

5.3.7 Network Bandwidth and Latency

The Network Bandwidth and Latency benchmark measures the time to send MPI messages between cluster processes. Latency is measured using 8 byte messages, and bandwidth is measured using 2,000,000 byte messages.

The major benchmark results are presented in table 5.10.

OpenBLAS Serial
Max Ping Pong Latency: 0.189604 msecs
Randomly Ordered Ring Latency: 0.192830 msecs
Min Ping Pong Bandwidth: 107.306358 MB/s
Naturally Ordered Ring Bandwidth: 61.385709 MB/s
Randomly Ordered Ring Bandwidth: 16.907255 MB/s
OpenBLAS OpenMP
Max Ping Pong Latency: 0.099864 msecs
Randomly Ordered Ring Latency: 0.075013 msecs
Min Ping Pong Bandwidth: 112.574381 MB/s
Naturally Ordered Ring Bandwidth: 70.511474 MB/s
Randomly Ordered Ring Bandwidth: 73.119334 MB/s
BLIS Serial
Max Ping Pong Latency: 0.191411 msecs
Randomly Ordered Ring Latency: 0.189172 msecs
Min Ping Pong Bandwidth: 107.075234 MB/s
Naturally Ordered Ring Bandwidth: 51.395031 MB/s
Randomly Ordered Ring Bandwidth: 16.492559 MB/s
BLIS OpenMP
Max Ping Pong Latency: 0.101356 msecs
Randomly Ordered Ring Latency: 0.070292 msecs
Min Ping Pong Bandwidth: 112.788813 MB/s
Naturally Ordered Ring Bandwidth: 99.948389 MB/s
Randomly Ordered Ring Bandwidth: 72.585888 MB/s

Table 5.10: **HPCC Network Bandwidth and Latency.**

The BLAS libraries are not used, but the results for all four library combinations are included for consistency with other benchmarks results.

A clear distinction needs to be made between Gigabits and Gigabytes/Megabytes. Using the standard *byte* of 8 bits, 1 Gigabits per second (1 Gb/s) is 125 Megabytes per second (125 MB/s). This is the theoretical maximum bandwidth of the Raspberry Pi 4's Gigabit Ethernet interface.

Observations

The benchmark results are specified in Megabytes per second (MB/s).

The average OpenMP topology bandwidth is 112.682 MB/s, which is 90% of the maximum theoretical bandwidth. The is consistent with 92.2% of *node-to-node* maximum theoretical bandwidth observed using the Linux `iperf` command in Section 5.54.

The average serial topology bandwidth is 107.548 MB/s, which is 86% of the maximum theoretical bandwidth. It is expected to observe a lower bandwidth with the serial topology due to the increased MPI process count compared to the the OpenMP topology.

5.4 HPCG Baseline

Similar to the TOP500 List, the HPCG List ranks computer systems in order of HPCG benchmark performance. The June 2020 HPCG List ranks 169 computer systems.

Ranking number 1 is the Fugaku supercomputer. And, ranking 169 is the Spaceborne Computer. The Spaceborne Computer, onboard the International Space Station (ISS), is a 32 core system based on the Intel Xeon E5-2620 v4 8 core CPU, clocked at 2.1GHz, with an Infiniband interconnect.

An extract from the list for these two computers is in Table 5.11.

HPCG Rank	**Name**	**Cores**	**HPL R_{max} Pflops**	**HPCG Pflops**	**Fraction of Peak**
1	Fugaku	6,635,520	415.530	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 HPCG benchmark results for the Aerin Cluster, obtained running the benchmark for 60 minutes using 75% of available memory, are presented in Table 5.12.

HPCG Rank	Name	Cores	HPL R_{max} Gflops	HPCG Gflops	Fraction of Peak
-	OpenBLAS Serial	32	9.5525e+01	3.49084	1.8%
-	OpenBLAS OpenMP	32	9.6630e+01	2.90942	1.5%
-	BLIS Serial	32	1.0341e+02	3.44246	1.8%
-	BLIS OpenMP	32	1.0041e+02	2.95205	1.5%

Table 5.12: **Aerin Cluster HPCG Benchmark Results.**

Observations

The HPCG benchmark performance is approximately 20% greater for a pure OpenMPI topology. This suggests it is more efficient to spread sparse data across a larger number of cores for processing. Since there is less opportunity for data blocking with sparse data, and less reuse of data already resident in caches, a finer grained approach to data distribution appears more efficient.

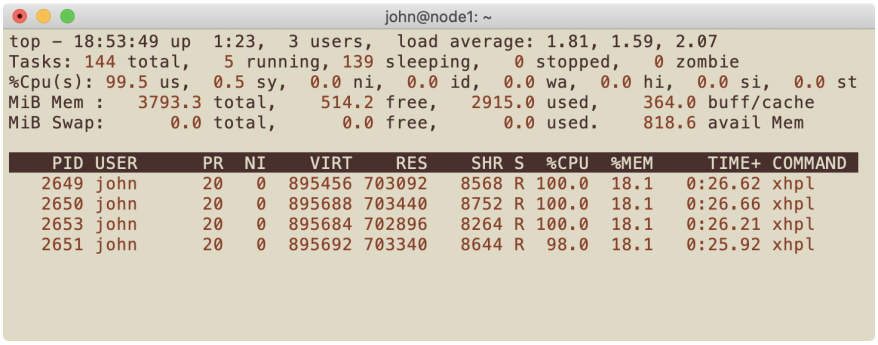
5.5 Optimisations

When running on a single node, as indicated in Figures 5.6, almost 100% of CPU time is spent in *user space*, i.e. running benchmark code. Therefore to gain benchmark performance improvement requires improvement to the architecture, algorithms, and/or supporting libraries, such as OpenBLAS and BLIS. Some work was done to improve OpenBLAS and BLIS performance on the Raspberry Pi 4, but this proved to be beyond the scope of this project (for the moment).

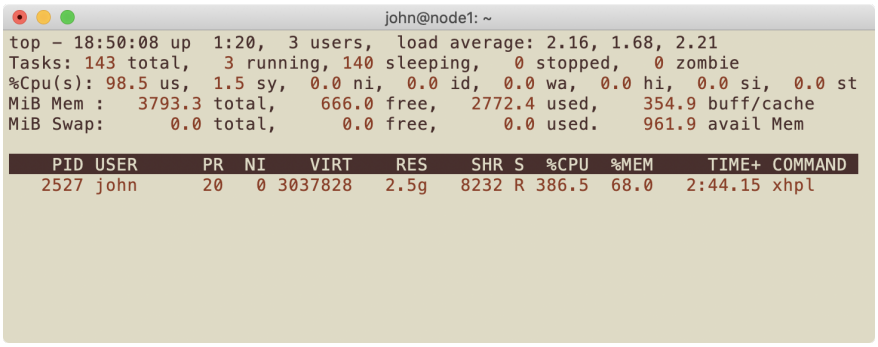
As seen in the 2 Node and Whole Cluster baseline results, when cores/nodes are networked together in cluster there is a reduction in performance per node. This is seen in Figure 5.7, where significant proportions of each node's CPU time is spent servicing *software interrupts* generated by network *receive* activity, and, in the case of OpenMP benchmarks, waiting for data.

The above observations suggested that network efficiency should be the focus of potential cluster optimisations.

The observations of Figure 5.7 can be summarised as follows:

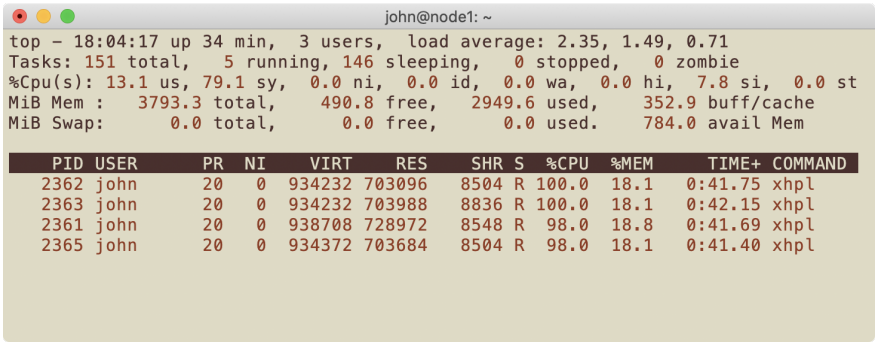


(a) Serial Benchmark.

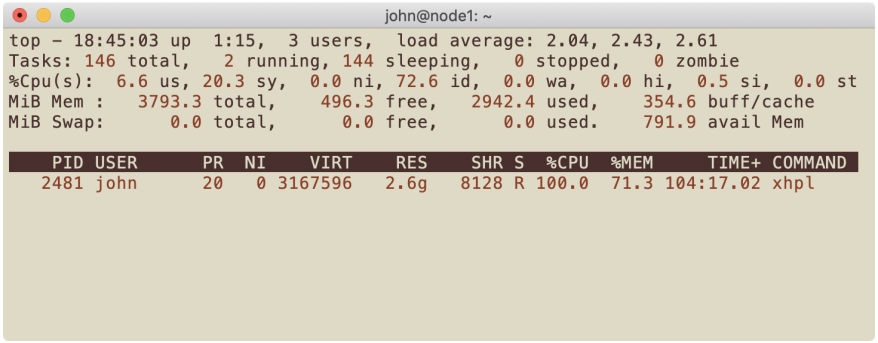


(b) OpenMP Benchmark.

Figure 5.6: **Single node `top` command output.** CPU time is almost exclusively spent in *user space* (`us`) when running both serial and OpenMP benchmarks. Minimal time is spent *idling* (`id`) or servicing software interrupts (`si`).



(a) Serial Benchmark.



(b) OpenMP Benchmark.

Figure 5.7: **8 node `top` command output.** Once networked together in a cluster, a significant proportion of each node's CPU time is spent in *kernel space* (`sy`) when running both serial and OpenMP benchmarks, and *idling* (`id`) in the OpenMP case. The images show observed worst case. Using `cat /proc/softirqs`, the software interrupt (`si`) CPU usage was observed to relate to network receive activity.

1. The majority of the time in *kernel space* is spent servicing software interrupts generated by network activity.
2. These interrupts are exclusively handled by CPU0 (core 0) of each node. This is seen using the Linux command `cat /proc/softirqs`.

The strategy for improving network performance was therefore decided to be:

1. Reduce the numbers of interrupts - enable *Interrupt Coalescing*
2. Distribute the interrupt servicing burden evenly across all node cores - enable *Receive Packet Steering*
3. Improve core data locality - enable *Receive Flow Steering*
4. Improve kernel processing throughput - implement a *No Forced Preemption* kernel.
5. Improve network efficiency - implement a Jumbo Frames enabled kernel.

The implementation of each of these optimisations is described in the following sections.

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 5.8.

Using the value of the block size NB which achieved optimum performance running the `linpack-profiler` tool, the HPL benchmark results using 80% of available memory are presented in Table 5.13.

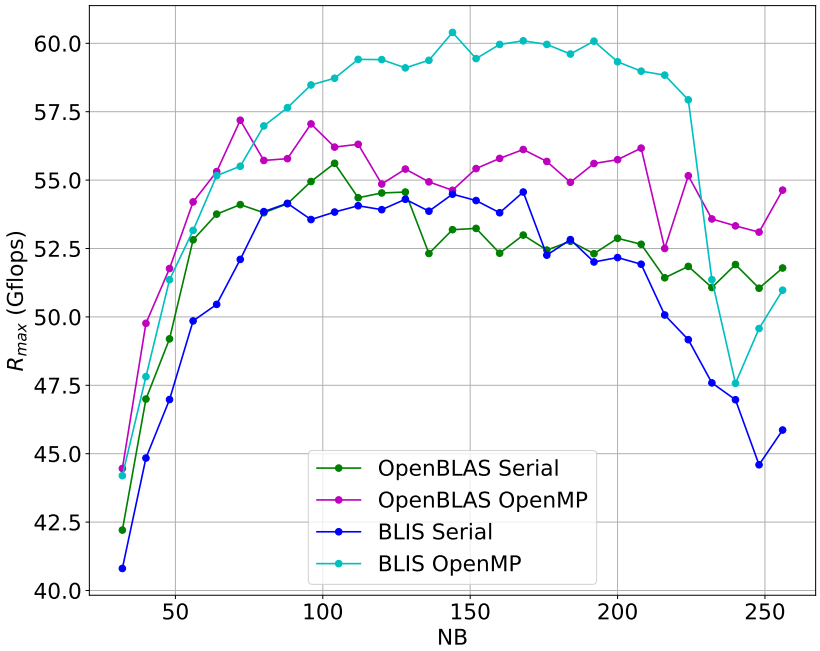


Figure 5.8: Pi Cluster Tools `linpack-profiler` R_{max} with *Adaptive RX* interrupt coalescing enabled.

					$\mathbf{R_{max}}$ **(Gflops)**	
BLAS	**N**	**NB**	**P**	**Q**	**Baseline**	**Adaptive RX Coalescing**
OpenBLAS Serial	52416	104	2	16	9.5525e+01	9.6078e+01
OpenBLAS OpenMP	52416	72	2	4	9.6630e+01	9.3222e+01
BLIS Serial	52416	168	4	8	1.0341e+02	1.0205e+02
BLIS OpenMP	52416	144	2	4	1.0041e+02	1.0582e+02

Table 5.13: **HPL $\mathbf{R_{max}}$ with *Adaptive RX* interrupt coalescing enabled.**

Observations

Observations and discussion...

5.5.2 Receive Packet Steering and Receive Flow Steering

The results of running the `linpack-profiler` tool with *Receive Packet Steering* and *Receive Flow Steering* enabled are plotted in Figure 5.9.

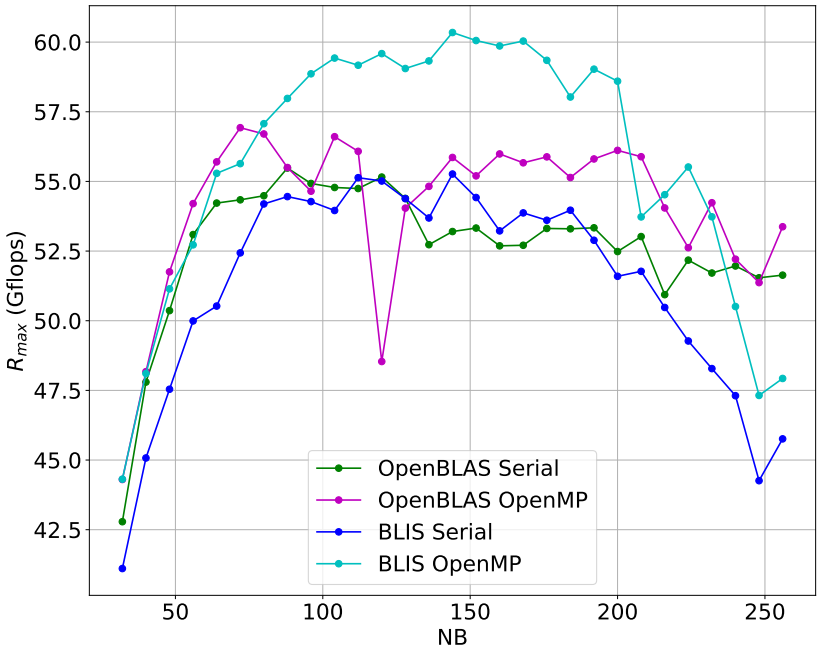


Figure 5.9: Pi Cluster Tools `linpack-profiler` R_{max} with Receive Packet Steering and Receive Flow Sterring enabled.

Using the value of the block size NB which achieved optimum performance running the `linpack-profiler` tool, the HPL benchmark results using 80% of available memory are presented in Table 5.14.

					R_{max} (Gflops)	
BLAS	**N**	**NB**	**P**	**Q**	**Baseline**	**Adaptive RX Coalescing**
OpenBLAS Serial	52360	88	2	16	9.5525e+01	9.5672e+01
OpenBLAS OpenMP	52416	72	2	4	9.6630e+01	9.2954e+01
BLIS Serial	52416	144	2	16	1.0341e+02	1.0117e+02
BLIS OpenMP	51416	144	2	4	1.0041e+02	1.0597e+02

Table 5.14: **HPL R_{max} with *Receive Packet Steering* and *Receive Flow Steering* enabled.**

Observations

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 5.10.

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 5.15.

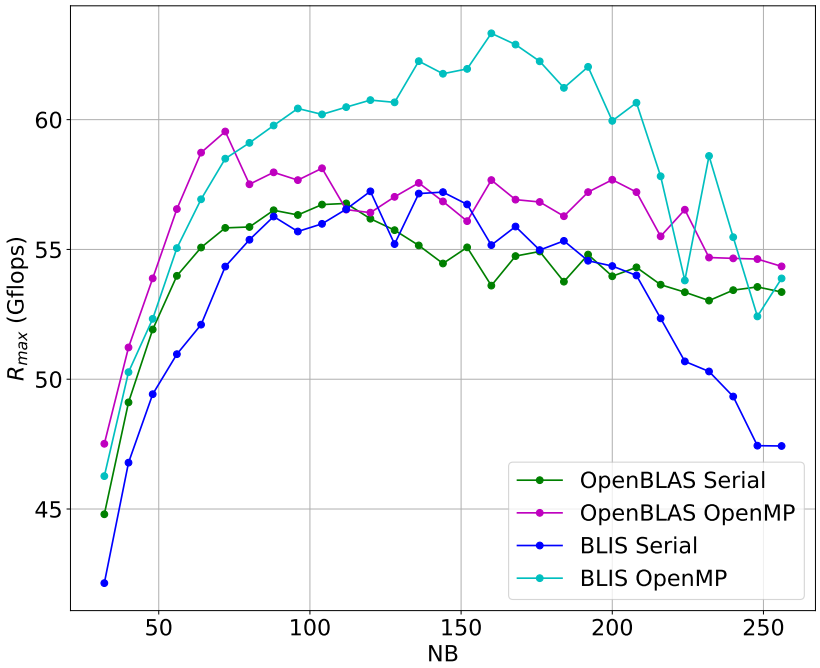


Figure 5.10: Pi Cluster Tools `linpack-profiler` R_{max} with a *No Forced Preemption* kernel.

					$\mathbf{R_{max}}$ **(Gflops)**	
BLAS	**N**	**NB**	**P**	**Q**	**Baseline**	**No Forced Preemption**
OpenBLAS Serial	52416	112	2	16	9.5525e+01	9.4729e+01
OpenBLAS OpenMP	52416	72	2	4	9.6630e+01	9.1107e+01
BLIS Serial	52320	120	2	16	1.0341e+02	9.9248e+01
BLIS OpenMP	52320	160	2	4	1.0041e+02	1.0694e+02

Table 5.15: Comparison of the Baseline HPL R_{max} with a *No Forced Preemption* kernel.

Observations

Observations and discussion...

5.5.4 Jumbo Frames

The kernel installed by Ubuntu 20.04 LTS 64-bit uses the standard MTU size of 1500 bytes. To enable Jumbo Frames requires modifications to the kernel Ethernet network driver, and a recompilation of the kernel. The network driver modifications requires a high level of kernel development expertise, and proved to be only partially successful. Detailed instructions on how to do this are included in the *Kernel Build With Jumbo Frames Support* project repository wiki page.

Figure 5.11 indicates the bandwidth improvement with increased MTU size.

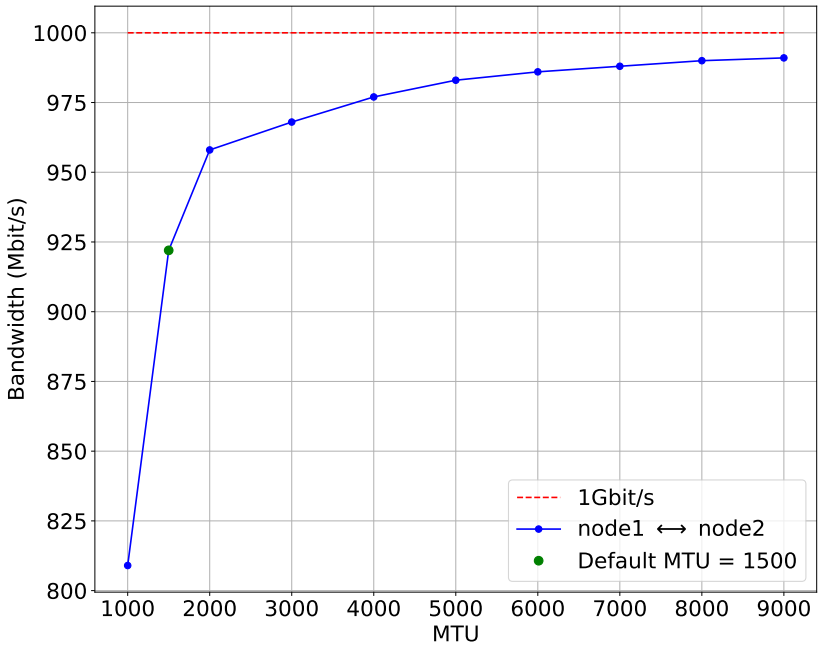


Figure 5.11: **Node to Node Bandwidth.** The node-to-node bandwidth measured using the Linux `iperf` command.

The kernel with Jumbo Frame support, as implemented, was able to run HPL with a flat P x Q processor grid of 1 x 32 for serial benchmarks, and 1 x 8 for OpenMP benchmarks. However, for less flat processor grids, where the previous maximum performance had been observed, the network stack on one or more nodes became corrupted and the node became uncontactable. When this occurred, messages relating to network driver queues errors were observed in kernel message log.

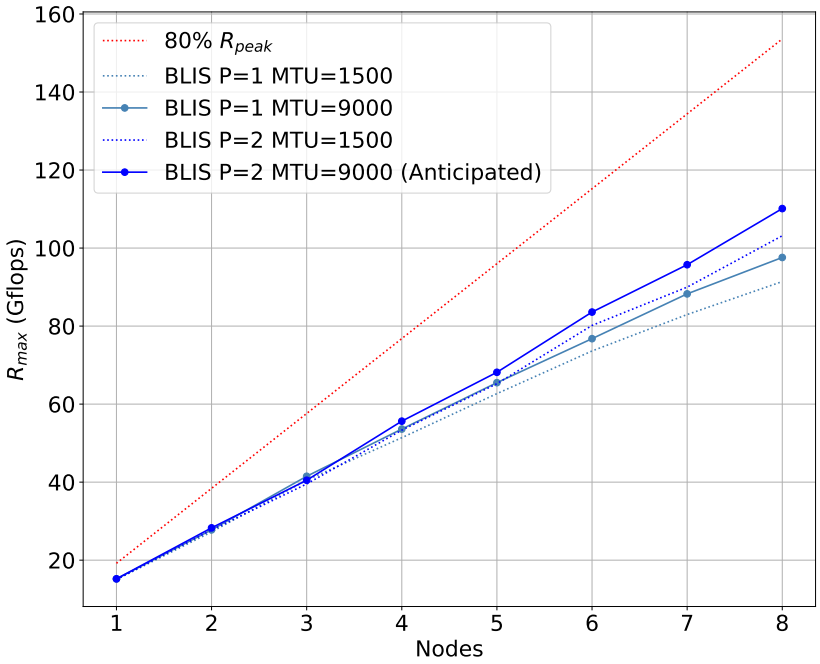


Figure 5.12: **HPL R_{max} verus Node Count with MTU = 9000.**

Figure 5.12 indicates the potential increased performance with an expertly implemented MTU increase to 9000 bytes. In this figure, the increased performance using a flat P x Q grid of 1 x 32 is extrapolated to a 2 x 16 grid for a serial BLIS benchmark.

Observations

As indicated in Figure 5.12, expertly implemented Jumbo Frames support may yield a performance increase to 110 Gflops, a 7% improvement.

5.6 Performance Per Watt

To determine the power usage of the Aerin Cluster a power meter was placed at the power outlet supplying the cluster. The power measured is presented in Table 5.16.

Configuration	Power Usage (Watts)
Router/firewall only	5
Network switch only	4
Router/firewall & network switch only	9
Aerin Cluster at idle	35
Aerin Cluster running benchmarks	70

Table 5.16: **Aerin Cluster Power Usage.** The 70 Watt power usage is the estimated average power usage observed when running serial and OpenMP benchmarks. The two Aerin Cluster power measurements include the router/firewall and network switch.

When the Aerin Cluster is relocated to UCL, and the cluster is connected to the UCL internal network, the router/firewall will be redundant. It is therefore reasonable to state that the estimated average power usage for the Aerin Cluster 65 Watts when running benchmarks.

The maximum observed HPL performance is 103.41 Gflops.

Therefore, the Aerin Cluster's equivalent Green500 List performance is:

$$Gflops/Watt = \frac{103.41}{65} \tag{5.5}$$

$$= 1.591 \tag{5.6}$$

The is equivalent to ranking 170 in the June 2020 Green500 List, as indicated in Table 5.17.

Rank	System	Cores	R_{max} TFlops	Power (KWatt)	Efficiency (Gflops/Watt)
169	ISystem Sugon TC6000	97,920	1683.0	1,050	1.603
-	Aerin Cluster	32	0.10341	0.065	1.591
170	MCSystem Sugon TC6000	74,400	1266.0	800	1.583

Table 5.17: **Extract from June 2020 Green500 List.**

Chapter 6

Conclusions and 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] Raspberry Pi Foundation. *Raspberry Pi 4 Model B*. 2020. URL: https://www.raspberrypi.org/products/raspberry-pi-4-model-b/.

[5] Raspberry Pi Foundation. *Raspberry Pi Zero*. 2020. URL: https://www.raspberrypi.org/products/raspberry-pi-zero/.

[6] Raspberry Pi Foundation. *Raspberry Pi CM3+*. 2020. URL: https://www.raspberrypi.org/products/compute-module-3-plus/.

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

[8] A. M. Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society* s2-42.1 (Jan. 1937), pp. 230–265. ISSN: 0024-6115. DOI: 10.1112/plms/s2-42.1.230. eprint: https://academic.oup.com/plms/article-pdf/s2-42/1/230/4317544/s2-42-1-230.pdf. URL: https://doi.org/10.1112/plms/s2-42.1.230.

[9] UK Met Office. *Numerical Weather Prediction Models*. 2020. URL: https://www.metoffice.gov.uk/research/approach/modelling-systems/unified-model/weather-forecasting.

[10] UK Met Office. *The Cray XC40 Supercomputing System*. 2020. URL: https://www.metoffice.gov.uk/about-us/what/technology/supercomputer.

[11] Tom Herbert and Willem de Bruijn. *Scaling in the Linux Network Stack*. 2020. URL: https://www.kernel.org/doc/Documentation/networking/scaling.txt.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] James W. Demmel, Nicholas J. Higham, and Robert S. Schreiber. *Block LU factorization*. 1995.

[18] Petitet et al. *HPL Algorithm*. 2018.

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

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

[21] Petitet et al. *HPL Frequently Asked Questions*. 2018. URL: https://www.netlib.org/benchmark/hpl/faqs.html.