MSc Scientific Computing Dissertation
Benchmarking a Raspberry Pi 4 Cluster

John Duffy

September 2020

Abstract

Dedication

Declaration

I declare that..

Acknowledgements

I want to thank...

Contents

Chapter 1

Introduction

1.1 Arm

Since the release of the Acorn Computers Arm1 in 1985, as a second coprocessor for the BBC Micro, through to powering today's fastest supercomputer, the Japanese 8 million core Fugaku supercomputer, Arm has steadily grown to become a dominant force in the microprocessor industry, with more than 170+ billion Arm-based processors shipped to date.

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

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

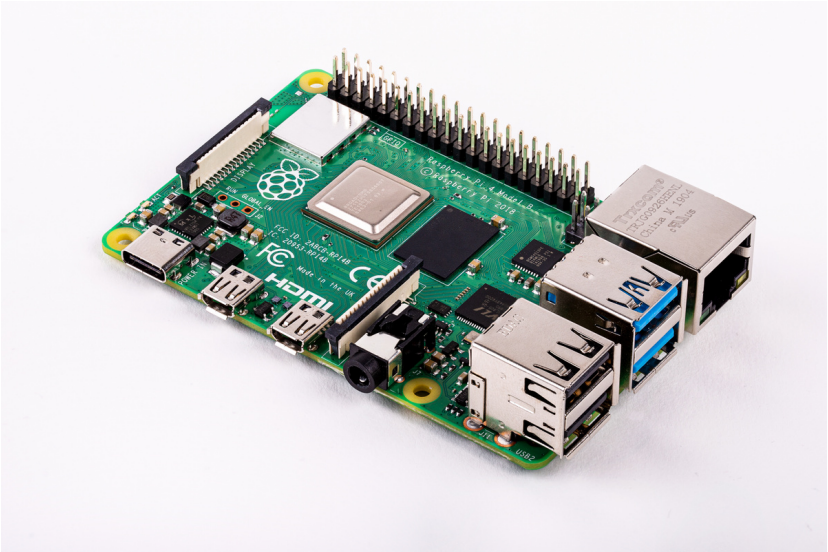


Figure 1.1: **The Raspberry Pi 4 Model B**.

1.2 Raspberry Pi

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

At the heart of every Raspberry Pi is a Broadcom "System on a Chip" (SoC). The SoC integrates Arm processor 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 processor core count or performance, clock speed, connectivity and available memory.

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

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

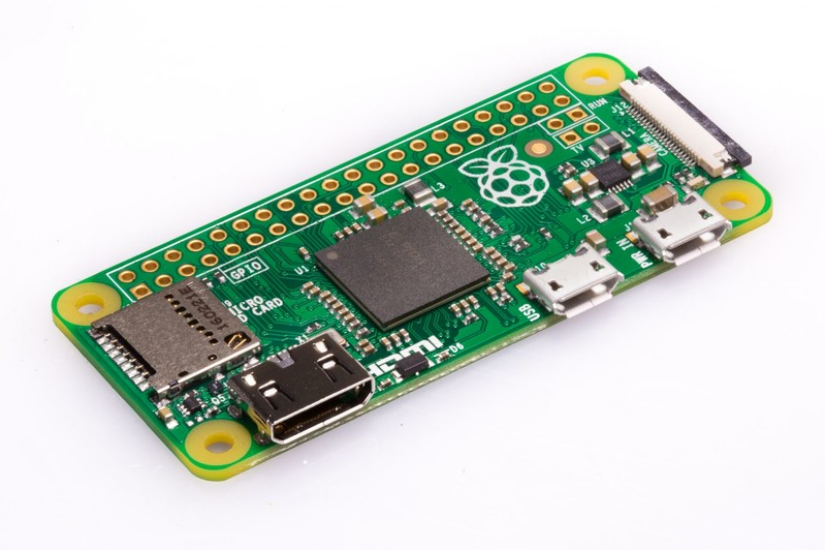


Figure 1.2: **The Raspberry Pi Zero**.

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

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

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

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

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

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

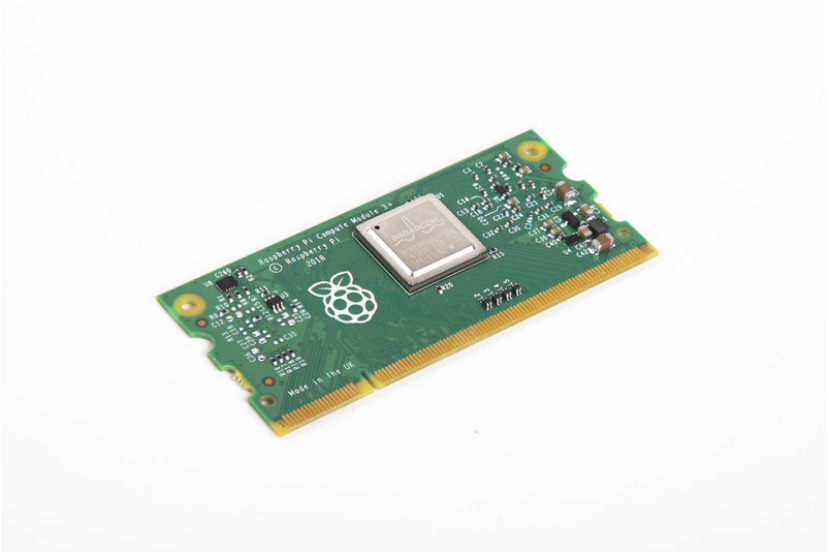


Figure 1.3: **The Raspberry Pi Compute Module 3+**.

also NOOBS (New Out of the Box Software), together with a MicroSD card OS image burning tool called Raspberry Pi Imager. NOOBS and Raspberry Pi Imager make it easy to install operating systems such as Ubuntu, RISC 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, 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 in a standard DDR2 SODIMM connector.

1.3 Aims

1.3.1 Benchmark Performance

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

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

1.3.2 Performance Optimisations

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

1.3.3 Investigate Gflops/Watt

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

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

1.4 Typography

This has been a very hands-on computing project with lots of Linux command line use. To enable a reader to replicate the cluster build and results, the Linux commands, output and file listings are included in the dissertation text, and colour coded as follows to aid readability.

This is a computer name:

`node1`

This is a command to type:

```
$ cat /proc/softirqs
```

This is the command output:

```
                    CPU0       CPU1       CPU2       CPU3
          HI:          1          0          0          1
       TIMER:    3835342    3454143    3431155    3431023
      NET_TX:      36635          0          0          0
      NET_RX:     509189        146        105        121
       BLOCK:      95326       4367       4311       4256
    IRQ_POLL:          0          0          0          0
     TASKLET:       4900          3          4         25
       SCHED:     444569     267214     218701     189120
     HRTIMER:         67          0          0          0
         RCU:     604466     281455     260784     277699
```

This is a long command to type.

```
$ mpirun --bind-to socket -host node1:1,node2:1,node3:1 -np
    ↪ 3 -x OMP_NUM_THREADS=4 xhpl
```

In the command above, it is possible to reduce typing by putting the host names and corresponding processor slots information in a `hostfile`, but this illustrates the typography.

This is a file listing, note the line numbering commencing at 1:

Listing 1.1: /etc/hosts

```
1  ##
2  # Host Database
3  #
4  # localhost is used to configure the loopback interface
5  # when the system is booting.  Do not change this entry.
6  ##
7  127.0.0.1 localhost
8  255.255.255.255 broadcasthost
9  ::1             localhost
10 192.168.0.1 node1
11 192.168.0.2 node2
12 192.168.0.3 node3
13 192.168.0.4 node4
14 192.168.0.5 node5
15 192.168.0.6 node6
16 192.168.0.7 node7
17 192.168.0.8 node8
18 192.168.0.9 node9
```

This is a partial file listing, note the lack of line numbering:

Listing 1.2: /etc/hosts

```
##
# Host Database
#
# localhost is used to configure the loopback interface
# when the system is booting.  Do not change this entry.
##
127.0.0.1 localhost
255.255.255.255 broadcasthost
::1             localhost
192.168.0.1 node1
192.168.0.2 node2
192.168.0.3 node3
192.168.0.4 node4
192.168.0.5 node5
192.168.0.6 node6
192.168.0.7 node7
192.168.0.8 node8
192.168.0.9 node9
```

And this is something to take note of:

```
This is a "gotcha", or
This differs from a similar build procedure, or
This is a "hack" to be fixed permanently later, or
Something similar
```

1.5 Project GitHub Repositories

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

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

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

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

Chapter 2

Computer Architecture and HPC Benchmarks

2.1 Introduction

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

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

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

high-end desktop.

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

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

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

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

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

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

Matrix-matrix multiplication plays an important role in computer benchmarking because the multiplication of large matrices tests processing, memory, and network performance. A more detailed discussion of this topic is also included in this chapter.

2.2 CPU Architecture

2.2.1 Threads

2.2.2 Processes

2.2.3 Context Switch

2.2.4 Concurrency

2.2.5 Parallel Computation

2.2.6 Scheduling

2.2.7 Interrrupts

2.2.8 Kernel Preemption Model

2.2.9 ARM Architecture

Fugatku...

Numer of Arm-based in Top/Green 500...

48 core workstation...

RISC paper...

RISC/CISC

Load/Store architecture

Simplicity of design...

Transistor count...

Electrical power...

armv7...

armv8... 64-bit

armv8.1...

armv8.2...

SVE...

Fugaku chip...

Fujitsu chip...

2.3 Main Memory

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

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

Computers systems without an operating system, such as embedded systems, permit direct access to main memory from software. In this case there is a

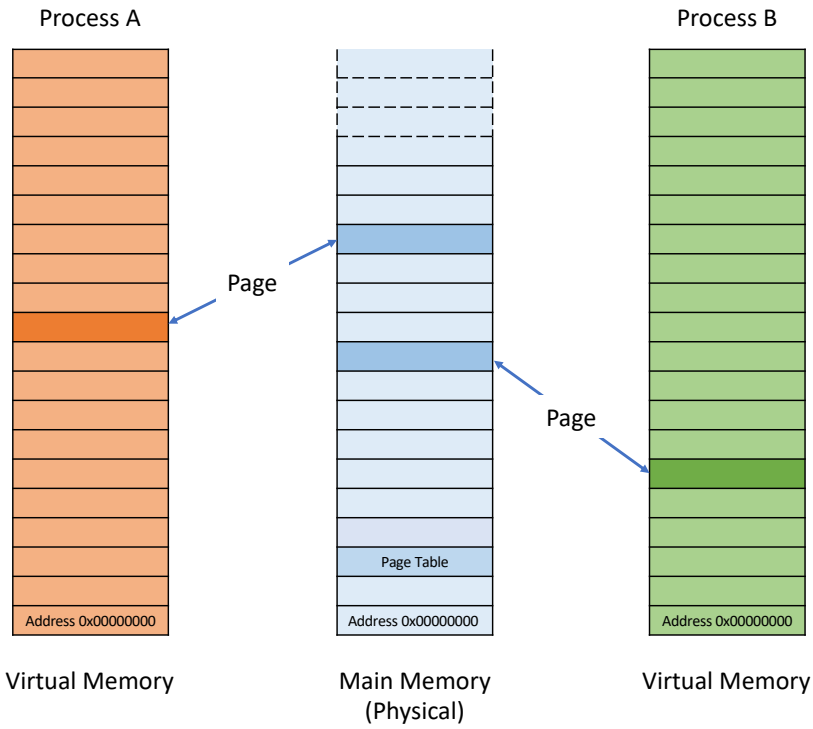


Figure 2.1: **Virtual Memory**. Each process has a *virtual address* space mapped to main memory in *pages* by a *page table* which resides in main memory. A smaller page table called the *Translation Lookaside Buffer* (TLB) is a *cache* in close proximity to each core. The TLB enables fast lookup of 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*.

2.3.1 Virtual Memory

Virtual memory is the abstracted view of main memory presented to a running program by the operating system. Virtual memory requires both hardware support, through the Memory Management Unit (MMU), and software support by the operating system. Contiguous regions of virtual memory are organised

into *pages*, typically 4 KB in size. Each page of virtual memory maps to a page of physical memory through a *page table* which resides in main memory. A smaller page table called the *Translation Lookaside Buffer* (TLB), which is a *cache* in close proximity to each processing core, is discussed later.

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

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

2.3.2 UMA versus NUMA

2.3.3 Shared Memory

2.3.4 Distributed Memory

2.4 Caches

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

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

Caches... lines

Cache coherency...

TLB...

The MMU has the following features:

48-entry fully-associative L1 instruction TLB. 32-entry fully-associative L1 data TLB for data load and store pipelines. 4-way set-associative 1024-entry L2 TLB in each processor.

The L1 instruction memory system has the following features:

48KB 3-way set-associative instruction cache. Fixed line length of 64 bytes.

The L1 data memory system has the following features:

32KB 2-way set-associative data cache. Fixed line length of 64 bytes.

The features of the L2 memory system include:

Configurable L2 cache size of 512KB, 1MB, 2MB and 4MB. Fixed line length of 64 bytes. Physically indexed and tagged cache. 16-way set-associative cache structure.

The SCU uses hybrid Modified Exclusive Shared Invalid (MESI) and Modified Owned Exclusive Shared Invalid (MOESI) protocols to maintain coherency between the individual L1 data caches and the L2 cache.

The L2 memory system requires support for inclusion between the L1 data caches and the L2 cache. A line that resides in any of the L1 data caches must also reside in the L2 cache. However, the data can differ between the two caches when the L1 cache line is in a dirty state. If another agent, a core in the cluster or another cluster, accesses this line in the L2 then it knows the line is present

in the L1 of a processor and then it queries that core for the most recent data.

2.5 Networking

2.5.1 Ethernet

2.5.2 MTU

2.5.3 Interrupt Coalescing

2.5.4 Receive Side Scaling

2.5.5 Receive Packet Steering

2.5.6 Receive Flow Steering

2.6 Matrix Multipication

2.7 Benchmarks

2.7.1 High Performance Linpack

2.7.2 HPC Challenge

2.7.3 HP Conjugate Gradients

2.8 Landscape

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

whole system performance, including processing power, memory bandwidth, and network speed and latency, in relation to standard HPC algorithms such as FFT and CG.

HPL has been the main focus of this project, mainly because it is the industry standard HPC benchmark, but also because tuning performance for HPL will also produce optimum results for HPCC (HPCC includes HPL) and HPCG.

Because BLAS (Basic Linear Algebra Subroutine) library performance and cluster topology, pure OpenMPI and hybrid OpenMPI/OpenMP, have a direct impact on benchmark performance, a discussion of these topics is also included in this chapter.

A detailed description of each benchmark follows.

2.9 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}.

2.9.1 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.

2.9.2 Running HPL

HPL uses the input file `HPL.dat` to specify the input parameters for each benchmark run.

Each of these has a preceding count parameter (`#`) which specifies the number of each parameter (for example, there may be more than 1 processor grid shape).

The problem size N and block size NB are set as follows, in this case there being

a single problem size and a single block size:

Listing 2.1: HPL.dat

```
1            # of problems sizes (N)
52360        Ns
1            # of NBs
88           NBs
```

The processor grid shapre parameters P and Q are set as follows, in this case there being 32 cores/slots with 3 possible grid shapes, 1 x 32, 2 x 16 and 4 x 8:

Listing 2.2: HPL.dat

```
3            # of process grids (P x Q)
1 2 4        Ps
32 16 8      Qs
```

For each benchmark run, the above parameters are set accordingly.

For this project the remaining parameters, which have a lesser effect on benchmark results, were set in accordance with the advice in HPL Tuning, with `swapping threshold` being set to match NB, as follows:

Listing 2.3: HPL.dat

```
16.0         threshold
1            # of panel fact
1            PFACTs (0=left, 1=Crout, 2=Right)
2            # of recursive stopping criterium
4 8          NBMINs (>= 1)
1            # of panels in recursion
2            NDIVs
1            # of recursive panel fact.
2            RFACTs (0=left, 1=Crout, 2=Right)
2            # of broadcast
1 3          BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)
2            # of lookahead depth
0 1          DEPTHs (>=0)
2            SWAP (0=bin-exch,1=long,2=mix)
88           swapping threshold
0            L1 in (0=transposed,1=no-transposed) form
0            U  in (0=transposed,1=no-transposed) form
1            Equilibration (0=no,1=yes)
8            memory alignment in double (> 0)
```

HPL is run using a serial BLAS library as follows, in this case using 2 nodes with 4 cores/slots:

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

HPL is run using a multi-threaded BLAS library as follows, again, in this case using 2 nodes with 4 cores/slots:

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

The results generated by each benchmark run are either printed on `stdout`, `stderr`, or placed in a file, depending on the `HPL.dat` parameter `device out`.

For this project, all benchmark results were placed in a file called `HPL.out` by specifying the filename and setting `device out` to zero, as follows:

Listing 2.4: HPL.dat

```
HPL.out      output file name (if any)
0            device out (6=stdout,7=stderr,file)
```

The `HPL.out` file from each benchmark run was renamed to reflect the N, NB, P and Q parameters used, and also the BLAS library used. For example:

```
$ mv HPL.out HPL.out.6_node_45320_88_1_6_2_3.openblas_openmp
```

Each results file was then stored appropriately in the `picluster/results` directory structure.

2.10 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.

2.10.1 HPL

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

2.10.2 DGEMM

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

2.10.3 STREAM

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

2.10.4 PTRANS

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

2.10.5 RandomAccess

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

2.10.6 FFT

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

2.10.7 Network Bandwidth and Latency

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

2.11 Running HPCC

HPCC is run in the same manner as HPL. An input file `hpccinf.txt` is created, which is of same format as `HPL.dat`, but may contain additional problem sizes and block sizes for the PTRANS benchmark.

HPCC is run using a serial BLAS library as follows, in this case using 2 nodes with 4 cores/slots:

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

HPCC is run using a multi-threaded BLAS library as follows, again, in this case using 2 nodes with 4 cores/slots:

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

The benchmark results are placed in an output file `hpccoutf.txt`.

For this project each benchmark `hpccoutf.txt` file was renamed to reflect the N, NB, P and Q parameters used, and also the BLAS library used. For example:

```
$ mv hpccoutf.txt hpccoutf.txt.8_node_52400_200_1_8_2_4.
↪ blis_openmp
```

Each results file was then stored appropriately in the `picluster/results` directory structure.

2.12 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."

2.12.1 Running HPCG

In a similar manner to HPL and HPCC, HPCG uses an input file `hpcg.dat` for benchmark run configuration.

The format of `hpcg.dat` with default parameter values is:

```
HPCG benchmark input file
Sandia National Laboratories; University of Tennessee,
↪ Knoxville
104 104 104
60
```

Lines 1 and 2 are comments, line 3 specifies the 3 dimensions of the problem size, and line 4 specifies the run time in seconds. The problem size is per OpenMPI process. For benchmark runs to be submitted for official performance ranking the problem size memory usage must exceed 25% of available memory and the run time must exceed 30 minutes (1800 seconds).

HPCG can be built in serial and OpenMP versions. See Part II Chapter 10 for details.

For the serial version each node runs 4 `xhpcg` processes, one on each core. Foe the OpenMP version a single `xhpcg` process is run on each node.

The serial version of HPCG is run as follows, in this case on 2 nodes each with 4 cores:

```
mpirun --bind-by core -host node1:4,node2:4 -np 8 xhpcg
```

The OpenMP version of HPCG is run as follows, again, in this case on 2 nodes each with 4 cores:

```
mpirun --bind-by socket -host node1:1,node2:1 -np 2 -x
↪ OMP_NUM_THREADS=4 xhpcg
```

The output from each benchmark run is placed is an output file with a name including a timestamp, for example:

`HPCG-Benchmark_3.1_2020-08-30_15-00-29.txt`

2.13 BLAS Libraries

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

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

And then look at the stack trace report:

```
$ sudo perf report
```

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

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

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

The mathematical operation implemented by `dgemm` is:

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

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

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

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

Maximum performance is achieved when the matrix multiplication problem size

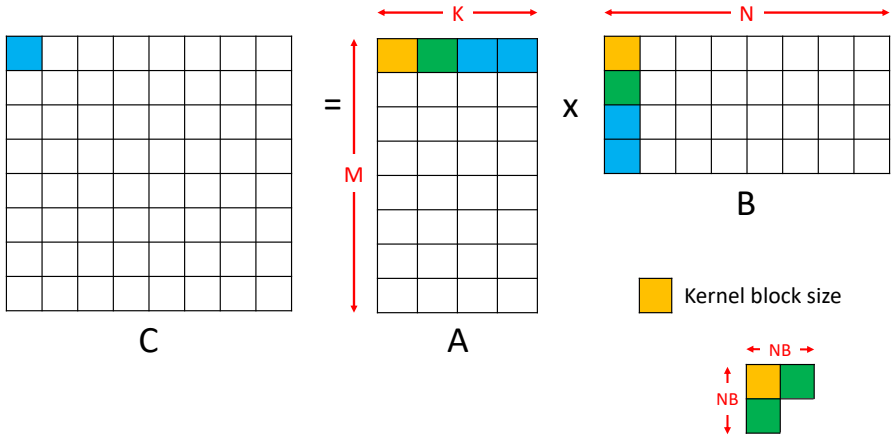


Figure 2.2: `dgemm` *kernel* Matrix-Matrix Multiplication.

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

The above is depicted in Figure ??.

2.13.1 GotoBLAS

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

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

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

2.13.2 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

2.13.3 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

2.14 Pure OpenMPI Topology

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

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

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

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

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

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

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

Where `nodes` is a file containing the following:

Listing 2.5: nodes

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

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

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

2.15 Hybrid OpenMPI/OpenMP Topology

In a hybrid OpenMPI/OpenMP topology, OpenMPI is used to distribute work between the nodes. Each node runs a single work process. OpenMP is then

used to distribute the work of this single process between the node cores using a multi-threaded BLAS library.

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

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

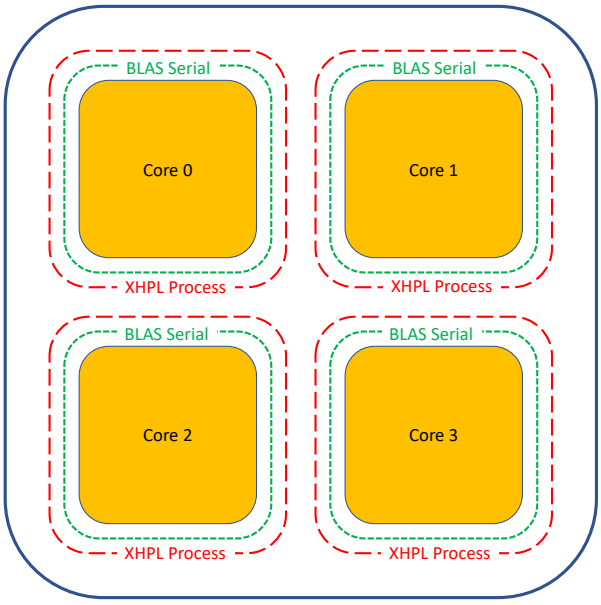
Using 2 nodes, and 4 cores per node, as per the pure OpenMPI example, 2 `xhpl` processes are run, one on each node, with the multi-threaded BLAS library utilising 4 cores, as follows:

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

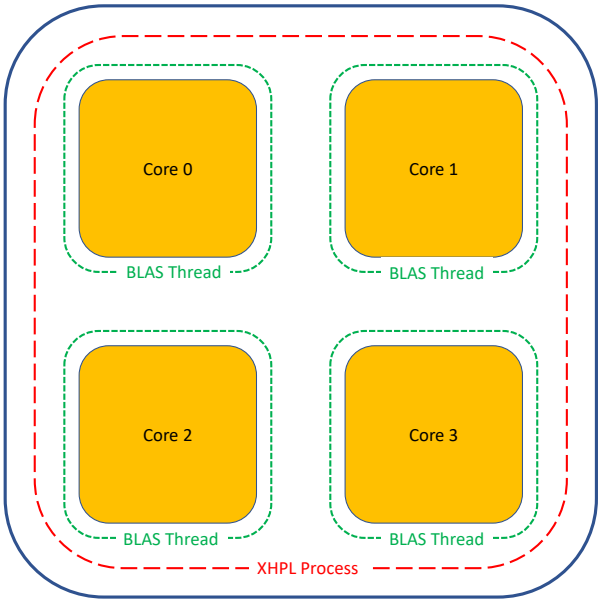
The `--bind-to socket` parameter indicates to `mpirun` that the `xhpl` process is not associated with a particular core, it is not to be *bound* to a specific core. The OpenMP runtime will determine which core(s) are used to run the `xhpl` process.

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

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



(a) Pure OpenMPI



(b) Hybrid OpenMPI/OpenMP

Figure 2.3: Single Node Toplologies.

Chapter 3

Mathematical Background of HPC Benchmarks

Chapter 4

The Aerin Cluster

This chapter describes the components of the Aerin Cluster, and includes some advice and lessons learned. Detailed build instructions are included in Part II Chapter 7.

Photo...

The Aerin Cluster consists of the following hardware and software components.

4.1 Hardware

- 8 x Raspberry Pi 4 Model B compute nodes, `node1` to `node8`
- 1 x Raspberry Pi 4 Model B build node, `node9`
- 9 x Official Raspberry Pi 4 power supplies
- 9 x Class 10 A1 MicroSD cards
- 9 x Heatsinks with integrated fans
- 1 x Netgear FVS318G 8 Port Gigabit Router/Firewall
- 1 x Netgear GS316 16 Port Gigabit Switch (with Jumbo Frame Support)
- Cat 7 cabling

4.1.1 Raspberry Pi's

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

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

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

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

4.1.2 Power Supplies

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

4.1.3 MicroSD Cards

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

4.1.4 Heatsinks

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

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

4.1.5 Network Considerations

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

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

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

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

4.1.6 Router/Firewall

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

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

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

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

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

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

4.1.7 Network Switch

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

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

4.1.8 Cabling

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

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

4.2 Software

4.2.1 Operating System

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

4.2.2 `cloud-init`

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

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

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

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

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

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

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

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

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

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

4.2.3 Benchmark Software

The HPL, HPCC and HPCG benchmark software was all compiled locally from source. The instructions for how to do this are in Part II Chapter 7.

4.2.4 BLAS Library Management

On the Aerin Cluster we have two different BLAS libraries installed, OpenBLAS and BLIS, both in serial and OpenMP versions. It is obviously critical to have the same BLAS library configured as the "the BLAS library in use" on each node at the same time.

Debian/Ubuntu have a very clever mechanism for setting a particular version of a library to be the "the BLAS library in use". The is called the "alternatives"

mechanism, and is not just used for BLAS libraries, there are lots of software packages with "alternatives".

Each "alternative" has a name, in the case of the BLAS libraries it is called `libblas.so.3`. What is really clever is that you can build software, such as HPL, to link against `libblas.so.3`, and then change then change what this "alternative" points to without having to rebuild the software.

For example, to set the serial version of OpenBLAS to "the BLAS library in use" we update the "alternative" with the following command:

```
$ sudo update-alternatives --set libblas.so.3-aarch64-linux-
↪ gnu /usr/lib/aarch64-linux-gnu/openblas-serial/libblas.so
↪ .3
```

Alternatively, there is an interactive version of the command which allows you to select a BLAS library from a list of options:

```
$ sudo update-alternatives --config libblas.so.3-aarch64-
↪ linux-gnu
```

The "alternatives" mechanism is very clever, but when you need to set the BLAS library on 8 nodes quite frequently this is a lot of typing and also error prone. So to make life easier I wrote two BLAS library management wrapper scripts described in the section below.

4.2.5 Pi Cluster Tools

Even a cluster of only 8 nodes requires quite a bit of effort, and typing, to keep the system up to date and to ensure the same BLAS library is running on each node. Logging in to each node individually to do this is a chore, and more importantly it is very error prone.

To get around this problem, a number of `bash` scripts were written as Pi Cluster Tools. Each script loops over a list of node names and uses `ssh` to run a command remotely on each node in turn.

The following scripts are included as Pi Cluster Tools.

- upgrade
- reboot
- shutdown
- do

- libblas-query
- libblas-set

Listings of the scripts are included in Part II Chapter 17.

To run a particular tool, for example `upgrade`, type the following command which will upgrade all of the nodes sequentially.

```
$ ~/picluster/tools/upgrade
```

The `do` command "does" the same command on each node. Note the required quotation marks.

```
$ ~/picluster/tools/do "mkdir -p ~/picluster/hpl/hpl-2.3/bin
↪ /picluster"
```

Probably the most useful of the Pi Cluster Tools are the two BLAS library tools.

`libblas-query` queries the "the BLAS library in use" on each node. This is extremely useful for ensuring the same library is in use on each node.

For example.

```
$ ~/picluster/tools/libblas-query
```

```
node8... /usr/lib/aarch64-linux-gnu/blis-openmp/libblas.so.3
node7... /usr/lib/aarch64-linux-gnu/blis-openmp/libblas.so.3
node6... /usr/lib/aarch64-linux-gnu/blis-openmp/libblas.so.3
node5... /usr/lib/aarch64-linux-gnu/blis-openmp/libblas.so.3
node4... /usr/lib/aarch64-linux-gnu/blis-openmp/libblas.so.3
node3... /usr/lib/aarch64-linux-gnu/blis-openmp/libblas.so.3
node2... /usr/lib/aarch64-linux-gnu/blis-openmp/libblas.so.3
node1... /usr/lib/aarch64-linux-gnu/blis-openmp/libblas.so.3
```

`libblas-set` takes a single argument, openblas-serial, openblas-openmp, blis-serial, or blis-openmp, and then uses the "alternatives" mechanism to set the "BLAS library in use" on each node.

For example.

```
$ ~/picluster/tools/libblas-set openblas-serial
```

```
node8... done
node7... done
node6... done
node5... done
```

```
node4... done
node3... done
node2... done
node1... done
```

```
Pi Cluster Tools are not production quality.
```

Chapter 5

Benchmark Results and Optimisations

5.1 Theoretical Maximum Performance (Gflops)

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

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

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

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

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

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

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

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

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

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

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

```
$ dmesg | grep Memory
```

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

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

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

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

5.2 HPL Baseline

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

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

80% of memory

Methodology...

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

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

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

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

5.2.1 HPL 1 Core Baseline

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

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

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

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

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

The HPL input file HPL.dat is populated with these NB and N combinations as follows, in this example using an NB of 32 and an N of 18528:

```
1              # of problems sizes (N)
18528          Ns
1              # of NBs
32             NBs
```

For this baseline a single `xhpl` process is run on both the pure OpenMPI and Hybrid OpenMPI/OpenMP topologies. In both of these cases HPL.dat is populated with processor grid parameters P and Q as follows:

```
1              # of process grids (P x Q)
1              Ps
1              Qs
```

This baseline is run on a pure OpenMPI topology with the following command:

```
$ mpirun --bind-to core -host node1:1 -np 1 xhpl
```

Explain bind to core...

This baseline is run on a hybrid OpenMPI/OpenMP topology with the following command:

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

Explain bind to socket...

The results are plotted in Figure ??.

Observations

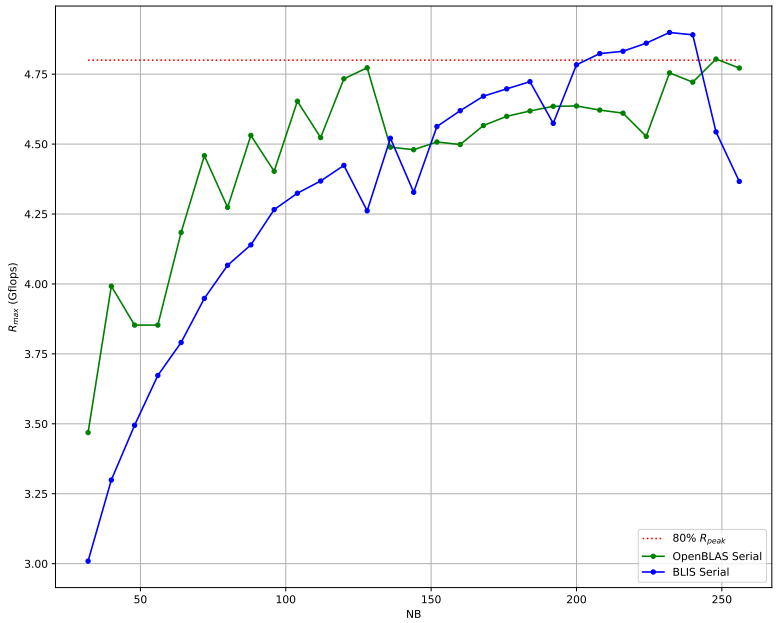
As expected, there is no noticeable performance difference between a pure OpenMPI and hybrid OpenMPI/OpenMP topology for a single `xhpl` process running on a single core.

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

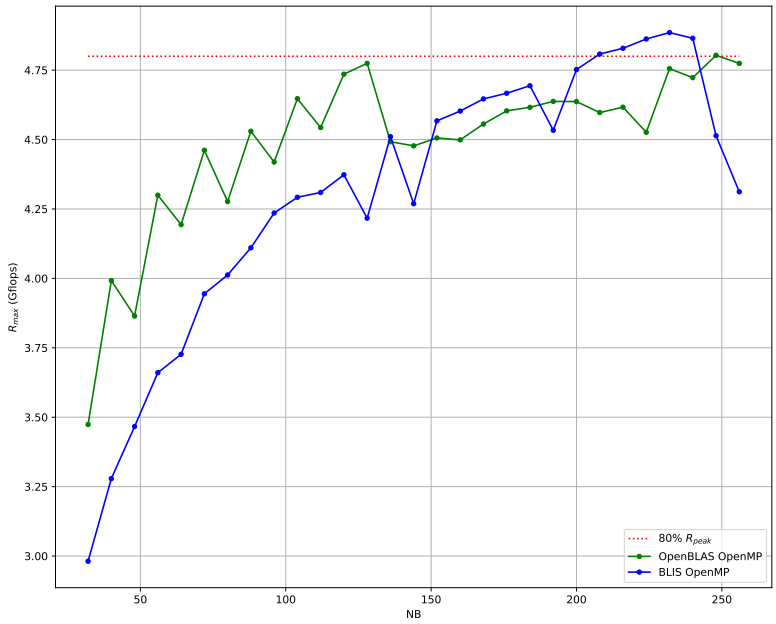
Discussion about OpenBLAS and BLIS internal kernel blocking...

5.2.2 HPL 1 Node Baseline

The purpose of this baseline is to determine the performance of the 4 cores of a single node. In this case each core has to share access to the L2 cache, which



(a) Pure OpenMPI



(b) Hybrid OpenMPI/OpenMP

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

will result in more main memory accesses. It is therefore anticipated that this will result in a performance reduction, per core, compared to the single core case.

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

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

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

For the pure OpenMPI topology, 4 `xhpl` processes are run, one on each core. In this case HPL.dat is populated with processor grid parameters P and Q as follows:

```
1             # of process grids (P x Q)
1             Ps
4             Qs
```

And the pure OpenMPI topology baseline is run with the following command:

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

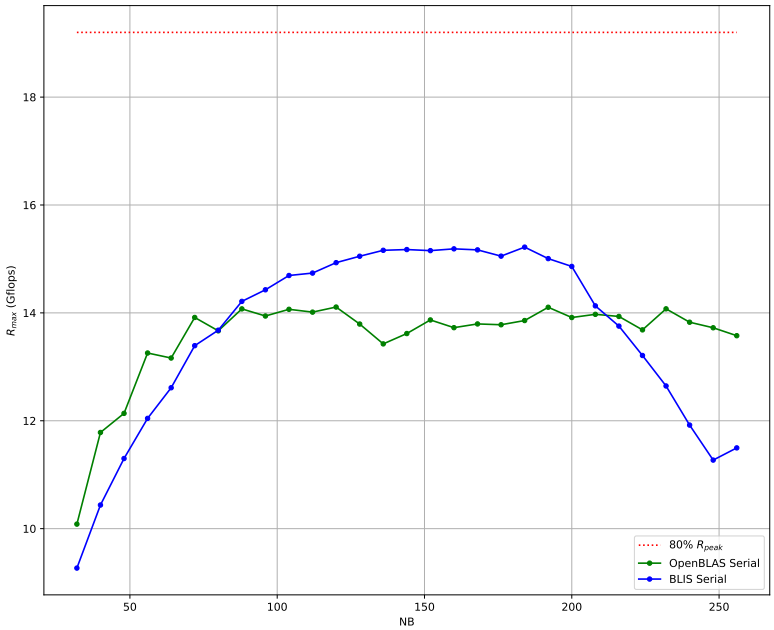
For the hybrid OpenMPI/OpenMP topology, a single `xhpl` process is run on the node. In this case the HPL.dat P and Q processor grid parameters are populated as follows:

```
1             # of process grids (P x Q)
1             Ps
1             Qs
```

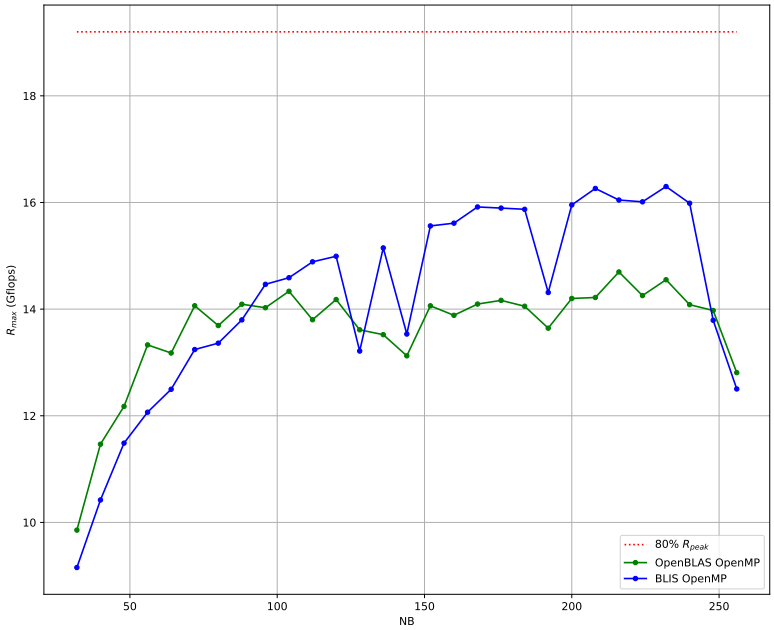
With 4 cores available to the multi-threaded BLAS library, the hybrid OpenMPI/OpenMP topology baseline is run with the following command:

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

The results are plotted in Figure ??



(a) Pure OpenMPI



(b) Hybrid OpenMPI/OpenMP

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

Observations

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

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

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

5.2.3 HPL 2 Node Baseline

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

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

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

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

For the pure OpenMPI topology, 8 `xhpl` processes are run, one on each core of each of the 2 nodes. Now it is possible to have 2 processor grid shapes, 1 x 8 and 2 x 4. In this case HPL.dat is populated with processor grid parameters P and Q as follows:

```
2            # of process grids (P x Q)
1 2          Ps
8 4          Qs
```

The pure OpenMPI topology baseline is run with the following command:

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

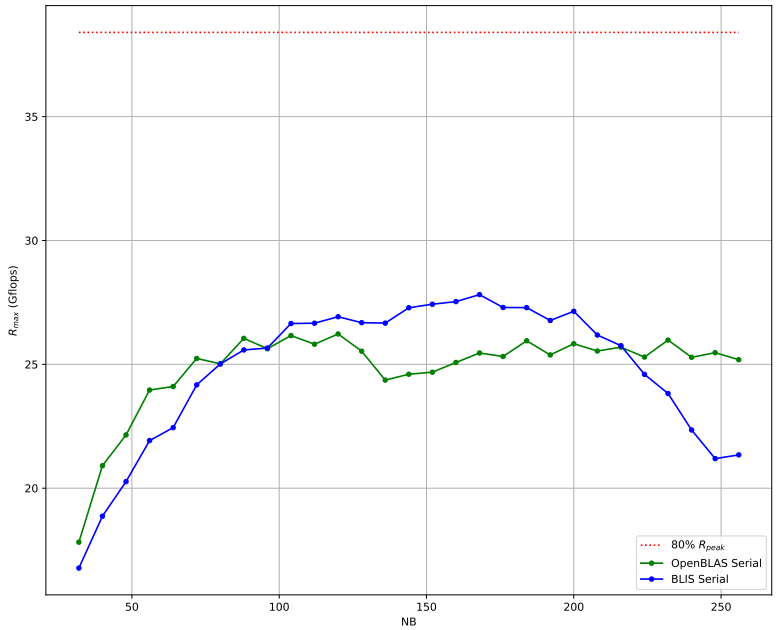
For the hybrid OpenMPI/OpenMP topology, a single `xhpl` process is run on each node. This results in single processor grid shape of 1 x 2, and the HPL.dat P and Q processor grid parameters are populated as follows:

```
1            # of process grids (P x Q)
1            Ps
2            Qs
```

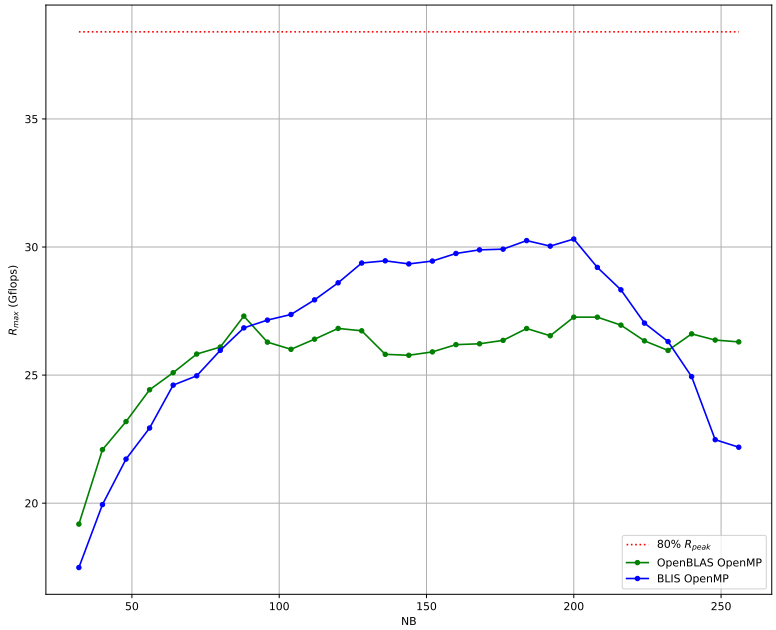
With the BLAS library utilising the 4 cores on each node, the hybrid OpenMPI/OpenMP topology baseline is run with the following command:

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

The results are plotted in Figure ??



(a) Pure OpenMPI



(b) Hybrid OpenMPI/OpenMP

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

Observations

5.2.4 HPL Cluster Baseline

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

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

These NB and N combinations are used to populate HPL.dat, as per the example below for the 3 node OpenBLAS Serial case.

```
1            # of problems sizes (N)
32000        Ns
1            # of NBs
000          NBs
```

For the 3 node pure OpenMPI baseline, the following HPL.dat processor grid shapes are used:

```
3            # of process grids (P x Q)
1 2 3        Ps
12 6 4       Qs
```

And the 3 node pure OpenMPI baseline is run with the following command:

```
$ mpirun --bind-to core -host node1:4,node2:4,node3:4 -np 12
↪  xhpl
```

For the 3 node hybrid OpenMPI/OpenMP baseline, the following HPL.dat processor grid shapes are used:

```
1            # of process grids (P x Q)
1            Ps
3            Qs
```

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

Table 5.4: HPL Cluster Baseline using OpenBLAS.

And the 3 node hybrid OpenMPI/OpenMP baseline is run with the following command:

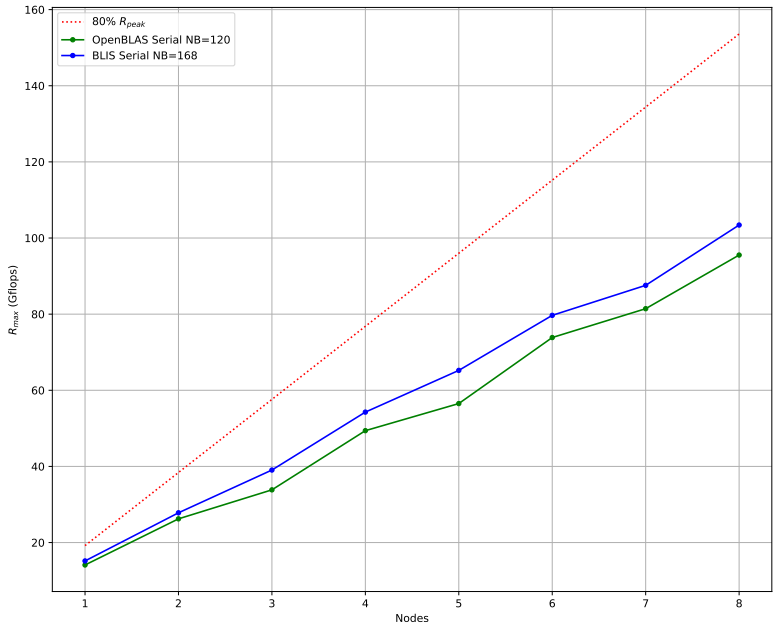
```
$ mpirun --bind-to socket -host node1:1,node2:1,node3:1 -np
↪ 3 -x OMP_NUM_THREADS=4 xhpl
```

For the 4, 5, 6, 7 and 8 node pure OpenMPI and hybrid OpenMPI/OpenMP baselines, HPL.dat is populated with the processor grid parameters P and Q in a similar manner. Likewise, the baselines are run in a similar manner.

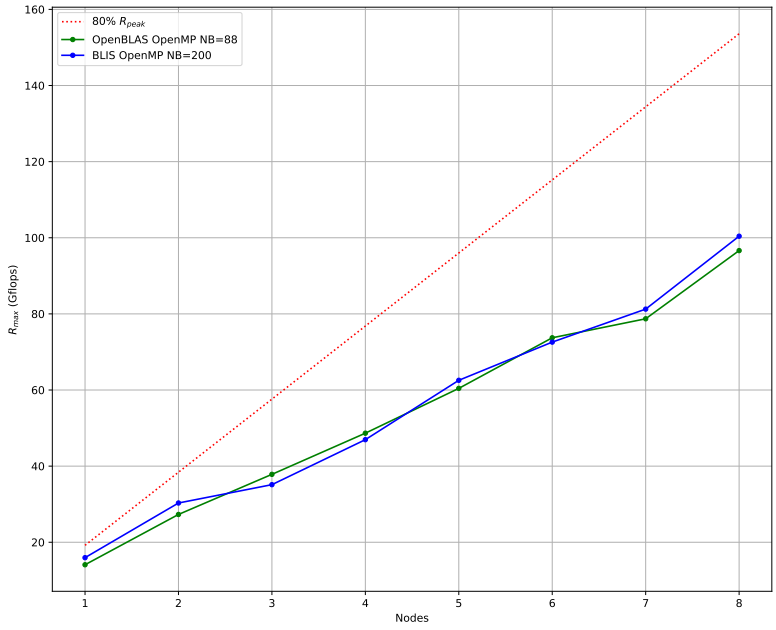
The baseline results are presented in Figure ??.

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

Table 5.5: HPL Cluster Baseline using BLIS.



(a) Pure OpenMPI



(b) Hybrid OpenMPI/OpenMP

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

5.2.5 Observations

Best NB...

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

Iperf...

htop...

top...

perf...

cache misses...

software interrupts...

Suggests... improve network efficiency?

5.3 HPCC Baseline

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

5.3.1 HPL

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

Table 5.6: HPCC HPL.

5.3.2 DGEMM

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

Table 5.7: HPCC DGEMM.

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

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

5.3.3 STREAM

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

Table 5.8: HPCC STREAM.

5.3.4 PTRANS

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

Table 5.9: HPCC PTRANS.

5.3.5 Random Access

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

Table 5.10: HPCC Random Access.

5.3.6 FFT

5.3.7 Network Bandwidth and Latency

5.4 HPCG Baseline

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

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

Table 5.11: June 2020 HPCG List.

The Aerin Cluster...

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

Table 5.12: The Aerin Cluster HPCG Benchmark.

5.4.1 Serial HPCG

5.4.2 OpenMP HPCG

5.5 Optimisations

5.5.1 Rebuild BLAS Libraries

The Debian Science Wiki suggests that for optimum performance, for architectures other than x86, the Debian BLAS library packages should be rebuilt locally. The instructions for doing so are included in the Debian source package for each library.

Rebuild OpenBLAS

The OpenBLAS build process attempts to detect the architecture in use and build OpenBLAS optimised for this architecture. The architecture is detected by checking specific "magic numbers" in known processor registers. Having detected the architecture, source code macros are defined specific to the architecture.

The architecture detection functionality is implemented in the file `cpuid_arm64.c`, in which the function `get_cpuconfig()` is used to define the source code macros. It was noticed that some macro definitions in this function which may affect performance were not correct for the BCM2711.

Listing 5.1: cpuid_arm64.c

```
...
  case CPU_CORTEXA57:
  case CPU_CORTEXA72:
  case CPU_CORTEXA73:
    // Common minimum settings for these Arm cores
    // Can change a lot, but we need to be conservative
    // TODO: detect info from /sys if possible
    printf("#define %s\n", cpuname[d]);
    printf("#define L1_CODE_SIZE 49152\n");
    printf("#define L1_CODE_LINESIZE 64\n");
    printf("#define L1_CODE_ASSOCIATIVE 3\n");
    printf("#define L1_DATA_SIZE 32768\n");
    printf("#define L1_DATA_LINESIZE 64\n");
    printf("#define L1_DATA_ASSOCIATIVE 2\n");
    printf("#define L2_SIZE 524288\n");
    printf("#define L2_LINESIZE 64\n");
```

```
    printf("#define L2_ASSOCIATIVE 16\n");
    printf("#define DTB_DEFAULT_ENTRIES 64\n");
    printf("#define DTB_SIZE 4096\n");
    break;
...
```

Listing ?? shows the macro definitions for the Arm Cortex-A72. The following two lines are incorrect for the L2 cache size and Data Translation Lookaside Buffer (DTB) for the BMC2711:

```
    printf("#define L2_SIZE 524288\n");
    printf("#define DTB_DEFAULT_ENTRIES 64\n");
```

To reflect the 1 MB L2 cache and DTB default entries of the BCM2711 these should be:

```
    printf("#define L2_SIZE 1048576\n");
    printf("#define DTB_DEFAULT_ENTRIES 32\n");
```

These values were changed and OpenBLAS was rebuilt following the instructions in the source package. The `config.h` file generated during the build process accurately reflects the changes to `cpuid_arm64.c`:

Listing 5.2: config.h

```
1  #define OS_LINUX 1
2  #define ARCH_ARM64 1
3  #define C_GCC 1
4  #define __64BIT__ 1
5  #define PTHREAD_CREATE_FUNC pthread_create
6  #define BUNDERSCORE _
7  #define NEEDBUNDERSCORE 1
8  #define ARMV8
9  #define HAVE_NEON
10 #define HAVE_VFPV4
11 #define CORTEXA72
12 #define L1_CODE_SIZE 49152
13 #define L1_CODE_LINESIZE 64
14 #define L1_CODE_ASSOCIATIVE 3
15 #define L1_DATA_SIZE 32768
16 #define L1_DATA_LINESIZE 64
17 #define L1_DATA_ASSOCIATIVE 2
18 #define L2_SIZE 1048576
19 #define L2_LINESIZE 64
20 #define L2_ASSOCIATIVE 16
21 #define DTB_DEFAULT_ENTRIES 64
22 #define DTB_SIZE 4096
23 #define NUM_CORES 4
24 #define CHAR_CORENAME "CORTEXA72"
```

```
25 #define GEMM_MULTITHREAD_THRESHOLD 4
```

On completion of the build process, the original OpenBLAS library was uninstalled and replaced by the locally rebuilt library.

It was anticipated that a doubling of the L2 cache size from 0.5 MB to 1 MB would have a positive impact on OpenBLAS performance, but this was not the case. There was no impact on performance, positive or negative, at all. The Linpack performance was exactly the same.

Since the L2 cache size of any processor is a major feature of the processor architecture, it is reasonable to expect a doubling of L2 cache size to result in a positive impact on performance, even marginal. The fact that there was no impact on performance at all seemed odd. So, an experiment was conducted with the L2 cache size set to 0 MB, and OpenBLAS was rebuilt. And the resulting Linpack performance was exactly the same.

In light of this experiment, a search through the OpenBLAS source code was conducted for L2_SIZE, and it was determined that this macro definition is not used.

Following an email exchange with an OpenBLAS developer, it would appear that a previous change to the source code to improve performance on server-class Arm architectures has somehow resulted in source code issues for smaller Arm architectures.

Time permitting, it would be an interesting project to fix these source code issues for smaller Arm architectures. But for this project, the locally built library was uninstalled, and the Debian/Ubuntu package re-installed.

Rebuild BLIS

5.5.2 Kernel Preemption Model

The Linux kernel has 3 Preemption Models...

1... 2... The default 3...

As per the Help in the Kernel Configuration...

Listing 5.3: Kernel Configuration Preemption Model Help

```
CONFIG_PREEMPT_NONE:

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 maximize the
raw processing power of the kernel, irrespective of scheduling
latencies.
```

So, kernel rebuilt with CONFIG_PREEMPT_NONE=y

See Appendix ? on how to rebuild the kernel...

Installed on each node...

So, although this optimisation applies to single node, the benefits of applying this optimisation may not be apparent until the kernel has to juggle networking etc...

RESULTS...

Recieve Queues

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

Running xhpl on 8 nodes using OpenBLAS...

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

SHORTLY AFTER PROGRAM START...

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

top...

```
top - 20:33:15 up 8 days,  6:02,  1 user,  load average:
↪ 4.02, 4.03, 4.00
Tasks: 140 total,   5 running, 135 sleeping,   0 stopped,
↪ 0 zombie
%Cpu(s): 72.5 us, 21.7 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0
↪ hi,  5.8 si,  0.0 st
MiB Mem :   3793.3 total,    330.1 free,   3034.9 used,
↪ 428.3 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.
↪ 698.7 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM
    ↪       TIME+ COMMAND
```

```
  34884 john      20   0  932964 732156   7980 R 100.3  18.8
  ↪  106:40.29 xhpl
  34881 john      20   0  933692 732272   7916 R 100.0  18.9
  ↪  107:29.75 xhpl
  34883 john      20   0  932932 731720   8136 R  99.3  18.8
  ↪  107:33.25 xhpl
  34882 john      20   0  932932 731784   8208 R  97.7  18.8
  ↪  107:33.64 xhpl
```

SOFTIRQS...

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

IPERF!!!

On node8, running the top command...

```
$ top
```

We can see...

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

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM
    ↪      TIME+ COMMAND
  23928 john      20   0  883880 682456   8200 R 100.0  17.6
  ↪   13:14.17 xhpl
  23927 john      20   0  883988 682432   7932 R  99.7  17.6
  ↪   13:12.58 xhpl
  23930 john      20   0  883912 682664   7832 R  99.7  17.6
  ↪   13:17.01 xhpl
  23929 john      20   0  883880 682640   8376 R  99.3  17.6
  ↪   13:16.25 xhpl
```

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

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

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

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

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

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

As can be seen...

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

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

What is there to be done?...

1. Reduce the numbers of interrupts...

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

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

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

5.5.3 Network Optimisation

On node2 start the Iperf server...

```
$ iperf -s
```

On node1 start the Iperf client...

```
$ iperf -c
```

ping tests of MTU...

iperf network speed...

Jumbo Frames

Requires a network switch capable of Jumbo frames...

```
$ ip link show eth0
```

```
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq
↪  state UP mode DEFAULT group default qlen 1000
    link/ether dc:a6:32:60:7b:cd brd ff:ff:ff:ff:ff:ff
```

```
$ ping -c 1 -s 1500 -M do node2
```

```
PING node2 (192.168.0.2) 1500(1528) bytes of data.
ping: local error: message too long, mtu=1500
```

```
$ ping -c 1 -s 1472 -M do node2
```

```
PING node2 (192.168.0.2) 1472(1500) bytes of data.
1480 bytes from node2 (192.168.0.2): icmp_seq=1 ttl=64 time
↪ =0.392 ms
```

Trying to set the MTU to 9000 bytes...

```
$ sudo ip link set eth0 mtu 9000
```

... results with...

```
Error: mtu greater than device maximum.
```

In fact, attempting to set the MTU to anything greater than 1500 bytes...

```
$ sudo ip link set eth0 mtu 1501
```

... results with...

```
Error: mtu greater than device maximum.
```

Need to build a kernel with Jumbo frame support...

See Appendix ?...

```
$ ip link show eth0
```

```
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9000 qdisc mq
↪  state UP mode DEFAULT group default qlen 1000
    link/ether dc:a6:32:60:7b:cd brd ff:ff:ff:ff:ff:ff
```

```
$ ping -c 1 -s 9000 -M do node2
```

```
PING node2 (192.168.0.2) 9000(9028) bytes of data.
ping: local error: message too long, mtu=9000
```

```
$ ping -c 1 -s 8972 -M do node2
```

```
PING node2 (192.168.0.2) 8972(9000) bytes of data.
8980 bytes from node2 (192.168.0.2): icmp_seq=1 ttl=64 time
↪ =0.847 ms
```

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

```
$ iperf -s
```

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

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

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

5.5.4 Kernel TCP Parameters Tuning

REFERENCE...

https://www.open-mpi.org/faq/?category=tcp

Listing 5.4: /etc/sysctl.d/picluster.conf

```
1 net.core.rmem_max = 16777216
2 net.core.wmem_max = 16777216
3 net.ipv4.tcp_rmem = 4096 87380 16777216
4 net.ipv4.tcp_wmem = 4096 65536 16777216
```

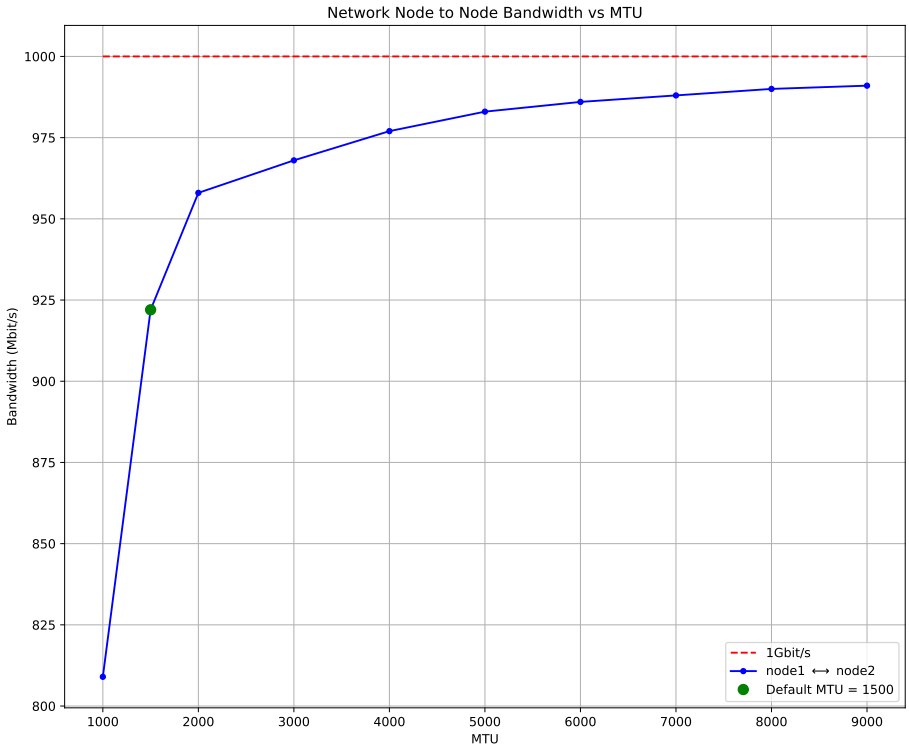


Figure 5.5: Network Node to Node Bandwidth vs MTU.

```
net.core.netdev_max_backlog = 30000
net.core.rmem_default = 16777216
net.core.wmem_default = 16777216
net.ipv4.tcp_mem= 16777216 16777216 16777216
net.ipv4.route.flush = 1
```

```
sudo sysctl --system
```

or

```
sudo shutdown -r now
```

```
Aug 11 03:35:40 node5 kernel: [19256.425779] bcmgenet
↪ fd580000.ethernet eth0: bcmgenet_xmit: tx ring 1 full when
↪  queue 2 awake
```

5.5.5 reclaim memory

A closer look at the memory use above indicates that 65536k of memory is being used as *cma-reserved*. This Contiguous Memory Allocator (CMA) memory is reserved at boot time for certain kernel drivers, in particular some video drivers. Since the Aerin Cluster is not using video, it may be possible to reclaim some of this memory to increase the amount available for the benchmark problem size.

The `/proc` filesystem enables access to kernel data structures at run time. Running the following command it is possible to see how the *cma-reserved* memory is being utilised:

```
$ cat /proc/meminfo | grep Cma
```

```
CmaTotal:          65536 kB
CmaFree:           63732 kB
```

As can be seen, the majority of *cma-reserved* memory is not being used. So, although this is a relatively small amount of memory on a single node, it is approximately 0.5 GB across all 8 nodes. This is worth trying to reclaim for the benchmark problem size via a rebuild of the kernel, something that is investigated later.

Chapter 6

Summary