High Performance Task

Jamie Atiyah

The entirety of this work is my own. Files used in this project are available at
https://stummuac-my.sharepoint.com/:u:/g/personal/22552426_stu_mmu_ac_uk/
EZGl7g92YwJAgAtwL8LxKsUB8rMFdpdccW1CVVHgfzNszQ?email=M.Bane%40mmu.ac.uk&e=sRDjrv

Faculty of Science and Engineering
Manchester Metropolitan University
UK
16.05.2023

I. Introduction

Computational power of modern day technology is increasing at a rapid rate. This progression was first observed in 1965 in Moore's law (Moore 1965). Moore's Law predicted that computer performance was doubling every two years due to an increase in speed of the transistors (a semi conductor device). Figure 1 shows the calculations per second per constant dollar on a logarithmic scale against the year. The plot shows a strong positive correlation, illustrating the near exponential increase in computational power over the years. With the increase in available processing power, solutions are being found to more problems, which are often characterised by higher levels of complexity. High performance computing (HPC) systems are being utilised in a wide range of scenarios, from meteorological scientists (Nakaegawa 2022) to engineers who look to advance vehicle technology through the use of quantum computing technology (Ignácz, Feszty, and Lakatos 2022). However, as the clock speedy increases, the power consumed rapidly increased 1. Whilst it is possible to deal with the complications that arise from an increase in power consumption, it was found that increasing the number of cores and to have them work concurrently is of more use than increasing clock speed (Zhen et al. n.d.).

$$p \propto f^n \tag{1}$$

This report will examine the parallel scalability of provided source codes which determine the integral of a given function between two points on both ARCHER2 central processing unit (CPU) and graphics processing unit (GPU) hardware. The performance of both the parallel code written in CUDA C, which has been ran on the GPU of the Manchester Metropolitan University's Department Of Computing And Mathematics' Linux machines, and the C code that calls message passing interface (MPI) to run on compute nodes of ARCHER2 was assessed.

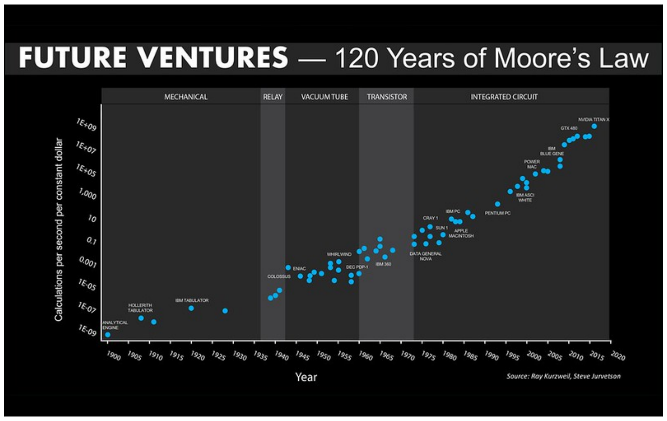


Figure 1. Moore's Law over 120 Years (Jurvetson and Kurzwell 2016)

II. Background

A. Terminology

HPC clusters are typically comprised of many nodes (Bane 2023), with each node having one or more sockets which can each hold a processor. Each of these processors has one or more cores.

B. Working in Parallel

Within a given problem, if the problem is able to be split into elements such that each element can be processed independently and the order of how the elements are processed is unimportant, the problem can therefore be parallelised. The benefits of dividing the work up by distributing to parallel processing elements is a quicker time to solution. Furthermore, if the work is divided up by using more nodes, the amount of total memory available increases. In parallel programming, the aim is to distribute the work evenly so that the work is finished at the same time, so as to prevent a load imbalance. Load imbalance in HPC systems in HPC has been found to cause resource contention (Neuwirth et al. 2017), reducing the efficiency of the parallel program. When used effectively, Parallelism is "central to the efficient use of resources" (Buell 2011) in HPC.

C. Amdahl's Law

In 1967, Dr. Gene M. Amdahl (Amdahl 2007) published an article which reviewed the capabilities of running computer programs in parallel. Amdahl stated that the computer will never go faster than the sum of the parts that do not run in parallel, no matter how many processing elements are used. The sections of the program that do not run in parallel are called the serial portions and run in a one object at a time process. When letting α denote the serial proportion of the code, p the number of cores, and T_p the time on p cores, S_p the speed-up on p cores,

$$T_p = \alpha T_1 + (1-\alpha)\frac{T_1}{p} \tag{2}$$

$$S_p = \frac{T_1}{T_p} = \frac{1}{(\alpha + (1-\alpha)/p)} \tag{3}$$

In regards to the maximum speed-up, S_{max} the maximum speed up is only dependent on the proportion of serial code when p tends to infinity 4.

$$S_{max\, p\to\infty} = \frac{1}{\alpha} \tag{4}$$

It can be argued that Amdahl's law lacks practicality. For instance, Amdahl's law fails to take into account the time for the parallel nodes to communicate to each other. Additionally, in 1987, Gustafson's highlights how Amdahl's law does not account for the availability of computing power as the machine grows (Ślesicki, Kawalec, and Ślesicka 2022). Whilst flaws within Amdahl's law exist, the law does well at illustrating the need to reduce the serial proportion as much as possible.

III. CPU

CPU architecture is excellent for general purpose. There are generally several cores involved, which each core being highly powerful with a fast clock cycle. At any given time, each of these cores could be doing different work. The cores can do out of order, speculative execution.

A. MPI Architecture

MPI is recognised as being one of the main programming architectures for writing efficient parallel applications (Mamidala et al. 2008). MPI can be used as a distribute memory architecture and also as a shared memory architecture. MPI utilises many nodes that are all connected together. The aim of MPI is to fairly distribute work over multiple cores spread over multiple nodes. When MPI is used as a distributed memory architecture, the same program is ran on each physical core of each node. Each program runs the same source code but using its own data as the actual value of the variables differs which leads to differing computations. To differentiate the different MPI processors on the physical processor cores, each MPI processor has a unique rank. There is a requirement for any method that the nodes communicate with one another. There are various models by how the message passing takes place.

1) Collective: In a collective, all processes participate. Collectives generally allow the sending of a vector values. There are several collective methods that exist.

MPI Broadcast	The same data is broadcast to all ranks from the root rank.
MPI Scatter	The data is distributed between all of the ranks with each rank being allocated a data chunk of the same size.
MPI Gather	The inverse of the MPI Scatter collective. The data is scattered across all the ranks in equally sized data chunks and is gathered into one big array on rank 0.
MPI Reduce	Implements mathematical reduction which brings together the partial results from each rank.

2) Point to Point: Point to point communication is a pairwise exchange between two MPI processes where one MPI process is a sender of data and the other is a receiver of data. Each message has an envelope which has the ranks of the receiver and sender process, a tag and a communicator. Each message also has data. For a successful completion of the message, the following requirements need to be met

1) The receiver's envelope must match the envelope of the sender.
2) The data types must match

IV. GPU Architecture

In comparison to the CPU based MPI architecture, a GPU architecture employs cores with much less capability. Whilst the cores have been reduced to being of low capability, GPU employs a large number of cores, which contributes to the peak performance of GPU outperforming CPU (Asano, Maruyama, and Yamaguchi 2009). GPU cores look to work as quickly as possible and concurrently. Within the GPU, there can be 1000s of lightweight threads. Each thread has a unique identifier (index) and runs a copy of the kernel. Therefore, potentially 1000s of copies of the kernel is ran which brings the potential for 1000 times speed-up. The unique identifier helps with parallelising the work of the kernel. The threads are held in thread blocks which are executed by streaming multiprocessors (SM). Threads within a block can be synchronised and have access to memory shared between the threads within the block. Thread blocks also have access to the GPU global memory.

The GPU sits on a PCI express so there is a requirement for a data transfer from the host. Therefore the off loading and subsequent cost of the data transfer needs to be considered.

V. Methodology

A. The Problem Domain

The problem domain for which the performance of the GPU and the CPU hardware is determining the integral of $f(x)$ from $x = a$ to $x = b$ 7. The integral can be approximated by taking the sum of areas beneath the curve where each area is approximated by a rectangle. Figure 2 shows a simple example of the problem with $f(x) = x^2 + 4$ and the area of beneath the curve between $x = 1$ and $x = 5$ being approximated by 8 rectangles. Scaling up this simple problem is complex and will require a high number of computations to be performed. However, using similar principles illustrated in figure 2 by splitting the area into local areas the problem can is able to calculated in parallel as each local area can be calculated independently. For this report, the bounds of the function $f(x)$ were set to $a = 0.1$ and $b = 500.1$ with 4096000 set as the number of quads.

$$\int_a^b f(x)\,dx \tag{5}$$

B. Experimental Setup

1) ARCHER2: The C source code that calls MPI ran on compute nodes of ARCHER2. Each processor in ARCHER2 has 64 cores and each node has 2 processors. The code was compiled using the Cray compiler and an executable code is created.

```
cc ASSESS.c –o ASSESS.exe
```

The executable code was moved to a work directory so the compute nodes can be used. The code was then submitted to batch and the code was run using various parameters.

```
mv ASSESS.exe ${WORK}/
```

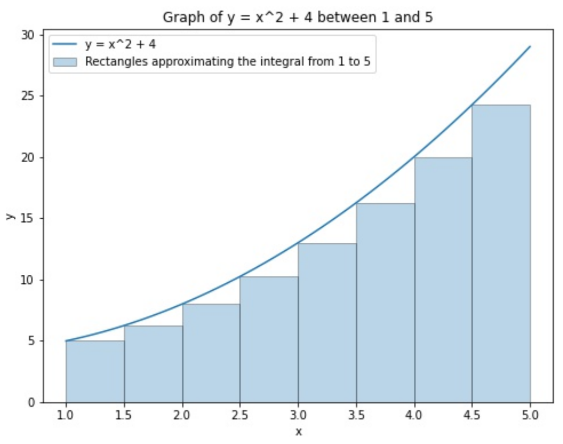
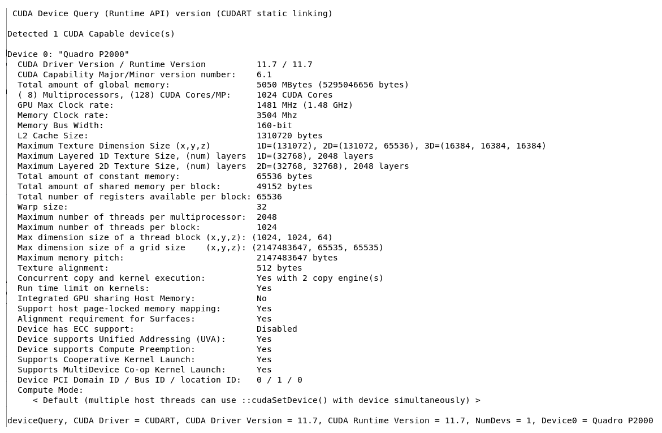


Figure 2. Simple Illustration of the Problem Domain

2) NVIDIA CUDA: The CUDA C code was ran on the GPU of the Manchester Metropolitan University's Department of Computing and Mathematics' Linux machines. Using the 'deviceQuery' call (in the below code), the properties of machine's GPU was provided. The GPU is the Quadro P2000. There are 8 streaming multiprocessors, each with 128 CUDA cores resulting in a total number of 1024 CUDA cores available.

```
/usr/local/cuda-11.7/extras/demo_suite/
    ↪ deviceQuery
```



```
 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "Quadro P2000"
  CUDA Driver Version / Runtime Version          11.7 / 11.7
  CUDA Capability Major/Minor version number:    6.1
  Total amount of global memory:                 5050 MBytes (5295046656 bytes)
  ( 8) Multiprocessors, (128) CUDA Cores/MP:     1024 CUDA Cores
  GPU Max Clock rate:                            1481 MHz (1.48 GHz)
  Memory Clock rate:                             3504 Mhz
  Memory Bus Width:                              160-bit
  L2 Cache Size:                                 1310720 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 11.7, CUDA Runtime Version = 11.7, NumDevs = 1, Device0 = Quadro P2000
```

Figure 3. Device Query Output

The NVIDIA CUDA/C compiler was used to compile the CUDA C code. There was no necessity for the CUDA C executable to be ran in a work directory as the Linux machine had no compute node so there was no need for a fast interconnect.

```
nvcc quad-cuda-ASSESS.cu -o quad-cuda-ASSESS.
    ↪ exe
```

The executable is then ran with the value of the integral and the wall clock time being outputed.

C. Provided Code Overview

1) MPI Solution with MPI reduction: The MPI environment is called in the command MPI_Init. The number of available MPI process 'numProcesses' is defined using the MPI_Comm_size function and is stored in the communicator MPI_COMM_WORLD. The variable myNum is the same for all the MPI processors. As stated in section 3.A, each MPI process has a unique rank. The rank is defined using MPI_Comm_Rank and puts it into the variable 'rankNum'.

```
 MPI_Comm_size(MPI_COMM_WORLD, &numProcesses);
MPI_Comm_rank(MPI_COMM_WORLD, &rankNum);
```

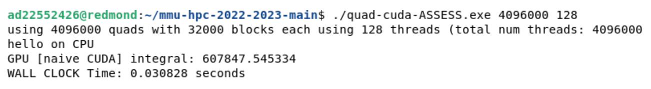
The number of quads per MPI process ('numberQuads_perProcess') is calculated by dividing the number of quads (4096000) by the number of available MPI processes. The variables start_perProcess ↪ and finish_perProcess, which denote the starting and ending index for the elements that an MPI process respectively, will be allocated are calculated using the established variables rankNum, numberQuads_perProcess, numProcesses, and numberQuads.

```
int start_perProcess = rankNum*
    ↪ numberQuads_perProcess;
if (rankNum == numProcesses-1)
    ↪ numberQuads_perProcess = numberQuads - (
    ↪ numberQuads_perProcess*rankNum);
int finish_perProcess = start_perProcess +
    ↪ numberQuads_perProcess - 1;
```

The work was divided up between the MPI processes and local sums calculated. In the provided MPI ASSESS.c code, MPI_Reduce is utilised to bring together the partial results from each rank into a global sum called integral.

```
MPI_Reduce(&localSum, &integral, 1, MPI_DOUBLE,
    ↪ MPI_SUM, 0, MPI_COMM_WORLD);
```

The wall clock time is returned with the value of the approximated integral (as shown in figure 4).



```
ad22552426@redmond:~/mmu-hpc-2022-2023-main$ ./quad-cuda-ASSESS.exe 4096000 128
using 4096000 quads with 32000 blocks each using 128 threads (total num threads: 4096000
hello on CPU
GPU [naive CUDA] integral: 607847.545334
WALL CLOCK Time: 0.030828 seconds
```

Figure 4. Example GPU output when executed with 128 threads per thread-block

2) Point to Point Solution: The provided code was also amended so that a point to point communication method was employed instead of the reduction method. The point to point communication uses the methods MPI_Send() and MPI_Recv().

3) Batch scripts: Two different batch scripts were provided in the experiment. The batch_mpi_scaling.sh script uses scaling to assess the optimal value of cores by using a loop where the executable is run of a varying number of cores. Furthermore, the flag '-N x' also changes the number of nodes to x number of nodes.

4) CUDA solution for Quadrature: A parallel kernel is invoked by the use of CUDA which looks at running multiples copies of the kernel by the command __global__. The code sets the maximum number of threads as 1024 as shared memory is utilised which works faster than using global memory. Each thread within a given block has a unique number which is denoted as threadIdx.x. Thread blocks are uniquely numbered and denoted as blockIdx.x. Each thread is then assigned a unique value i which denotes its thread number out of all threads across all thread blocks. Each thread is also assigned a value p which denotes the thread number within a given block.

The total area to be calculated is split into subareas. Each thread calculates a subarea which is then reduced into block subareas (i.e. the subareas of the threads within a block are all added up). The space needed for the device block area is determined using the cudaMalloc call.

```
for (int t=1; t<myNumThreads; t++) {
        myBlockArea += myThreadArea[t];
```

The area underneath the curve between the bounds established in section 5.A is determined by calling the below command which requests the number of blocks and threads per block and passes values for the lower bound, the calculated width of each trapezoidal, the number of quads (which is required to be entered as 4096000) and the newly freed memory space deviceBlockArea.

```
calcMyArea<<<blks, tpb>>> (a, width,
    ↪ numberQuads, deviceBlockArea);
```

D. Performance Metrics

The performance was assessed by evaluating the speed-up and the efficiency. The speed-up (S) and efficiency (E) are calculated using the time t and number of cores p. The ideal speed up is found when $S(p) = p$.

$$S(p) = \frac{t_1}{t_p} \tag{6}$$

$$E(p) = \frac{S(p)}{p} \tag{7}$$

VI. Evaluation

A. Value of the Integral

The codes used in this experiment all obtained the value of the integral as 607847.545334.

B. MPI Performance

The performance of the MPI algorithm was assessed over multiple nodes for multiple cores. Figure 5 shows the performance of the MPI algorithm for the metrics speed up, total time taken, and efficiency. There is a clear positive correlation between the number of cores and the speed up when using 1, 2 or 4 nodes. The increase of the speed up as the number of cores for 1 and 2 nodes is nearly constant whilst the rate by which the speed up increases when using 4 nodes is more variable, with a decrease in rate between 80 and 100 cores and then a sudden, dramatic increase between 100 and 127 cores. The total time taken follows a negative correlation as increasing the number of cores decreases the total time taken. Between 0 and 20 cores there is a sharp decline in the total time taken for all the node values. When increasing the number of cores beyond 40, the total time decreases at a much slower rate. When looking at the the time taken for the number of cores on a log-log scale, the varying gradients for 4 nodes indicates that the distribution does not always follow a negative exponential. This may be caused by overhead which can reduce the impact of the parallel processing. Many of the efficiency values are above 100 %, indicating a 'superlinear speedup' (Ristov et al. 2016) as the speed up value on a particular number of cores exceeds the number of cores (as shown in Table 1). The superlinearity ends (the efficiency falls beneath 100%) for a varying number of cores for the different number of nodes. The superlinearity ends with the highest amount of cores for 2 nodes and ends with the lowest amount of cores for 4 nodes. This highlights how overhead effects the efficiency of the parallel program, as the superlinearity ends earlier when more nodes are used, hence increasing the number of cores.

The performance of the MPI solution was assessed depending on whether MPI reduction or point to point communication was employed. Figure 6 shows that the performance of the MPI solution is drastically better when MPI reduction communication in comparison to when point to point communication is used. The dramatic difference in speed up and efficiency may be due to the MPI_Send() and MPI_Recv() functions are blocking methods, resulting in sending processes having to wait until the message has been sent correctly, and the receiving process having to wait until the message has been received correctly.

Table 2 shows the time taken for each rank for the MPI code with MPI reduction to run when ran with 2 nodes and 32 cores. The difference in timings between the ranks may suggest that the work could be split up better.

C. GPU Performance

The performance of the GPU algorithm was assessed as a function of the number of thread-blocks and threads running on the GPU. The values used for the number of threads per GPU thread-block were all multiples of 32 as the warp size is 32 (shown in Figure 3). The number of quads was set to 4096000 as required for this experimentation. Figure 7 shows the wall clock time for varying number of threads per GPU thread-block. Generally, as the number of threads per GPU thread-block increases, the wall clock time increases. The results are stored in table 3 and show that setting the number of threads per GPU thread-block as 128 provides the smallest wall clock time. Using 128 threads per thread-block results in there being 16 thread-blocks on a streaming multiprocessor with the maximum number of

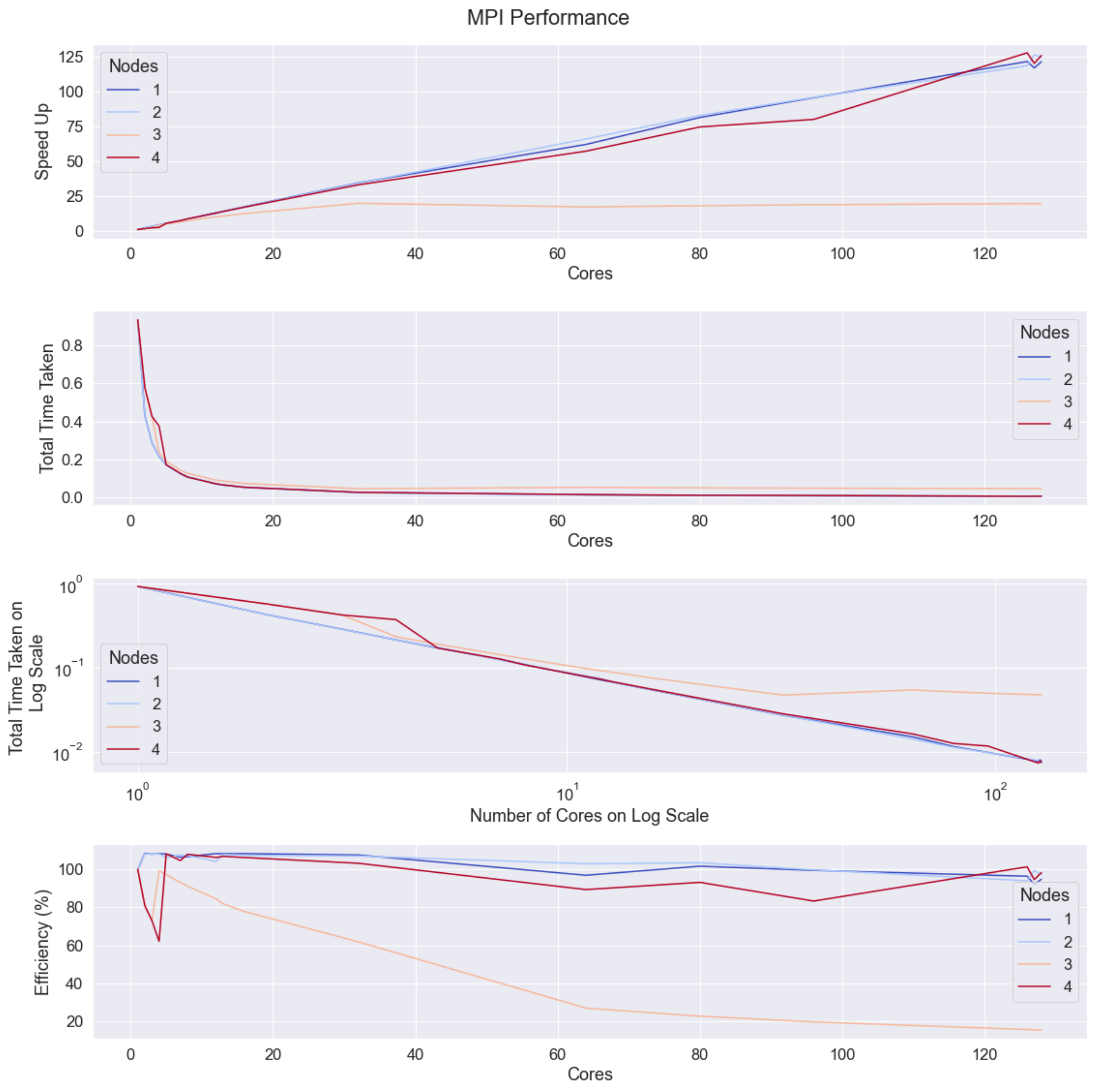


Figure 5. Speed Up, Total Time Taken, and Efficiency for varying number of cores and nodes

threads on a streaming multiprocessor being used (2048 threads).

D. Comparison

Comparing the times taken between the GPU algorithm when a value of 128 threads per thread-block is used against the MPI reduction algorithm with 2 nodes and 80 cores used reveals that the CPU algorithm performs quicker than the GPU algorithm (figure 8). A look at the distribution of the total time taken for the GPU and MPI algorithms reveals that the distribution of the total time taken for the MPI algorithm is highly dispersed in comparison to the GPU algorithm (shown in figure 9), highlighting that whilst better times can be acheived using the MPI architecture, the number of nodes and cores specified is highly important. The timings of the GPU algorithm may have been high due to the communication time between the GPU and CPU as data transfer from the host is a requirement due to the GPU

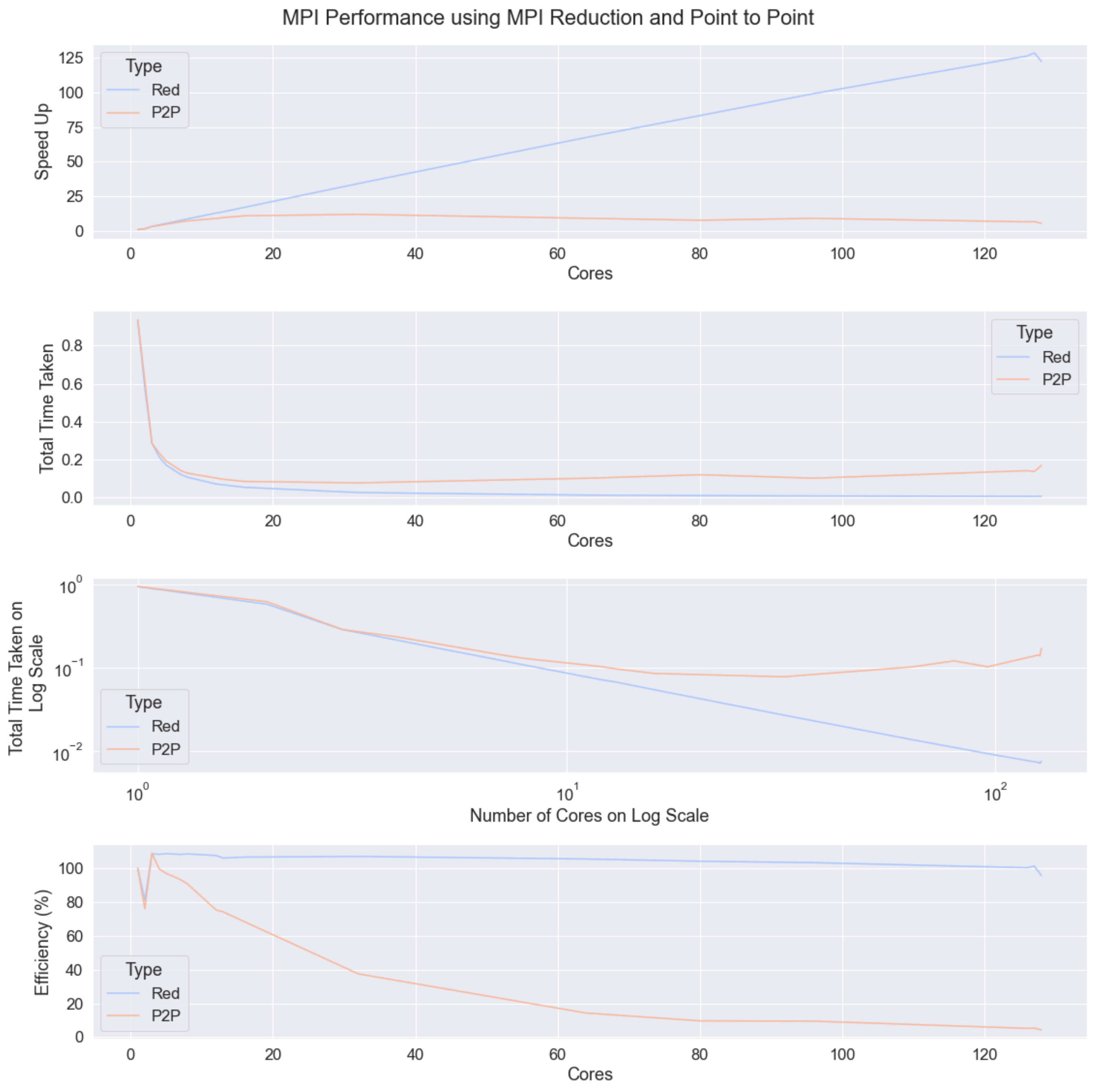


Figure 6. Speed Up, Total Time Taken, and Efficiency for MPI reduction & Point to Point communication methods

sitting on a PCI express.

VII. Conclusion

The objective of this report was to examine the parallel scalability of provided source codes which determine the integral of a given function between two points on both ARCHER2 central processing unit (CPU) and graphics processing unit (GPU) hardware. The experiments found that the CPU source code completed the code quicker than the GPU, indicating that the work was paralellised better on the CPU compared to the GPU. Further work in improving the performance of the GPU algorithm could look at studying the time spent performing the data transfer to the host and assess if any improvement can be gained. Further work in improving the performance of the MPI algorithm could look to study the varying communication methods and assess if opting to use a collective method such as MPI gather leads to an improvement in performance.

Table I
MPI PERFORMANCE

MPI Performance				
Nodes	**Cores**	**Total Time Taken**	**Speed Up**	**Efficiency (%)**
1	1	0.933461	1	100
1	2	0.430018	2.170748666	108.5374333
1	3	0.288043	3.240700173	108.0233391
1	4	0.215373	4.334159806	108.3539952
1	5	0.172754	5.403411788	108.0682358
1	7	0.125266	7.451830505	106.4547215
1	8	0.109318	8.538950585	106.7368823
1	12	0.071779	13.00465317	108.3721098
1	13	0.066306	14.0780774	108.2929031
1	16	0.05385	17.33446611	108.3404132
1	32	0.027102	34.44251347	107.6328546
1	64	0.01505	62.02398671	96.91247924
1	80	0.011475	81.34736383	101.6842048
1	96	0.009772	95.5240483	99.50421698
1	126	0.007685	121.4653221	96.40104925
1	127	0.007985	116.9018159	92.04867394
1	128	0.007693	121.3390095	94.79610116
2	1	0.930929	1	100
2	2	0.429935	2.165278472	108.2639236
2	3	0.287206	3.241328524	108.0442841
2	4	0.215043	4.329036518	108.225913
2	5	0.175234	5.312490727	106.2498145
2	7	0.123569	7.53367754	107.6239649
2	8	0.108309	8.595121366	107.4390171
2	12	0.074477	12.4995502	104.1629183
2	13	0.066421	14.01558242	107.8121725
2	16	0.054137	17.19579955	107.4737472
2	32	0.027176	34.25555637	107.0486137
2	64	0.014122	65.92047869	103.0007479
2	80	0.011244	82.79340092	103.4917512
2	96	0.009721	95.76473614	99.75493348
2	126	0.007859	118.4538745	94.01101154
2	127	0.007373	126.2619015	99.4188201
2	128	0.007477	124.5056841	97.2700657
3	1	0.932139	1	100
3	2	0.575392	1.620006882	81.00034411
3	3	0.425309	2.191674759	73.05582529
3	4	0.234598	3.973345894	99.33364735
3	5	0.191693	4.862665825	97.2533165
3	7	0.143108	6.513535232	93.05050332
3	8	0.12782	7.292591144	91.1573893
3	12	0.091982	10.13392838	84.44940314
3	13	0.087591	10.64194952	81.86115012
3	16	0.074897	12.44561197	77.78507484
3	32	0.047145	19.77174674	61.78670856
3	64	0.054322	17.1595118	26.81173719
3	80	0.051658	18.04442681	22.55553351
3	96	0.049725	18.74588235	19.52696078
3	126	0.047653	19.56097203	15.52458097
3	127	0.047816	19.49429061	15.34983513
3	128	0.047512	19.61902256	15.32736138
4	1	0.932055	1	100
4	2	0.576682	1.616237372	80.81186859
4	3	0.426039	2.187722251	72.92407503
4	4	0.375427	2.482653086	62.06632714
4	5	0.172229	5.411719281	108.2343856
4	7	0.127213	7.326727614	104.6675373
4	8	0.107875	8.64013905	108.0017381
4	12	0.073056	12.75808968	106.317414
4	13	0.06708	13.894678	106.8821384
4	16	0.054819	17.00240792	106.2650495
4	32	0.028214	33.03519529	103.2349853
4	64	0.016297	57.19181444	89.36221007
4	80	0.012504	74.54054702	93.17568378
4	96	0.011655	79.97039897	83.30249893
4	126	0.007298	127.7137572	101.3601248
4	127	0.007755	120.1876209	94.63592196
4	128	0.007407	125.8343459	98.30808273

Table II
TIMINGS PER RANK

Timings per rank		
Rank	**Whole Code Timing**	**Loop Timing**
12	0.789919	0.654922
29	0.656426	0.65635
19	0.651789	0.651688
21	0.651408	0.651339
26	0.668147	0.656002
20	0.708944	0.65196
25	0.655591	0.655487
27	0.668113	0.668054
17	0.654566	0.654512
22	0.708919	0.705331
31	0.708914	0.708831
3	0.708887	0.708837
18	0.708179	0.708033
24	0.70896	0.668514
28	0.708961	0.656105
30	0.708935	0.656347
16	0.708978	0.653034
1	0.653983	0.653902
2	0.790709	0.656336
6	0.655716	0.655434
5	0.775241	0.775166
8	0.789931	0.653131
10	0.777242	0.655358
3	0.79069	0.790611
7	0.655698	0.655643
9	0.655096	0.654984
4	0.775249	0.652559
11	0.777236	0.777171
13	0.656617	0.656519
15	0.657598	0.65754
14	0.789905	0.789835
0	0.790722	0.655873

Table III
GPU PERFORMANCE FOR THE NUMBER OF THREADS PER GPU THREAD-BLOCK

Number of threads per GPU thread-block	**Wall Clock Time**	**Total Number of Threads**
1024	0.036047	4096000
896	0.035263	4096512
768	0.037444	4096512
640	0.031535	4096000
512	0.029998	4096000
384	0.028978	4096128
256	0.029124	4096000
128	0.028498	4096000
64	0.02897	4096000
32	0.029834	4096000

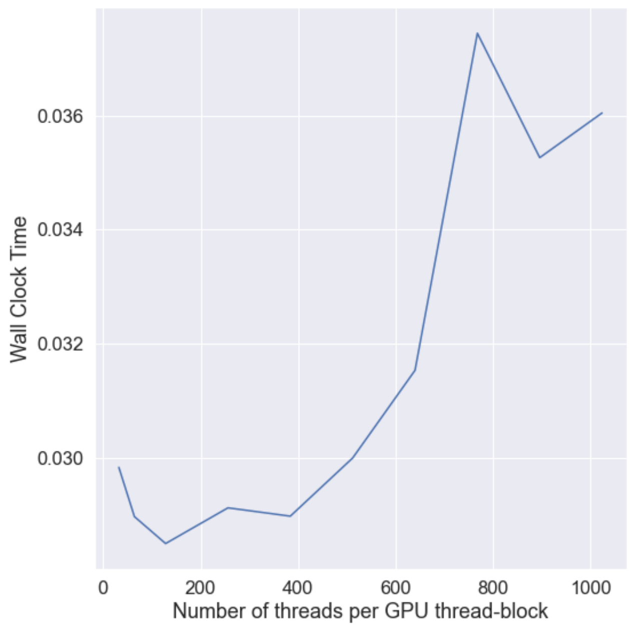


Figure 7. Total Time Taken per Number of Threads per GPU Threadblock

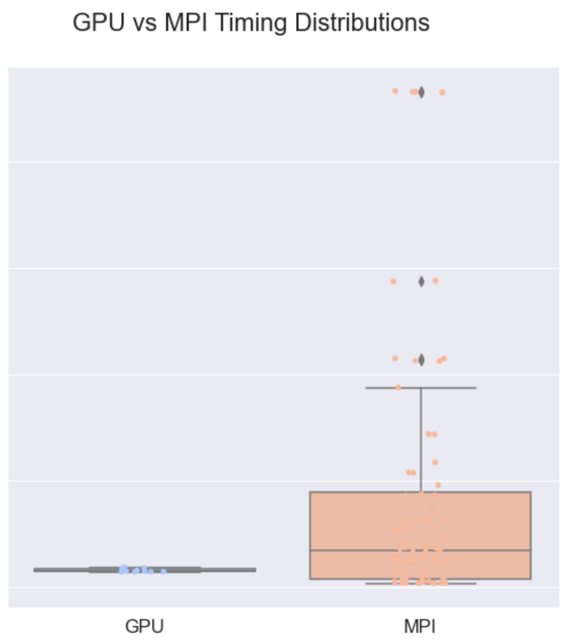


Figure 9. Distribution of timings for MPI and GPU)

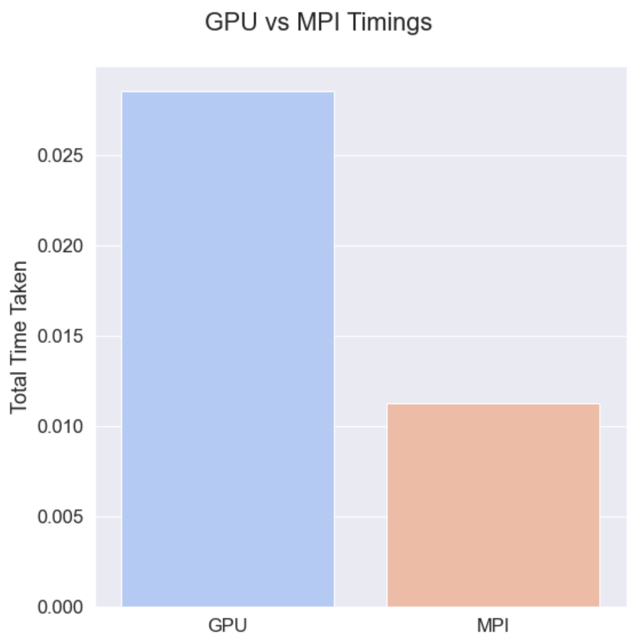


Figure 8. Total Time taken for MPI (4 nodes, 80 cores) and GPU (128 threads per thread-block)

References

Amdahl, Gene M (2007). "Validity of the single processor approach to achieving large scale computing capabilities, reprinted from the afips conference proceedings, vol. 30 (atlantic city, nj, apr. 18–20), afips press, reston, va., 1967, pp. 483–485, when dr. amdahl was at international business machines corporation, sunnyvale, california". In: *IEEE Solid-State Circuits Society Newsletter* 12.3, pp. 19–20.

Asano, Shuichi, Tsutomu Maruyama, and Yoshiki Yamaguchi (2009). "Performance comparison of FPGA, GPU and CPU in image processing". In: *2009 international conference on field programmable logic and applications*. IEEE, pp. 126–131.

Bane, Michael (2023). *Glossary of HPC terminology*. URL: https://moodle.mmu.ac.uk/mod/glossary/view.php?id=3789596 (visited on 03/23/2023).

Buell, Duncan (2011). *In Praise of An Introduction to Parallel Programming: An introduction to parallel programming by Pacheco, Peter*. Elsevier.

Ignácz, Ferenc, Dániel Feszty, and István Lakatos (2022). "Possible Applications of Quantum Computing, Especially in Vehicle Technology: A Review Article". In: *Periodica Polytechnica Transportation Engineering* 50.3, pp. 245–251.

Jurvetson, Steve and Ray Kurzwell (2016). URL: https://www.flickr.com/photos/jurvetson/31409423572/in/photostream/.

Mamidala, Amith R et al. (2008). "MPI collectives on modern multicore clusters: Performance optimizations and communication characteristics". In: *2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*. IEEE, pp. 130–137.

Moore, Gordon E (1965). *Cramming more components onto integrated circuits*.

Nakaegawa, Tosiyuki (2022). "High-Performance Computing in Meteorology under a Context of an Era of Graphical Processing Units". In: *Computers* 11.7, p. 114.

Neuwirth, Sarah et al. (2017). "Automatic and transparent resource contention mitigation for improving large-scale parallel file system performance". In: *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, pp. 604–613.

Ristov, Sasko et al. (2016). "Superlinear speedup in HPC systems: Why and when?" In: *2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*. IEEE, pp. 889–898.

Ślesicki, Błażej, Adam Marian Kawalec, and Anna Ślesicka (2022). "The study of the possibility of applying parallel programming to the algorithms of Space-time adaptive processing". In: *Problemy Mechatroniki. Uzbrojenie, lotnictwo, inżynieria bezpieczeństwa* 13.3.

Zhen, Meiyuan et al. (n.d.). "Time Parallel Computation of Unsteady Flows with Multigrid Reduction in Time". In: *Available at SSRN 4353552* ().