Bite-sized Bytes

David Nelson

May 2021

Contents

Abstract

This dissertation explores how uses parallelism to enhance common problems in modern computing. We aim to decompose standard computer algorithms such as Monte Carlo Analysis and Matrix Multiplication into parallel counterparts to achieve good parallel performance. We obtain solid performance, especially from the MPI parallel program, by substantially decreasing program runtimes for high processor counts. Further improvements can be made to parallel systems by incorporating an array of techniques and system structures.

Chapter 1
Introduction

Imagine owning a business that makes hand-made blanket. You only have one employee, yourself. Sales are increasing rapidly, so you want to enhance the product output, but there is a physical limit to how fast one person can make a blanket. So you hire another person, and now the business can make blankets twice as fast! That is the idea behind parallel computing; a Processor can only get so fast given current technology, so rather than trying to make a computer run more quickly, we can put two computers or "cores" inside one computer to double the performance.

This document aims to research the nature of *Parallel computation*, defining what parallelism means, why it is valuable, and its limitations. We aim to demonstrate significant improvements in the runtime of a program thanks to the power of parallelism.

Parallelism has applications in many areas, particularly in areas of research where enormous amounts of data are required, such as creating numerical weather forecasting models and digital material modelling. Moreover, there are applications more abstract areas, like linear algebra operations, for example, parallel Eigenvalue solvers.

We begin by looking briefly at serial computation to get an idea of how a computer works without parallelism. We follow this by defining Parallel computation as splitting a typical serial computation into multiple parallel parts that can run simultaneously on separate cores. Parallelism allows us to dramatically reduce the run time of specific problems, especially those containing many 'independent' parts which do not need to run in any specific order.

We establish some critical metrics for analysing the performance of parallel systems such as Parallel Speedup, Parallel Efficiency and FLOPS. Then we look at the concept of scalability in parallelism, which measures how parallel performance varies relative to the number of processors used. We split scalability into two main groups; Strong Scaling, in which the size of the overall problem stays constant with respect to the number of processors and Weak Scaling, in which overall problem size scales linearly with the number of processors. Next, we look at the two main hardware formats that parallel programs can run on; Shared memory, in which all of the processors in a parallel program have access to the same data and Distributed memory, in which all the processors have a separate data source.

Next, we set the stage for the two problems we will focus on for the rest of the report; Monte Carlo Analysis and Matrix Multiplication. We describe the nature of these problems and show how we will decompose them from serial algorithms into parallel algorithms. Monte Carlo Analysis is an example of an "Embarrassingly Parallel" problem because it is easily decomposed into a parallel program, as the tasks can be handed down to workers without much thought. We then introduce the problem of Matrix Multiplication, which is interesting for this report because it is quite an intensive problem in terms of data size and processing power. The complexity of this problem forces us to use rigorous techniques of problem decomposition and data management. Moreover, matrix multiplication has applications in many modern research areas, such as data science and computer modelling.

We will implement parallel matrix multiplication programs in the shared memory and distributed memory formats to see how performance varies. We execute most of the tests on Python, a prevalent high-level programming language. We also run some tests on Fortran, a programming language commonly used for scientific computing.

Chapter 2
Serial Computation

We define a *core* as the part of a CPU that performs Arithmetic and Logical operations on local memory. A modern CPU can have many cores to take advantage of parallel computation. Traditionally, computers have followed serial computation, meaning that a CPU will have one core, and to execute a program the CPU will execute one instruction at a time, in sequential order. At this point, parallel computers are commonplace, even budget laptops will come with at least a dual-core processor. But it hasn't always been this way, for a long time, computers were restricted to serial computation. This led to advancements in other directions such as 'multitasking', which is a core giving the illusion that it is executing multiple tasks simultaneously, but in actuality, it is performing lots of tasks very fast, one after the other. For example, if you are watching a video on your computer, the time between individual frames may be imperceptible to us, but to a CPU this is enough of a gap to break off from the video do basic system tasks like updating the clock and then come back to the next frame of the video. This is thanks to the idea of multithreading, meaning that tasks are broken down into "threads" that can be executed concurrently, either on one core as seen in (Fig. 2.1) or on multiple cores. So, multithreading can also be implemented in a parallel format; tasks can be broken into threads and sent out to individual cores to execute. We will see later that this of often implemented into library functions in programming languages like Python. For example, if we call a matrix multiplication function in python, such as "matmul" from the library "NumPy", the operating system will see what cores are available and send threads of the function to different cores accordingly.

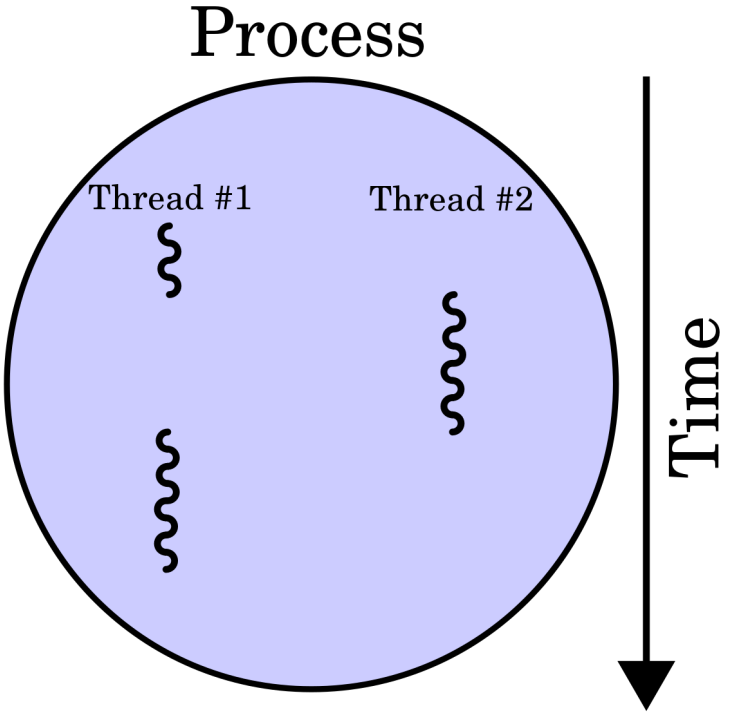

Figure 2.1: Two threads running on a single processor. [9]

We now look into the von Neumann architecture. While computer design has come on a long way since its creation, it can still give insight into the fundamental components of a modern computer. The von Neumann architecture, as seen in (Fig. 2.2), describes the main elements of a computer:

- An input device, such as a keyboard or mouse

- A CPU, containing the Arithmetic/Logic Unit (ALU) which will basic operations and a Control Unit which manages the ALU by performing interrupts;
- A memory unit which contains primary memory i.e. RAM (Random Access Memory) and secondary memory, i.e. HDD (Hard Disk Drive) storage;
- An output device, such as a monitor screen.

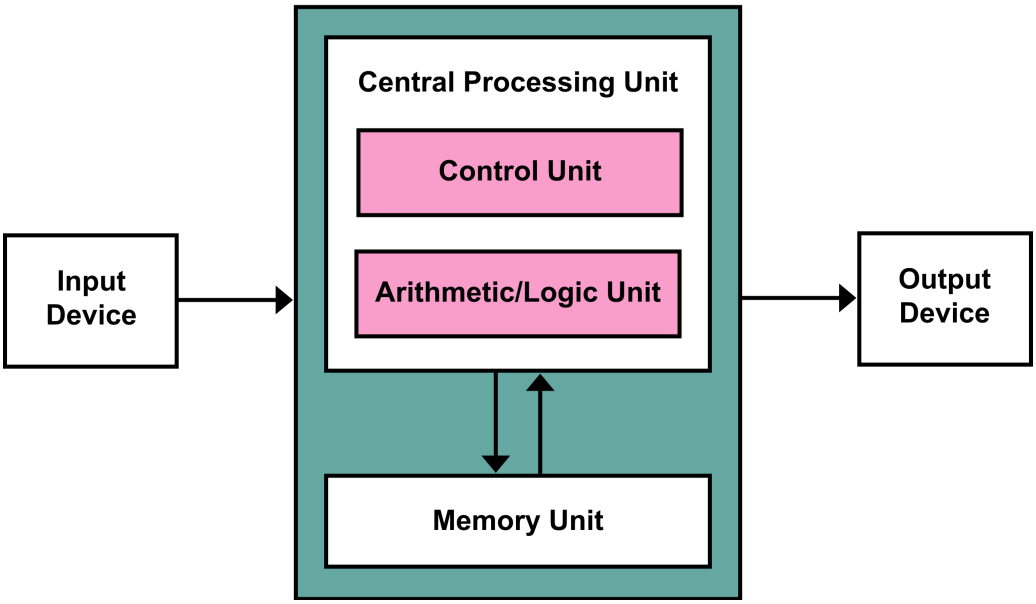

Figure 2.2: Von Neumann Architecture [11]

Following this paradigm, a computer works by following the fetch-decode-execute cycle. Before beginning this cycle, relevant programs and data are loaded into primary memory (RAM) from secondary storage such as a hard disk. Now, the CPU finds the instruction line to be executed, which is in local memory, this instruction is "fetched" and loaded into the CPU's local cache, which is then "decoded" by the control unit, and then "executed" by the ALU, now the next cycle will begin, until the end of the program is reached.

Uniprocessor (serial) computer design has served us well historically; it is relatively easy to program as everything is self-contained. Although, there are physical/technological limits that cannot be surpassed with only one processor, such as short term limits of how small a chip can be made based on current technology and more long term limits like how tiny the transistors within processors will be able to get due to the size of atoms. Even with a focus on advanced techniques of processor design, there comes the point at which we cannot execute a program faster on a single processor, but we can split up the program and execute it on many different processors, leading to a decrease in overall runtime.

Chapter 3
Parallel Computation

We now look at parallel computing, the benefits and drawbacks that it brings and the key terms that we will use to evaluate the performance of parallel systems. In a parallel program, the problem to be executed will be broken down into many smaller tasks which will be executed simultaneously on many cores.

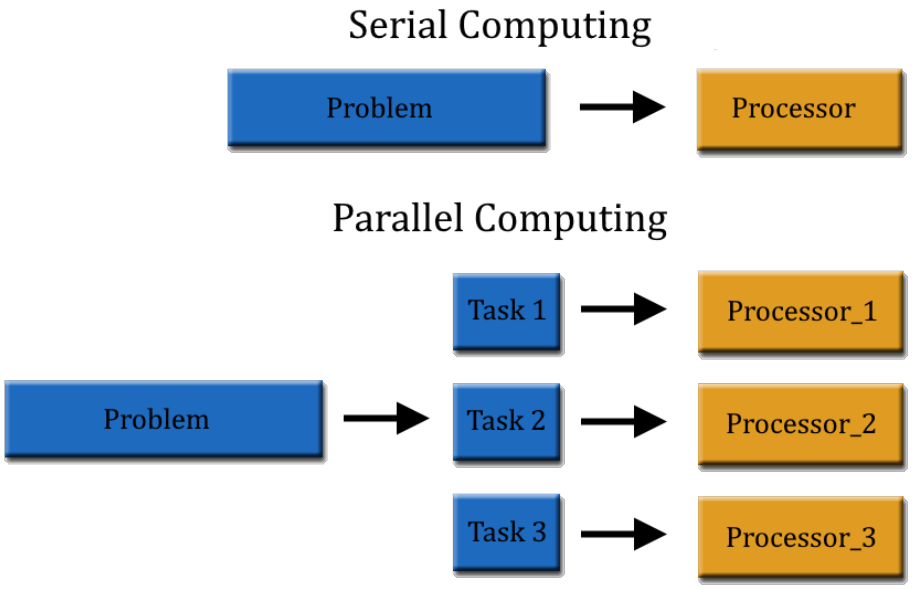

Figure 3.1: A serial program running on 1 processor compared to a parallel program running on 3 processors. [6]

A typical parallel program will consist of one core, referred to as the *master core*, dividing up all of the data to be worked on, and sending it out to all of the other cores, referred to as *worker cores*, to perform various operations. The worker cores will then return some data to the master core, and the program is complete. Some programs are decomposed into this similar format very easily because none of the workers ever need to communicate. However, some programs require workers to constantly send data between each other, making it much harder to implement effectively and leading to worse performance. However, as we will see in Chapter 9, there are parallel programs in which no core is the master, and all of the workers automatically work out whatever work they need to do based on their "rank" in the system. For most of the report, we will view the programs as working in this Master/Slave paradigm as it conveys a lot of the critical concepts of parallelism.

Key Metrics

We Now look at some of the key metrics for evaluating parallel programs that will be used throughout the report. Firstly, we define $T(N, P)$ to be the runtime of a program of problem size N running on P processors.

Speedup

Speedup $S(N,P)$, shows how much faster a program with size N runs on P processors compared to running serially, i.e. $P=1$ processors. [1]

$$S(N,P) = \frac{T(N,1)}{T(N,P)} \tag{3.1}$$

For example, in an ideal situation, a program running on $P=2$ processors should run in half the time that it did on $P=1$ processors, giving a speedup of $S(N,2)=2$. In general we should have $0 \leq S(N,P) \leq P$.

Efficiency

Parallel Efficiency $E(N,P)$ is the ratio of the speedup compared to the number of processors P that are used. Efficiency can be thought of as a measure of the actual speedup that is attained compared to the theoretical maximum speedup that could be achieved on P processors. [1]

$$E(N,P) = \frac{1}{P}\frac{T(N,1)}{T(N,P)} = \frac{1}{P}S(N,P) \tag{3.2}$$

Following from the previous example, the efficiency $E(N,2) = S(N,P)/P = 2/2 = 1$ would be a maximum of 1 as the highest speedup is being achieved for $P=2$. In general we should have $0 \leq E(N,P) \leq 1$.

Normalised Runtime

In some parallel programs, we are expecting the runtime to increase so we use the metric of normalised runtime. We define normalised runtime to be the runtime of the program on P processors normalised with respect to the runtime of the same problem on one processor.

$$NR(N,P) = T(N,P)/T(N,1) \tag{3.3}$$

FLOPS

First, we must define Work Done $W(N)$, which refers to the amount of work that a program must perform to execute a problem of size N, measured in fundamental operations such as multiplication or addition of 2 numbers. Finally, we define FLOPS, meaning "Floating Point Operations Per Second", which measures the work done of a program compared to its runtime.

$$FLOPS(N,P) = W(N)/T(N,P) \tag{3.4}$$

This is a somewhat simplified approach for performing FLOPS, as it is a metric that is often used for evaluating the performance of processors rather than program. In the case of evaluating processors, FLOPS will refer to the number of cycles that a processor can perform per second. However, are more concerned with using it to evaluate how many operations a program will perform per second. [8]

Moreover, we note that 1 KFLOPS $= 10^3$ FLOPS, 1 MFLOPS $= 10^6$ FLOPS, 1 GFLOPS $= 10^9$ FLOPS and 1TFLOPS $= 10^{12}$ FLOPS.

Scalability

Scaling is how the performance of a parallel application changes as the number of processors increases. A program that scales effectively will have runtimes that improve proportionally to the number of processors P. We can break scalability into 2 main measures, Strong scaling and Weak scaling:

Strong Scaling

To measure strong scaling, the size of the total problem N will stay the same as the number of processors P increases, meaning that the metric we will use is speedup $S(N, P)$ as defined previously. In an ideal scenario, the speedup will scale linearly with the number of cores. So, for example, if we are running a perfect strong scaling program on 1 processor, doubling the number of processors to 2 processors should half the program's run time, i.e. a speedup of 2. So in this perfect scenario, the maximum speedup is being achieved, so we would have a parallel efficiency $E(N, 2) = 1$. It follows that in an ideal world, the speedup of a strong scaling program with $E(N, P) = 1$ should follow a linear $y = x$ graph. However, due to limitations that we soon look at, the speedup will begin to plateau at a certain number of processors, as seen in (Fig. 3.2). Typically, for any work that can be parallelised,

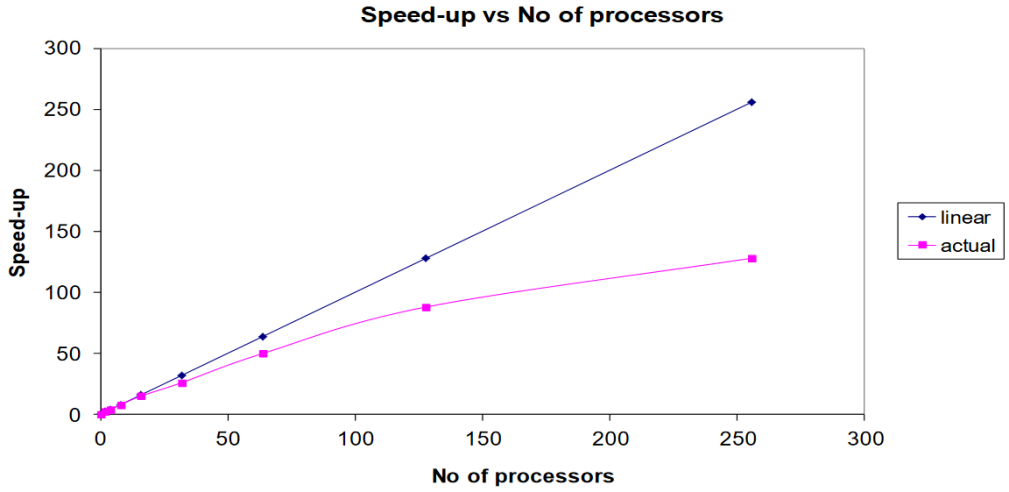

Figure 3.2: How speedup should look for a strong scaling algorithm [1]

some parts must be executed serially. For example, following the blanket factory story from earlier, we find now this factory owner has enlisted 10 employees, but he really prides his business on the quality of his products. So even though he knows that workers are competent, he still will not let a single blanket leave his factory without being quality assured by him. So, although most of the work is parallelised, there is still a bottleneck at the end of the production, which has to be performed serially.

Amdahl's Law states that the speedup of a program as its parallelised is limited by the size of the serial portion, α. Any program has 2 components, one which must be done serially, $\alpha, 0 \leq \alpha \leq 1$, and one which can be parallelised, $1 - \alpha$. Assuming the parallel portion is 100% efficient, the parallel runtime will take the form:

$$T(N, P) = \alpha T(N, 1) + \frac{(1 - \alpha)T(N, 1)}{P}, \tag{3.5}$$

which gives a speedup of:

$$S(N, P) = \frac{T(N, 1)}{T(N, P)} = \frac{P}{\alpha P + (1 - \alpha)}. \tag{3.6}$$

So, as $\alpha \to 0$, $S(N, P) \to P$, but as we can see in (Fig. 3.3), if $\alpha > 0$, there can be a large difference to the maximum speedup: if $\alpha = 0.001$, the speedup is limited to $S(N, 32) = 31.04$, but if $\alpha = 0.01$, the speedup is limited to $S(N, 32) = 24.43$, meaning that if 1% of a program is serial, the maximum speedup at 32 cores is limited to 76.34% of its potential. This means that when creating parallel programs, there has to be much thought put into minimising the serial portion of the problem and parallelising as much as possible.

Weak Scaling

For a weak scaling program, the total problem size N increases linearly with the number of processors P, meaning that each processor will have a regular problem size N to execute. The metric we use to measure the performance of a weak scaling program is normalised runtime $NR(N, P)$, as we are

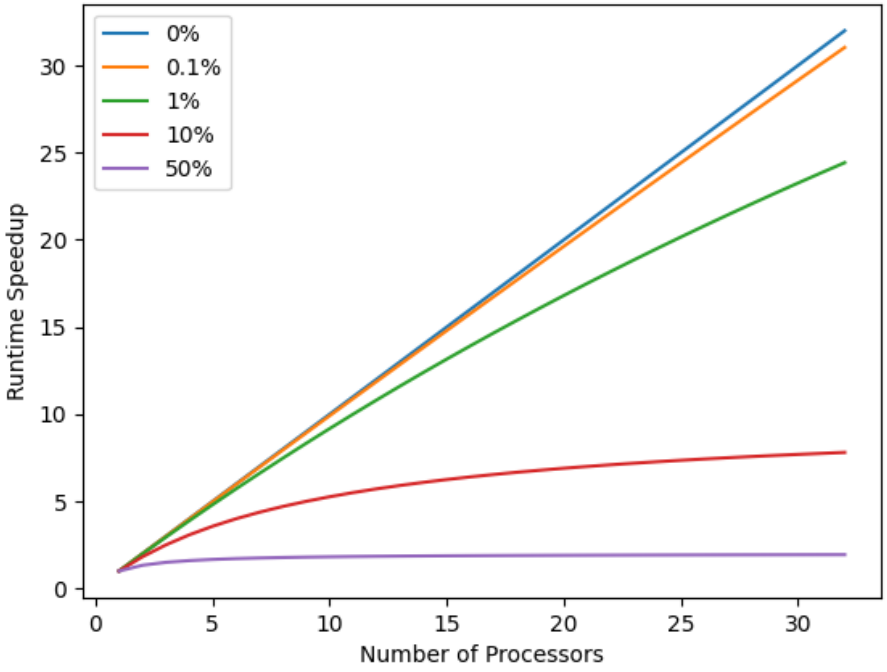

Figure 3.3: The theoretical maximum speedup of a strong scaling program when the serial portion $\alpha = 0, 0.001, 0.01, 0.1, 0.5$ and number of processors $P = 1, 2, 3, \ldots, 31, 32$.

constantly increasing the overall problem size, the runtime will likely increase, or in an ideal case, stay constant. So, if we were to run an ideal weak scaling program on 1 processor, then doubling the number of processors to 2 processors should keep the run time the same. (Fig. 3.4)

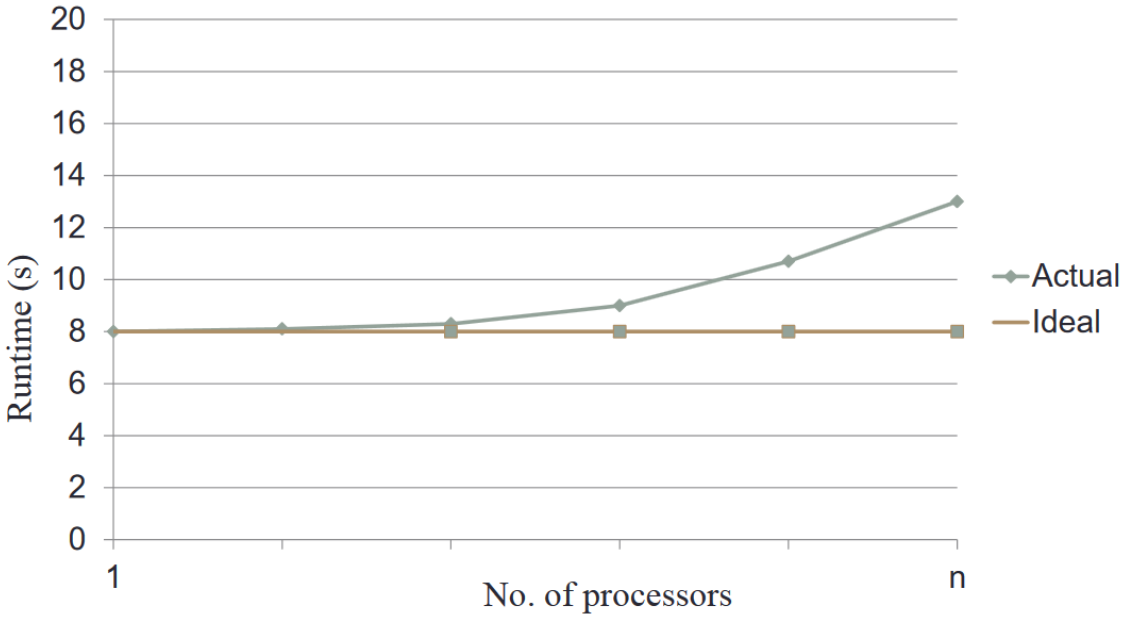

Figure 3.4: Runtime of a weak scaling problem. The ideal case of this is problem type is for the runtime to remain constant as the number of processors increases. [1]

Gustafson's law states that we need more significant problems to justify larger numbers of processors. This parallelism is only effective if we can increase the total problem size, while keeping the serial portion α relatively constant, meaning that the parallel section of the code will grow massively. So, if

we can increase N, without increasing the size of the serial portion, then $\alpha \to 0$ and the normalised runtime $NR(N, P)$ will tend towards ideal, i.e. constant runtime.

The motivation behind Gustafson's law is that rather than just using parallelism to decrease the runtime of more minor problems, we should use the power of parallelism to strive for problem sizes that would otherwise be impossible to achieve on current technology. Following again from our factory owner example, he uses weak scaling parallelism by adding more workers to the factory to scale up to production sizes that he could not do by himself. If he requires higher production volumes, he can keep employing more workers to keep scaling up. This type of scaling is especially applicable in the computer modelling of an object. For example, if each core models a particular volume of the object in question, we can keep adding more cores to model a larger object.

We note that we will be focusing more on strong scaling programs for analysis for the majority of this report.

Shared memory

On a typical consumer machine, for example a quad core laptop, all of the processors have the same access to a shared memory source. Shared memory means that all processors in the system can access the same memory source via a system bus - a pathway that allows processors to request and receive data from memory. (Fig. 3.5). The advantages of this method are that all processors will have the same access to all of the memory, so developers do not need to worry about which processors can readily access what data. Although for highly memory/data-intensive programs in a shared memory system, there can be slowdowns when multiple processes are trying to access memory simultaneously because only a certain number of processors can use the system bus at one time.

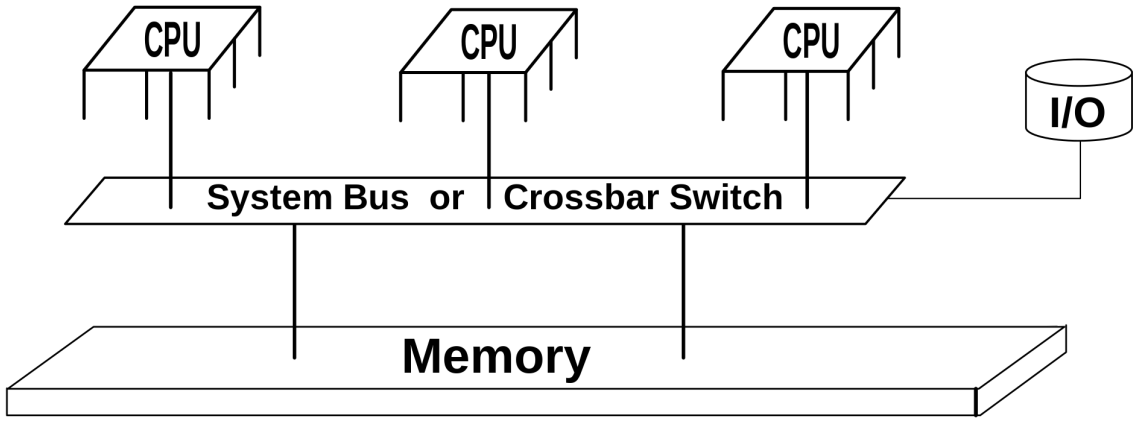

Figure 3.5: A diagram of a Shared memory system [10]

The most common framework for utilising shared memory is "OpenMP", which is a programming interface that allows users to work with shared memory. However, the programming language that is mainly used throughout this report is Python, which has its own shared memory implementation called "Multiprocessing" that we will use.

We now look at an example of a shared memory program, implemented in the Python Multiprocessing library, which will be described more in depth later in the report. This program will generate an array of values [1,2,3,...,8] and place this array into shared memory. Each worker will be given an index of this array, access the array in shared memory and multiply the element at their respective index by a number from $9, 10, \ldots, 16$.

```
from multiprocessing import Pool, shared_memory
```

```python
import numpy as np

#Sgemm version of matrix multiplicaiton
def change_value(i,factor,shm_name):
    existing_shm = shared_memory.SharedMemory(name=shm_name)
    shared_array = np.ndarray(8, dtype=np.float32, buffer=existing_shm.buf)
    shared_array[i] *= factor
    existing_shm.close()


def main():
    #Generates an array of values [1,2,3,..8]
    data = np.arange(1,9)
    #Defines shared memory blocks array to be placed into
    shm = shared_memory.SharedMemory(create=True, size=data.nbytes)
    #Places array into shared memory
    shared_array = np.ndarray(data.shape, dtype=data.dtype, buffer=shm.buf)
    shared_array[:] = data[:]
    print(f"Array␣contents␣before␣operation:␣{shared_array}")
    #Gets the name of the sahred memory blocks so that they can be accessed by workers
    shm_name = shm.name
    #Creates a list of all parameters to be sent to workers. These paramaters include the
        "coordinates" of the matrices that should be work on and the name of the shared
       memory blocks that the matrices are sotred in.
    send_list = [[i,i+9,shm_name] for i in range(8)]
    #Opens a pool of worker processes
    p = Pool(processes=8)
    #Passes the parameters to each of the workers. Some timing results are then returned
    res_list = p.starmap(change_value, send_list)
    print(f"Array␣contents␣after␣operation:␣{shared_array}")
    #Closes the worker processes
    p.close()
    #Close and unlink the shared memory block
    shm.close()
    shm.unlink()


if __name__ == '__main__':
    main()
```

Listing 3.1: Python Multiprocessing Example [5]

A shared memory system can take take two main forms, UMA and NUMA:

UMA

In a Uniform Memory Access (UMA) system, every core has the same, i.e. "uniform" access to one shared memory source. This type of system is often found in personal laptops and desktop computers.

NUMA

At the end of the previous section, we described the von Neumann architecture; we will now look at the von Neumann bottleneck. This bottleneck occurs because, in a modern CPU, the CPU is far faster than the communication times to memory, meaning that the CPU is often stuck waiting or idling while it is waiting for data from memory, this can add up to severe performance losses. This is exaggerated for a parallel CPU with one shared memory source; if one core is occupying the memory access, then all the other processors are waiting, leading to massive performance losses. This resulted in the creation of NUMA (non-uniform memory access), which means that each core will have its own local primary memory source.

A Non-Uniform Memory Access (NUMA) system will take a form more similar to the distributed memory that we will look at soon. In a NUMA system, the processors will be split into groups called

sockets; each socket will have its local memory, which can be accessed at high speed. All of the memory that is allocated to other sockets will still be accessible via a high-speed interconnect. However, this will still be a longer access time than its own local memory, hence "non-uniform" memory access. NUMA is the system format that will typically be used on nodes of High-Performance Computing (HPC) Clusters, for example, Queens' Aigis cluster, which I will be using for all of the testings in this report.

Distributed Memory

Within a Distributed memory system, each of the processors has its local memory source. Thus, there is no way for processors to access each other's memory directly than by communicating by sending and receiving messages. However, in the NUMA shared memory architecture, each core can directly access memory from another socket. The distributed memory system allows for programs to scale much higher than any shared memory platform as the problem size is not bounded by the amount of memory that can fit on a single chip. Although, distributed memory does come with its pitfalls. Compared to shared memory, it is much harder to implement as there has to be more focus on what processors have access to specific data. Moreover, some hardware limitations still come with the speed of connections between individual processors, as everything is not accessing the same local memory. Moreover, using a distributed memory system introduces extra overheads when running a program, as all of the necessary data must be split up and sent from the Master core to the worker cores for the program to begin, in a shared memory system this is not the case, all of the processors will already have access to the same memory.

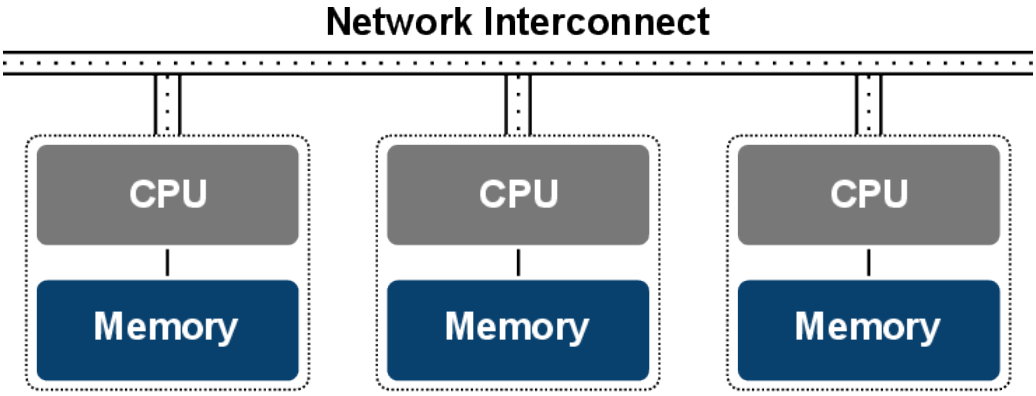

Figure 3.6: A diagram of a Distributed Memory system [7]

In order to harness distributed memory, we use the MPI (Message Passing Interface) paradigm. MPI is the standard by which information is communicated between cores in a distributed system. In the MPI standard, each core is assigned a rank, which is a number in the range $0, 1, \ldots, P-1$ which can be thought of as an ID for each core, helping to track communication.

For now, we look at an example of MPI code in Python. This program will run on two cores, core 1 (rank 1) will generate a random number, "randNum", print this number for the user and then send this number using the "Send" communicator, to core 0 (rank 0) which will "Receive" the number. Core 0 will now print this number to the user, double the number and send the result back to core 1. Core 1 will now print its new received number to show that the data has been communicated between the cores effectively.

```
import numpy
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

randNum = numpy.zeros(1)
```

```
#This part is only executed by core 1
if rank == 1:
        #Generate a random number
        randNum = numpy.random.random_sample(1)
        print(f"Process {rank} drew the number: {randNum[0]}")
        #Send the number to core 0
        comm.Send(randNum, dest=0)
        #Receive the new number from core 0
        comm.Recv(randNum, source=0)
        print(f"Process {rank} received the number: {randNum[0]}")

##This part is only executed by core 0
if rank == 0:
        print(f"Process {rank} before receiving has the number: {randNum[0]}")
        #Receive the number from core 1
        comm.Recv(randNum, source=1)
        #Multiply the number by 2
        print(f"Process {rank} received the number: {randNum[0]}")
        randNum *= 2
        #Send the new number back to core 1
        comm.Send(randNum, dest=1)
```

Listing 3.2: Python MPI Example [5]

Different bounds on parallel scalability

We now look at a few different real-world limits that stop up from always reaching peak theoretical parallel performance.

We could take advantage of Gustafson's law in an ideal world and continue scaling up program size indefinitely, leading to better overall performance. However, certain limits prevent us from adding more cores or adding more memory to a system.

Firstly, the most apparent problem is that high-performance computing hardware is not cheap, so meticulous considerations must be made to maximise performance when designing these systems. Thus, the speed of the processor must be weighed against other factors, such as the amount of memory and the speed of the network. Different systems can exploit different problems; for example, the matrix multiplication program that we will look at later could benefit more from having more memory so that all of the matrices can be stored in memory at once in the distributed memory implementation. Moreover, in the shared memory format the program could benefit from improved access to memory via a higher bandwidth system bus, as we will see there are slowdowns when many processors are trying to access shared memory simultaneously. In problems like these, the data operated on is the system's lifeblood, so there must be as much attention paid to data management as processing power. However, in the Monte Carlo Analysis problem that we look at later, each of the processors is only working on small amounts of data that it randomly generates as it goes along. Therefore, memory would not be crucial in a system designed for this purpose.

Secondly, even if the price was not a massive issue, we still cannot just throw more processors and memory together and continue making a faster computer. After a certain number of processors, commercially purchased computers cannot keep being stacked into one space. Instead, they must be split into racks of smaller computers that are all then linked together via a network interconnect, like is (Fig. 3.6). So once a computer becomes large enough to be split into many different racks, it can suddenly not be run on a shared memory format anymore. Thus, we must now implement a distributed memory approach to keep all of the data in order. This scenario is why MPI (Message Passing Interface) is so important; it allows developers to manage data and work in systems that can scale up to many racks of computers, all working in harmony.

Chapter 4

The Monte-Carlo Pi Approximation problem

We will also use the Monte-Carlo algorithm to approximate the value of π. We choose this problem as it is an excellent example of an "embarrassingly parallel" algorithm, meaning that it requires no communication between worker processes, allowing it to obtain speedups that are close to ideal. This algorithm will generate N pairs of pseudo-random numbers $0 \leq x, y \leq 1$, representing coordinates in the 2-D plane.

Then the program will determine whether $x^2 + y^2 \leq 1$, which will show if (x, y) lies within the upper right quadrant of the unit circle; if so, it will be counted as a "hit", and we will add 1 to the hit counter h. The value h/N will allow for an approximation of the area of the unit circle, $A \approx h/N$, which can be used to approximate the value of pi using:

$$A = \pi r^2 \tag{4.1}$$

we know that for one quadrant of the unit circle, $r = 1$ and using the approximation of the unit circle $A \approx h/N$, we have:

$$\pi \approx 4\frac{h}{N}. \tag{4.2}$$

The algorithm can be very easily parallelised as each pair (x, y) is randomly and independently generated; there is no need for communication between processes. The only communication needed is for the master process to send out one integer value, N, telling the worker processes how many pairs they must calculate. Followed by the worker processes sending back an integer telling the master process how many hits, h, they had. This is a minimal communication overhead, which should allow for some excellent parallelisation.

We will now see the results of a strong scaling and weak scaling version of this problem that I have implemented in Python using the Multiprocessing library. The Multiprocessing will be explained in more detail later. However, for now, all we must know is that it is a shared memory parallelism implementation. The memory format the Monte Carlo program is not that important because it is a more computationally intensive problem. Any of the data that is operated by the individual worker processes is randomly generated by the workers as they go along, so their access to memory is insignificant. The program for this can be seen in Appendix A.1, or "BitesizeBytes/TesterFiles/Monte-Carlo/MonteCarlo_Tester.py" in my GitHub repository.

Strong Scaling Results

In (Fig. 4.2) we can see that as N increases, the speedup tends closer and closer to the ideal speedup, this occurs because of Amdahl's law - for small N, the serial portion α will dominate. However, as we increase N, α will decrease relative to the parallel problem size. So, as N grows, the serial fraction of the problem, α, falls allowing for a maximum speedup of $S(10^{10}, 32) = 23.69$ at $P = 32$ cores and $N = 10^{10}$ attempts.

We were not able to perform the test for any more significant numbers of pairs than $N = 10^{10}$, as it must be appreciated that 10^{10} pairs of Python "floats" (floating point numbers) account for 149GB of data to be operated upon which takes a considerable amount of time.

We can now use this data to calculate the FLOPS for every instance of the Monte Carlo strong scaling program. In this case, we will assume that the floating point operations under consideration are: (1)

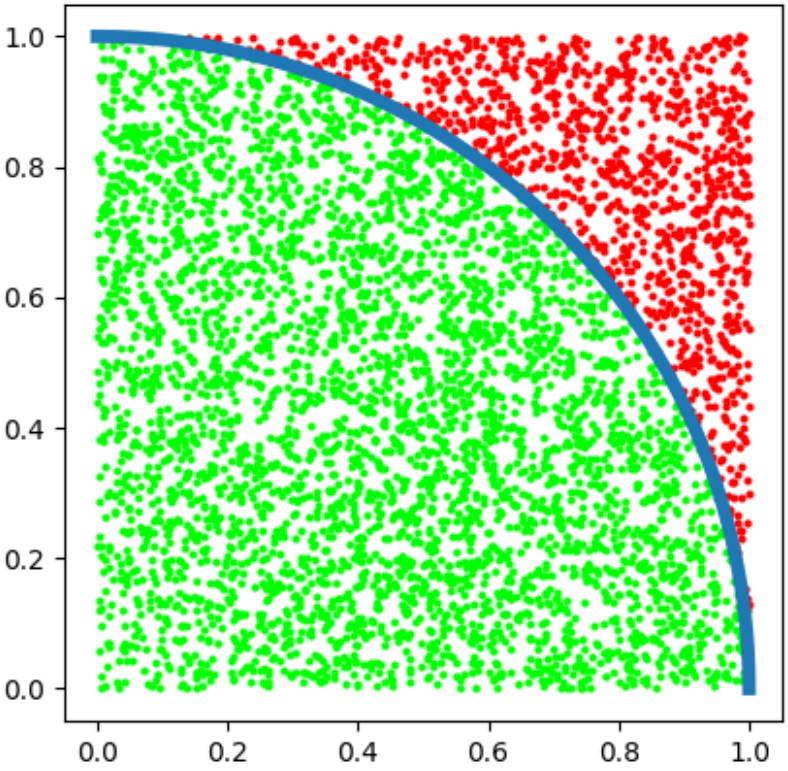

Figure 4.1: Visualisation of the Monte-Carlo approximation of π. Green dots represent 'hits', i.e. points that lie within the upper right quadrant of the unit circle (shown as a blue quarter circle), and red dots represent 'misses', i.e. points that do not.

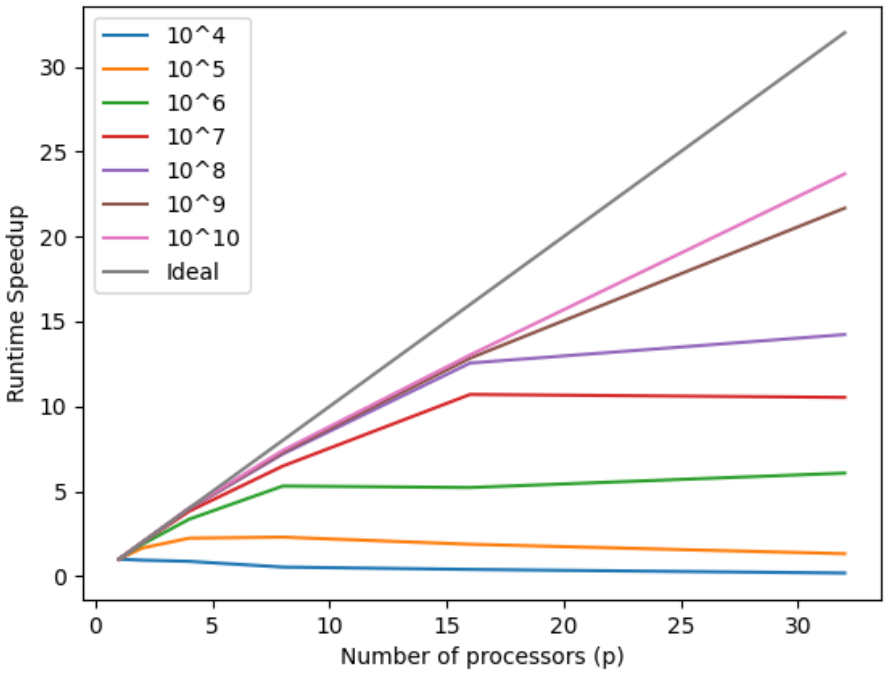

Figure 4.2: The parallel speedup of the Monte Carlo approximation running on $p = 1, 2, 4, \ldots, 32$ cores. Each line represents the program calculating $N = 10^m$, for $m = 4, 5, \ldots, 10$, pairs of co-ordinates in the x,y plane. This is a strong scaling problem, so each core will be calculating $N = 10^m/p$ pairs of co-ordinates.

MFLOPS						
Processors (P)/ Attempts (N)	1	2	4	8	16	32
10^4	7.36	6.93	6.41	3.94	2.91	1.35
10^5	9.09	15.08	20.33	20.88	17.06	12.01
10^6	10.37	19.50	34.80	55.02	54.11	62.92
10^7	10.54	20.95	40.33	68.37	112.82	110.97
10^8	10.53	21.06	41.78	75.73	132.13	149.79
10^9	10.53	21.08	41.98	76.75	135.22	228.28
10^{10}	10.36	20.79	41.63	76.73	135.00	245.50

Table 4.1: The MFLOPS of the MonteCarlo as the problem size $N = 10^4, 10^5, \ldots, 10^{10}$ and over $P = 1, 2, 4, \ldots, 32$ cores.

Generating x, y - 2 operations; (2) Calculating x^2, y^2 - 2 operations; (3) adding $x^2 + y^2$ together - 1 operation; (4) comparing $x^2 + y^2 \leq 1$ - 1 operation; and finally (5) Increment the hit counter h - 1 operation. This gives a total of 7 operations per cycle of the for loop, which will be performed $N = 2^m$ times, so W(N) = 7×10^m. So, we have MFLOPS(N) = $7 \times 10^m/(t \times 10^6)$ where t is the time taken. As we can see in (Table. 4.1), the peak occurs at $N = 10^{10}$, $P = 32$ for a maximum of 245.5MFLOPS.

Weak Scaling Results

The idea behind Gustafson's law and weak scaling programs is to push for higher problem sizes N, meaning that N will scale linearly with P. So, in this program, each core will be performing $N = 10^m$ attempts.

We can see the results of the weak scaling results in (Fig. 4.3). This figure shows that for the largest problem size $N = 10^8$, a normalised runtime of $NR(10^8, 32) = 1.63$ is achieved. This result means that the program has executed $32\times$ times as much work in $1.63\times$ the runtime. This is a very good result and displays how we can perform programs of larger overall problem sizes through weak scaling parallelism.

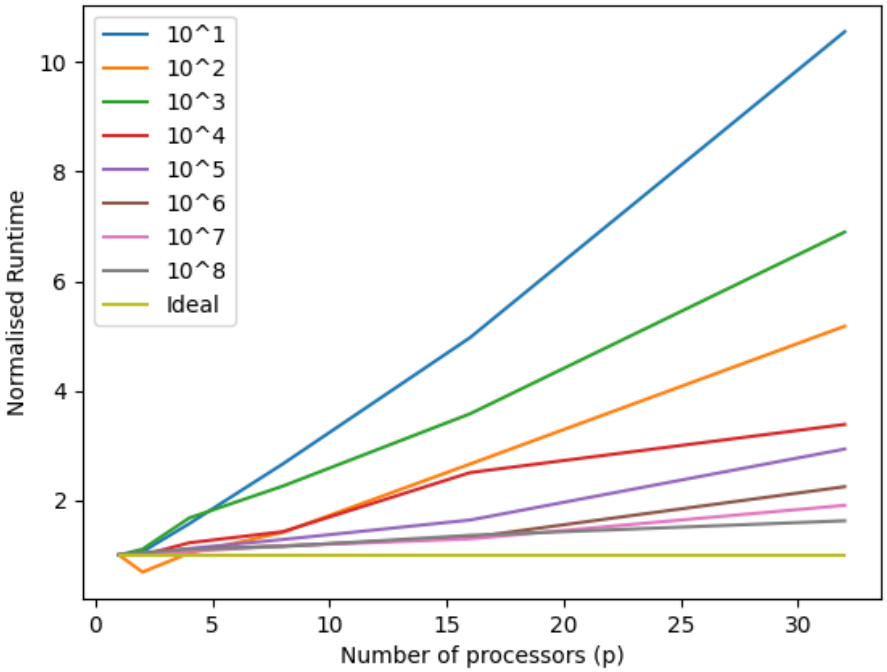

Figure 4.3: The parallel runtime of the Monte Carlo approximation running on $p = 1, 2, 4, \ldots, 32$ cores. Each line represents the program calculating 10^m, for $m = 1, 2, \ldots, 8$, pairs of co-ordinates in the x,y plane. This is a weak scaling problem, so each core will be calculating 10^m pairs of co-ordinates.

Chapter 5
The Matrix Multiplication problem

For the remainder of the report, the main problem we will focus on is matrix multiplication performed 2 square, dense matrices, A, B of equal order $N \times N$ where $N = 2^m$ for some $m \in \mathbb{N}$. Matrix multiplication is an interesting problem for our case because it presents itself as a problem that is quite complex to decompose into a parallel solution. However, it is also a problem that is useful in many real-world applications such as computer graphics. We also note that for the remainder of this report, the number of processors P that we will work on will be a power of 2, $P = 1, 2, 4, 8, 16, 32$, similar to the matrix orders N. We make this simplification in order to streamline the problem decomposition. Moreover, as A and B are always square matrices, we will refer to them as being order N rather than $N \times N$

We strive to calculate matrix $A \times B = C$. So generally, matrix multiplication will follow the form:

$$\begin{pmatrix} A_{0,0} & A_{0,1} & \dots & A_{0,N} \\ A_{1,0} & A_{1,1} & \dots & A_{1,N} \\ \vdots & \vdots & \ddots & \vdots \\ A_{N,0} & A_{N,1} & \dots & A_{N,N} \end{pmatrix} \times \begin{pmatrix} B_{0,0} & B_{0,1} & \dots & B_{0,N} \\ A_{1,0} & B_{1,1} & \dots & B_{1,N} \\ \vdots & \vdots & \ddots & \vdots \\ B_{N,0} & B_{N,1} & \dots & B_{N,N} \end{pmatrix} = \begin{pmatrix} C_{0,0} & C_{0,1} & \dots & C_{0,N} \\ C_{1,0} & C_{1,1} & \dots & C_{1,N} \\ \vdots & \vdots & \ddots & \vdots \\ C_{N,0} & C_{N,1} & \dots & C_{N,N} \end{pmatrix}. \quad (5.1)$$

Where each element element $C_{i,j}$ is calculated by the dot product of the ith row of A and the jth column of B, where $0 \leq i, j \leq n - 1$.

We first analyse the serial method matrix multiplication commonly used, as seen in (Fig. 5.1). The problem size complexity, i.e. the work done, of this algorithm can be evaluated as $W(N) = N^2(2N - 1)\tau$ where τ represents the time to perform an elementary operation such as multiplying or adding two numbers.[4] So, we say that the matrix multiplication has a computational time complexity of $O(N^3)$, meaning that the number of operations and the runtimes scale with N^3 where N is the matrix order. This is reflected in (Fig. 6.1). To calculate FLOPS later, we also denote the work done of matrix multiplication for 2 square matrices of order N to be $W(N) = N^2(2N - 1)$.

In order to parallelise this serial algorithm, the most obvious approach to take would be to assign $\frac{N}{P}$ (which is an integer as N, P are powers of 2) rows to each processor, which will then calculate $\frac{N}{P}$ rows of the resultant matrix C. However, when a distributed memory approach is adopted later in the report, this means that each processor's local memory will contain $\frac{N}{P}$ rows of A and N columns of matrix B, i.e. the entirety of B. This appears to be a relatively inefficient method of distributing data, so going forward, a "checkerboard" approach is adopted to calculating C. This refers to the fact that

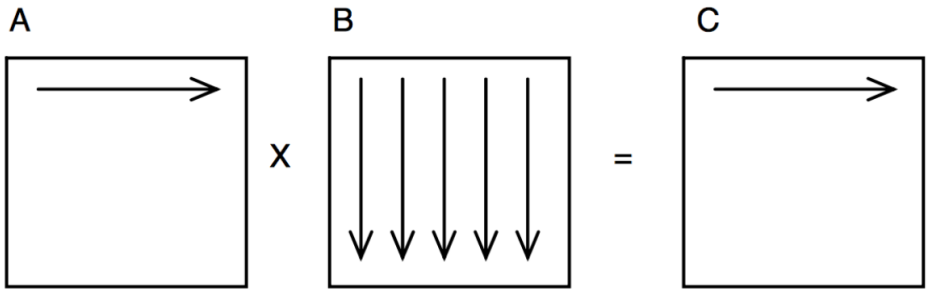

Figure 5.1: The serial algorithm for matrix multiplication $A \times B = C$, where each row in C is calculated by a row in A multiplied by each column in B. [3]

matrix C is split into a checkerboard structure, with each processor calculating one submatrix of C, referred to as $C'_{i,j}$.

Again, we assume we are operating on P processors, where $P = 2^k$ for some $k \in \mathbb{N}$, and we have matrix order $N \geq P$. Making the simplifications of N, P both being a power of 2 allows us to streamline the problem decomposition and focus more on results. We now devise an algorithm that will allow us to divide the work evenly among P processors.

We know that A,B are both square matrices of order $N \times N$, giving resultant square matrix C of order $N \times N$. So we strive to divide C into P evenly sized blocks. We note that for $P = 2^k$, if k is odd, then partitioning C is a bit harder as we cannot split into $\sqrt{P}$ blocks, as $\sqrt{P}$ is not an integer for odd k. So, we define two variables, ϕ_i, ϕ_j which denote the number of partitions that must be made in the i and j axes, respectively, as follows:

$$\phi_i = 2^{\lceil k/2 \rceil} \tag{5.2}$$

$$\phi_j = 2^{\lfloor k/2 \rfloor}. \tag{5.3}$$

The length of each partition ϕ_i and ϕ_j will then be ψ_i and ψ_j, respectively, are calculated as follows:

$$\psi_i = \frac{m}{\phi_i} = \frac{2^n}{2^{\lceil k/2 \rceil}} = 2^{n - \lceil k/2 \rceil} \tag{5.4}$$

$$\psi_j = \frac{m}{\phi_j} = \frac{2^n}{2^{\lfloor k/2 \rfloor}} = 2^{n - \lfloor k/2 \rfloor} \tag{5.5}$$

We note that ψ_i, ψ_j are integers.

We now look at and example of decomposition of matrix $C = A \times B$ of order $N = 4$ and $P = 4$ processors:

$$\begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix} = \begin{pmatrix} \begin{pmatrix} C_{0,0} & C_{0,1} \\ C_{1,0} & C_{1,1} \end{pmatrix} & \begin{pmatrix} C_{0,2} & C_{0,3} \\ C_{1,2} & C_{2,3} \end{pmatrix} \\ \begin{pmatrix} C_{2,0} & C_{2,1} \\ C_{3,0} & C_{3,1} \end{pmatrix} & \begin{pmatrix} C_{2,2} & C_{2,3} \\ C_{3,2} & C_{3,3} \end{pmatrix} \end{pmatrix} = \begin{pmatrix} C'_{0,0} & C'_{0,1} \\ C'_{1,0} & C'_{1,1} \end{pmatrix}. \tag{5.6}$$

Where $C'_{i,j}$ is a sub-matrix of order $\psi_i \times \psi_j$. Within this example, $C'_{0,0}$ will be calculated as follows:

$$\begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \end{pmatrix} \times \begin{pmatrix} B_{0,0} & B_{0,1} \\ B_{1,0} & B_{1,1} \\ B_{2,0} & B_{2,1} \\ B_{3,0} & B_{3,1} \end{pmatrix} = \begin{pmatrix} C_{0,0} & C_{0,1} \\ C_{1,0} & C_{1,1} \end{pmatrix} = C'_{0,0}. \tag{5.7}$$

It is useful to use this notation for the resultant submatrices $C'_{n,n}$ as they will represent the each work to be done by one processor. We now note that there will be $\phi_i \times \phi_j$ submatrices $C'_{n,n}$, each of length $\psi_i \times \psi_j$.

This example can also be seen in (Fig. 5.2).

So, in general, for matrices A and B of order N, this will take the form:

$$\begin{gathered} A \times B = \begin{pmatrix} A_{0,0} & A_{0,1} & \dots & A_{0,N} \\ A_{1,0} & A_{1,1} & \dots & A_{1,N} \\ \vdots & \vdots & \ddots & \vdots \\ A_{N,0} & A_{N,1} & \dots & A_{N,N} \end{pmatrix} \times \begin{pmatrix} B_{0,0} & B_{0,1} & \dots & B_{0,N} \\ B_{1,0} & B_{1,1} & \dots & B_{1,N} \\ \vdots & \vdots & \ddots & \vdots \\ B_{N,0} & B_{N,1} & \dots & B_{N,N} \end{pmatrix} = \\ C = \begin{pmatrix} C_{0,0} & C_{0,1} & \dots & C_{0,N} \\ C_{1,0} & C_{1,1} & \dots & C_{1,N} \\ \vdots & \vdots & \ddots & \vdots \\ C_{N,0} & C_{N,1} & \dots & C_{N,N} \end{pmatrix} = \begin{pmatrix} C'_{0,0} & \dots & C'_{0,\phi_j} \\ \vdots & \ddots & \vdots \\ C'_{\phi_i,0} & \dots & C'_{\phi_i,\phi_j} \end{pmatrix} \end{gathered} \tag{5.8}$$

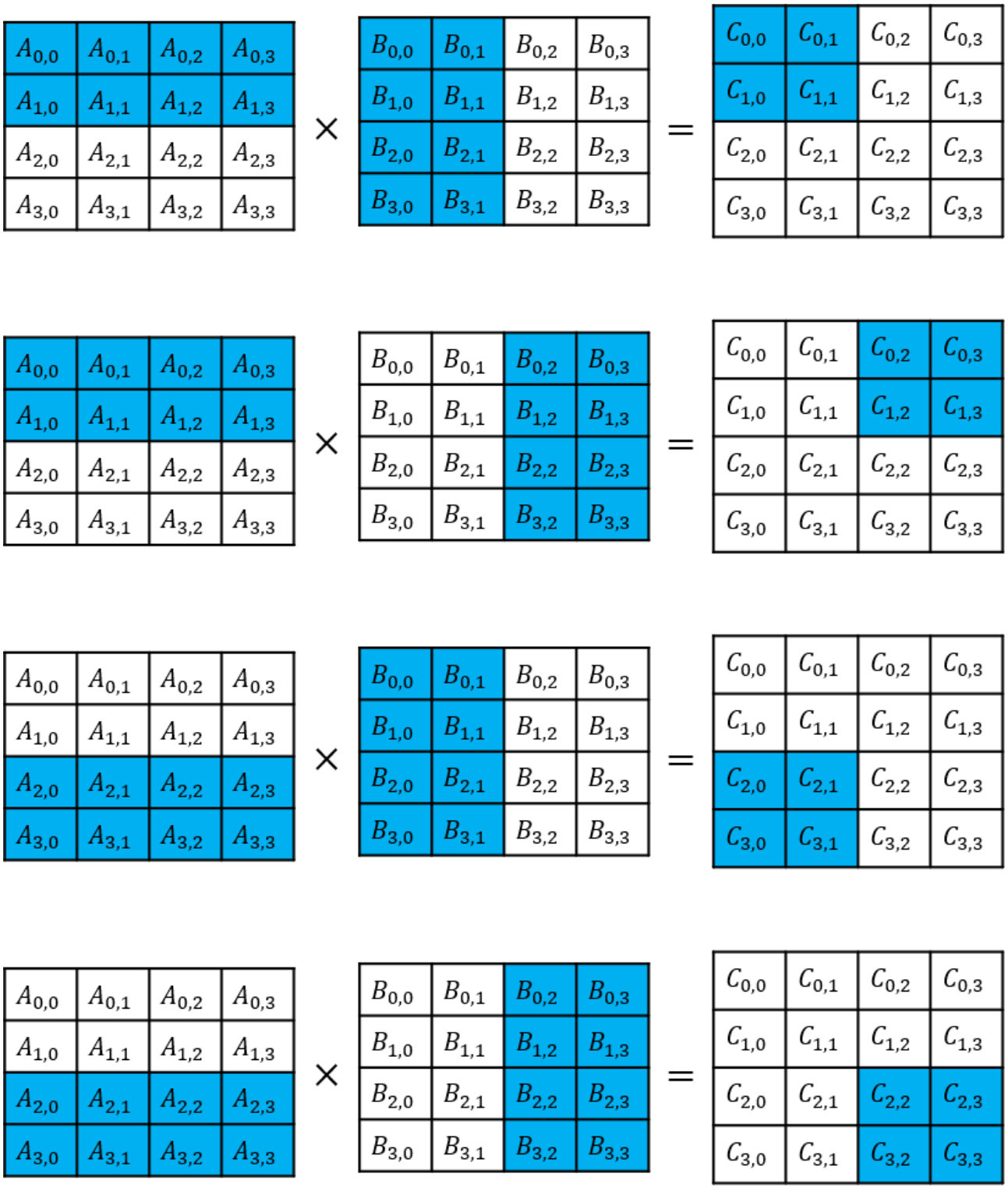

Figure 5.2: Example of the matrix multiplication parallel decomposition for matrix order $N = 4$ and number of processors $P = 4$. Each processor will calculate one submatrix of C, highlighted in blue.

where each $C'_{i,j}$ is a submatrix of dimension $\psi_i \times \psi_j$ calculated as follows:

$$\begin{pmatrix} A_{0,0} & A_{0,1} & \dots & A_{0,N} \\ \vdots & \vdots & \ddots & \vdots \\ A_{\psi_i,0} & A_{\psi_i,1} & \dots & A_{\psi_i,N} \end{pmatrix} \times \begin{pmatrix} B_{0,0} & \dots & B_{0,\psi_j} \\ B_{1,0} & \dots & B_{1,\psi_j} \\ \vdots & \ddots & \vdots \\ B_{N,0} & \dots & B_{N,\psi_j} \end{pmatrix} = C'_{i,j}. \tag{5.9}$$

Each element of $C'_{i,j}$ will then take the following co-ordinates within C:

$$C'_{i,j} = \begin{pmatrix} C_{i\psi_i,j\psi_j} & C_{i\psi_i,j\psi_j+1} & \dots & C_{i\psi_i,(j+1)\psi_j-1} \\ C_{i\psi_i+1,j\psi_j} & C_{i\psi_i+1,j\psi_j+1} & \dots & C_{i\psi_i+1,(j+1)\psi_j-1} \\ \vdots & \vdots & \ddots & \vdots \\ C_{(i+1)\psi_i-1,j\psi_j} & C_{(i+1)\psi_i-1,j\psi_j+1} & \dots & C_{(i+1)\psi_i-1,(j+1)\psi_j-1} \end{pmatrix} \tag{5.10}$$

.

So, the matrix C is effectively broken into a grid of block submatrices each of which is calculated by a single core. An example of this decomposition for matrix order $N = 4$ and number of processors $P = 4$ can be seen in (Fig. 5.2).

Chapter 6
Programming Languages

Before we begin looking at the results from testing the matrix multiplication program, we will first examine the nature of the programming language I have chosen, Python, and how I have utilised it to implement it. In order to write executable programs, we must use a programming language. When writing a program, the typical programmer will use a "High-Level" programming language, typical examples of which are C, Python or Fortran. A high-level language means that it is designed to be closer to human's natural language and the way we think, making it much easier to write and understand code.

However, computers do not think the same way as us, so for a program to be executed, it must be translated from the high-level "Source Code" written by a human into a "Low-level" language understandable by a CPU. This is known as "Machine Code" and consists of fundamental binary instructions. There are two main methods of translating high-level code to low-level code - using a Compiler or an Interpreter.

When source code is run through a compiler, it will analyse the source code structure to check for syntax or logical errors; if none are found, then the source code is translated to executable machine code, which is then executed.

When source code is run through an interpreter, each line is run through an interpreter one line at a time, meaning that the interpreter is constantly running. Moreover, the program is never directly translated to machine code. [2]

We will mainly be using Python, an interpreted language. This comes with substantial benefits, such as "dynamic typing", which means that the type of a variable is only determined when it is run through the interpreter at runtime, in contrast with a compiled language which uses "static typing" in which all variable types must be explicitly declared at compilation. Dynamic typing makes it far easier to write programs quickly, making languages like Python more user-friendly to less experienced programmers, hence their popularity. Although, giving up so much control to the interpreter can make these languages less appealing to more experienced programmers who are willing to immerse themselves into gritty, lower-level details to reap runtime benefits.

We will also use some results from Fortran, a compiled language that is very common in scientific research.

Because of its popularity and open-source nature, Python has a wide array of useful libraries, which are implemented throughout this report. We will now give a brief description of the libraries that are most important to use and why they are important:

- NumPy - Storing large arrays and matrices of data, also for performing numerical operations on these data structures. We mainly use NumPy for storing matrices and performing mathematical operations such as logarithms when splitting up data.
- SciPy - Provides general scientific computing functions. We only use this function to get "sgemm" matrix multiplication function from its Linear Algebra package.
- Multiprocessing - Process-based parallelism implementation in Python. We use it as our shared memory implementation.
- mpi4py - Implementation of the MPI (Message Passing Interface) standard. We use mpi4py to run a distributed memory parallel program.

Time Comparison						
Process type/ Matrix Order	My function (Python)	My function (32 Cores Python)	matmul (NumPy Python)	dgemm (Python)	sgemm (Python)	sgemm (Fortran)
256	15.1771	1.5430	NaN	NaN	NaN	NaN
512	119.3400	7.9496	NaN	NaN	NaN	NaN
1024	942.7839	45.0310	0.0269	0.0576	0.0291	0.0242
2048	7541.8886	349.7939	0.1804	0.4299	0.2201	0.1766
4096	NaN	NaN	1.5075	3.4347	1.8574	1.4094
8192	NaN	NaN	11.9587	25.9035	13.5588	11.1977
16384	NaN	NaN	95.2846	199.2615	101.5215	89.4562
32768	NaN	NaN	864.0640	1562.4091	923.9083	738.7508

Table 6.1: Runtime of matrix multiplication via various different Python functions and one Fortran function for $N = 256, 512, \ldots, 16384, 32768$.

- Pandas - Creating Data structures for data storage and analysis. We use Pandas to store and analyse timing data in our programs.
- Matplotlib - Plotting Library in Python. We use Matplotlib to generate all of the original figures in this report, such as speedup graphs.

Though Python is often discredited as a "slow" language, some of these libraries such as NumPy and SciPy have been written and optimised with HPC and scientific research in mind, meaning that some of their functions can be nearly on par with what is considered "fast" compiled languages such as Fortran or C. Many of the functions, for example, sgemm are implemented in C and directly called from Python code.

Since the main problem of this report is parallelising the matrix multiplication problem, we will select an array of matrix multiplication functions from different Python libraries and contrast their runtime results with results from their counterparts in Fortran and a "naive" function that I have written myself in Python. We refer to this handwritten function as "naive" because it has no consideration of optimising data or processor usage by exploiting the properties of the function it is working on. This "naive" function just runs over the rows and columns of A, B using basic for loops.

The first library functions that we will use are the "sgemm" and "dgemm" functions from Lapack, which are used for generic matrix multiplication. The names of which can be decomposed as follows "s" and "d" refer to single precision and double precision data types, which are 4 and 8 bytes of data each. The following "ge" means that the two matrices can take any general form, as we are working with square matrices. However, there are functions that are optimised for matrices of specific structures, for example, band matrices where all non-zero elements are confined to a "band" along the diagonal. Finally, the "mm" means matrix multiplication. These two functions have representations in both Python and Fortran, meaning that we can compare them directly.

Next, we will use the "matmul" function from NumPy, another matrix multiplication function.

Finally, we will compare my own "naive" handwritten function for matrix multiplication, called "matrix_mult". Moreover, we will also run this function on 32 cores using the Python Multiprocessing library to see how the results compare to the highly optimised functions.

Note that I could only test my "matrix_mult" function up to matrix order $N = 2048$ as it was already taking approximately 2 hours to run one matrix multiplication on that matrix order. However, the results of (Table. 6.2) show that at this maximum matrix order, the Python "sgemm" function was able to execute approximately in 0.05 seconds, which is on the order of 10^5 times faster than my "matrix_mult" function, showing the extent of the optimisations are made in the built-in Python functions.

(Table. 6.1) shows the runtimes of various Python functions, and one Fortran function of matrix

Time Comparison				
Process type/ Matrix Order	matmul (NumPy Python)	dgemm (Python)	sgemm (Python)	sgemm (Fortran)
1024	0.0123	0.0184	0.0095	0.0242
2048	0.0186	0.0794	0.0529	0.1766
4096	0.1035	0.6620	0.4583	1.4094
8192	0.8293	3.9984	2.6495	11.1977
16384	5.1190	24.7832	15.1972	89.4562
32768	76.6318	176.4925	159.3846	738.7508

Table 6.2: Runtime of matrix multiplication via various different Python functions and one Fortran function for $N = 256, 512, \ldots, 16384, 32768$. These functions are running with Multithreading enabled.

multiplication. We can see that as we increase N by a factor of 2, most of the runtimes increase by a factor of 8. This is what we expect for a matrix multiplication function, as it is an N^3 function, doubling the problem size should have a result of $2^3 = 8$ on the runtime. This can be see more clearly in (Fig. 6.1) where we plot the runtimes of (Table. 6.2 and 6.1) as a function of N to see if they match up to the $O(N^3)$ nature of the matrix multiplication problem. We see that applying this thread lock has exactly the desired effect, in the top figure with no lock that the 3 Python functions do not follow the "ideal" N^3 line, but when the thread lock is applied in the bottom figure, we see that all functions follow the N^3 line quite closely.

Although, from (Table. 6.2), we can see that the Python library functions (matmul, sgemm, dgemm) are not scaling with what we would expect the computational time complexity of the matrix multiplication problem. As this problem is $O(N^3)$, each time we double the matrix size N, the time should go up by a factor of $2^3 = 8$. This occurred because these Python functions are taking advantage of Multithreading, allowing the programs to run faster at larger problem sizes N. In a typical case, this would be good as the program should run faster with minimal effort from the programmer. However, we want to display matrix multiplication as a strictly $O(N^3)$ problem as a reference point for future analysis. So we apply some Environment variables, which lock the number of threads that each function can have to 1, making the problem scales as we expect. The results of this can be seen in (Table. 6.1) We choose not to impose this lock upon the Fortran program as it is already running at the expected timings.

We can see the normalised runtimes of (Table. 6.2) and (Table. 6.1) in (Fig. 6.1). Note that we did not include my handwritten "matrix_mult" function in this as it did not have enough time data to be plotted effectively. We can see that all of the functions follow the desired $O(N^3)$ runtime relatively well in the top figure of (Fig. 6.1). So, we will choose the Python sgemm function from Lapack to be the main focus of our analysis because its solid performance and scaling, almost identical to the NumPy matmul results. Moreover, the Python sgemm function has an analogous counterpart in Fortran which will be helpful to cross-examine results in some circumstances. Finally, we choose the sgemm function because it is "single precision", which means that the data that it takes as input and output will take up far less memory. As a single-precision float is 4 bytes and a double-precision float is 8 bytes, each matrix will take up half as much data, which means 6 times less memory needed for the total 3 matrices. This will help alleviate some memory obstacles and push for larger N in order to focus more on computational performance differences.

Note that we will also use some results from my naive function in future analysis, but not any other built-in Python functions.

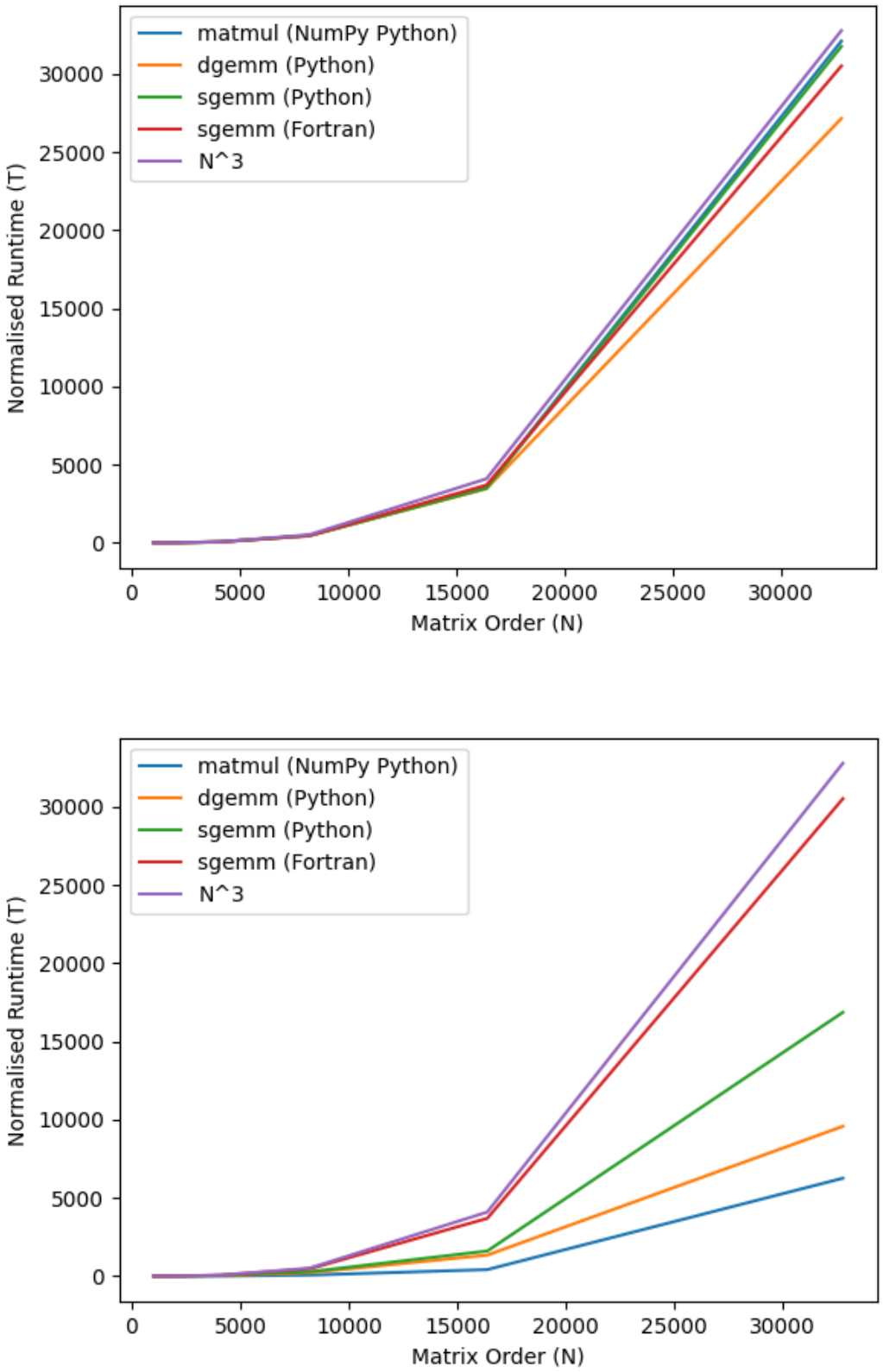

Figure 6.1: Normalised runtime of matrix multiplication functions matmul, dgemm and sgemm from Python, sgemm from Fortran, and N^3 for $N = 1024, 2048, \ldots, 32768$. The top figure shows the functions running normally, with no parallelism. The bottom figure shows the functions with multithreading enabled.

Chapter 7
Multiprocessing

This section uses Python's Multiprocessing library, which allows the user to perform shared memory parallelism within the Python Programming language. The Multiprocessing library can be seen as the Python implementation of the OpenMP paradigm. However, they are not entirely analogous; Multiprocessing is a higher-level interface that works with Python data structures, whereas OpenMP has a lower-level interface with individual cores. Despite their differences, they do focus on a shared memory format, and they are in a lot of ways close enough for the purposes of this report. Within Multiprocessing, we will focus specifically on the Pool class, which allows the user to control a "Pool" of worker processes.

Program Description

We now look at the general format that the testing programs will take in this Multiprocessing section. The master process will generate a pair of matrices, A and B, made up of randomly generated floating-point numbers in the range $(0, 1)$ and placed them into shared memory such that all the worker processes can access them. The master process will calculate the submatrices of A and B that each worker process will work on and pass the "coordinates" of the submatrices to each worker process via the starmap function from the Pool class. Each sub-process will calculate its own allocated submatrix C', and write this into a block of shared memory that is reserved for the matrix C. So, this shared memory implementation has very little communication between processes; the master core sends out the coordinates to work on and the workers don't need to send anything back. For the purposes of timing, the workers will actually send back the runtimes of the various portions of the program to the master core, but this communication is very small in comparison to the runtime of the matrix multiplication.

Moreover, this program will time the four main parts of the program - 1. The "Scatter" time, i.e. the time for the master process to divide up the work and send the coordinates of the submatrices to the respective worker processes; 2. The "Calculation" time, i.e. the time for each worker process to read their respective matrices from the shared memory and then calculate their submatrix C'; 3. The "Gather" time, i.e. the time for each of the C' submatrices to be sent back to the master core and assembled into the total result C; and 4. The "Total" time, which is just the sum of the previous three timings, i.e. the total time of the parallel matrix multiplication. Each of these timings is saved to separate "Pandas" data frames so that they can be easily stored, manipulated and plotted.

As the Aigis 11 node has 32 cores, we will look at the program running for a range of processors $P = 1, 2, 4, 8, 16, 32$, and various matrix orders $N = 2^m$ for $m \in \mathbb{N}$ in order to see how the speedups change as the time problem size increases. To ensure the effect of randomly generating the matrices is minimised, each instance of the test will be run ten times, and the meantime will be calculated in order to account for fluctuations in runtime. We first look at how the timings of the program using my handwritten "matrix_mult" function, followed by using the Python "sgemm" function from Lapack.

Results from handwritten function

For the first test, I have written a Python program (BitesizeBytes/ TesterFiles/ Multiprocessing/ MyFunc/ MatrixMult_Multiprocessing_MyFunc.py) that will perform matrix multiplication using my handwritten "matrix_mult" function on two square matrices as described above. The matrices A and B will take order $N = 32, 64, \ldots, 1024, 2048$ so that we can see how the speedup changes as we increase the problem size N. As stated in Chapter 6, it is only feasible to run the "matrix_mult" function up to matrix order $N = 2048$ as at this size; the function takes approximately 2 hours to run on one core.

As seen in (Fig. 7.1), the maximum speedup $S = 24.2$ at matrix order $N = 2048$ and $P = 32$ cores,

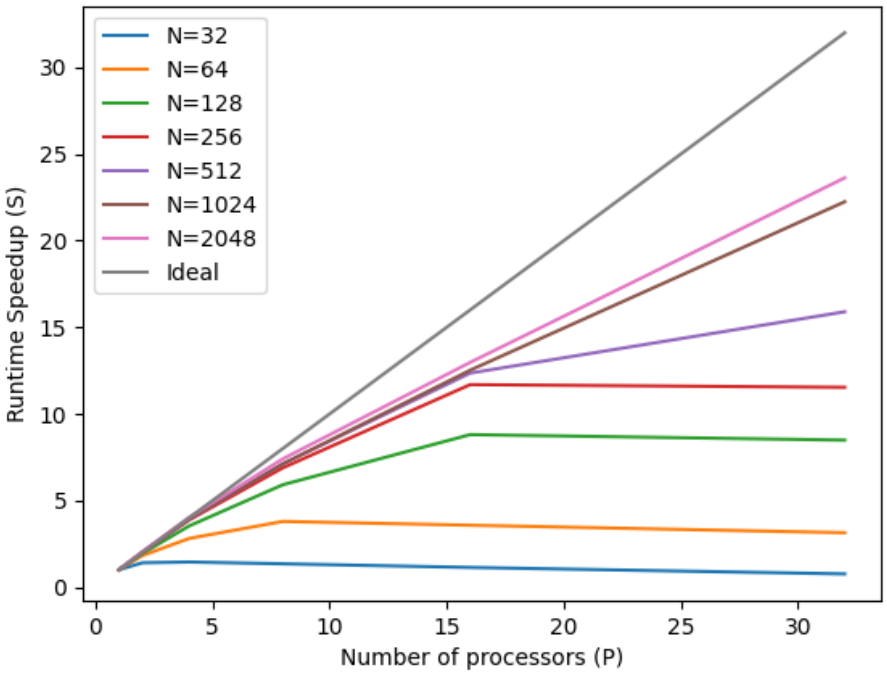

Figure 7.1: The speedup of my handwritten "matrix_mult" function for matrix order $N = 32, 64, \ldots, 1024, 2048$ and number of processors $P = 1, 2, 4, 8, 16, 32$. This function is parallelised using the Python Multiprocessing library. The maximum speedup $S = 24.2$ at matrix order $N = 2048$ and $P = 32$ cores.

which is quite an impressive speedup, but for such a time intensive matrix multiplication function, I would expect slightly closer to an ideal speedup, so we will investigate the runtimes further.

As we can see in (Fig. 7.2) as N increases the calculation portion i.e. the parallel portion begins to dominate, allowing for larger speedup. The "Scatter" times start to disappear as N grows because the action of sending the co-ordinates of the submatrices to the respective worker processes is constant with respect to N as the matrices are placed into shared memory and don't have to be directly sent individually from the master process to the worker processes, so the "Scatter" time only depends on P. The dependence on P comes because more worker processes mean sending more sets of co-ordinates, this can be seen especially at $N = 32$: as P increases, the scatter time dominates the runtime, but as N starts to increase, the action of sending out between 1 and 32 pairs of co-ordinates pales in comparison to the work that is being done by the calculation section. Although the serial section of the program still doesn't fully disappear as N increases, as the "Gather" time is actually dependent on N and P, because larger matrices A and B and more worker processes means that larger resultant submatrices C' have to be sent back from each of the worker processes to the master process to be constructed into C.

So, an ideal speedup is not able to be achieved by the "my func" parallel program because, as seen in (Fig. 7.2) the gathering time still plays a role at $N = 2048$, while it may not look like a significant size of the serial portion, as we have seen in Chapter 3 & (Fig. 3.3), even a serial portion of $\alpha = 0.01$ can limit a parallel program to a maximum speedup of $S = 24.43$ for $P = 32$ cores.

Results from sgemm

Realistically, when having to perform matrix multiplication or similar tasks in an HPC environment, one would use a well optimised function such as "sgemm" from Lapack, rather than using a handwritten

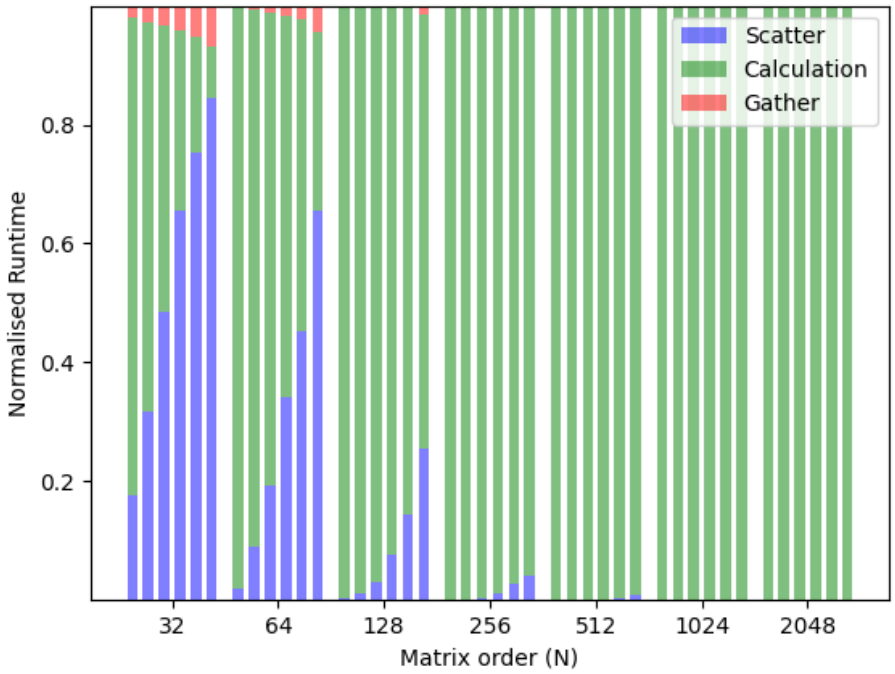

Figure 7.2: The normalised timings of "Scatter", "Calculation" and "Gather" sections of the parallel program (BitesizeBytes/ TesterFiles/ Multiprocessing/ MyFunc/ Matrix-Mult_Multiprocessing_MyFunc.py) which is performing matrix multiplication, using my handwritten "matrix_mult" function for matrix order $N = 32, 64, \ldots, 1024, 2048$ and number of processors $P = 1, 2, 4, 8, 16, 32$. This function is parallelised using the Python Multiprocessing library. Each group of bars shows the function performing matrix multiplication of A and B a matrix order N and every bar in each group shows the normalised runtimes on $P = 1, 2, 4, 8, 16, 32$, from left to right, respectively.

function. So we now run the same test as before, except with the main matrix multiplication function being "sgemm" and matrix order ranging from $N = 128, 256, \ldots, 16384, 32768$.

The speedup results can be seen in (Fig. 7.1), which follows a trend that we expect according to Amdahl's law, as N grows to 32768, the parallel portion of the program grows, which minimises the serial portion α, allowing for larger speedups. Thus, achieving a maximum speedup of $S(32768, 32) = 12.3$ at matrix order $N = 32768$ and $P = 32$ cores.

Although the speedup results using the sgemm function, seen in (Fig. 7.3) aren't as strong as those seen using the handwritten function in (Fig. 7.1), the runtimes will still be significantly better when using sgemm. It is often much easier to parallelise a badly written algorithm.

So again, we look to a breakdown of the parts of the program per iteration, which can be seen in (Fig. 7.4). Similarly to the previous results, the serial portion α effectively disappears past $N = 8192$, with $\alpha = 0.4\%$ for the largest matrix order $N = 32768$. However, using (Eq. 3.6) for $\alpha = 0.004$, there should theoretically be a speedup of S(N,32)=32/(0.004*32 + 0.996)=28.5. But we are only achieving a speedup of $S(32768, 32) = 12.3$, which is only around half of what we are expecting. This is likely because in a shared memory program that is so data-intensive like matrix multiplication when multiple cores are trying to access memory at the same time to write their submatrix C' there are a lot of hidden serial slowdowns occurring because there is limited bandwidth to the shared memory source, only a certain amount of processors can access the data simultaneously. However, their serial portions cannot be effectively shown in (Fig. 7.4) as they are peppered throughout the program, rather than happening in large chunks at either end of the program.

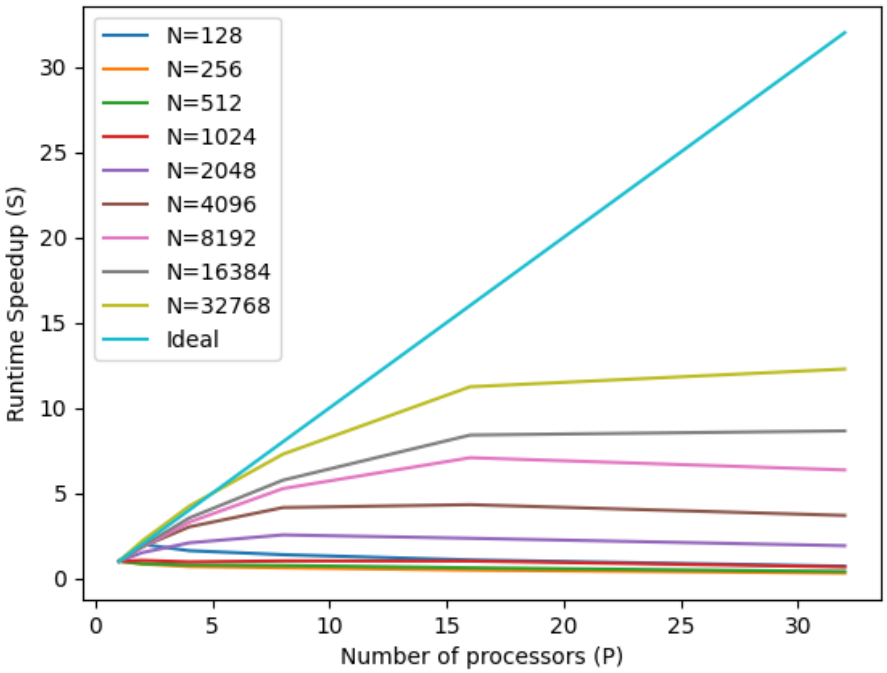

Figure 7.3: The parallel speedup of Python "sgemm" function from Lapack, for matrix order $N = 128, 256, \ldots, 16384, 32768$ and number of processors $P = 1, 2, 4, 8, 16, 32$. This function is parallelised using the Python Multiprocessing library. The maximum speedup $S(32768, 32) = 12.3$ at matrix order $N = 32768$ and $P = 32$ cores.

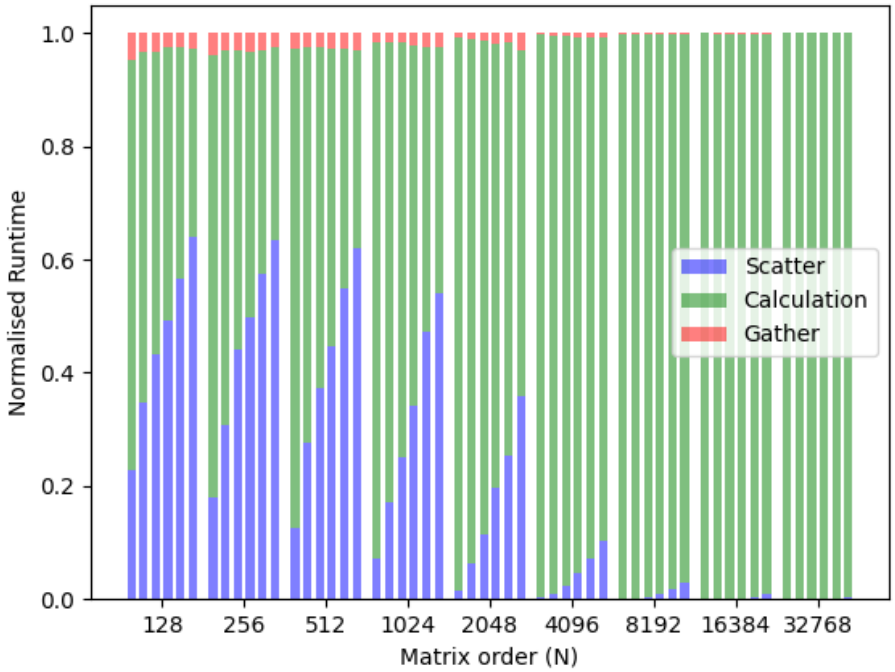

Figure 7.4: The normalised timings of "Scatter", "Calculation" and "Gather" sections of the parallel program (BitesizeBytes/ TesterFiles/ Multiprocessing/ Lapack/ Matrix-Mult_Multiprocessing_Lapack.py) which is performing matrix multiplication using the Python "sgemm" function from Lapack for matrix order $N = 128, 256, \ldots, 16384, 32768$ and number of processors $P = 1, 2, 4, 8, 16, 32$. This function is parallelised using the Python Multiprocessing library. Each group of bars shows the function performing matrix multiplication of A and B a matrix order N and every bar in each group shows the normalised runtimes on $P = 1, 2, 4, 8, 16, 32$, from left to right, respectively.

Chapter 8

MPI

I will now look at the Message Passing method of parallelism, specifically focusing on the Message Passing Interface (MPI). MPI encourages distributed memory parallelism by splitting a program into a user-defined amount of P 'processes', each of which is assigned a rank in the range $0, 1, 2, \ldots, P$. Cores communicate data with each other using many different point-to-point communicators, such "Send" and "Receive" which will communicate packets of data between individual processes or group actions such as "Scatter" and "Gather" between multiple processes. The Python implementation of MPI, which we will use, is a library called "mpi4py". Since Python is built upon the C programming language, and C has an implementation of MPI, this "mpi4py" library is simply a "layer" of Python over the standard C/C++ MPI.

Because MPI by nature is a distributed memory format of parallelism, there will be larger overheads for certain aspects of the program. For example, the "scatter" time in the Multiprocessing implementation, as seen in (Fig. 7.4), is able to essentially disappear as N grows because the data is being read from shared memory. In this format, the master process isn't sending out the submatrices to the worker cores but rather the coordinates of the submatrices, so the overheads should be quite low.

In contrast, in the MPI, scatter time can take a large portion of the program because packaging large data objects into "pickles" and sending them between cores will take considerable time compared to reading from shared memory, but it must be noted that MPI is optimised to make sends of large data as fast as possible through a communicator such as "Gather". Gather is a "collective" communicator, which means that all of the cores will perform it collectively; in our use case, it will be used to gather in all of the submatrices to the master core at the end of the program. Gather can be thought of as every core performing a "Send" to a single "root" core, and this root core performing a "Receive" on all the data.

Program Description

We now look at the general form of the Python program, which I will use to test parallel matrix multiplication via MPI and mpi4py.

First, the number of processors P and the order N of matrices A and B will be read from the command line. The master process will then generate two random matrices A and B of order N; then it will calculate the "coordinates" of the submatrices A and B that each worker process will work on, and send the submatrices to the respective process using a series of "Send" communicators. I had intended for this to be performed using a "Scatter" instead - a scatter the opposite of gather; it is a collective operation that sends data from one root to each respective core. But, when running the program with a Scatter communicator, the program wouldn't run past a $N = 8096$. Nevertheless, this doesn't have a significant impact on the runtimes. Each processor will then calculate their resultant submatrix C' and send it to the master process via the "Gatherv" communicator. The "Gatherv" communicator is the same as the Gather communicator except that it allows "vectors" or arrays of data of different sizes to be collected and ordered at the root. The master process will then construct all of the submatrices into the final result C.

The timings of the four main portions of the program are then put into Pandas data frames and saved to file so that they can be analysed later. Similarly to the previous section, the four main timings that are recorded are: 1. The "Scatter" time - how long it takes the master core to calculate and send out all of the relevant submatrices; 2. The "Calculation" time - how long it takes to calculate all of the C' submatrices; 3. The "Gather" time - how long it takes to send all of the submatrix C' to be sent back

to the master core and be assembled into C; 4. The "Total" time - the time of the whole parallel matrix multiplication, i.e. the sum of the previous three timings.

Because there isn't an easy way to loop over an MPI program with different numbers of processors within Python, we create a BASH file that will loop over the varying number of processors P, and matrix order N that we want to run the program on.

Similarly to the previous section, we will show the results of the program I just described for a number of processors $P = 1, 2, 4, 8, 16, 32$, and we will show the results for varying N using two different matrix multiplication function 1. My handwritten "matrix_mult" function and 2. The Lapack "sgemm" function. However, in contrast to the previous section, we will also show the results of a similar program running on Fortran.

Results from handwritten function

For the first test, I have written a Python program (BitesizeBytes/ TesterFiles/ MPI/ MyFunc/ MatrixMult_MPI_MyFunc.py) that will perform matrix multiplication using my handwritten "matrix_mult" function on two square matrices as described above. The matrices A and B will take order $N = 32, 64, \ldots, 1024, 2048$ so that we can see how the speedup changes as we increase the problem size N. As stated in Chapter 6, it is only feasible to run the "matrix_mult" function up to matrix order $N = 2048$ as at this size the function takes approximately 2 hours to run on 1 core.

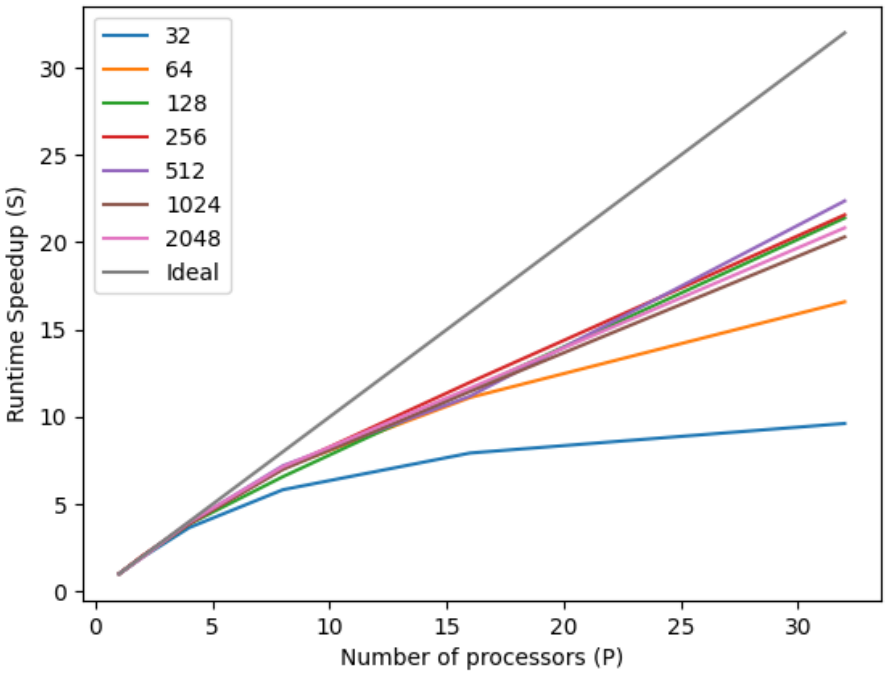

Figure 8.1: The parallel speedup of my handwritten "matrix_mult" function for matrix order $N = 32, 64, \ldots, 1024, 2048$ and number of processors $P = 1, 2, 4, 8, 16, 32$. This function is parallelised using the Python mpi4py library. The maximum speedup $S = 24.2$ at matrix order $N = 2048$ and $P = 32$ cores.

As shown in (Fig. 8.1), this program achieves quite impressive speedups, with a maximum $S(512, 32) = 22.37$, however, the speedup tends to fall slightly after this, with $S(2048, 32) = 20.83$ for the largest N. So, we look to (Fig. 8.2) for a breakdown of the different runtime sections, which shows that the serial α falls disappears almost immediately, with both the Scatter and Gather bars non-existent for $N > 64$. However, given this extremely low α, we should be expecting maximum speedups of $S(N, 32) = 32$.

This is likely occurring because of hidden serial slowdowns that are not effectively tracked during with my timing results.

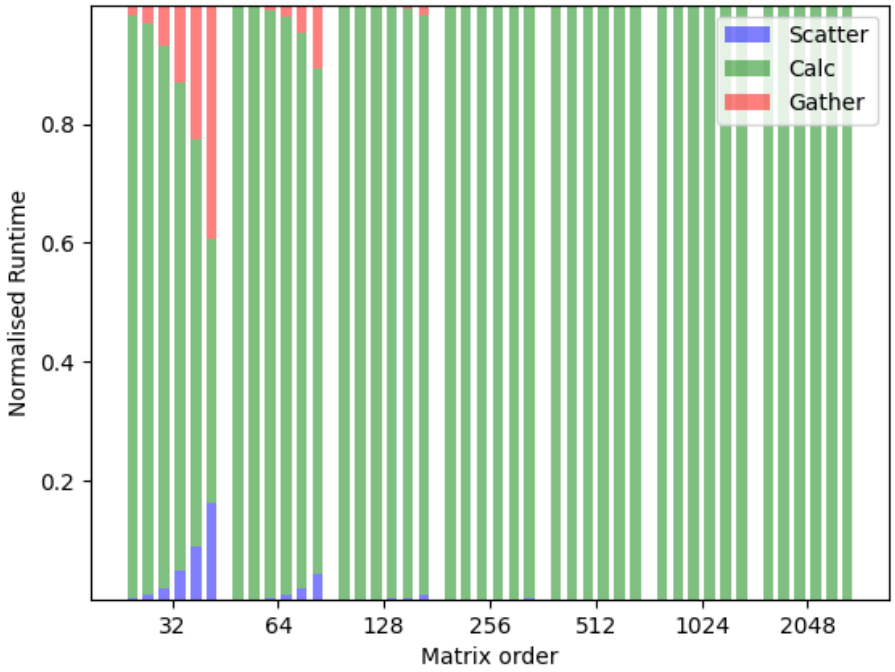

Figure 8.2: The normalised timings of "Scatter", "Calculation" and "Gather" sections of the parallel program (BitesizeBytes/ TesterFiles/ MPI/ MyFunc/ MatrixMult_MPI_MyFunc.py) which is performing matrix multiplication using the handwritten "matrix_mult" for matrix order $N = 32, 64, \ldots, 1024, 2048$ and number of processors $P = 1, 2, 4, 8, 16, 32$. This function is parallelised using the Python mpi4py library. Each group of bars shows the function performing matrix multiplication of A and B a matrix order N and every bar in each group shows the normalised runtimes on $P = 1, 2, 4, 8, 16, 32$, from left to right, respectively.

Similarly to the previous section, we see very close to ideal speedup results for the MPI implementation of the handwritten "matrix_mult" function. However, this is to be expected for a function that is so poorly optimised. There is an adage in the field of parallelism that if a program scales perfectly, it wasn't very well optimised in the first place. So we again move on to the "sgemm" function to see if the parallel speedup is just as good.

Results from sgemm

We now look at the results of Python MPI running on the "sgemm" function. From (Fig. 8.3), we can see that the best parallel speedup $S(32, 32768) = 15.8$ is naturally achieved by the largest $N = 32768$.

We now look at a breakdown of the different runtime sections in (Fig. 8.4). As MPI is a distributed memory implementation, the scatter and gather overheads remain as N grows, although they do fall from $\alpha = 49\%$ for $N = 2048$ to $\alpha = 13\%$ for the largest $N = 32768$, allowing for the best speedup. In contrast to the previous MPI results, the runtime seems to continue scaling towards ideal as N grows.

When compared to the sgemm performance on Python Multiprocessing, we see that MPI wins with a maximum speedup of $S(32, 32768) = 15.8$ compared to Multiprocessing's $S(32, 32768) = 12.3$. This is because MPI is a much lower level interface, meaning that the user has much more control over the data in the program. Having more control clearly allows for better performance, but it must be noted that MPI is much less user-friendly and is quite overwhelming to an inexperienced user. Converting

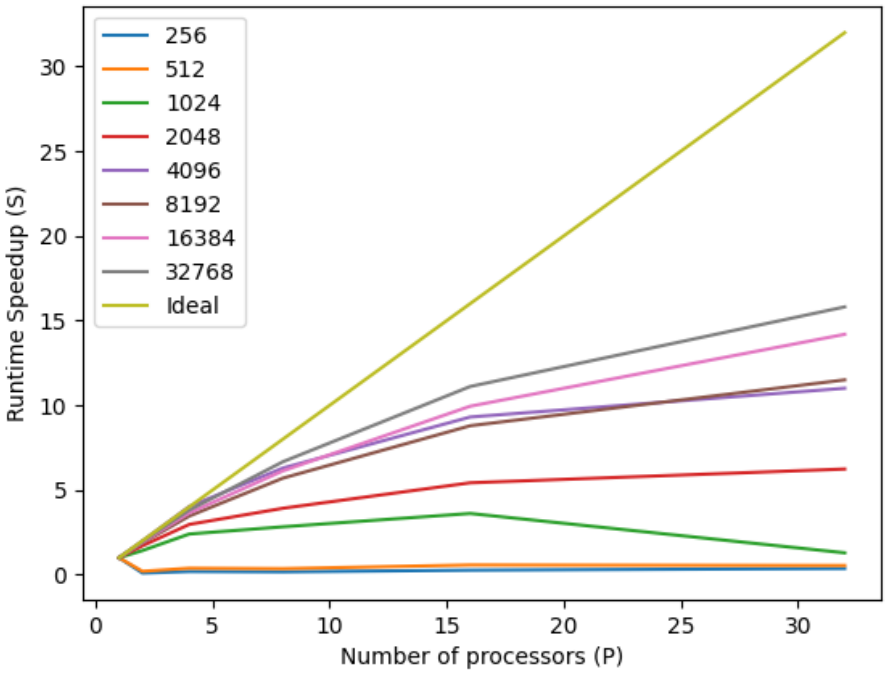

Figure 8.3: The parallel speedup of Python "sgemm" function from Lapack, for matrix order $N = 128, 256, \ldots, 16384, 32768$ and number of processors $P = 1, 2, 4, 8, 16, 32$. This function is parallelised using the Python mpi4py library, which is an implementation of MPI. The maximum speedup $S(32768, 32) = 15.8$ at matrix order $N = 32768$ and $P = 32$ cores.

a problem into the MPI paradigm can lead to restructuring the entire program. In contrast, much of the parallelism in the Multiprocessing library can often be achieved by just adding a few lines in order to set up the worker processes.

Results from Fortran

We have seen the speedup results from Python MPI, so we now look at results from a similar program in the compiled programming language Fortran.

The speedup results can be seen in (Fig. 8.5). This shows a maximum speedup of $S(8192, 32) = 21.5$ at matrix order $N = 8192$ and $P = 32$ cores. However, after matrix order $N = 8192$, the speedup begins to fall for number of processors $P > 16$. The fall in speedup likely occurs because of the size of the data that is being operated on at matrix order $N = 16384, 32768$ with each matrix occupying 8GB and 8GB, respectively in Fortran. Because of this size, the program struggles to keep all of the data in memory at one time, meaning that some of the data have to be stored in virtual memory. This can lead to large slowdowns as reading data from disk storage is extremely slow compared to reading from memory.

Despite the drop in Fortran's speedup for larger N, it still outperforms Python best speedup when running on the sgemm function. For the largest N, Fortran MPI has a speedup of $S(32768, 32) = 17$, compared to Python MPI's best speedup of $S(32768, 32) = 15.8$.

[0]Although all of the other programs in this report have been developed by myself entirely, my supervisor had a large hand in developing this Fortran program, so I don't have as much control over it. Because of this, I don't have the in depth runtime breakdowns as for all the other programs. Nevertheless, we can still speculate on the parallel performance of this algorithm.

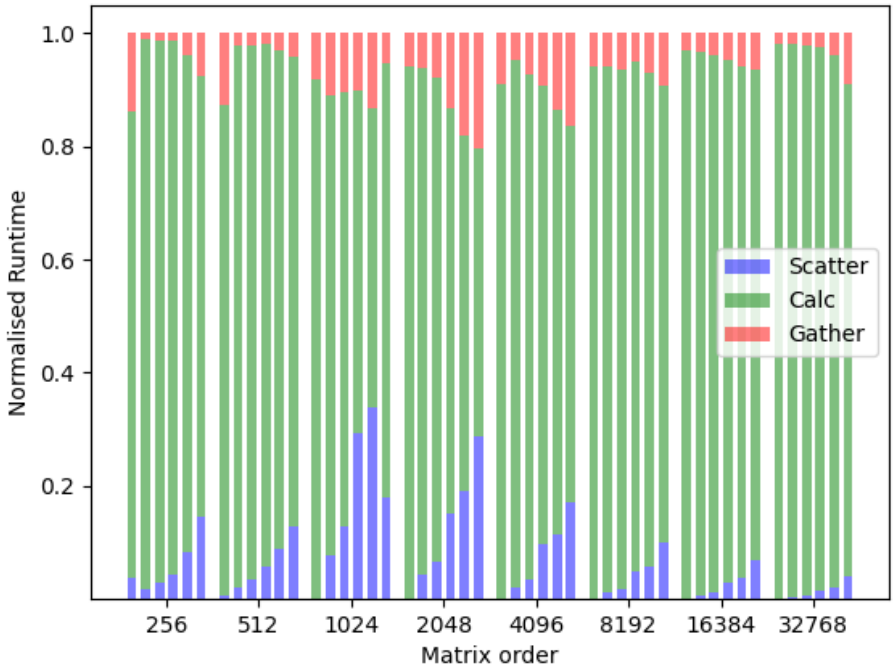

Figure 8.4: The normalised timings of "Scatter", "Calculation" and "Gather" sections of the parallel program (BitesizeBytes/ TesterFiles/ MPI/ Lapack/ MatrixMult_MPI_Lapack.py) which is performing matrix multiplication using the Python "sgemm" function from Lapack for matrix order $N = 128, 256, \ldots, 16384, 32768$ and number of processors $P = 1, 2, 4, 8, 16, 32$. This function is parallelised using the Python mpi4py library. Each group of bars shows the function performing matrix multiplication of A and B a matrix order N and every bar in each group shows the normalised runtimes on $P = 1, 2, 4, 8, 16, 32$, from left to right, respectively.

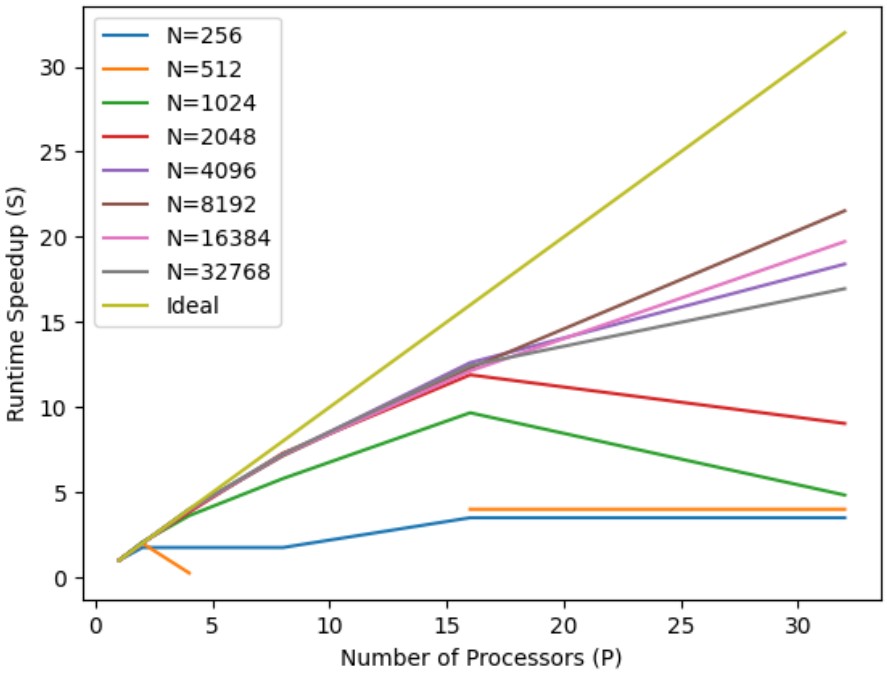

Figure 8.5: The parallel speedup of "sgemm" function from Lapack, for matrix order $N = 128, 256, \ldots, 16384, 32768$ and number of processors $P = 1, 2, 4, 8, 16, 32$. This program is running on the Fortran language and is parallelised with MPI. The maximum speedup $S(8192, 32) = 21.5$ at matrix order $N = 8192$ and $P = 32$ cores.

Chapter 9
MPI IO

The parallelism that can be achieved is bounded by communication under the MPI program; the matrices 4GB each will soon be too large to be sent effectively via an MPI communicator. This is because the size of the matrices grows in an $O(N^2)$ fashion, meaning that the next matrix order would be $N = 65536$, and so each matrix would take four times as much space, so we would have to manage three matrices of size 16GB. Moreover, MPI has an actual bound on the size of data that can be sent between processes which is a warning sign itself.

So, we now approach the problem using the MPI IO paradigm, which allows users to leverage high speed parallel IO in order to sidestep system memory limits. There are useful communicators such as "allRead" that groups data from file together when multiple cores are accessing the same file in order to minimise read time and "allWrite", which allows multiple cores to write data simultaneously to a single file. The approach to creating this program alleviates all communication between processes; it is assumed that the necessary data is already written to storage, which is then read separately by individual processes, operated on and then written to a new file. All of this IO is efficiently managed by MPI to minimise read and write times. Though as we will see, the MPI IO paradigm is not perfect because it relies heavily on reading data from secondary storage; as we have seen earlier in Chapter 3, reading from secondary storage is much slower than reading from memory. This program format accurately resembles the parallelism used in HPC and Scientific research and is very scalable because of the minimal communication between processes.

Instead of the matrices being created on the master core and being distributed throughout the different cores using the scatter function, the matrices are have already been created, and they are stored as files on system storage and called into local memory by individual cores. While this method has much larger overhead for accessing much slower storage, it allows us to avoid the bound on memory and run on much larger matrices.

Program Description

This will be controlled by the BASH script "run_MPI_IO.sh", which will call many different Python scripts. This BASH file will loop over matrix order $N = 1024, 2048, \ldots, 32768 = 2^{10}, \ldots, 2^{15}$. For each matrix order, m, it will call "GenMatrices.py", which generates the two matrices, A and B, and saves them to a text file. The script "MatrixMult_MPI_IO.py" will then be called, which will read in the necessary submatrices of A and B into each core, perform matrix multiplication with the sgemm function from Lapack, generating P submatrices, C', which will all be written to a file storing the final matrix C. The script "CheckResults.py", which will read in the original A and B, calculate C and check that the result calculated by the previous script is correct. The script "Delete_C.py" will then delete the newly generated C. The previous three scripts will be performed for processor count $P = 1, 2, 4, 8, 16, 32$ for the same matrix A and B files, following which, the script "Delete_A_B.py" will delete the files that were created to hold matrix A and B. This will be performed ten times for each matrix order, N, in order to get an average of the time taken. Note that for the largest size $m = 2^{14}, 2^{15}$, the number of iterations was lowered in order to make run times more manageable, as it can be assumed that for matrices of order so large, any element of randomness is minimised.

Similarly to the previous 2 test scenarios, we will measure four main timings in this program: 1. The "Read" time - the time taken for all of the matrices to be read in from file; 2. The "Calculation" time - the time for the parallel matrix multiplication to occur; 3. The "Write" time - the time taken for all of the resultant matrices C' to be written to the C file; 4. The "Total" time - the time for the whole program to run, i.e. the sum of the previous three results.

Test Results

The runtime speedup graph for the program total matrix multiplication program as described above can be seen in (Fig. 9.3). This figure shows that this program achieves a very poor overall speedup, with maximum speedup $S(32768, 32) = 4.6$. This likely comes because of the massive overhead related to accessing secondary storage, so we probe further into the results.

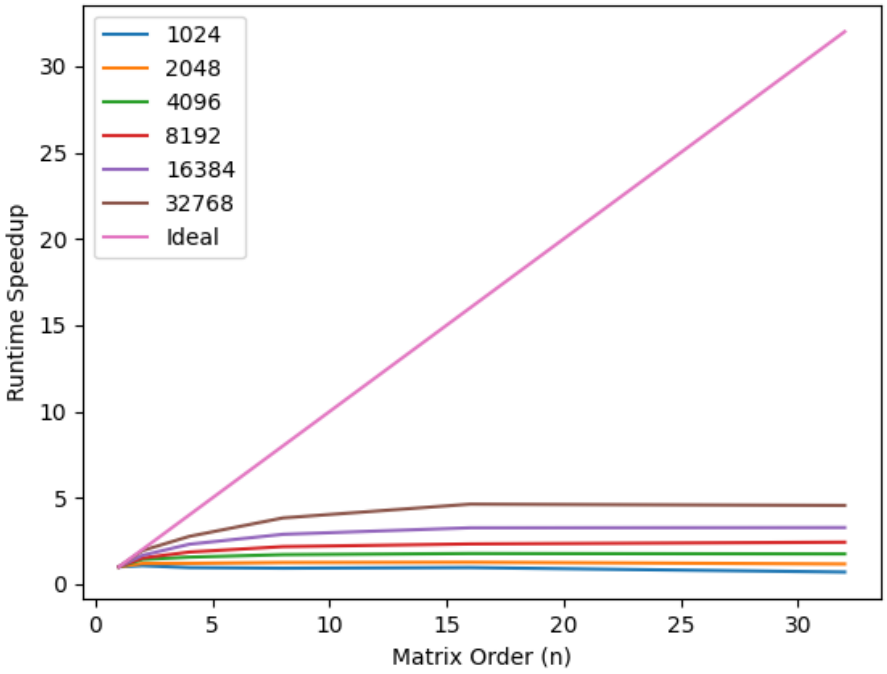

Figure 9.1: The speedup of "MatrixMult_MPI_IO.py" when performing the entire process of reading in matrix A and B from file, performing matrix multiplication and storing the result to a matrix C file. This graph displays the speedup of the script as the number of processors P ranges $P = 1, 2, 4, 8, 16, 32$, over different matrix orders $N = 128, \ldots, 32768$.

So, we again look at the breakdown of the separate "Read", "Calculation" and "Write" timings in the program, which can be seen in (Fig. 9.2). This shows that as N increases, the read and write times are still taking a substantial amount of the program time, with serial portion $\alpha = 70\%$ for the largest matrix order $N = 32768$ and processor count $P = 32$.

To get an idea of how much parallelism there is being achieved, we now look at the runtime speedup graph of only the parallel portion of only the parallel matrix multiplication portion of the program, shown in (Fig. 9.3). This shows that there is a very good speedup for up to matrix order $N = 8192$, with the highest speed up being achieved $S(8192, 32) = 16.9$. Although, for matrix orders $N = 16384, 32768$, the calculation time speedup begins to fall, similarly to the Fortran MPI results. Again, this is likely a result of all of the data not being able to fit in memory at one time because the data of the matrices are being read to many different processors for larger P, meaning there may be some data being unnecessarily replicated in memory. A possible solution to this would be an implementation of MPI IO, where all of the data is read in from external memory but used as shared memory within the node. However, this is too complex to implement for the purposes of this report.

In many of the typical use cases of a program like this, the data would be read in from storage, and multiple operations would be performed on the data before the results are written to file again. So, the situation that we are running under doesn't really justify the massive read and write times for a program like this. Furthermore, in a massively distributed system in a real implementation of a program like this, each of the nodes would have their own local file system that they would be

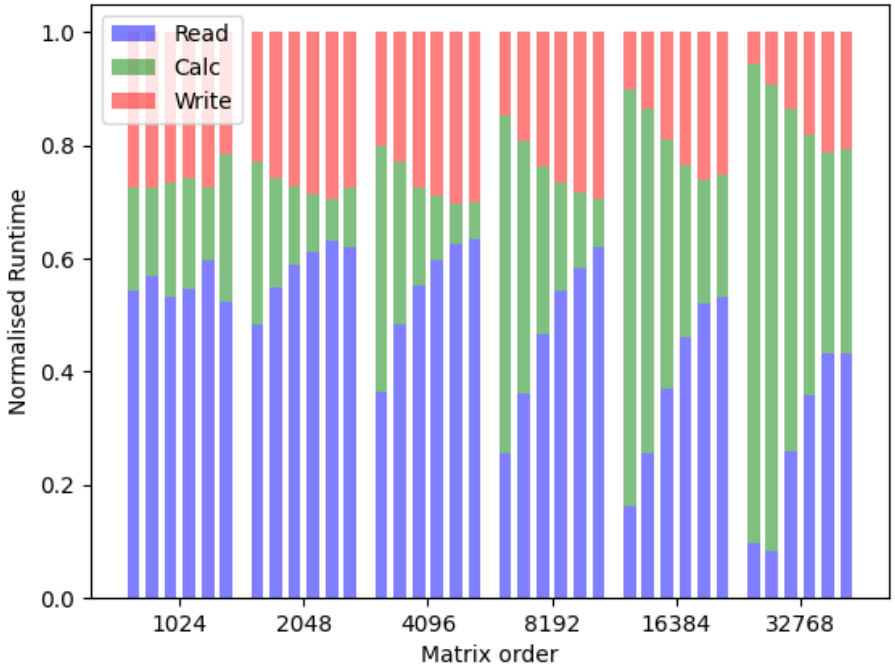

Figure 9.2: The normalised timings of "Scatter", "Calculation" and "Gather" sections of the parallel program (BitesizeBytes/ TesterFiles/ MPI/ Lapack/ MatrixMult_MPI_Lapack.py) which is performing matrix multiplication using the Python "sgemm" function from Lapack for matrix order $N = 1024, \ldots, 16384, 32768$ and number of processors $P = 1, 2, 4, 8, 16, 32$. This function is parallelised using the Python mpi4py library. This is the MPI IO program results, so each of the matrices A,B and C that are worked on are saved to file. Each group of bars shows the function performing matrix multiplication of A and B a matrix order N and every bar in each group shows the normalised runtimes on $P = 1, 2, 4, 8, 16, 32$, from left to right, respectively.

reading their data in form, which can again be more parallelised, whereas, in our environment, each core must be reading/writing from the same file system, which likely is increasing the IO times even more significantly. That is to say, in many circumstances, the only way to effectively run a program of massive scale is to use the MPI IO format. It is an interesting format and a good insight into how parallel programs are run in HPC environments with many nodes. However, in the context of the problem that we are working on, MPI IO is not the most appropriate solution.

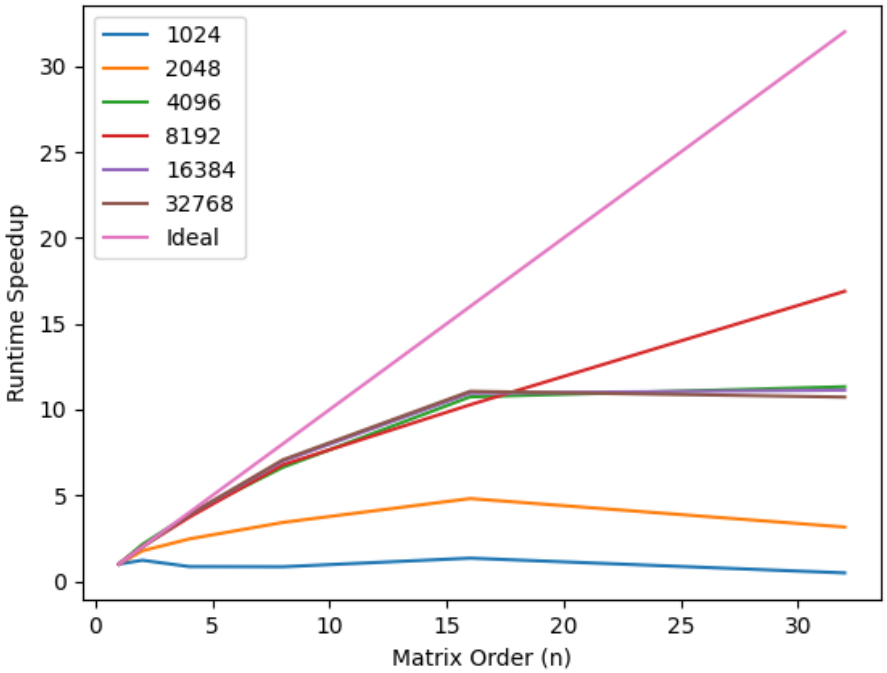

Figure 9.3: The speedup of "MatrixMult_MPI_IO.py" when performing only the calculation portion of the script. This graph displays the speedup of the calculation as the number of processors P ranges $P = 1, 2, 4, 8, 16, 32$, over different matrix orders $N = 128, \ldots, 32768$.

Chapter 10
Runtime Comparison

So far, we have seen that MPI is achieving the best speedups in both Python and Fortran, however, we are yet to see whether impressive speedups necessarily equate to better overall runtimes. The runtimes of the different parallel implementations can be seen in (Fig. 10.1), which shows that MPI does have the best performance in terms of raw runtimes.

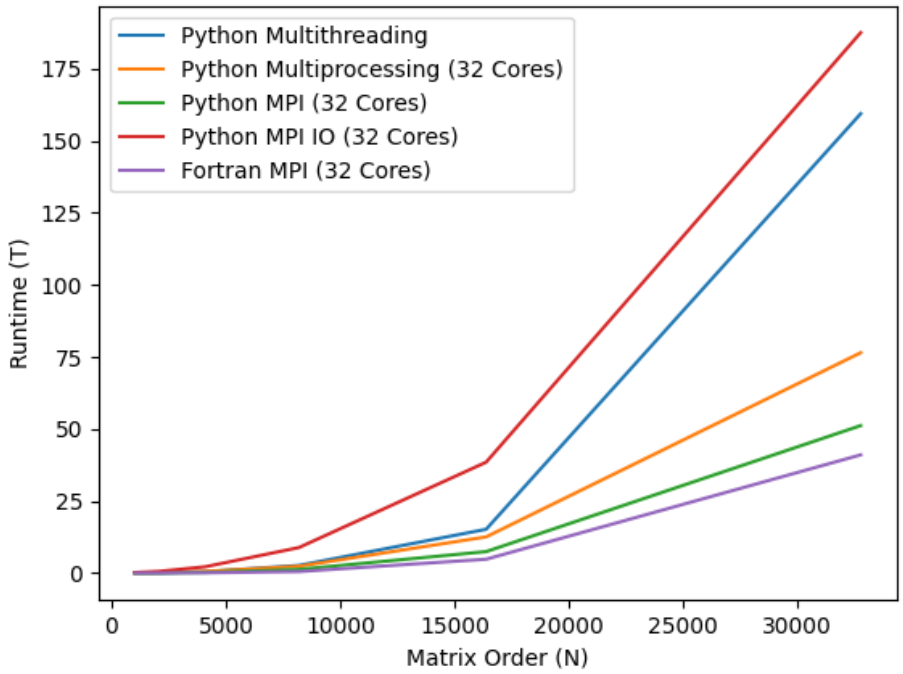

Figure 10.1: Runtime comparison of "sgemm" running on various different parallel formats.

The slowest runtime is Python MPI IO, which is not surprising because of how slow reading from file is.

The second slowest runtime comes from "Multithreading", which is expected as this is a form of parallelism entirely handled by the Operating System and takes no input from the user, so given the ease of implementation, any "free" performance gains welcome to a user.

The third fastest Python runtime comes from the Multiprocessing implementation. It is to be expected that this has a slower runtime than MPI as this is a much higher level interface to the parallelism than what the user has in MPI. Having less control over the data leads to performance losses; as discussed in Chapter 7, there are many slowdowns in this shared memory format when multiple cores are waiting to access shared memory simultaneously.

The best performer of all of the Python runtimes if MPI; this is because it is the lowest level interface, which gives the user much control over the data that is being used. Moreover, MPI is very well optimised for handling such large amounts of data, especially when computation is contained to one node. However, the real power of MPI comes from its ability to scale up to running programs on multiple nodes, which we have not shown in our tests.

TFLOPS					
Process type/ Matrix Order (N)	Python Multithreading	Python Multiprocessing	Python MPI	Python MPI IO	Fortran MPI
8192	0.415	0.465	0.864	0.124318	2.1
16384	0.579	0.701	1.181	0.228624	1.832
32768	0.441	0.92	1.375	0.375301	1.716

Table 10.1: TFLOPS comparison of "sgemm" running on various different parallel formats.

We can also see that Fortran MPI outperforms Python MPI. Although we see the speedups for Fortran being go fall for the largest sizes of N, we see that is it still outperforming Python in the runtimes.

We can see the performance of these different implementations in a different light in (Table 10.1), which shows the performance in terms of TFLOPS. Fortran MPI has the peak performance of 2.1TFLOPS for matrix order $N = 8192$, which falls in line with the highest parallel speedup that Fortran MPI achieved, of $S(8192, 32) = 21.5$. While the performance of Fortran MPI does begin to fall to 1.716TFLOPS after this peak, it does still outperform Python MPI's 1.375TFLOPS for the largest matrix order $N = 32768$. Interestingly, where the Fortran performance begins to fall for larger N, the Python performance in Multiprocessing, MPI and MPI IO continues to rise quite steadily. However, this again brings us back to the idea shown earlier that if something scales perfectly, it likely wasn't well optimised in the first place.

Chapter 11
Conclusion

This project demonstrated the ability of parallel systems to improve the runtime of different programs. We showed strong speedups in both Monte Carlo Analysis and Matrix Multiplication, which provided two completely different challenges of parallel decomposition.

The Main Result was that MPI offered the opportunity to achieve the best parallel speedups and overall runtime performance. However, given that Multiprocessing is much easier to implement, it is still a viable option for smaller problem sizes. MPI is the only option to scale a program up to a massively distributed system for larger problem sizes. We demonstrated how this could be achieved using MPI IO, in which there was no communication between processes.

If this project were to be continued, many different avenues could be explored. Firstly, as discussed in Chapter 9, I would like to research a parallel performance on a hybrid of shared and distributed memory. When running on multiple nodes, the individual nodes can be treated as a distributed system, but it will be treated as a shared memory system within each of the nodes. Another exciting avenue is the field of GPU (Graphics Processing Unit) parallelism. Where CPUs may have 16, 32 or even 64 cores, GPUs will have thousands of cores. These devices are specialised for highly parallel workloads, such as computer graphics rendering and scientific simulation.

Bibliography

[1] Archer. *Performance Scaling*. URL: https://www.archer.ac.uk/training/course-material/2016/12/mpi_scaling_manc/Slides/Scaling.pdf. (accessed: 28/05/21).

[2] Guru99. *Compiler vs Interpreter*. URL: https://www.guru99.com/difference-compiler-vs-interpreter.html. (accessed: 28/05/21).

[3] University of Nizhni Novgorod. *Introduction to Parallel Programming*. URL: http://www.lac.inpe.br/~stephan/CAP-372/matrixmult_microsoft.pdf. (accessed: 28/05/21).

[4] University of Nizhni Novgorod. *Introduction to Parallel Programming*. URL: http://www.cs.csi.cuny.edu/~gu/teaching/courses/csc76010/slides/Matrix%5C%20Multiplication%5C%20by%5C%20Nur.pdf. (accessed: 28/05/21).

[5] NYU. *Python MPI - Message Passing*. URL: https://nyu-cds.github.io/python-mpi/02-messagepassing/. (accessed: 28/05/21).

[6] Teldat. *Parallel computing via multicore computers allow high processing capacity*. URL: https://www.teldat.com/blog/en/parallel-computing-bit-instruction-task-level-parallelism-multicore-computers/. (accessed: 28/05/21).

[7] Tu Wein. *Distributed-Memory Systems*. URL: https://www.iue.tuwien.ac.at/phd/weinbub/dissertationsu17.html. (accessed: 28/05/21).

[8] WikiChip. *Floating-Point Operations Per Second (FLOPS)*. URL: https://en.wikichip.org/wiki/flops. (accessed: 28/05/21).

[9] Wikipedia. *Multithreading (computer architecture)*. URL: https://en.wikipedia.org/wiki/Multithreading_(computer_architecture). (accessed: 28/05/21).

[10] Wikipedia. *Shared memory*. URL: https://en.wikipedia.org/wiki/Shared_memory. (accessed: 28/05/21).

[11] Wikipedia. *von Neumann architecture*. URL: https://en.wikipedia.org/wiki/Von_Neumann_architecture#/media/File:Von_Neumann_Architecture.svg. (accessed: 28/05/21).

Appendix A

Appendix: Code

This appendix contains distilled versions of the programs that I have used for the results in this report. The entire code-base that I have created for this project can be seen at https://github.com/dnelson17/BitesizeBytes.git. The README file contains a short guide.

Python Monte Carlo Approximation Code

```
from multiprocessing import Pool
import pandas as pd
import random
import time
import sys

#This is the parallel Monte Carlo function, so each worker process will be executing this
    function
def monte_carlo(attempts):
    i = 0
    hits = 0
    for i in range(attempts):
        #Generates random points for x,y
        x = random.uniform(0, 1)
        y = random.uniform(0, 1)
        #Checks whether (x,y) lies within unit circle
        if x**2 + y**2 <= 1:
            #if so, increments hits counter by 1
            hits += 1
    #Returns the number of his that we found
    return hits


def gen_time_results(attempts, no_cores, scaling_type):
    print(f"{no_cores} core(s)")
    #If we are running the "strong scaling" version, each core will perform N/P attempts
    if scaling_type == "strong":
        attempts_split = (10**attempts)//no_cores
    #If we are running the "weak scaling" version, each core will perform N attempts
    else:
        attempts_split = 10**attempts
    #Starts the timer
    start = time.perf_counter()
    #Creates a list to send the number of attempts to each core
    send_list = [(attempts_split,) for _ in range(no_cores)]
    #Opens a pool of worker processes
    p = Pool(processes=no_cores)
    #This is where the parallelism is taking place, each worker is given their number of
        attempts. The "monte_carlo" function will be called on each of them and they will
        return their number of "hits"
    hits_list = p.starmap(monte_carlo, (send_list))
    #Closes the pool of processes
    p.close()
    #Stops the timer
    finish = time.perf_counter()
    #Sums all of the hits from the workers
    total_hits = sum(hits_list)
    #Outputs the approximation of Pi to the user
    approx_pi = 4*total_hits/(attempts*no_cores)
    print(f"Estimation: {approx_pi}")
```

```
    #Calculates the time taken of the monte carlo calculation
    time_taken = round(finish-start,10)
    #Returns the time taken to be saved to a dataframe
    return time_taken
```

Listing A.1: Python Monte Carlo program.

Python Multiprocessing Code

```
from multiprocessing import Pool, shared_memory
from scipy.linalg import blas as FB
import pandas as pd
import numpy as np
import time

#Below are the parallel matrix multiplcation functions that will be called. In the real
    files, they will be in two separate files. The functions only vary in the part that
    is between the "-----", so I will only explain the first function

#Sgemm version of matrix multiplicaiton
def matrix_mult(i,len_i,j,len_j,mat_size,name_A,name_B,name_C):
    #The different cores often  have separate the timeer, so we mark the time before any
        calculation so that results can be offset to the same starting time
    stabiliser_time = time.time()
    #Identifies the shared memory blocks of A,B,C
    existing_shm_A = shared_memory.SharedMemory(name=name_A)
    existing_shm_B = shared_memory.SharedMemory(name=name_B)
    existing_shm_C = shared_memory.SharedMemory(name=name_C)
    #Calculates the (i,j) coodinates of the submatrices that must be worked on
    i1 = i*len_i
    i2 = (i+1)*len_i
    j1 = j*len_j
    j2 = (j+1)*len_j
    #Reads the relevant block of A,B,C from shared memory
    sub_mat_A = np.ndarray((mat_size,mat_size), dtype=np.float32, buffer=existing_shm_A.
        buf)[i1:i2,:]
    sub_mat_B = np.ndarray((mat_size,mat_size), dtype=np.float32, buffer=existing_shm_B.
        buf)[:,j1:j2]
    sub_mat_C = np.ndarray((mat_size,mat_size), dtype=np.float32, buffer=existing_shm_C.
        buf)
    #Marks the start of the calculation time
    calc_start = time.time()
    #-----------------------------------------------
    #Calculates the submatrix C' using sgemm and saves it to shared memory
    sub_mat_C[i1:i2,j1:j2] = FB.sgemm(alpha=1.0, a=sub_mat_A, b=sub_mat_B)
    #-----------------------------------------------
    #<arks the end of the calculation time
    calc_finish = time.time()
    #Closes the link to the shared memory blocks
    existing_shm_A.close()
    existing_shm_B.close()
    existing_shm_C.close()
    #Returns all the timiing results
    return stabiliser_time, calc_start, calc_finish


#My handwritten version of matrix multiplicaiton
def matrix_mult(i,len_i,j,len_j,mat_size,name_A,name_B,name_C):
    stabiliser_time = time.time()
    existing_shm_A = shared_memory.SharedMemory(name=name_A)
    existing_shm_B = shared_memory.SharedMemory(name=name_B)
    existing_shm_C = shared_memory.SharedMemory(name=name_C)
    i1 = i*len_i
    i2 = (i+1)*len_i
    j1 = j*len_j
    j2 = (j+1)*len_j
    sub_mat_A = np.ndarray((mat_size,mat_size), dtype=np.float32, buffer=existing_shm_A.
        buf)
    sub_mat_B = np.ndarray((mat_size,mat_size), dtype=np.float32, buffer=existing_shm_B.
        buf)
    sub_mat_C = np.ndarray((mat_size,mat_size), dtype=np.float32, buffer=existing_shm_C.
        buf)
    calc_start = time.time()
```

```
    #-------------------------------------------------
    #Calulates matrix C' using hadwritten matrix multiplication function
    for i in range(i1,i2):
        for j in range(j1,j2):
            for k in range(mat_size):
                sub_mat_C[i,j] += sub_mat_A[i,k] * sub_mat_B[k,j]
    #-------------------------------------------------
    calc_finish = time.time()
    existing_shm_A.close()
    existing_shm_B.close()
    existing_shm_C.close()
    return stabiliser_time, calc_start, calc_finish


def gen_time_results(mat_size, core_list):
    #Generates 2 random matrices A,B
    data_A = np.random.rand((mat_size,mat_size)).astype(np.float32)
    data_B = np.random.rand((mat_size,mat_size)).astype(np.float32)
    #Generates empty matrix C for results to be written
    data_C = np.empty((mat_size,mat_size),dtype=np.float32)
    #Defines shared memory blocks for matrices to be place into
    shm_A = shared_memory.SharedMemory(create=True, size=data_A.nbytes)
    shm_B = shared_memory.SharedMemory(create=True, size=data_B.nbytes)
    shm_C = shared_memory.SharedMemory(create=True, size=data_C.nbytes)
    #Places matrix A,B,C into shared memory
    mat_A = np.ndarray(data_A.shape, dtype=data_A.dtype, buffer=shm_A.buf)
    mat_A[:] = data_A[:]
    mat_B = np.ndarray(data_B.shape, dtype=data_B.dtype, buffer=shm_B.buf)
    mat_B[:] = data_B[:]
    mat_C = np.ndarray(data_C.shape, dtype=data_C.dtype, buffer=shm_C.buf)
    mat_C[:] = data_C[:]
    #Gets the name of the sahred memory blocks so that they can be accessed by workers
    name_A = shm_A.name
    name_B = shm_B.name
    name_C = shm_C.name
    #Define empty lists for various time results to be saved to
    total_times = []
    send_times = []
    calc_times = []
    recv_times = []
    #A list of cores to run on (typically [1,2,4,8,16,32]) will be passed from the main
        function. This loop will perform the matrix mutlplication on each of these
        processor counts.
    for no_cores in core_list:
        #Resets the values of matrix C to zeros after the previous matrix multiplcation
        mat_C[:] = np.zeros((mat_size,mat_size),dtype=np.float32)
        #Assuming the matrix is of size 2^n for int N, we take log2 to find the value of
            n
        power = np.log2(no_cores)/2
        #Represents the number of partitons that must be calculated in the result matrix
            C, phi_i and phi_j, respectively
        pars_i = int(2**(np.ceil(power)))
        pars_j = int(2**(np.floor(power)))
        #Represents the size of each partiton in the i and j axis, psi_i and psi_j,
            respectively
        len_i = int(mat_size/pars_i)
        len_j = int(mat_size/pars_j)
        #Starts overall timing
        total_start = time.time()
        #Creates a list of all parameters to be sent to workers. These paramaters include
             the "coordinates" of the matrices that should be work on and the name of the
             shared memory blocks that the matrices are sotred in.
        send_list = [[i,len_i,j,len_j,mat_size,name_A,name_B,name_C] for j in range(
            pars_j) for i in range(pars_i)]
        #Opens a pool of worker processes
        p = Pool(processes=no_cores)
```

```
        #Passes the parameters to each of the workers. Some timing results are then
            returned
        res_list = p.starmap(matrix_mult, send_list)
        #Closes the worker processes
        p.close()
        #Everything in the "----" below is just calculating timing results
        # ----------------------
        total_finish = time.time()
        calc_start_list = []
        calc_finish_list = []
        res_list = list(res_list)
        for i in range(len(res_list)):
            time_difference = res_list[0][0] - res_list[i][0]
            calc_start_list.append(res_list[i][1]+time_difference)
            calc_finish_list.append(res_list[i][2]+time_difference)
        calc_start = min(calc_start_list)
        calc_finish = max(calc_finish_list)
        send_time = calc_start-total_start
        calc_time = calc_finish-calc_start
        gather_time = total_finish-calc_finish
        total_time = total_finish-total_start
        assert send_time + calc_time + gather_time == total_time
        send_times.append( round(send_time,10) )
        calc_times.append( round(calc_time,10) )
        recv_times.append( round(gather_time,10) )
        total_times.append( round(total_time,10) )
        # ----------------------
    #Closes and unlinks the shared memory memory blocks that were created for A,B,C
    shm_A.close()
    shm_B.close()
    shm_C.close()
    shm_A.unlink()
    shm_B.unlink()
    shm_C.unlink()
    #Returns timing results to be saved to dataframes
    return tuple(send_times), tuple(calc_times), tuple(recv_times), tuple(total_times)
```

Listing A.2: Python Multiprocessing matrix multiplication program.

Python MPI Code

```
from scipy.linalg import blas as FB
from mpi4py import MPI
import pandas as pd
import numpy as np
import time
import sys

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

#Reads in the order of the matrix that will be worked on from the command line
mat_power = int(sys.argv[1])
mat_size = 2**mat_power

# Initialize the 2 random matrices A and B only if this is rank 0, the master core
mat_A = None
mat_B = None
if rank == 0:
    mat_A = np.random.rand(mat_size,mat_size).astype(np.float32)
    mat_B = np.random.rand(mat_size,mat_size).astype(np.float32)
    #Transpose B so that it is easier to send to workers
    mat_B = np.transpose(mat_B)
    mat_B = np.ascontiguousarray(mat_B, dtype=np.float32)

#Timing starts for the "scatter" portion
comm.Barrier()
total_start = MPI.Wtime()

#Calclates x for number of processors P=2^x
power = np.log2(size)/2
#Calclulates the number of partitions in the i,j axes, phi_i and phi_j, respectively
pars_i = int(2**(np.ceil(power)))
pars_j = int(2**(np.floor(power)))
#Calclulates the length of partitions in the i,j axes, psi_i and psi_j, respectively
len_i = int(mat_size/pars_i)
len_j = int(mat_size/pars_j)
#Determines whether x is even or odd for P=2^x
factor = 2**(int(np.log2(size))%2)
#Creates a list of the coordinates that each core will be working on
displ_A = [len_i * (factor * list_rank // pars_i) for list_rank in range(size)]
displ_B = [len_j * (list_rank % pars_j) for list_rank in range(size)]

#Creates empty matrices for each worker's submatrices of A,B to be sent to
sub_mat_A = np.empty((len_i,mat_size),dtype=np.float32)
sub_mat_B = np.empty((len_j,mat_size),dtype=np.float32)

#The master core will iterate over every worker and send them their respective
    submatrices
if rank == 0:
    for i in range(1,size):
        comm.Send([mat_A[displ_A[i]:displ_A[i]+len_i],MPI.FLOAT],dest=i,tag=25)
        comm.Send([mat_B[displ_B[i]:displ_B[i]+len_j],MPI.FLOAT],dest=i,tag=25)
    #Defines the matrices that the master core will be operating on
    sub_mat_A = mat_A[displ_A[0]:displ_A[0]+len_i]
    sub_mat_B = mat_B[displ_B[0]:displ_B[0]+len_j]
else:
    #Every worker core receives their submatrices of A,B
    comm.Recv([sub_mat_A,MPI.FLOAT],source=0)
    comm.Recv([sub_mat_B,MPI.FLOAT],source=0)

#Starts the timer for the beginning of the "calculation" portion
comm.Barrier()
calc_start = MPI.Wtime()
```

```
#Each core calculates their submatrix C' using sgemm. In the handwritten version of this,
     the "matrix_mult" function will be called instead
sub_mat_C = FB.sgemm(alpha=1.0, a=sub_mat_A, b=sub_mat_B, trans_b=True)

#Stops the timer for the "calculation" portion, starting the timer for the "gather"
    portion
comm.Barrier()
calc_finish = MPI.Wtime()

#Creates an empty matrix for the submatrices C' to be gathered into
mat_C = None
if rank == 0:
    mat_C = np.empty(mat_size*mat_size,dtype=np.float32)

#Gathers all of the submatrices C'
count_C = [len_i*len_j for _ in range(size)]
displ_C = [len_i*len_j*list_rank for list_rank in range(size)]
sub_mat_C = np.ascontiguousarray(sub_mat_C, dtype=np.float32)
comm.Gatherv(sub_mat_C,[mat_C,count_C,displ_C,MPI.FLOAT],root=0)

#Restructures all of the submatrices into the final result matrix C
if rank == 0:
    mat_C = np.split(mat_C, size, axis=0)
    mat_C = np.apply_along_axis(func1d=np.reshape, axis=1, arr=mat_C, newshape=(len_i,
        len_j) )
    mat_C = np.vstack( np.split( np.concatenate(mat_C,axis=1) , pars_i, axis=1) )

#Stops the timer as the matrix multiplication is now finished
comm.Barrier()
total_finish = MPI.Wtime()

#The matrix multiplication is now done, everything past this is just saving timing
    results.
```

Listing A.3: Python MPI sgemm version program.

```
#In the "matrix_mult" handwritten verision of this, we only difference will be that the
    line that calls sgemm will instead call "matrix_mult" to perform tha matrix
    multiplication.

#So the difference is that this:
sub_mat_C = FB.sgemm(alpha=1.0, a=sub_mat_A, b=sub_mat_B, trans_b=True)
#will change to this:
sub_mat_C = matrix_mult(sub_mat_A, sub_mat_B)

#Where this is the matrix_mult function:
def matrix_mult(mat_A, mat_B):
    mat_C = np.zeros((mat_A.shape[0],mat_B.shape[0]),dtype=np.float32)
    for i in range(mat_A.shape[0]):
        for j in range(mat_B.shape[0]):
            for k in range(mat_B.shape[1]):
                mat_C[i,j] += mat_A[i,k] * mat_B[j,k]
    return mat_C
```

Listing A.4: The handwritten "matrix_mult" function that is used in the Python MPI program.

Fortran MPI Code

```
module precisn
    implicit none
    INTEGER, PARAMETER     :: wp = SELECTED_REAL_KIND(6)  ! 'single' precision

end module precisn

program parallel_matrix
    USE precisn
    USE MPI
    implicit none
    real(wp), allocatable :: my_M(:,:)
    real(wp), allocatable :: my_V(:,:)
    real(wp), allocatable :: my_R(:,:)
    integer               :: ierr, ii, jj
    integer               :: ncpus
    integer               :: myrank
    integer               :: m_size
    integer               :: numrows
    real(wp)              :: start_time, end_time
    real(wp), parameter   :: alpha = 1.0_wp
    real(wp), parameter   :: beta  = 0.0_wp
    real(wp)              :: sumtime
    integer, parameter    :: niter=10

    call MPI_Init(ierr)
    call MPI_comm_size(mpi_comm_world, ncpus, ierr)
    call MPI_comm_rank(mpi_comm_world, myrank, ierr)

    do jj=8,15
        m_size = 2**jj

        numrows = m_size/ncpus

        call setup_matrix(numrows, m_size)

        call MPI_Barrier(mpi_comm_world, ierr)
        start_time = hel_time()

        do ii=1,niter
            call sgemm( 'n','n', m_size, m_size, numrows,  alpha, my_M, m_size, my_V,
                numrows, beta, my_R,m_size )
        end do

        call MPI_Barrier(mpi_comm_world, ierr)
        end_time   = hel_time()

        if (myrank==0) then
            print *, ncpus, m_size, (end_time - start_time)/niter
        end if

        call tear_down_matrix
    end do

    call MPI_Finalize(ierr)



contains

subroutine setup_matrix (nrows,m_size)
    implicit none
    integer, intent(in) :: nrows
    integer, intent(in) :: m_size
    integer             :: ierr, ii, jj
```

```
    allocate(my_M(m_size,nrows),stat=ierr)
    if (ierr /= 0) print *, "allocation error"
    allocate(my_V(nrows,m_size))
    allocate(my_R(m_size,m_size))

    call random_number(my_M)
    call random_number(my_V)
    my_R = 0.0_wp

end subroutine setup_matrix

subroutine tear_down_matrix
    implicit none
    integer :: ierr

    deallocate(my_M, my_V, my_R, stat=ierr)
    if (ierr /= 0) print *, "allocation error"

end subroutine tear_down_matrix

REAL(wp) FUNCTION hel_time()

    INTEGER :: it0, count_rate

    it0 = 0
    count_rate = 1

    CALL SYSTEM_CLOCK(it0, count_rate)
    hel_time = REAL(it0, wp) / REAL(count_rate, wp)

END FUNCTION hel_time
end program parallel_matrix
```

Listing A.5: Fortran matrix multiplication program.

Python MPI IO Code

```
from scipy.linalg import blas as FB
from mpi4py import MPI
import pandas as pd
import numpy as np
import time
import sys

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

amode_A = MPI.MODE_RDONLY
amode_B = MPI.MODE_RDONLY
amode_C = MPI.MODE_WRONLY|MPI.MODE_CREATE

#Reads the Matrix Size from the command line
mat_power = int(sys.argv[1])
iteration = int(sys.argv[2])
mat_size = 2**mat_power

#Assuming the number of processors is of size 2^n for int n, we take log2 to find the
    value of n
power = np.log2(size)/2
#the number of partitons that must be calculated in the result matrix C in the i and j
    dimensions
pars_i = int(2**(np.ceil(power)))
pars_j = int(2**(np.floor(power)))
#the size of each partiton in the i and j axis
i_size = int(mat_size/pars_i)
j_size = int(mat_size/pars_j)
#Adjusts partition sizes for odd values of n
factor = 2**(int(np.log2(size))%2)
#Calculates to coordinates of the result block matrix in mat C
i_coord = factor * rank // pars_i
j_coord = rank % pars_j

comm.Barrier()
io_start = MPI.Wtime()

#Opening and reading matrix A
fh_A = MPI.File.Open(comm, f"mat_A/mat_A_{mat_size}_{iteration}.txt", amode_A)
buf_mat_A = np.empty((i_size,mat_size), dtype=np.float32)
offset_A = i_coord*buf_mat_A.nbytes
fh_A.Read_at_all(offset_A, buf_mat_A)
fh_A.Close()
#Opening and reading matrix B
fh_B = MPI.File.Open(comm, f"mat_B/mat_B_{mat_size}_{iteration}.txt", amode_B)
buf_mat_B = np.empty((j_size,mat_size), dtype=np.float32)
offset_B = j_coord*buf_mat_B.nbytes
fh_B.Read_at_all(offset_B, buf_mat_B)
mat_B = np.transpose(buf_mat_B)
fh_B.Close()

comm.Barrier()
calc_start = MPI.Wtime()

#Calculate submatrix of C
mat_C = FB.sgemm(alpha=1.0, a=buf_mat_A, b=mat_B)

comm.Barrier()
calc_finish = MPI.Wtime()

#Each core writes their submatrix C' into the file for matrix C
buf_mat_C = np.ascontiguousarray(mat_C)
```

```
fh_C = MPI.File.Open(comm, f"mat_C/mat_C_{mat_size}_{iteration}.txt", amode_C)
filetype = MPI.FLOAT.Create_vector(i_size, j_size, mat_size)
filetype.Commit()
offset_C = (mat_size*i_coord*i_size + j_coord*j_size)*MPI.FLOAT.Get_size()
fh_C.Set_view(offset_C, filetype=filetype)
fh_C.Write_all(buf_mat_C)
filetype.Free()
fh_C.Close()

comm.Barrier()
io_finish = MPI.Wtime()
```

Listing A.6: Python MPI IO matrix multiplication program.