CM30225 Parallel Computing
Coursework 2: Distributed Memory Architecture

January 6, 2017

Contents

1 Introduction

The *distributed memory architecture* is where processors each have their own individual address space in memory. Typically multiple processors are connected via a network and communicate with each other through message passing. Message passing has a considerable overhead, and so the design of distributed programs must consider the costs associated with communication carefully. Indeed, each process should ideally work on local data and avoid accessing remote data as far as possible.

Scaling distributed architectures can be powerful as the computational power begins to outweigh the expense of explicit message passing. However, communication on small problem sizes is still expensive and can offset the benefits of parallelisation, as can inefficient network topologies between nodes. The Open MPI implementation of the MPI standard supports SPMD (and MPMD), and will be used in this project to investigate the results of matrix relaxation across a distributed architecture.

2 Approach to Distributed Parallelism

Any unsynchronised, concurrent access to data involving a write results in a data race. As established in Coursework 1, avoiding data races is crucial to keeping data both consistent and accurate across the program. Therefore, my approach of using two matrices for the relaxation computation remains unchanged. In this approach, the program reads from the first matrix and concurrently writes once to every cell in the second matrix – during this time no read access occurs. Importantly this avoids potential data races when relaxing an $n \times n$ matrix composed of $(n-2)^2$ cells to relax. $c_{i+1,j}$, $c_{i-1,j}$, $c_{i,j+1}$, $c_{i,j-1}$ for each $c_{i,j}$ are relaxed into the second matrix, and precision is calculated through read access on both matrices – in both scenarios, avoiding data races. There is additional space complexity involved with a second matrix – $\mathcal{O}(n^2)$ – meaning that further memory will be dynamically allocated in the heap. Nevertheless, in the context of distributed systems, I believe that this is preferable in comparison to the overheads of maintaining a single matrix through additional message passing.

Generating numbers for the initial matrix was also adapted. Rather than using fixed random numbers, matrixs are generated with a fixed non-random number pattern. This enables reproducibility in tests across the distributed program and gives more significant results as the matrix size scales. The psuedocode for the distributed approach is as follows:

```
DETERMINE environment variables (e.g matrix size, precision)
INIT matrix with fixed non-random numbers
WHILE outside precision threshold
    distribute work
    relax array
    set new precision
    swap matrices pointers
END WHILE
```

2.1 Allocation of Work

One approach to this solution is to broadcast the matrix from the root node to all processes in the communicator using *MPI_Bcast*, and to get each process to make calculations (in parallel) including working out their own starting position and the number of cells to work on using the matrix. Whilst getting each process to make its own calculations is a good idea, broadcasting the matrix to allow this is not an efficient solution, and scales very poorly. For instance consider a 10000 × 10000 matrix to be relaxed across 4 nodes with 10 processes each. Taking a standard *double* data type size on modern architectures (8 bytes), this is: $8 \times 10000 \times 10000$ bytes, or 800 megabytes, which needs to be broadcast from the root process to each of the 40 processes. Evidently this is not scalable as both the distributed architecture scales and the problem size scales. A further approach could be for each process to use the input dimension to calculate its own starting position and the number of cells to work on, but this still requires a full broadcast of the matrix after these calculations.

The solution was to use *MPI_Scatterv* to distribute sections of the matrix, such that each process only receives a segment of the matrix relevant to them. This hugely reduces the amount of data being passed around in messages on the network. $p - 1$ processes receive $\frac{n^2}{p} + 2n$ cells (where n = dimension, p = number of processes), with the final process receiving additional cells if the matrix does not divide equally. Clearly this is preferable to each process receiving n^2 cells. For instance, for a 5000x5000 matrix to be relaxed across 40 processes, scattering[1] uses $\approx 2.5\%$ of the network bandwidth which broadcasting[2] uses.

One of the considerations with this approach is that one process will get an unequal share of the matrix if the matrix does not divide equally. In the shared memory coursework, I remedied this by ensuring an equal split of work for threads by equally distributing the remaining number of cells over existing threads. However, this is not a feasible solution for a distributed system when using *MPI_Scatterv*, where subdivided rows of the matrix create an unneeded complexity and can result in an unnecessary overhead. For instance, subdivided rows require partial surrounding rows to also be sent for precision checking and, if avoiding this for simplicities sake, there is the overhead of the additional rows containing cells which the process won't actually use for any calculations. For simplicity, the final unequal section is simply communicated to the final process at once, who will deal with it faster than the costs of complex row splitting and communication elsewhere in the program.

Each process then relaxes its given section, starting at cell $n + 1$ cell on the second row of its section, and ending at the final cell on the penultimate row. The root node then gathers all relaxed cells, and reconstructs them in the second matrix using the displacement array to calculate positions in the matrix.

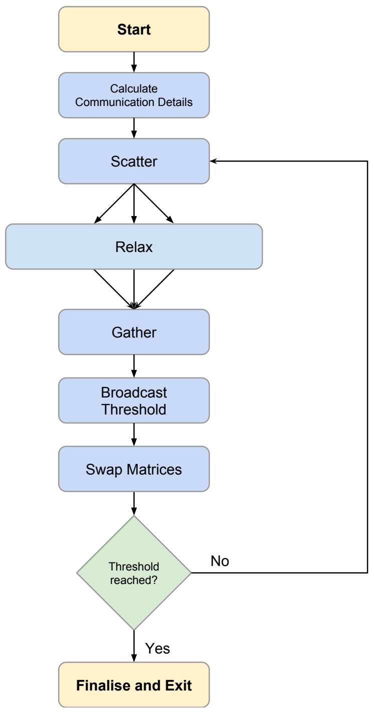


Figure 1: Flow Diagram

2.2 Precision Checking

Using the reconstructed matrix, the root node calculates the overall precision between both matrices and writes a 1 or a 0 to the *outside_threshold* variable. This variable is then broadcast from the root node to all processes in the communicator, to either move on to the collective operation of *MPI_Scatterv* for further relaxation, or to exit the while loop so all processes can call *MPI_Finalize*.

Unlike in the shared memory coursework, I did not choose to distribute precision checking to each process. There is added space complexity involved with this approach – each process has to store the the results of the most previous relaxation to compare to – and

[1] MPI_Scatterv: $(\frac{5000^2}{40} + (2 \times 5000)) \times 40 \times 8 \approx 0.2$ gigabytes
[2] MPI_Bcast: $5000^2 \times 40 \times 8 \approx 8$ gigabytes

there is additional relatively non-fast, albeit limited in size, message passing. Because the root node's precision check exits as soon as the first cell is discovered outside the precision threshold, I felt that the benefits of taking a simple approach outweighed the complexities (including code complexity) of a distributed precision checking method.

Nevertheless, it is worth considering that as the program progresses, the matrix converges and cells outside the threshold aren't as likely to be discovered straight away with my approach. During this period of the program's running time, a distributed approach would be beneficial as the problem size scales, although this was not explored within the constraints of this project. An alternative non-blocking solution using functions such as *MPI_Isend* and *MPI_Wait* could have been explored to investigate the effects on latency at this section. Another approach would be to use a further MPI function *MPI_Reduce* with a reduction operation such as *MPI_SUM* to quickly deduce the outcome (>0 or 0) with distributed precision checks across different processes. However, it is worth noting that the aggregated threshold outcome would still require broadcasting from the root node so each process can determine whether to continue relaxing or to call *MPI_Finalize*. A further approach would be for each process to calculate its own precision and then to use *MPI_ALLREDUCE* to communicate the results to each process in the communicator. Ultimately however, the benefits of exploring different precision techniques are relatively small in comparison to relaxation which is the expensive part of the program. For the scope of this project, the *MPI_Bcast* method was therefore chosen for simplicity when calculating precision.

A flow diagram representing the structure of the distributed program is included in Figure 1. The relax section can be best described by the features in Figure 2, where each process works on its own section in parallel to others. The distributed approach reported so far has aimed to minimise the time spent between relaxing in parallel, so that the rewards of parallelisation can be achieved through these supersteps.

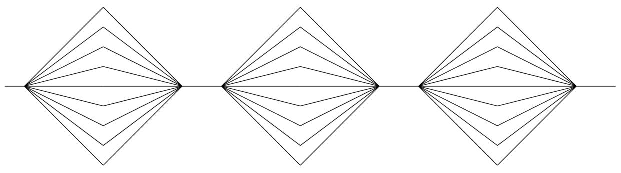


Figure 2: Superstep Diagram

3 Correctness Testing

Matrix relaxation has been completed by hand, as transcribed below. A simple 5×5 matrix is used for illustration purposes (note: the averaging operation of adjacent cells has not been detailed). The matrix is allocated and envisioned in this matrix as a one dimensional array. For instance for a 5×5 matrix: a cell with index 0 refers to the first element in the first row, a cell with index 5 refers to the first element in the second row, and so forth.

Using a precision of 0.2 and 3 processes, we take four full iterations to arrive at the final relaxed matrix. Each process has had its communication details for scattering and gathering calculated using the distributed algorithm as follows:

- **Process 0:** Scatter from cell 0 (*displacement*) for 15 cells (*send count*). Gather from cell 5 (*gather displacement*) for 5 cells (*receive count*).
- **Process 1:** Scatter from cell 5 for 15 cells. Gather from cell 10 for 5 cells.
- **Process 2:** Scatter from cell 10 for 15 cells. Gather from cell 15 for 5 cells.

1.00	1.00	1.00	1.00	1.00
1.00	0.00	0.00	0.00	1.00
1.00	0.00	0.00	0.00	1.00
1.00	0.00	0.00	0.00	1.00
1.00	1.00	1.00	1.00	1.00

The matrix has been initialised with a fixed non-random pattern. The working that follows shows the section which each process receives from the matrix (as detailed above) in the left column. The outcome of relaxing the inner cells for each section is displayed in the right column. These relaxed sections are then used to reconstruct the matrix for future iterations if the precision has not been reached.

Iteration 1

$$\textbf{Process 0:}\ \begin{array}{ccccc} 1.00 & 1.00 & 1.00 & 1.00 & 1.00 \\ 1.00 & 0.00 & 0.00 & 0.00 & 1.00 \\ 1.00 & 0.00 & 0.00 & 0.00 & 1.00 \end{array} \qquad \longrightarrow 1.00 \quad 0.50 \quad 0.25 \quad 0.50 \quad 1.00$$

$$\textbf{Process 1:}\ \begin{array}{ccccc} 1.00 & 0.00 & 0.00 & 0.00 & 1.00 \\ 1.00 & 0.00 & 0.00 & 0.00 & 1.00 \\ 1.00 & 0.00 & 0.00 & 0.00 & 1.00 \end{array} \qquad \longrightarrow 1.00 \quad 0.25 \quad 0.00 \quad 0.25 \quad 1.00$$

$$\textbf{Process 2:}\ \begin{array}{ccccc} 1.00 & 0.00 & 0.00 & 0.00 & 1.00 \\ 1.00 & 0.00 & 0.00 & 0.00 & 1.00 \\ 1.00 & 1.00 & 1.00 & 1.00 & 1.00 \end{array} \qquad \longrightarrow 1.00 \quad 0.50 \quad 0.25 \quad 0.50 \quad 1.00$$

1.00	1.00	1.00	1.00	1.00
1.00	0.50	0.25	0.50	1.00
1.00	0.25	0.00	0.25	1.00
1.00	0.50	0.25	0.50	1.00
1.00	1.00	1.00	1.00	1.00

The relaxed sections from all processes are gathered to reconstruct the new matrix. The precision check fails on the first inner cell, cell 6, which is outside of the precision threshold. Therefore, the matrix is scattered out to all processes to be relaxed again.

Iteration 2:

$$\textbf{Process 0:}\ \begin{array}{ccccc} 1.00 & 1.00 & 1.00 & 1.00 & 1.00 \\ 1.00 & 0.50 & 0.25 & 0.50 & 1.00 \\ 1.00 & 0.25 & 0.00 & 0.25 & 1.00 \end{array} \qquad \longrightarrow 1.00 \quad 0.625 \quad 0.500 \quad 0.625 \quad 1.00$$

$$\textbf{Process 1:}\ \begin{array}{ccccc} 1.00 & 0.50 & 0.25 & 0.50 & 1.00 \\ 1.00 & 0.25 & 0.00 & 0.25 & 1.00 \\ 1.00 & 0.50 & 0.25 & 0.50 & 1.00 \end{array} \qquad \longrightarrow 1.00 \quad 0.500 \quad 0.250 \quad 0.500 \quad 1.00$$

$$\textbf{Process 2:}\ \begin{array}{ccccc} 1.00 & 0.25 & 0.00 & 0.25 & 1.00 \\ 1.00 & 0.50 & 0.25 & 0.50 & 1.00 \\ 1.00 & 1.00 & 1.00 & 1.00 & 1.00 \end{array} \qquad \longrightarrow 1.00 \quad 0.625 \quad 0.500 \quad 0.625 \quad 1.00$$

$$\begin{array}{ccccc}
1.00 & 1.00 & 1.00 & 1.00 & 1.00 \\
1.00 & 0.625 & 0.500 & 0.625 & 1.00 \\
1.00 & 0.500 & 0.250 & 0.500 & 1.00 \\
1.00 & 0.625 & 0.500 & 0.625 & 1.00 \\
1.00 & 1.00 & 1.00 & 1.00 & 1.00
\end{array}$$

The relaxed sections from all processes are gathered to reconstruct the new matrix. The precision check fails on the second inner cell, cell 7, which is outside of the precision threshold. Therefore, the matrix is scattered out to all processes to be relaxed again.

Iteration 3:

$$\textbf{Process 0:}\ \begin{array}{ccccc}
1.00 & 1.00 & 1.00 & 1.00 & 1.00 \\
1.00 & 0.625 & 0.500 & 0.625 & 1.00 \\
1.00 & 0.500 & 0.250 & 0.500 & 1.00
\end{array} \longrightarrow 1.00 \quad 0.75 \quad 0.625 \quad 0.75 \quad 1.00$$

$$\textbf{Process 1:}\ \begin{array}{ccccc}
1.00 & 0.625 & 0.500 & 0.625 & 1.00 \\
1.00 & 0.500 & 0.250 & 0.500 & 1.00 \\
1.00 & 0.625 & 0.500 & 0.625 & 1.00
\end{array} \longrightarrow 1.00 \quad 0.625 \quad 0.50 \quad 0.625 \quad 1.00$$

$$\textbf{Process 2:}\ \begin{array}{ccccc}
1.00 & 0.500 & 0.250 & 0.500 & 1.00 \\
1.00 & 0.625 & 0.500 & 0.625 & 1.00 \\
1.00 & 1.00 & 1.00 & 1.00 & 1.00
\end{array} \longrightarrow 1.00 \quad 0.75 \quad 0.625 \quad 0.75 \quad 1.00$$

$$\begin{array}{ccccc}
1.00 & 1.00 & 1.00 & 1.00 & 1.00 \\
1.00 & 0.75 & 0.625 & 0.75 & 1.00 \\
1.00 & 0.625 & 0.50 & 0.625 & 1.00 \\
1.00 & 0.75 & 0.625 & 0.75 & 1.00 \\
1.00 & 1.00 & 1.00 & 1.00 & 1.00
\end{array}$$

The relaxed sections from all processes are gathered to reconstruct the new matrix. The precision check fails on the fifth inner cell, cell 12, which is outside of the precision threshold. Therefore, the matrix is scattered out to all processes to be relaxed again.

Iteration 4:

$$\textbf{Process 0:}\ \begin{array}{ccccc}
1.00 & 1.00 & 1.00 & 1.00 & 1.00 \\
1.00 & 0.75 & 0.625 & 0.75 & 1.00 \\
1.00 & 0.625 & 0.50 & 0.625 & 1.00
\end{array} \longrightarrow 1.00 \quad 0.8125 \quad 0.75 \quad 0.8125 \quad 1.00$$

$$\textbf{Process 1:}\ \begin{array}{ccccc}
1.00 & 0.75 & 0.625 & 0.75 & 1.00 \\
1.00 & 0.625 & 0.50 & 0.625 & 1.00 \\
1.00 & 0.75 & 0.625 & 0.75 & 1.00
\end{array} \longrightarrow 1.00 \quad 0.75 \quad 0.625 \quad 0.75 \quad 1.00$$

$$\textbf{Process 2:}\ \begin{array}{ccccc}
1.00 & 0.625 & 0.50 & 0.625 & 1.00 \\
1.00 & 0.75 & 0.625 & 0.75 & 1.00 \\
1.00 & 1.00 & 1.00 & 1.00 & 1.00
\end{array} \longrightarrow 1.00 \quad 0.8125 \quad 0.75 \quad 0.8125 \quad 1.00$$

1.00	1.00	1.00	1.00	1.00
1.00	0.8125	0.75	0.8125	1.00
1.00	0.75	0.625	0.75	1.00
1.00	0.8125	0.75	0.8125	1.00
1.00	1.00	1.00	1.00	1.00

The relaxed sections from all processes are gathered to reconstruct the new matrix. The precision check completes successfully and all cells fall within the precision threshold. At this point, the program would finalize all processes and terminate.

The results of relaxing this matrix by hand were compared with the results of the distributed program. With the same configuration and input matrix, the final relaxed matrix and number of iterations taken to reach this solution match between the program and manual, hand-worked version.

```
– Array A –
1.000000 1.000000 1.000000 1.000000 1.000000
1.000000 0.812500 0.750000 0.812500 1.000000
1.000000 0.750000 0.625000 0.750000 1.000000
1.000000 0.812500 0.750000 0.812500 1.000000
1.000000 1.000000 1.000000 1.000000 1.000000

Finished in 4 iterations
```

3.1 Further Test Cases

Reusing the small test framework from the shared memory coursework, I ran a series of repeated tests across different program configurations. As noted in the previous coursework's document, this does not ensure correctness – instead merely a suggestion of reliability in the program producing consistent outputs, regardless of the configuration used. Alongside the manual relaxation testing by hand however, I was comfortable that the distributed program performed to a degree which was suitable for the scope of this project. The results from the test framework are shown below. (M_1 is included in the Appendix - Figure 14)

Size	**Precision**	**Processes**	**Expected Output**	**Number of Runs**	**Pass**
10	0.01	1	M_1	20	Yes
10	0.01	2	M_1	20	Yes
10	0.01	5	M_1	20	Yes
100	0.01	1	M_2	20	Yes
100	0.01	5	M_2	20	Yes
100	0.01	15	M_2	20	Yes
100	0.01	30	M_2	20	Yes
100	0.01	60	M_2	20	Yes
500	0.01	1	M_3	20	Yes
500	0.01	5	M_3	20	Yes
500	0.01	15	M_3	20	Yes
500	0.01	30	M_3	20	Yes
500	0.01	60	M_3	20	Yes
1000	0.01	1	M_4	20	Yes
1000	0.01	5	M_4	20	Yes
1000	0.01	15	M_4	20	Yes
1000	0.01	30	M_4	20	Yes
1000	0.01	60	M_4	20	Yes

4 Scalability Investigation

The distributed parallel program was tested using Balena through a collection of varying program configurations. The unix *time* command (with the *real* output) was used to measure the total time of computations. Matrix dimension, precision and the number of processes were varied to discover the effects on speedup and efficiency.

As in the shared memory coursework, speedup and efficiency measures were calculated using the best possible sequential algorithm – in this scenario, a parallel version of the shared memory program using one thread. Given that I was following the master-slave paradigm with this comparative program, there are in fact two threads for both the master and slave potentially running simultaneously. Neither completes any substantial computations while the other is running but there is an overhead of creating and managing threads which needs to be considered. For this reason, it can be argued that the model sequential program isn't a true comparison because the program is not entirely sequential. Indeed, in hindsight the results may have more meaningful if the results were compared on the same architecture – e.g comparing the MPI program against itself with 1 process, rather than against a near-sequential threaded version. Nevertheless, the near-sequential parallel algorithm was used as it provided a common baseline for which both shared memory and distributed architectures could be scaled and compared against.

The first scalability investigation takes a fixed problem size and successively adds more processes to see the effects on time taken to fully relax the matrix.

4.1 Time taken for 1-64 processes, S=10000, P=0.01

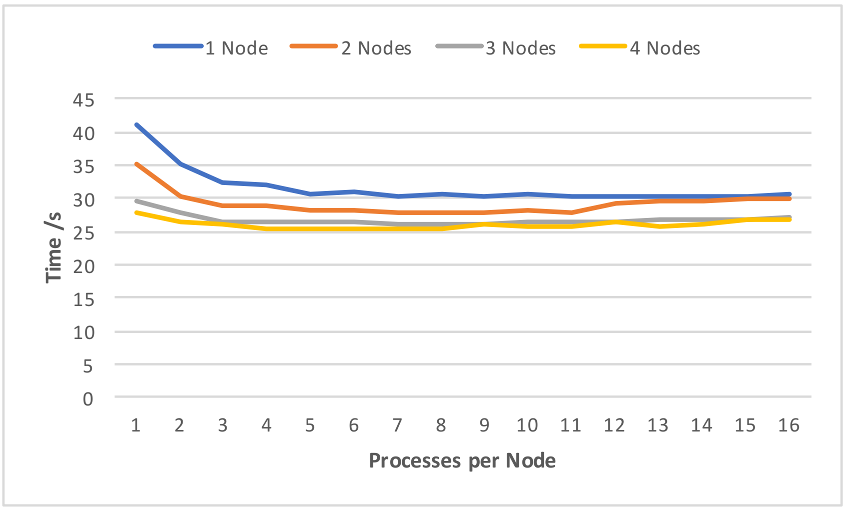


Figure 3: Time taken for $S = 10000, P = 0.01$ and $Processes = 1 - 64$

The law of diminishing returns applies in Figure 3 where successive processes results in smaller returns in time reduction. When the program is split across nodes, the time taken decreases due to additional processes running on each node – for instance, 41 seconds for 1 node with 1 process, versus 28 seconds for 4 nodes with 1 process each (for a total of 4 processes). The graph shows that ~5-13 processes on each node is very similar in terms of time taken, with additional processes above 13 on each node adding overhead and beginning to add small increases in the total time taken. However, most generally Figure 3 is consistent with Amdahl's Law in the sense that successive processes reach a natural limit on a fixed problem size.

4.2 Speedup for 1-64 processes, S=10000, P=0.01

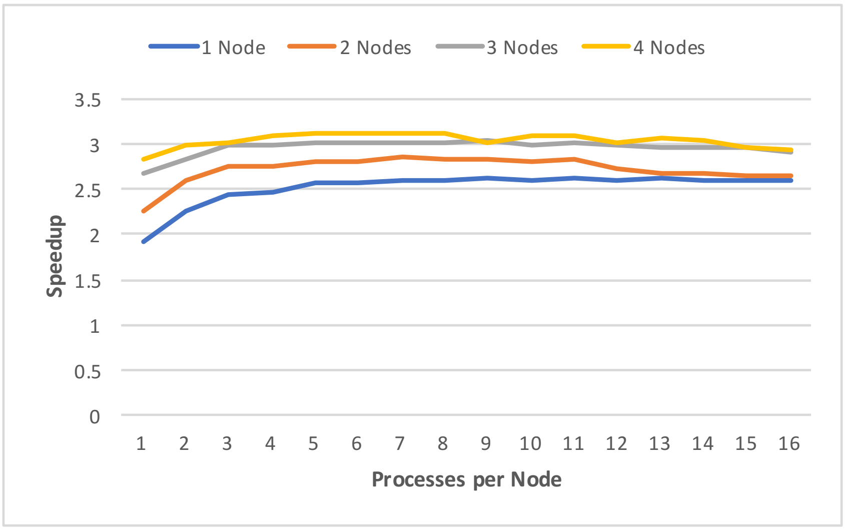


Figure 4: Speedup for $S = 10000, P = 0.01$ and $Processes = 1 - 64$

Figure 4 defines the program approaching this speedup limit through a weak asymptotic curve. Successive processes result in very little speedup, which will be reflected in poor efficiency results in Figure 5. Unlike the near-linear speedup achieved in the shared memory model, the speedup gained on this fixed problem size is fairly limited. The maximum speedup gained was 3.12, implying that the sequential portion of my program is approximately $1/3.12 \approx 0.32$ - 32% in the context of this problem size. Clearly this is substantial, and as Amdahl Law describes, this is representative of the upper limit on parallelisation for this fixed problem.

4.3 Efficiency for 1-64 processes, S=10000, P=0.01

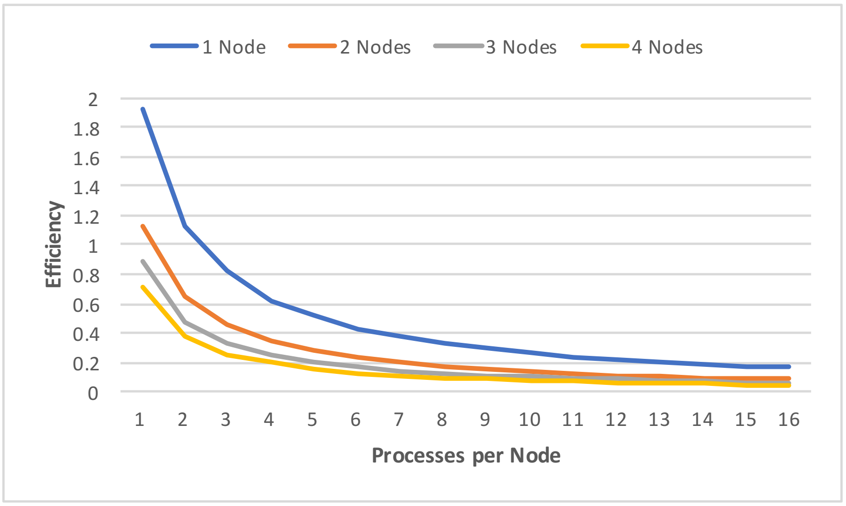


Figure 5: Efficiency for $S = 10000, P = 0.01$ and $Processes = 1 - 64$

The efficiency of this program with a fixed problem size is poor and is close to exponential decreases as additional processes are added. However, it is worth raising the caveat of the *best possible sequential algorithm* once more. With 1-2 processes, the distributed program performs significantly better in terms of efficiency than the chosen sequential algorithm, before rapidly decreasing. Given that in this context we may not be comparing against the most suitable sequential algorithm, the almost double gain in efficiency which the graph illustrates is perhaps a little misleading. Nevertheless, the meaningful trend of a deteriorating efficiency as processing power scales with a fixed problem size is apparent in Figure 5, consistent with Amdahl's law describing the limit on speedup, subsequently reflected through these efficiency measures.

4.4 Precision against time for S=10000, P=[0.01-0.001], Processes=[1,10,20,30,40,64]

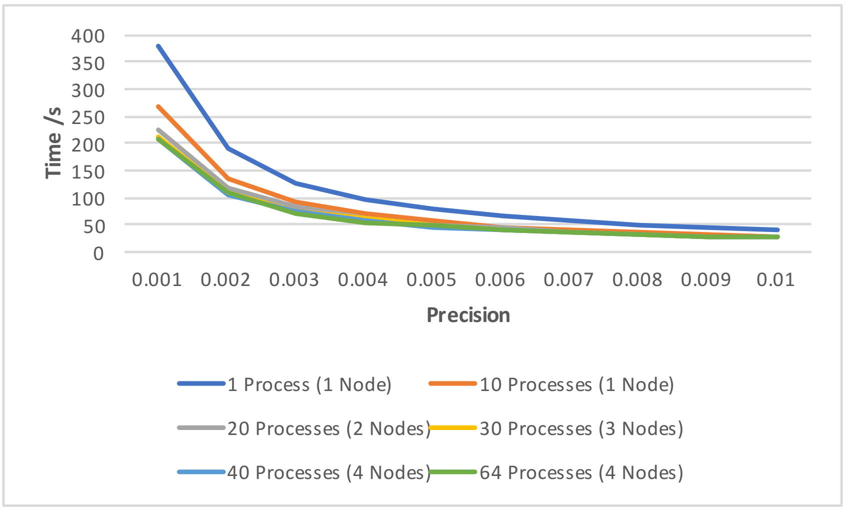


Figure 6: Time taken for $S = 10000$, $P = [0.01 - 0.001]$ and $Processes = [1, 10, 20, 30, 40, 64]$

Figures 6 and 9 demonstrates that with a scaling problem size (via the means of precision and dimension respectively), additional processing power can be used to reach the final solution in a faster time. For example, the time taken for 1 process vs the time taken for 10 processes on the example problem with 0.001 precision in Figure 6. Furthermore, as the problem size decreases, the gaps between different processing power and the time taken contracts. This corresponds with Gustafson's Law, where the problem size is increased resulting in a smaller sequential portion of the program, enabling the benefits of parallelisation.

4.5 Speedup for S=10000, P=[0.01-0.001], Processes=[1,10,20,30,40,64]

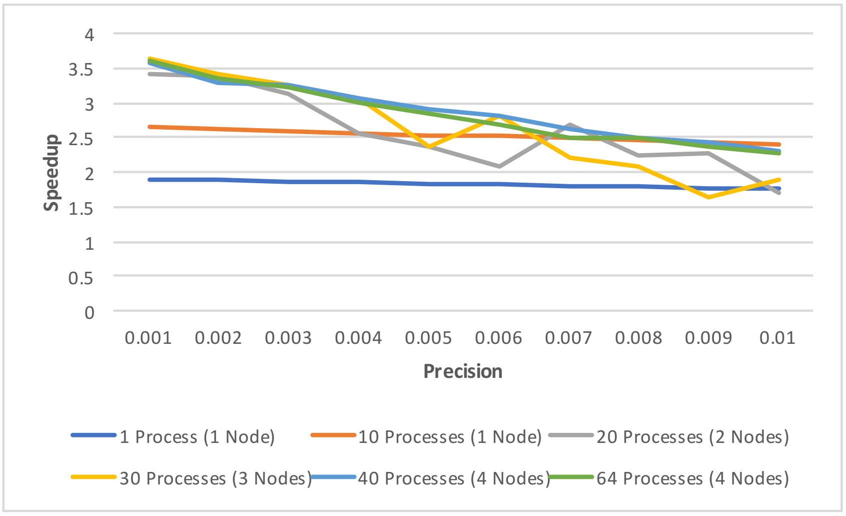


Figure 7: Speedup for $S = 10000, P = [0.01 - 0.001]$ and $Processes = [1, 10, 20, 30, 40, 64]$

The speedup gained for 1 node is largely consistent across all precisions and problem sizes. However, when more than 1 node is used the fluctuations apparent in Figure 7 begin to appear. When testing on Balena, jobs can be assigned to any node of the designated teaching set. For example, with 20 processes on 2 nodes, the 0.006 precision test used nodes 96 and 112, whereas the 0.007 precision test used nodes 103 and 163. While it is not clear whether the difference between numbers relates to the physical distance between nodes – this could be purely arbitrary on Balena – using different nodes between testing may well have caused the lack of stability in results. More specifically, this may be due to *asymmetry*. Nodes in proximity of each other can mean less network communication overheads when communicating and synchronising data between nodes. Albeit a minor factor, but certainly relevant if we are comparing between two chips on the same motherboard versus two chips on separate motherboards. In terms of the general trends, Figure 7 shows that for all nodes, as the precision/problem size increases, so does the speedup. Increased iterations finding the final solution means that a greater proportion of the program is spent running in parallel, consistent with Gustafson's law.

As a result, Figure 8 shows that increasing the problem size results in a greater efficiency – which is nevertheless a rather minor increase in efficiency across all ranges of processing power. The efficiency for the distributed program is generally rather low across all forms of testing, almost certainly due to the additional message passing overhead which distributed systems have. Indeed, as more processes are added, more messages will be generated (for scatter, gather and broadcasting). The costs of passing these messages across the network, and the costs of processes effectively being blocked at *MPI_Bcast* waiting for all calls of the function to return once all receive buffers are full, results in a poor efficiency (relative to the shared memory model).

4.6 Efficiency for S=10000, P=[0.01-0.001], Processes=[1,10,20,30,40,64]

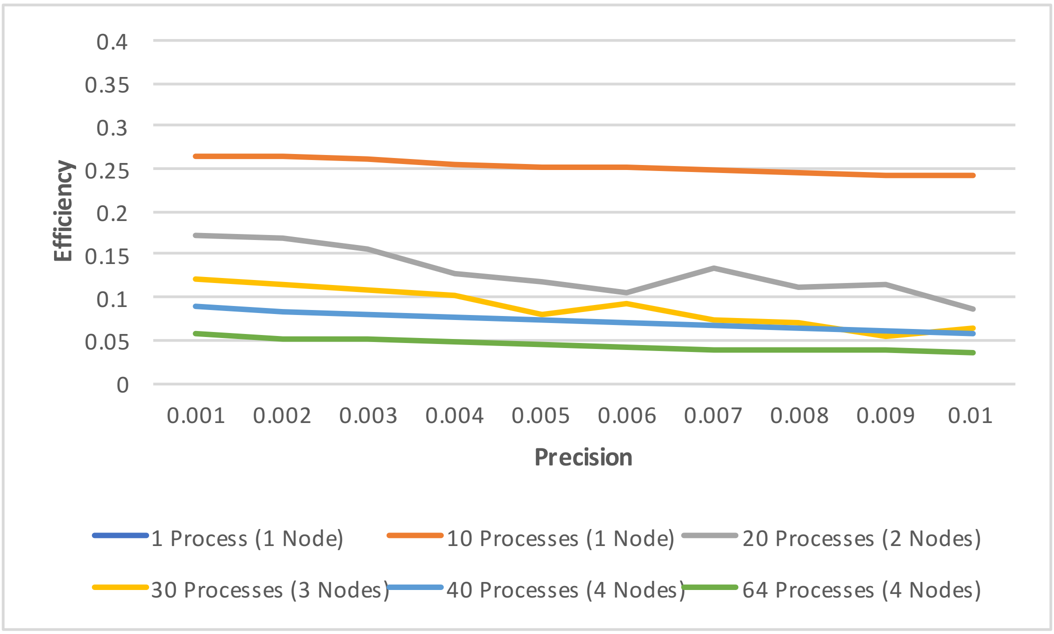


Figure 8: Efficiency for $S = 10000, P = [0.01 - 0.001]$ and $Processes = [1, 10, 20, 30, 40, 64]$

4.7 Dimension against time for S=[500-25000], P=0.01, Processes=[1,2,3,4,5]

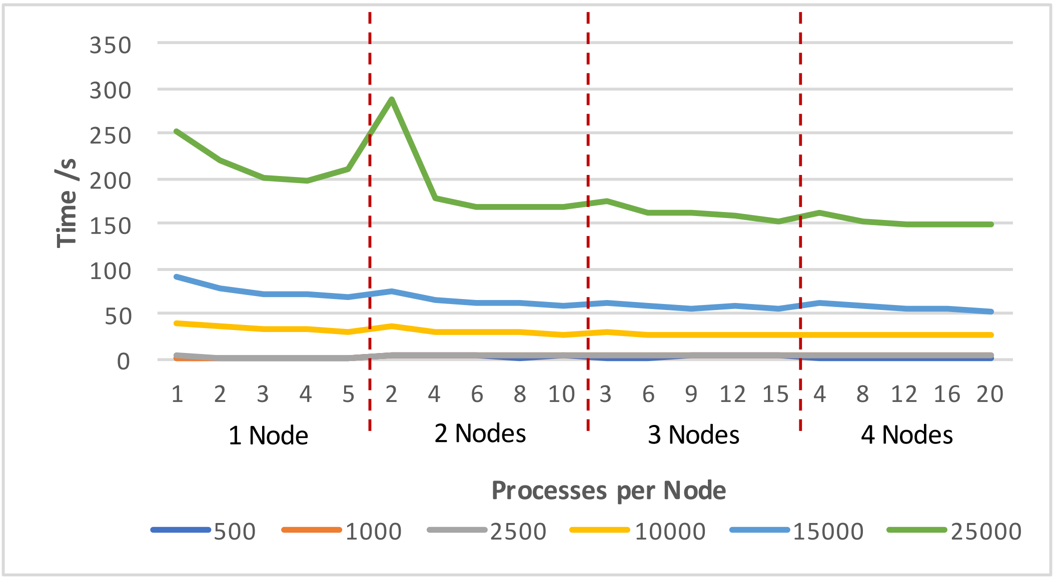


Figure 9: Time taken for $S = [500, 1000, 2500, 10000, 15000, 25000], P = [0.01 - 0.001]$ and $Processes = [1, 2, 3, 4, 5]$

4.8 Speedup for all dimensions and processes

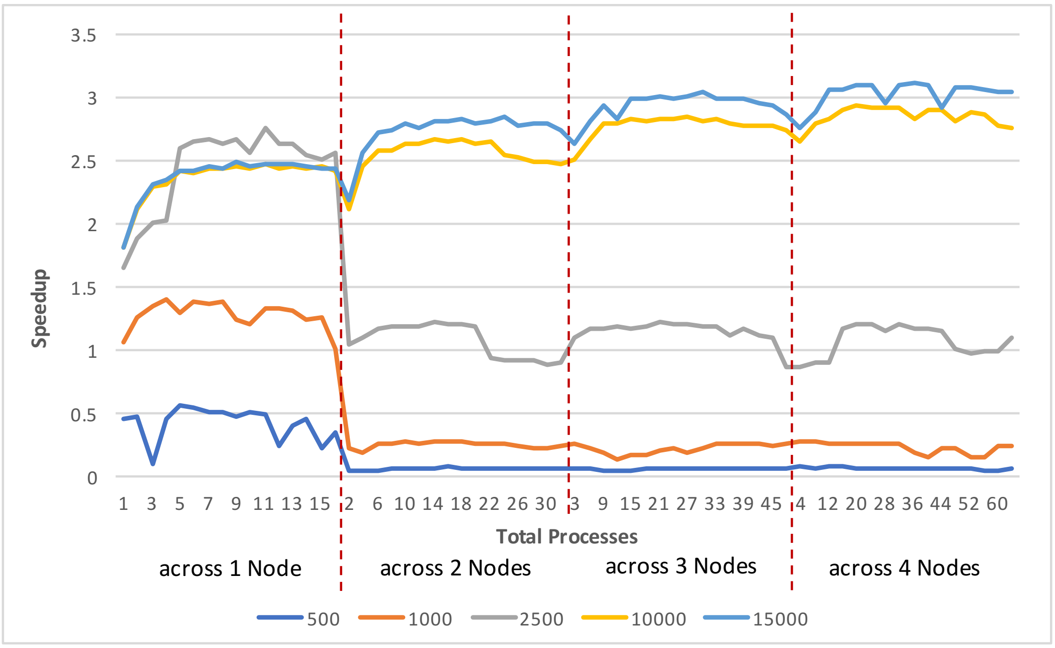


Figure 10: Speedup for $S = [500, 1000, 2500, 10000, 15000], P = [0.01]$ and $Processes = [1 - 64]$

4.9 Efficiency for all dimensions and processes

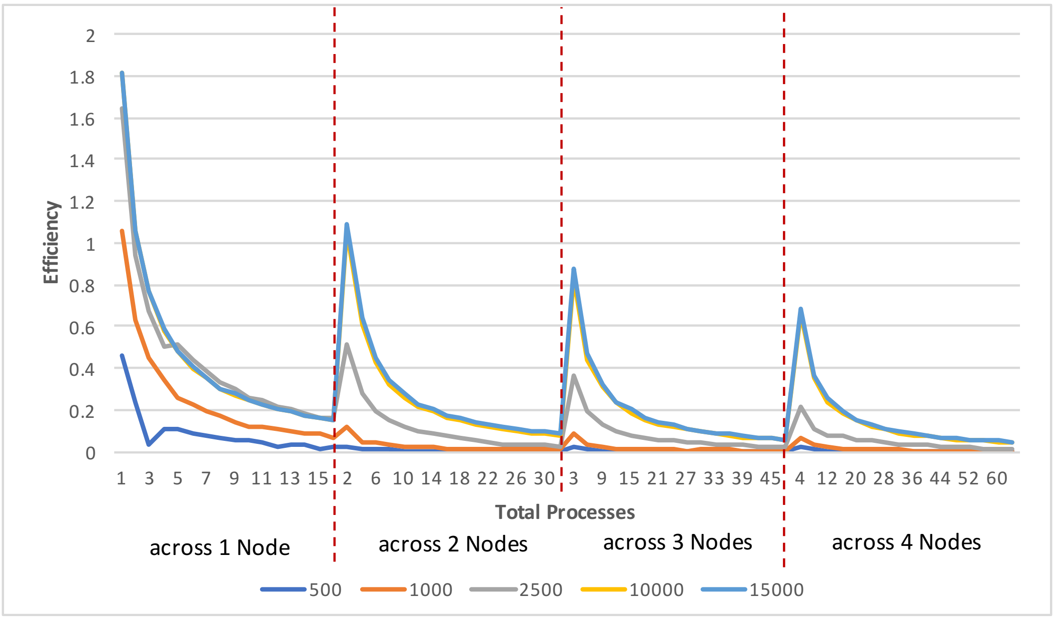


Figure 11: Efficiency for $S = [500, 1000, 2500, 10000, 15000], P = [0.01]$ and $Processes = [1 - 64]$

Metrics around speedup and efficiency can be used to find an ideal ratio between processing power and the problem sizes to be dealt with, especially when considering the best economical costs for resources. In the context of this project, extensive testing was completed with various matrix sizes across a wide range of processes to investigate the scalability of different configurations. The overall speedup results can be seen in Figure 10 and the overall efficiency results in Figure 11. Coherent with Amdahl's Law, realistically large matrices approach an upper limit on speedup in Figure 10. Furthermore, corresponding to Gustafson's Law, Figure 11 shows efficiency improving as the problem size grows through increased matrix dimensions. Speedup was greatest when working on large datasets where the costs of network message passing were alleviated by increasing amounts of parallelisation. Consequently, the most efficient computations were also on large problem sizes, where parallelisation of the problem played the largest part.

5 Running the Program

The distributed program utilises GNU's getopt library [3] enabling command-line arguments to configure the running of the program. mpicc is used to compile the program:

```
mpicc -Wall -std=gnu99 mpi_parallel.c -o mpi_parallel
```

The compiled program is then executed using mpirun with multiple command line arguments:

```
mpirun -np 3 mpi_parallel -S 10 -P 0.01 -V
```

The above command runs a distributed parallel program with 3 processes, on a matrix of size $10 \cdot 10$, with a precision of 0.01 and in verbose mode.

The list of command line arguments is below. Please note that *np* is an Open MPI argument used with mpirun.

Argument	Specification	Suggested Range	Default (if not provided)
np	**Number of Processes**	1-64	Open MPI defaults to 1 process
P	**Precision**	0.01-0.2	0.01
S	**Matrix Dimension**	6-2500	10
V	**Verbosity**	n/a	Turned off by default

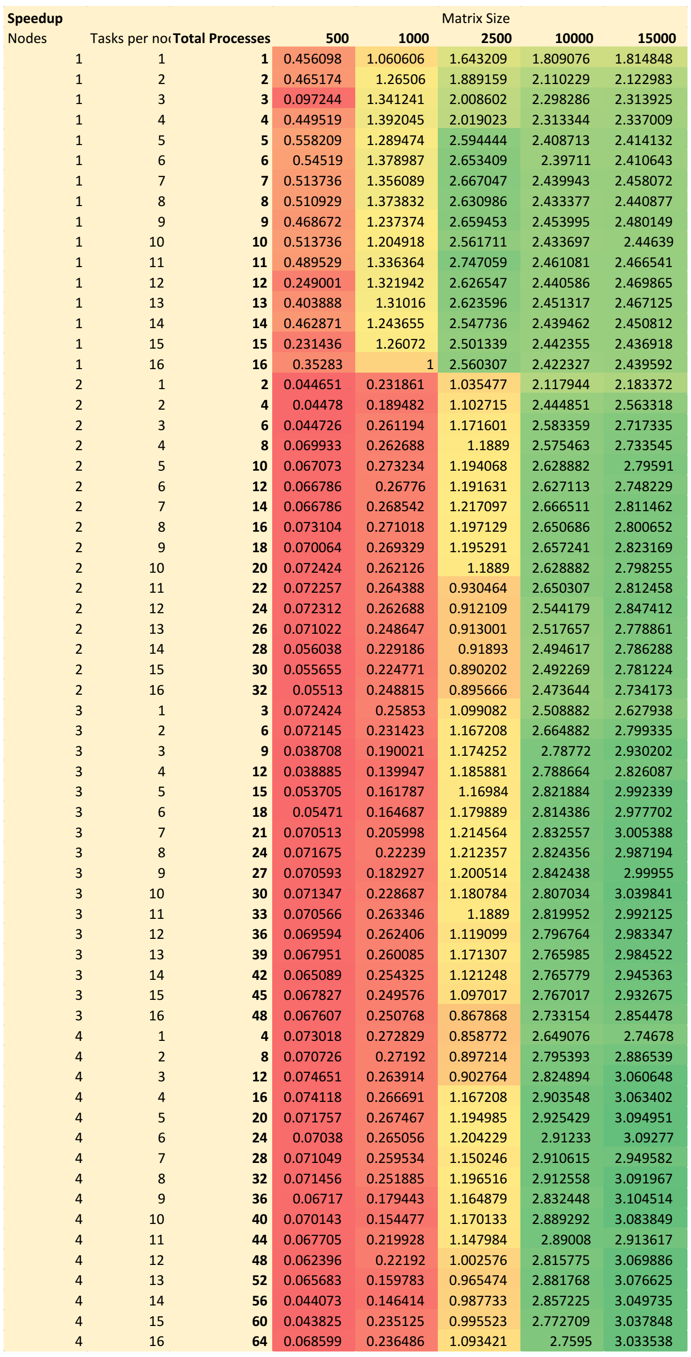
Further examples:

```
mpirun -np 3 mpi_parallel -S 100
mpirun -np 5 mpi_parallel -P 0.01 -V
mpirun -np 2 mpi_parallel
mpirun -np 42 mpi_parallel -S 2500 -P 0.01
```

Full validation has not been completed for these command line arguments – e.g the program behaviour for more processes than dimensions is not specified. Additionally, there are not checks with regards to the argument data type. If an argument is not passed, the default value specified above will be used in its place.

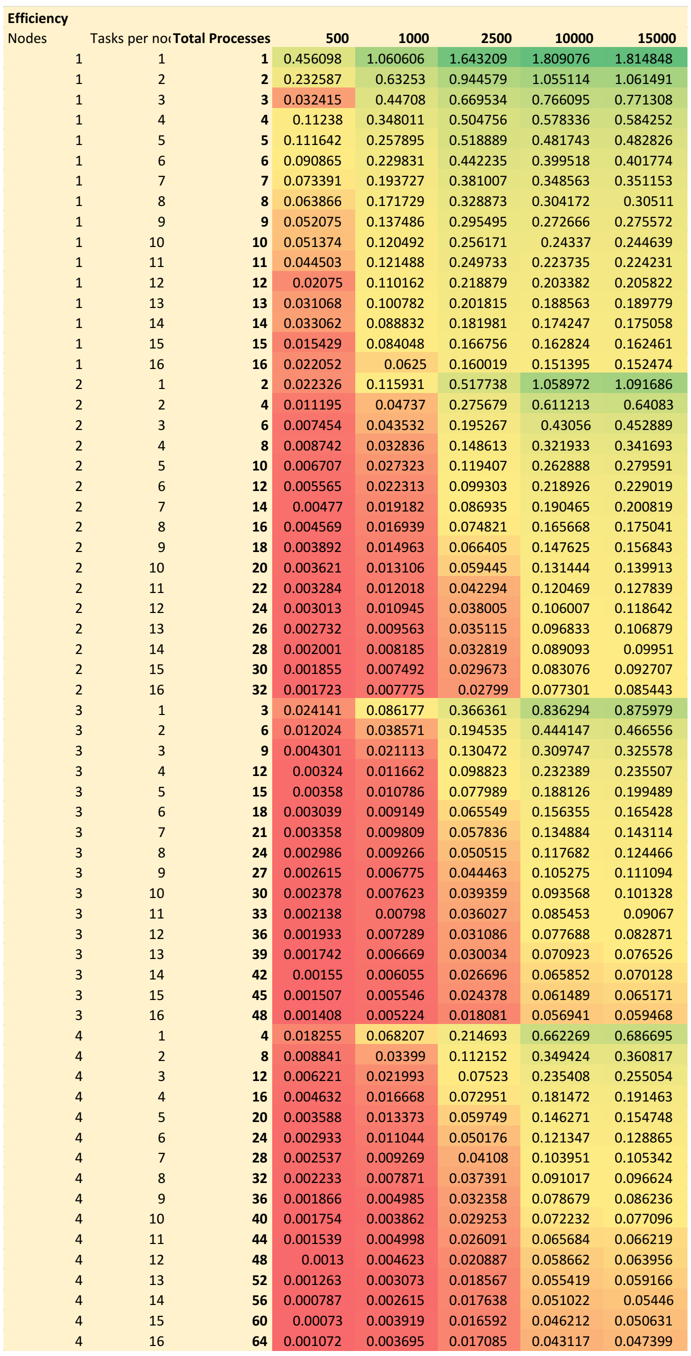
[3] https://www.gnu.org/software/libc/manual/html node/Getopt.html

6 Appendix



Speedup					Matrix Size		
Nodes	Tasks per no	**Total Processes**	**500**	**1000**	**2500**	**10000**	**15000**
1	1	**1**	0.456098	1.060606	1.643209	1.809076	1.814848
1	2	**2**	0.465174	1.26506	1.889159	2.110229	2.122983
1	3	**3**	0.097244	1.341241	2.008602	2.298286	2.313925
1	4	**4**	0.449519	1.392045	2.019023	2.313344	2.337009
1	5	**5**	0.558209	1.289474	2.594444	2.408713	2.414132
1	6	**6**	0.54519	1.378987	2.653409	2.39711	2.410643
1	7	**7**	0.513736	1.356089	2.667047	2.439943	2.458072
1	8	**8**	0.510929	1.373832	2.630986	2.433377	2.440877
1	9	**9**	0.468672	1.237374	2.659453	2.453995	2.480149
1	10	**10**	0.513736	1.204918	2.561711	2.433697	2.44639
1	11	**11**	0.489529	1.336364	2.747059	2.461081	2.466541
1	12	**12**	0.249001	1.321942	2.626547	2.440586	2.469865
1	13	**13**	0.403888	1.31016	2.623596	2.451317	2.467125
1	14	**14**	0.462871	1.243655	2.547736	2.439462	2.450812
1	15	**15**	0.231436	1.26072	2.501339	2.442355	2.436918
1	16	**16**	0.35283	1	2.560307	2.422327	2.439592
2	1	**2**	0.044651	0.231861	1.035477	2.117944	2.183372
2	2	**4**	0.04478	0.189482	1.102715	2.444851	2.563318
2	3	**6**	0.044726	0.261194	1.171601	2.583359	2.717335
2	4	**8**	0.069933	0.262688	1.1889	2.575463	2.733545
2	5	**10**	0.067073	0.273234	1.194068	2.628882	2.79591
2	6	**12**	0.066786	0.26776	1.191631	2.627113	2.748229
2	7	**14**	0.066786	0.268542	1.217097	2.666511	2.811462
2	8	**16**	0.073104	0.271018	1.197129	2.650686	2.800652
2	9	**18**	0.070064	0.269329	1.195291	2.657241	2.823169
2	10	**20**	0.072424	0.262126	1.1889	2.628882	2.798255
2	11	**22**	0.072257	0.264388	0.930464	2.650307	2.812458
2	12	**24**	0.072312	0.262688	0.912109	2.544179	2.847412
2	13	**26**	0.071022	0.248647	0.913001	2.517657	2.778861
2	14	**28**	0.056038	0.229186	0.91893	2.494617	2.786288
2	15	**30**	0.055655	0.224771	0.890202	2.492269	2.781224
2	16	**32**	0.05513	0.248815	0.895666	2.473644	2.734173
3	1	**3**	0.072424	0.25853	1.099082	2.508882	2.627938
3	2	**6**	0.072145	0.231423	1.167208	2.664882	2.799335
3	3	**9**	0.038708	0.190021	1.174252	2.78772	2.930202
3	4	**12**	0.038885	0.139947	1.185881	2.788664	2.826087
3	5	**15**	0.053705	0.161787	1.16984	2.821884	2.992339
3	6	**18**	0.05471	0.164687	1.179889	2.814386	2.977702
3	7	**21**	0.070513	0.205998	1.214564	2.832557	3.005388
3	8	**24**	0.071675	0.22239	1.212357	2.824356	2.987194
3	9	**27**	0.070593	0.182927	1.200514	2.842438	2.99955
3	10	**30**	0.071347	0.228687	1.180784	2.807034	3.039841
3	11	**33**	0.070566	0.263346	1.1889	2.819952	2.992125
3	12	**36**	0.069594	0.262406	1.119099	2.796764	2.983347
3	13	**39**	0.067951	0.260085	1.171307	2.765985	2.984522
3	14	**42**	0.065089	0.254325	1.121248	2.765779	2.945363
3	15	**45**	0.067827	0.249576	1.097017	2.767017	2.932675
3	16	**48**	0.067607	0.250768	0.867868	2.733154	2.854478
4	1	**4**	0.073018	0.272829	0.858772	2.649076	2.74678
4	2	**8**	0.070726	0.27192	0.897214	2.795393	2.886539
4	3	**12**	0.074651	0.263914	0.902764	2.824894	3.060648
4	4	**16**	0.074118	0.266691	1.167208	2.903548	3.063402
4	5	**20**	0.071757	0.267467	1.194985	2.925429	3.094951
4	6	**24**	0.07038	0.265056	1.204229	2.91233	3.09277
4	7	**28**	0.071049	0.259534	1.150246	2.910615	2.949582
4	8	**32**	0.071456	0.251885	1.196516	2.912558	3.091967
4	9	**36**	0.06717	0.179443	1.164879	2.832448	3.104514
4	10	**40**	0.070143	0.154477	1.170133	2.889292	3.083849
4	11	**44**	0.067705	0.219928	1.147984	2.89008	2.913617
4	12	**48**	0.062396	0.22192	1.002576	2.815775	3.069886
4	13	**52**	0.065683	0.159783	0.965474	2.881768	3.076625
4	14	**56**	0.044073	0.146414	0.987733	2.857225	3.049735
4	15	**60**	0.043825	0.235125	0.995523	2.772709	3.037848
4	16	**64**	0.068599	0.236486	1.093421	2.7595	3.033538

Figure 12: Speedup visualisation for $S = [500, 1000, 2500, 10000, 15000], P = [0.01]$ and $Processes = [1 - 64]$



Efficiency							
Nodes	Tasks per no	**Total Processes**	**500**	**1000**	**2500**	**10000**	**15000**
1	1	**1**	0.456098	1.060606	1.643209	1.809076	1.814848
1	2	**2**	0.232587	0.63253	0.944579	1.055114	1.061491
1	3	**3**	0.032415	0.44708	0.669534	0.766095	0.771308
1	4	**4**	0.11238	0.348011	0.504756	0.578336	0.584252
1	5	**5**	0.111642	0.257895	0.518889	0.481743	0.482826
1	6	**6**	0.090865	0.229831	0.442235	0.399518	0.401774
1	7	**7**	0.073391	0.193727	0.381007	0.348563	0.351153
1	8	**8**	0.063866	0.171729	0.328873	0.304172	0.30511
1	9	**9**	0.052075	0.137486	0.295495	0.272666	0.275572
1	10	**10**	0.051374	0.120492	0.256171	0.24337	0.244639
1	11	**11**	0.044503	0.121488	0.249733	0.223735	0.224231
1	12	**12**	0.02075	0.110162	0.218879	0.203382	0.205822
1	13	**13**	0.031068	0.100782	0.201815	0.188563	0.189779
1	14	**14**	0.033062	0.088832	0.181981	0.174247	0.175058
1	15	**15**	0.015429	0.084048	0.166756	0.162824	0.162461
1	16	**16**	0.022052	0.0625	0.160019	0.151395	0.152474
2	1	**2**	0.022326	0.115931	0.517738	1.058972	1.091686
2	2	**4**	0.011195	0.04737	0.275679	0.611213	0.64083
2	3	**6**	0.007454	0.043532	0.195267	0.43056	0.452889
2	4	**8**	0.008742	0.032836	0.148613	0.321933	0.341693
2	5	**10**	0.006707	0.027323	0.119407	0.262888	0.279591
2	6	**12**	0.005565	0.022313	0.099303	0.218926	0.229019
2	7	**14**	0.00477	0.019182	0.086935	0.190465	0.200819
2	8	**16**	0.004569	0.016939	0.074821	0.165668	0.175041
2	9	**18**	0.003892	0.014963	0.066405	0.147625	0.156843
2	10	**20**	0.003621	0.013106	0.059445	0.131444	0.139913
2	11	**22**	0.003284	0.012018	0.042294	0.120469	0.127839
2	12	**24**	0.003013	0.010945	0.038005	0.106007	0.118642
2	13	**26**	0.002732	0.009563	0.035115	0.096833	0.106879
2	14	**28**	0.002001	0.008185	0.032819	0.089093	0.09951
2	15	**30**	0.001855	0.007492	0.029673	0.083076	0.092707
2	16	**32**	0.001723	0.007775	0.02799	0.077301	0.085443
3	1	**3**	0.024141	0.086177	0.366361	0.836294	0.875979
3	2	**6**	0.012024	0.038571	0.194535	0.444147	0.466556
3	3	**9**	0.004301	0.021113	0.130472	0.309747	0.325578
3	4	**12**	0.00324	0.011662	0.098823	0.232389	0.235507
3	5	**15**	0.00358	0.010786	0.077989	0.188126	0.199489
3	6	**18**	0.003039	0.009149	0.065549	0.156355	0.165428
3	7	**21**	0.003358	0.009809	0.057836	0.134884	0.143114
3	8	**24**	0.002986	0.009266	0.050515	0.117682	0.124466
3	9	**27**	0.002615	0.006775	0.044463	0.105275	0.111094
3	10	**30**	0.002378	0.007623	0.039359	0.093568	0.101328
3	11	**33**	0.002138	0.00798	0.036027	0.085453	0.09067
3	12	**36**	0.001933	0.007289	0.031086	0.077688	0.082871
3	13	**39**	0.001742	0.006669	0.030034	0.070923	0.076526
3	14	**42**	0.00155	0.006055	0.026696	0.065852	0.070128
3	15	**45**	0.001507	0.005546	0.024378	0.061489	0.065171
3	16	**48**	0.001408	0.005224	0.018081	0.056941	0.059468
4	1	**4**	0.018255	0.068207	0.214693	0.662269	0.686695
4	2	**8**	0.008841	0.03399	0.112152	0.349424	0.360817
4	3	**12**	0.006221	0.021993	0.07523	0.235408	0.255054
4	4	**16**	0.004632	0.016668	0.072951	0.181472	0.191463
4	5	**20**	0.003588	0.013373	0.059749	0.146271	0.154748
4	6	**24**	0.002933	0.011044	0.050176	0.121347	0.128865
4	7	**28**	0.002537	0.009269	0.04108	0.103951	0.105342
4	8	**32**	0.002233	0.007871	0.037391	0.091017	0.096624
4	9	**36**	0.001866	0.004985	0.032358	0.078679	0.086236
4	10	**40**	0.001754	0.003862	0.029253	0.072232	0.077096
4	11	**44**	0.001539	0.004998	0.026091	0.065684	0.066219
4	12	**48**	0.0013	0.004623	0.020887	0.058662	0.063956
4	13	**52**	0.001263	0.003073	0.018567	0.055419	0.059166
4	14	**56**	0.000787	0.002615	0.017638	0.051022	0.05446
4	15	**60**	0.00073	0.003919	0.016592	0.046212	0.050631
4	16	**64**	0.001072	0.003695	0.017085	0.043117	0.047399

Figure 13: Efficiency visualisation for $S = [500, 1000, 2500, 10000, 15000], P = [0.01]$ and $Processes = [1 - 64]$

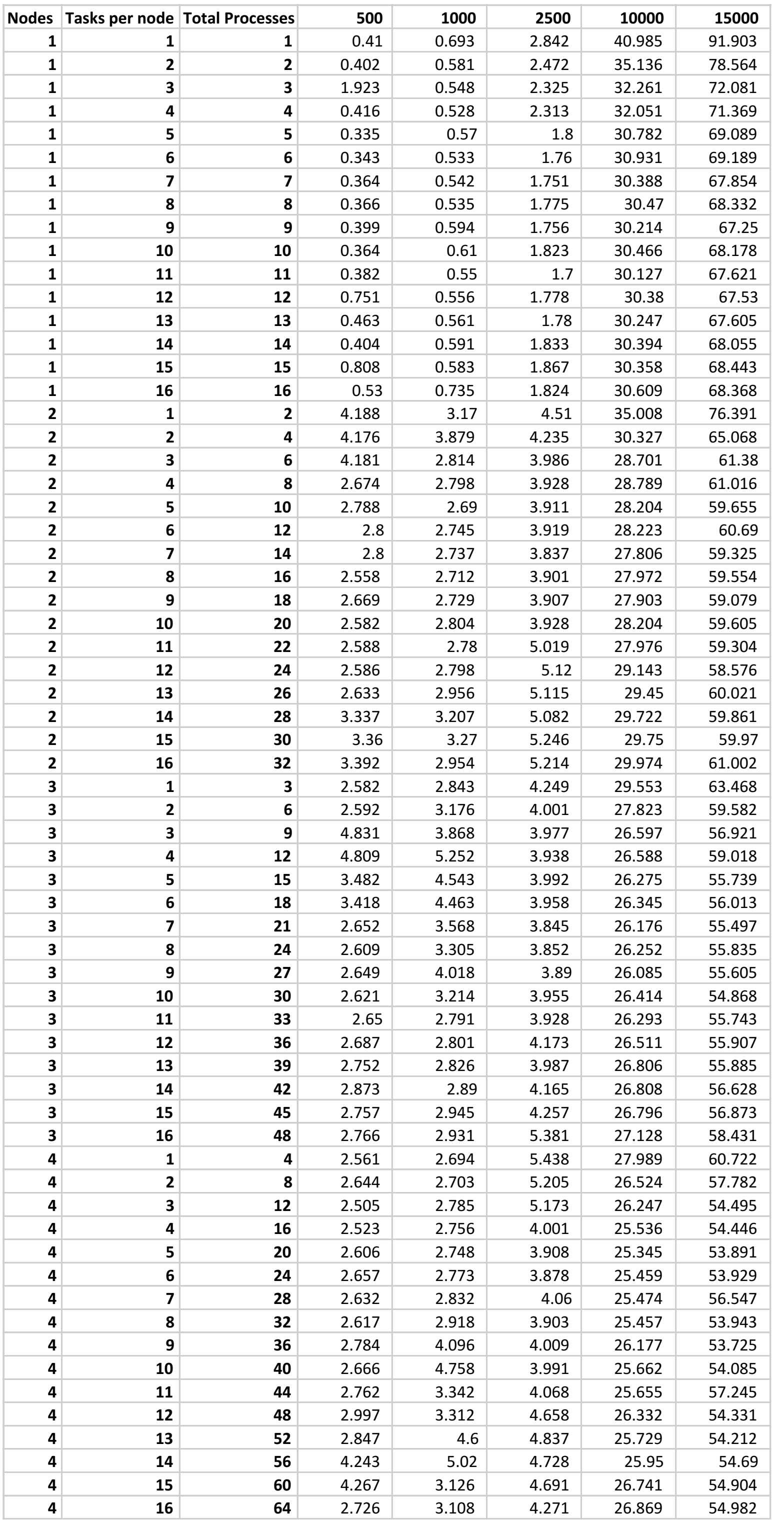
$$M_1 = \begin{matrix}
1.000000 & 1.000000 & 1.000000 & 1.000000 & 1.000000 & 1.000000 & 1.000000 & 1.000000 & 1.000000 & 1.000000 \\
1.000000 & 0.981399 & 0.965041 & 0.952901 & 0.946442 & 0.946442 & 0.952901 & 0.965041 & 0.981399 & 1.000000 \\
1.000000 & 0.965041 & 0.934300 & 0.911484 & 0.899345 & 0.899345 & 0.911484 & 0.934300 & 0.965041 & 1.000000 \\
1.000000 & 0.952901 & 0.911484 & 0.880746 & 0.864391 & 0.864391 & 0.880746 & 0.911484 & 0.952901 & 1.000000 \\
1.000000 & 0.946442 & 0.899345 & 0.864391 & 0.845793 & 0.845793 & 0.864391 & 0.899345 & 0.946442 & 1.000000 \\
1.000000 & 0.946442 & 0.899345 & 0.864391 & 0.845793 & 0.845793 & 0.864391 & 0.899345 & 0.946442 & 1.000000 \\
1.000000 & 0.952901 & 0.911484 & 0.880746 & 0.864391 & 0.864391 & 0.880746 & 0.911484 & 0.952901 & 1.000000 \\
1.000000 & 0.965041 & 0.934300 & 0.911484 & 0.899345 & 0.899345 & 0.911484 & 0.934300 & 0.965041 & 1.000000 \\
1.000000 & 0.981399 & 0.965041 & 0.952901 & 0.946442 & 0.946442 & 0.952901 & 0.965041 & 0.981399 & 1.000000 \\
1.000000 & 1.000000 & 1.000000 & 1.000000 & 1.000000 & 1.000000 & 1.000000 & 1.000000 & 1.000000 & 1.000000
\end{matrix}$$

Figure 14: Expected Output Matrix

6.1 Raw Data

	1 Node	**1 Node**	**2 Nodes**	**3 Nodes**	**4 Nodes**	**4 Nodes**
	1 Process	**10 Processes**	**20 Processes**	**30 Processes**	**40 Processes**	**64 Processes**
0.001	94.038	67.181	52.133	49.217	50.12	49.56
0.002	47.444	34.014	26.533	26.255	27.268	26.799
0.003	31.863	22.969	19.089	18.272	18.359	18.546
0.004	24.315	17.659	17.705	14.764	14.717	15.021
0.005	19.712	14.455	15.444	15.255	12.458	12.829
0.006	16.316	11.873	14.386	10.636	10.652	11.152
0.007	14.283	10.428	9.671	11.744	9.841	10.365
0.008	12.735	9.304	10.25	10.954	9.146	9.18
0.009	11.195	8.2	8.769	12.262	8.188	8.421
0.01	10.427	7.655	10.901	9.82	8.038	8.118

Figure 15: Raw Data for $S = 10000$, $P = [0.001 - 0.01]$ and $Processes = [1, 10, 20, 30, 40, 64]$



Nodes	Tasks per node	Total Processes	500	1000	2500	10000	15000
1	1	1	0.41	0.693	2.842	40.985	91.903
1	2	2	0.402	0.581	2.472	35.136	78.564
1	3	3	1.923	0.548	2.325	32.261	72.081
1	4	4	0.416	0.528	2.313	32.051	71.369
1	5	5	0.335	0.57	1.8	30.782	69.089
1	6	6	0.343	0.533	1.76	30.931	69.189
1	7	7	0.364	0.542	1.751	30.388	67.854
1	8	8	0.366	0.535	1.775	30.47	68.332
1	9	9	0.399	0.594	1.756	30.214	67.25
1	10	10	0.364	0.61	1.823	30.466	68.178
1	11	11	0.382	0.55	1.7	30.127	67.621
1	12	12	0.751	0.556	1.778	30.38	67.53
1	13	13	0.463	0.561	1.78	30.247	67.605
1	14	14	0.404	0.591	1.833	30.394	68.055
1	15	15	0.808	0.583	1.867	30.358	68.443
1	16	16	0.53	0.735	1.824	30.609	68.368
2	1	2	4.188	3.17	4.51	35.008	76.391
2	2	4	4.176	3.879	4.235	30.327	65.068
2	3	6	4.181	2.814	3.986	28.701	61.38
2	4	8	2.674	2.798	3.928	28.789	61.016
2	5	10	2.788	2.69	3.911	28.204	59.655
2	6	12	2.8	2.745	3.919	28.223	60.69
2	7	14	2.8	2.737	3.837	27.806	59.325
2	8	16	2.558	2.712	3.901	27.972	59.554
2	9	18	2.669	2.729	3.907	27.903	59.079
2	10	20	2.582	2.804	3.928	28.204	59.605
2	11	22	2.588	2.78	5.019	27.976	59.304
2	12	24	2.586	2.798	5.12	29.143	58.576
2	13	26	2.633	2.956	5.115	29.45	60.021
2	14	28	3.337	3.207	5.082	29.722	59.861
2	15	30	3.36	3.27	5.246	29.75	59.97
2	16	32	3.392	2.954	5.214	29.974	61.002
3	1	3	2.582	2.843	4.249	29.553	63.468
3	2	6	2.592	3.176	4.001	27.823	59.582
3	3	9	4.831	3.868	3.977	26.597	56.921
3	4	12	4.809	5.252	3.938	26.588	59.018
3	5	15	3.482	4.543	3.992	26.275	55.739
3	6	18	3.418	4.463	3.958	26.345	56.013
3	7	21	2.652	3.568	3.845	26.176	55.497
3	8	24	2.609	3.305	3.852	26.252	55.835
3	9	27	2.649	4.018	3.89	26.085	55.605
3	10	30	2.621	3.214	3.955	26.414	54.868
3	11	33	2.65	2.791	3.928	26.293	55.743
3	12	36	2.687	2.801	4.173	26.511	55.907
3	13	39	2.752	2.826	3.987	26.806	55.885
3	14	42	2.873	2.89	4.165	26.808	56.628
3	15	45	2.757	2.945	4.257	26.796	56.873
3	16	48	2.766	2.931	5.381	27.128	58.431
4	1	4	2.561	2.694	5.438	27.989	60.722
4	2	8	2.644	2.703	5.205	26.524	57.782
4	3	12	2.505	2.785	5.173	26.247	54.495
4	4	16	2.523	2.756	4.001	25.536	54.446
4	5	20	2.606	2.748	3.908	25.345	53.891
4	6	24	2.657	2.773	3.878	25.459	53.929
4	7	28	2.632	2.832	4.06	25.474	56.547
4	8	32	2.617	2.918	3.903	25.457	53.943
4	9	36	2.784	4.096	4.009	26.177	53.725
4	10	40	2.666	4.758	3.991	25.662	54.085
4	11	44	2.762	3.342	4.068	25.655	57.245
4	12	48	2.997	3.312	4.658	26.332	54.331
4	13	52	2.847	4.6	4.837	25.729	54.212
4	14	56	4.243	5.02	4.728	25.95	54.69
4	15	60	4.267	3.126	4.691	26.741	54.904
4	16	64	2.726	3.108	4.271	26.869	54.982

Figure 16: Raw Data for $S = [500, 1000, 2500, 10000, 15000], P = [0.01]$ and $Processes = [1 - 64]$