Distributed Computing Coursework

Alexander Dawkins

9 January 2022

Contents

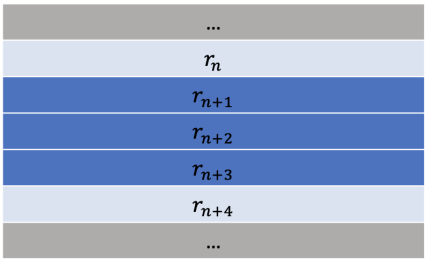

Figure 1: Diagram showing a series of rows r_n to r_{n+4}

1 Introduction

This is coursework for the Parallel Computing (CM30225) module at the University of Bath:

The task is to write a program using the MPI standard that will perform the "relaxation technique" to a matrix of numbers, by setting each cell to the average of its surrounding cells.

It will be run on a cluster hosted on Azure. This cluster consists of 8 nodes, with 42 usable cores on each. Nodes are connected by 100Gb InfiniBand networking.

2 Design of the Program

2.1 Overview of Solution

As the task is the same as the previous (shared memory) coursework, I had already settled on a design that I was happy with. For a board of size n^2 and with p processors, each processor was assigned $\lfloor \frac{n}{p} \rfloor$, and the first $n \mod p$ processors were assigned an extra row. In other words, the board's rows were split evenly amongst the processors. After each iteration of processing, each processor will send it's outermost rows with its neighbour's and receive its neighbour's outermost rows too.

As shown in figure 1, a processor with rank k assigned the processing of rows r_{n+1}, r_{n+2}, and r_{n+3}, it would send r_{n+1} to the processor working above it (rank $k-1$), and r_{n+3} to the processor working below it (rank $k+1$). It would in turn receive r_n from $k-1$, and r_{n+4} from $k+1$.

2.2 Communication, and avoiding race conditions

After each processor has generated the next generation for its assigned data, a *gather* occurs, where each processor's state (done or not done) is collected by the root node. The root node will then broadcast a signal indicating whether the processors should stop or continue. This check is sequential, and could be considered a bottleneck of the solution. Instead of all processors sending their state to the root processor, some checking could be done by all processors. An example could be that each evenly ranked processor n checks the processor with rank $n + 1$, and then submits the results to the root node. It is clear how this divide-and-conquer approach could then be extended further, where ranks that are multiples of 4 could check the next multiple of 4, and so on. I didn't implement this approach as it would over-complicate my code even further, and this current way still provided adequate results.

At the end of processing, the rows are all sent to the root processor.

As all processors are synchronised by the gather/broadcast procedure, we mitigate the possibility of a race condition.

2.2.1 Edge cases

For the processor working on the first and last rows, only one pair of rows will be swapped, as the edge rows are stationary.

2.3 Collecting results

After processing is complete, all rows are sent to the root processor using the gather MPI function. The gather operation will take n elements from each processor, and will populate a buffer of size $n \times p$ where p is the number of processors. As the program models the board as a 2D array, it was complicated to use the gather function. To solve this, I used a loop to re-shape the 2D array into a 1D array. This data could then be compiled with the gather procedure and re-shaped. The problem of variable workload sizing was solved by using buffers that were the maximum workload size and for unused values to be filled with NAN values.

Admittedly, this solution is complicated and took some time to debug, but it works. The code to reshape the array wouldn't be necessary if the board was stored as a 1D array to start with.

2.4 Reducing message receiving waiting times

To reduce the coupling between processors, in parts I used the MPI commands MPI_IRecv and MPI_Wait to allow the code to continue and

3 Correctness Testing

3.1 Manual Testing

I calculated the following 5x5 grid by hand, and compared it to the program's output. It provides a simple way to test that the program is getting the basics correct. While this may be considered too small, it brought attention to some problems in the program early in development.

The following values have been rounded to 2d.p.

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 0.5 & 0.25 & 0.25 & 0 \\ 1 & 0.25 & 0 & 0 & 0 \\ 1 & 0.25 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix} \rightarrow \dots$$

$$\dots \rightarrow \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 0.86 & 0.71 & 0.50 & 0 \\ 1 & 0.71 & 0.50 & 0.29 & 0 \\ 1 & 0.50 & 0.29 & 0.14 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

3.2 Example answers generated in Microsoft Excel

I also created an Excel sheet set up to generate large example output boards. While this doesn't guarantee the correct answers, it forced me to step through each iteration of the answer as I generated the next one, meaning that some mistakes were caught. These boards were compared to the output from the program and were identical.

3.3 Automated Testing

My code will also check every solution that it produces is precise - that the cells are all "equal" to the averages of the surrounding cells. If this isn't the case, information is provided about the location, difference, etc.

An excerpt of testing output can be found in the appendix

4 Results

4.1 The initial board layout

Each cell in the top row and left column was set to 1, and the rest of the board was set to 0. Throughout testing, a precision of $1e^{-3}$ was used as the limit of acceptable difference for two values to be considered equal.

We use the described layout as it is an intuitive problem, similar to that of the dissipation of heat. Moreover, we can do a quick validation on the output of the program, as there is a line of symmetry running from the top left to the

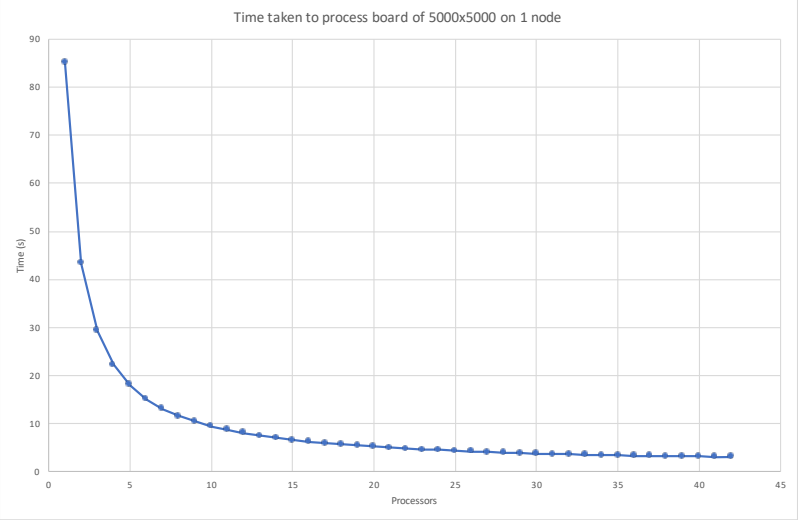

Figure 2: Graph showing the time taken to process board of 5000x5000 on 1 node

bottom right. We use the values 1 and 0 as they are "normal" values: we can easily scale/offset the results to change the initial values. For example, if we instead wanted the relaxation of the same shape but between 30 and 10, and can multiply each element by 20 and then add 10.

$$Board = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 0 & \cdots & 0 \end{bmatrix}$$

4.2 Time taken to process a board of size 5,000 on 1 node

Figure 2 shows that the time spent on processing decreases as we add more nodes; quickly at first, but then with diminishing returns. These results are similar to those achieved during the shared memory coursework, and assure us that this parallel solution works. The negative gradient quickly decreases, meaning that the efficiency of our work is decreasing as we add more processors: as we add more processors, the time saved per processor gets less and less.

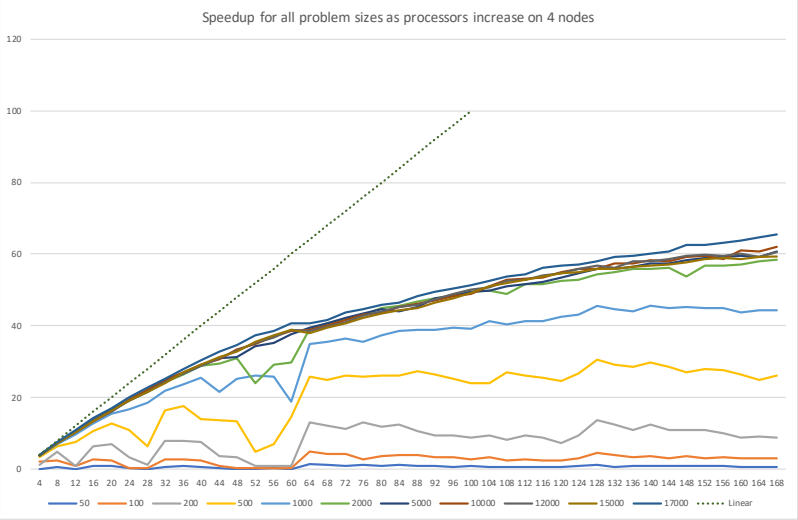

Figure 3: Graph showing the speedup for all problem sizes as processors increase.

5 Scalability

5.1 Speedup

All speedup values **are less than linear**, meaning that at no point did my code obtain super-linear speedup. This could suggest that my sequential code is a fair comparison for my parallel code. It could also suggest that cache wasn't used effectively. Perhaps I could have written the code to use less memory so that the problem could fit in each processor's cache.

Figure 3 shows that for 4 nodes, we do see some speedup amongst the larger problem sizes. As expected, the smaller problem sizes of 50 and 100 are simply too small for adding this much overhead to increase their performance.

Figure 4 shows only the smaller problems with an appropriate y-axis scale. The speedup is volatile and about constant.

Although the larger problems do see some close to linear speedup at smaller quantities of processors, a plateau is clear to see as we approach the right of the graph. My first thought was that maybe a larger problem size would make better use of the available resources, and it almost does: the largest problem size (17,000) *does* demonstrate the largest speedup. However, the fact that all problem sizes from 2,000 to 17,000 such similar results, I think we may be approaching the upper limit of the scalability of my code. This could be an example of **Amdahl's Law** in effect: for any problem, there is a limit of

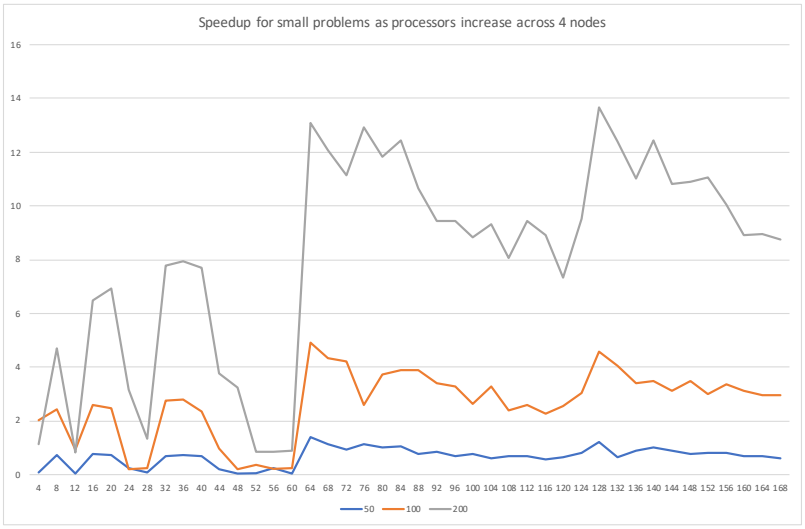

Figure 4: Graph showing the speedup for small problem sizes as processors increase.

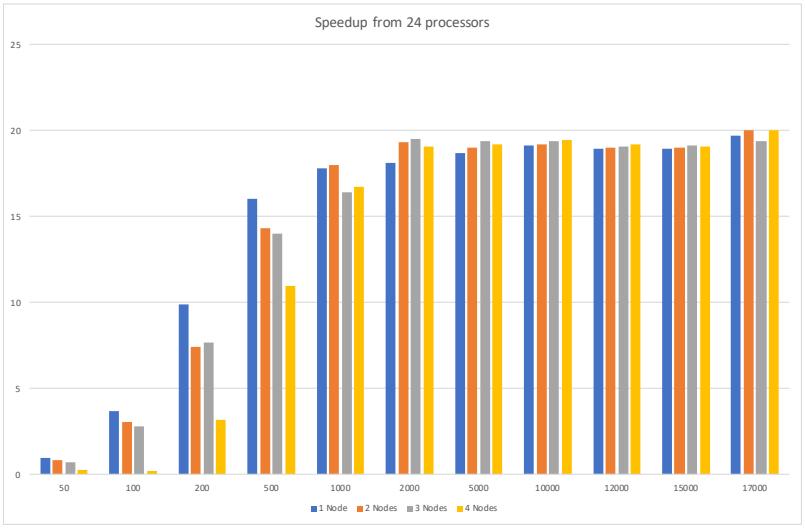

Figure 5: Graph showing the speedup for all problem sizes running on 24 processors, spread about across different numbers of nodes

possible speedup, regardless of processor count.

Figure 5 shows the comparison of speedup from 24 processors, as these processors are spread out evenly among n nodes. For small problems, using more nodes than necessary will decreases speedup, probably because its quicker to communicate between processors on the same nodes than over the network. However, as the problem size increases, the advantage of using local processors is reduced, and the difference in speedups is diminished.

5.1.1 OpenMPI Tuned Algorithm Selection

The volatility when the number of processors is below 64 shown in figures 3 and 4 could be to do with how OpenMPI attempts to using tuning to optimise some operations such as `gather`, which is used in my code. OpenMPI will select an algorithm to use for gathering data based on the number of processors and the size of the messages. This could explain the drastic changes in speedup before 64 processors, and the the smooth curves that can be seen after 64. As OpenMPI's algorithm selection is also to do with the message size, it is possible that it will pick one algorithm and stick with it regardless of processor size for the larger problems - hence the smoother curves.

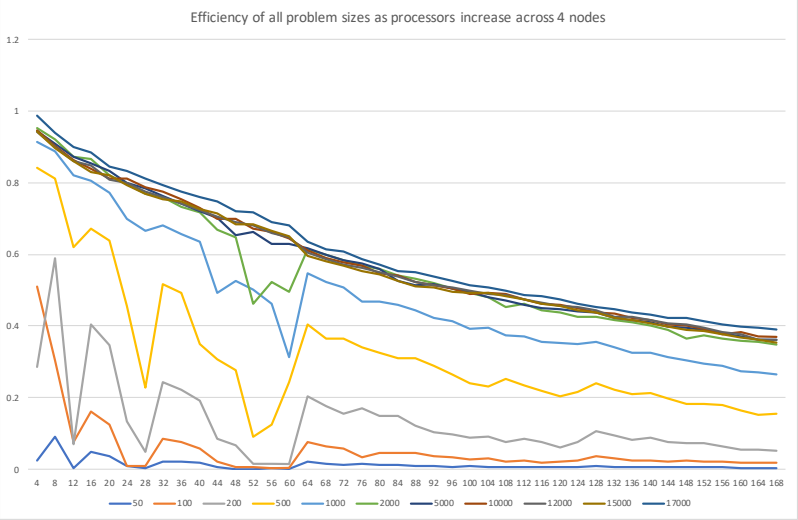

Figure 6: Graph efficiency as processor per node increases on 4 nodes.

5.2 Efficiency

Efficiency measures the speedup per processor. A low efficiency indicates that adding processors isn't having much of an impact on the execution time. This is because there is only so much work to be spread around, and adding more processors means that there are more processors idling, either for communication or for other processors to finish their work. We expect efficiency to decrease as we add more processors.

Figure 6 shows the efficiency of all problem sizes as we increase the number of processors per node on 4 nodes. As expected, efficiency generally decreases, as processors are doing less and less.

The volatility of efficiency when the number of processors is below 64 could be to do with coalescence being used for G

This could be explained in the way that the problem is split between processors. If the work doesn't evenly divide between the number of processors, then some processors must wait for other processors to process the remainders. This could cause significant decreases in efficiency as the processors will spend a lot of time waiting for the remainder rows to be worked on.

An expected, extremely low efficiency of less than 0.2 is observed for the smaller problems as we add more processors. The problem of size 50 (48 workable rows) reaches 0.0035 at 48 processors (12 processors per node, 4 nodes) as there is no advantage to having this many processors - there is not enough

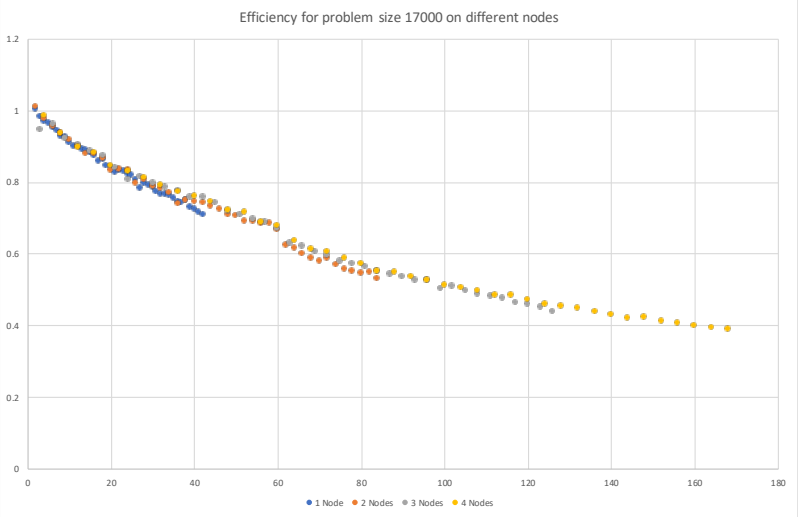

Figure 7: Graph showing efficiencies for problem size 17,000

work to be spread around for it to be worth the communication overhead.

I would have thought that spreading the processors out across different nodes would have decreased efficiency. However, figure 7 shows that efficiency is mostly independent of whether processors are on the same machine, or spread over different nodes. This is likely because the nodes that we're running these tests on are connected via 100Gb InfiniBand networking, and MPI doesn't optimise for local communication between cores. Maybe the InfiBand networking is preferable as it will be routed using external hardware, and the memory bus may have a lot of traffic.

5.2.1 Gustaffson's Law

Gustaffson's Law is **for a fixed number of processors, as the work increases, the fraction of sequential time to parallel time tends to 0, increasing efficiency**. Figure 8 demonstrates this law, as the efficiency for larger problems is smaller than that of smaller problems. The efficiency does plateau as the fraction of sequential work approaches zero.

5.3 Karp-Flatt Metric

$$e = \frac{\frac{1}{S_p} - \frac{1}{p}}{1 - \frac{1}{p}} \tag{1}$$

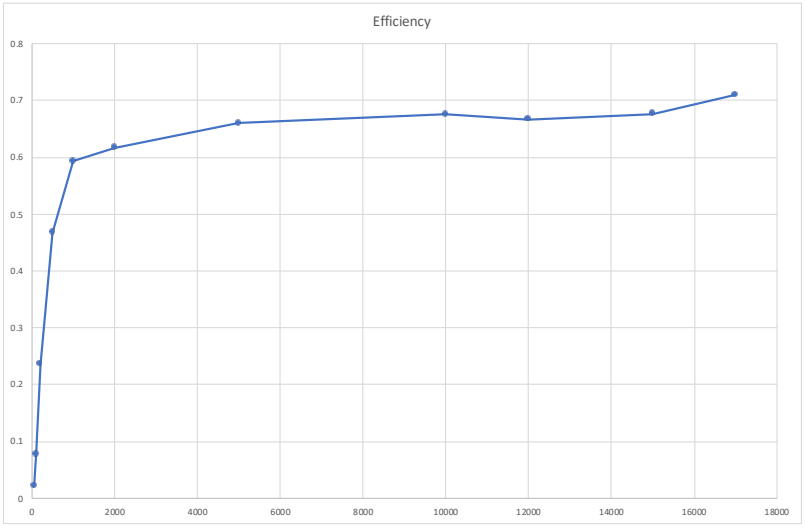

Figure 8: Graph showing efficiency of 42 processors as the problem size increases

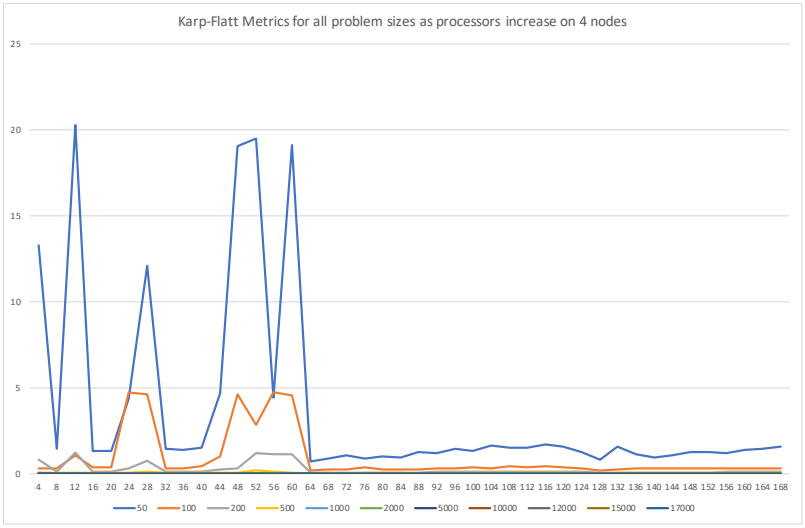

Figure 9: Graph showing the calculated Karp-Flatt value for all problem sizes

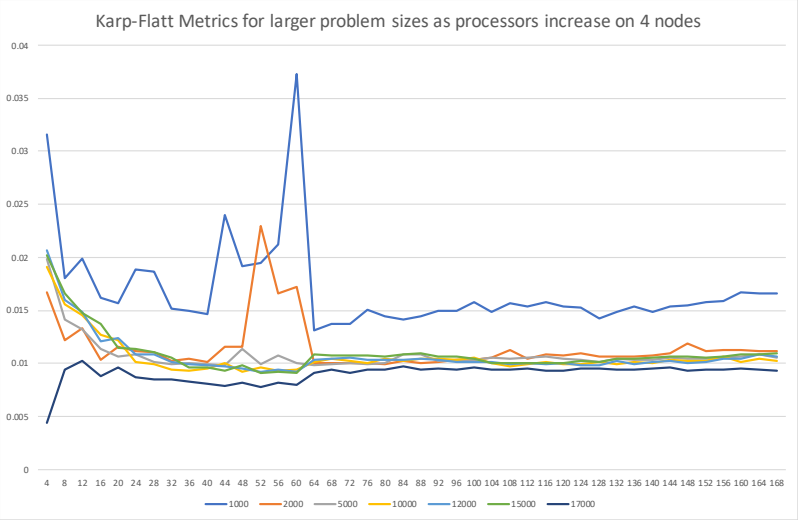

Figure 10: Graph showing the calculated Karp-Flatt value for large problem sizes

Using the calculated speedup values, we can calculate the Karp-Flatt metrics for each problem size. The Karp-Flatt metric (1) measures how much of the processing time is sequential: a higher value indicates a larger sequential part.

Figure **??** shows the calculated Karp-Flatt metric for all problem sizes as processors increase. The volatility before 64 will be due to communication overhead, discussed in section 5.1.1. The high KF values for the small problem sizes (reaching 20 !) imply an extremely large sequential section. The problem is dwarfed by the communication overhead from OpenMPI.

Figure **??** excludes the smaller problems, providing a suitable y-axis scale. These values are low, meaning that a large amount of the execution time for my implementation on these larger problems are spent in parallel. This suggests than the efficiency decrease we have observed is mostly not due to poor utilisation of additional cores.

5.4 Iso-Efficiency

For every problem, there is a relationship between the problem size n and the number of processors p called its **iso-efficiency** that ensures a constant efficiency is maintained after increasing p, by increasing the problem size n by some amount.

$$T_s = cT_o \tag{2}$$

$$T_o = pT_p - T_s \tag{3}$$

The time it takes for the parallel solution to solve the problem of size n is $T_p = \frac{n}{p} + \sqrt{n}$: the cells are evenly amongst the processors ($\frac{n}{p}$), and then the remaining rows are calculated in parallel ($\sqrt{n}$). All processors must wait for these extra rows to compute, independent of whether they are actually doing anything.

The sequential solution will simply step through each cell, calculating the value, so its time is $T_s = n$.

Using equation (3):

$$\begin{aligned} T_o &= pT_p - T_s \\ &= p(\frac{n}{p} + \sqrt{n}) - n \\ &= n + p\sqrt{n} - n \\ &= p\sqrt{n} \end{aligned}$$

Using equation (2), and T_o:

$$\begin{aligned} n &= c(p\sqrt{n}) \\ &= p^2c^2 \end{aligned}$$

In big-O notation...

$$n = O(p^2)$$

Nodes	Processors	Measured efficiency for $n = 25e^6$ (2 s.f.)	Difference
1	17	0.83	0.01
2	16	0.85	0.01
2	18	0.83	0.01
3	18	0.84	0.00
4	16	0.85	0.01

Table 1: Table showing the measured efficiencies of $p \approx 17$, $n = 25e^6$, across 1 to 4 nodes.

This results means that the number of processors must be increased by $\sqrt{x}$ to maintain efficiency, after the problem size increased by x.

5.4.1 Comparison to measured data

We can predict the number of processors to achieve the same measured efficiency 0.84 as $n = 4e^6$, $p = 7$ for a larger problem size.

$$7^2 = \frac{4e^6}{k}, \quad p^2 = \frac{25e^6}{k}$$

(make k the subject, and join both sides...)

$$\frac{4e^6}{7^2} = \frac{25e^6}{p^2}$$

$$p = \sqrt{\frac{25 \times 7^2}{4}}$$

$$= 17$$

The actual calculated efficiency for $n = 25e^6, p = 17$ spread across 1 node is 0.83, an error of only 0.01. The measured values matching with the predicted value shows that our model is accurate, for this data point.

Note this reading is still on 1 node, as 17 is a prime number. As shown in figure 7, efficiency isn't affected much by the fair distribution of processors across nodes. Table 1 shows the results of the closest number of processors, spread across more nodes.

6 Conclusion

In this report we've explored my methodology of writing a parallel program using OpenMPI, and then explored the results, including a scalability investigation that covered speedup, efficiency, Amdahl's Law, Gustafson's Law, Karp-Flatt metrics and finally Iso-Efficiency. We compared the Iso-Efficiency model with the measured data to verify its accuracy.

6.1 Improvements

Given more time, I would like to continue this project by doing the following.

6.1.1 An accurate measurement of sequential vs parallel time

I would like to spend some time in finding a way of measuring sequential time of calculation. This could be done by measuring how long each processor takes to wait for synchronisation. As indicated by the Karp-Flatt metrics, this value would be large comapred to parallel processing time, and much smaller for the larger problems.

Having close-to-accurate sequential times would allow us to calculate Work Efficiency and the maximum limit, given by Amdahl's Law (equation 4).

I did take naive, coarse measurements of the time to setup (sequential), and the time to solve the problem (parallel), but this doesn't take into account that each super-step is followed by a sequential check.

$$S_\infty \leq \frac{T_{seq} + T_{par}}{T_{seq}} \tag{4}$$

6.1.2 An improved order of row processing

To reduce waiting times, it might be worth having the workers calculate their first and last rows *first*, asynchronously send these rows to its neighbour and then continuing with the rows in between. This would mean that when the worker's neighbors are finished, the rows will be waiting for them. Of course, the worker would have to wait to ensurethe send had finished at some point, but this could be *after* the rest of the work has been done.

6.1.3 Reduced check frequency

A way of decreasing the coupling of multiple worker would be to use longer super steps: instead of checking whether the board is done after each iteration, each worker could instead run for some x number of iterations, before gathering for a check. This could potentially lead to unnecessary work being done, but it would also reduce the communication overhead between workers. The code is ready to support this, but I wasn't able to fully test it in time for submission.

6.2 Source code

The source code can be found as part of this coursework submission, or on GitHub.

A Results

Results can be found in `.csv` files as part of this coursework submission.

B Excerpt of testing output

```
...
nodes=3 proc=123 size=15000 correct!
nodes=3 proc=123 size=17000 correct!
nodes=3 proc=126 size=50 correct!
nodes=3 proc=126 size=100 correct!
nodes=3 proc=126 size=200 correct!
nodes=3 proc=126 size=500 correct!
nodes=3 proc=126 size=1000 correct!
nodes=3 proc=126 size=2000 correct!
nodes=3 proc=126 size=5000 correct!
nodes=3 proc=126 size=10000 correct!
nodes=3 proc=126 size=12000 correct!
nodes=3 proc=126 size=15000 correct!
nodes=3 proc=126 size=17000 correct!
nodes=4 proc=4 size=50 correct!
nodes=4 proc=4 size=100 correct!
nodes=4 proc=4 size=200 correct!
nodes=4 proc=4 size=500 correct!
nodes=4 proc=4 size=1000 correct!
nodes=4 proc=4 size=2000 correct!
nodes=4 proc=4 size=5000 correct!
nodes=4 proc=4 size=10000 correct!
nodes=4 proc=4 size=12000 correct!
nodes=4 proc=4 size=15000 correct!
nodes=4 proc=4 size=17000 correct!
nodes=4 proc=8 size=50 correct!
nodes=4 proc=8 size=100 correct!
nodes=4 proc=8 size=200 correct!
...
```