Shared Memory Coursework

Alexander Dawkins

16 November 2022

Contents

1 Introduction

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

> "The objective of this assignment is to give you experience of using a cluster with a batch processing system; using pthread parallelism on a shared memory architecture; plus a feel for how parallel problems scale on such architectures." [1]

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

2 Design of the Program

To become familiar with the problem, I first wrote a naive solution that would repeatedly calculate the next generation of the matrix until the difference between the current and next generation is less than the provided precision. This used only a single thread, and no considerations for efficiency were made.

2.1 Initial Parallel Solution

My first attempt at making my solution parallel was simply running the same code on many threads, with a mutex for each cell. Each thread would start at cell $(1, 1)$ and progress as if it was reading a book. If the next cell's mutex was locked, it would simply wait for it to become unlocked. As there are now many threads, a single board was used for both reading and writing simultaneously.

After learning more about Parallel Computing, I now see that there are many things wrong with this solution.

2.2 Reducing waiting time for Mutexes

As all the threads start in the same place, follow the same path, and stay in the same order, the thread that's in front limits the speed that new values can be created. This is extremely wasteful of time, as the rest of the board is available to be operated on. It makes sense to spread the threads out over all the space.

To fix this problem, I increased the amount of work a thread would do before waiting by assigning an entire row to each thread. When the row is complete a (mutex controlled) circular list is used to find a row that hasn't been worked on for the longest.

2.3 Ensuring accurate results

Using the same board for reading and writing will produce a "legal" result (i.e. each value is equal to the average of its neighbors). However, it is not guaranteed that the final board will be the same as a board that has been worked on with respect for generation-parity between each cell.

In some contexts this wouldn't matter, and the decrease in processing time would be preferential to a completely accurate result, but to challenge myself, I wanted to ensure that each board generated was as accurate as possible.

I used a secondary board for the answers of a generation, and once all threads had reached a barrier, an arbitrary thread would copy the new values back to the original board.

2.4 Reducing barrier waiting times

The program is now producing accurate results, but is extremely slow.

Instead of only working on a single row at a time, we can instead equally distribute the board (and therefore work) between the threads in larger chunks. As long as there is enough work to be shared among the threads, each thread should now be working on a larger chunk of the board in each super-step. Threads wait at a barrier when they have finished processing their batch of rows. As each thread now has a dedicated section of the board to work on (that it will never leave), **mutexes are no longer necessary** to protect the rows (or cells).

If the number of threads is not a factor of the height of the board, one of the remaining n rows is given to each of the first n threads. A "fairer" method of dividing the work would be to further divide the rows so that each thread would have the exact same number of cells, or to use a thread-pool design, where remaining work can be distributed to available threads. These ideas would be interesting to implement, but I believe them to be outside of the aims of this assignment.

2.5 Avoiding Race Conditions

To check if the board is finished, if a thread makes no meaningful change in a whole generation (all differences between the new values and old values are within the chosen precision), it will set its finished flag to 1 before waiting at the barrier. While the worker threads are working, the main thread will check if all flags are true, and if so, will change the global "run" flag to false before releasing the other threads. Upon waking, the threads will check the global run flag and return.

The design we have just arrived at naturally avoids race conditions: at no point in the code is there any writes to the same value / part of memory. Each thread works on its own discrete partition of the board, flags itself as done using

its own element of `threadDone[]`, and checks if the original thread has changed the `run` flag.

To prevent non-deterministic results, all values are read from one board, and the results are written to another. This ensures that the calculated values are consistent, and the order of work doesn't matter. It is allowed and expected that threads will read the same value simultaneously.

3 Correctness Testing

3.1 Manual Testing

I calculated the following 5x5 grid manually, and compared it to the output of the program. It provides a quick way to test that the program is getting the basics correct.

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

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

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

3.2 Example answers generated in Microsoft Excel

I then created an Excel sheet set up to generate 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

I then compared the a variety of single-threaded answers with the same problem calculated on multiple cores, to prove that the increasing the number of processors doesn't introduce any inaccuracies in the answer (or produces the exact same problems!)

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.

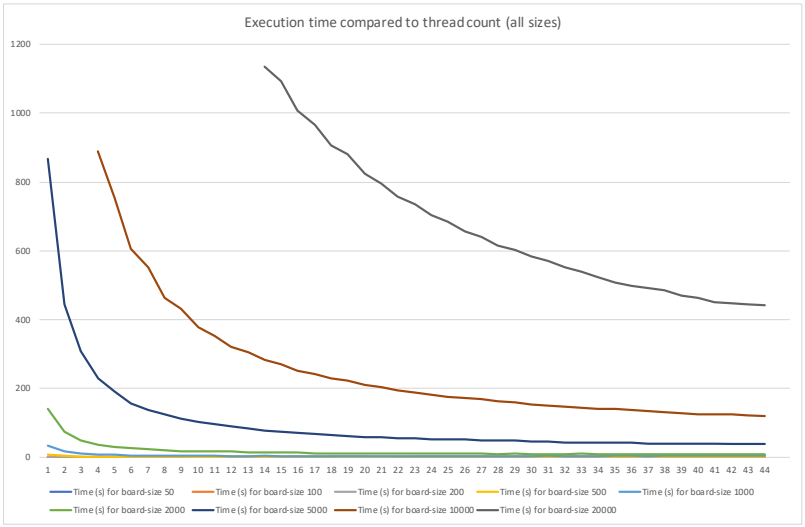

Figure 1: Graph showing the change in execution time as thread count increases

4 Results

The included results show the time that the program took to relax square boards of width 50 to 20,000. 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 $10e^{-4}$ was used as the limit of acceptable difference for two values to be considered equal.

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

4.1 Estimating single-threaded results

As the shared computer has an upper limit for execution time of **20 minutes** (1200 seconds), it was impossible to get real values for the execution times of the larger problems (10,000x10,000 and 20,000x20,000), on fewer threads. So that we may still explore the performance of the program on the larger problems, naive models were found to estimate the execution time T as a function of the number of threads used t.

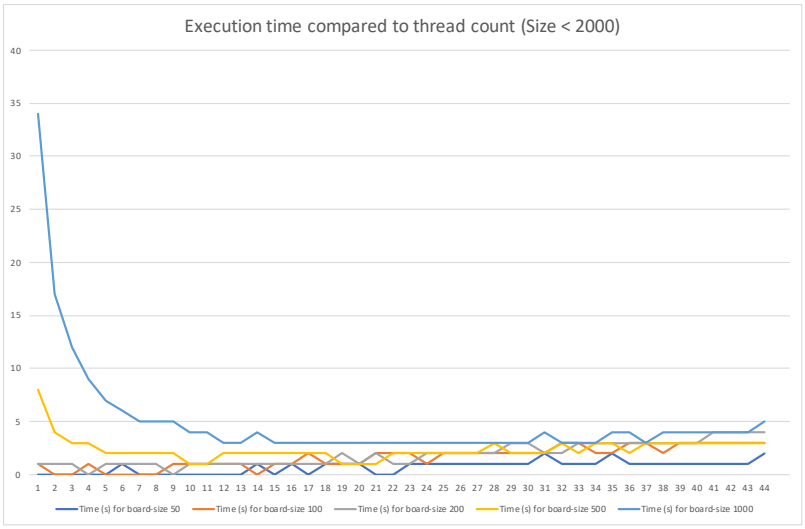

Figure 2: Graph showing the change in execution time of smaller problems as thread count increases

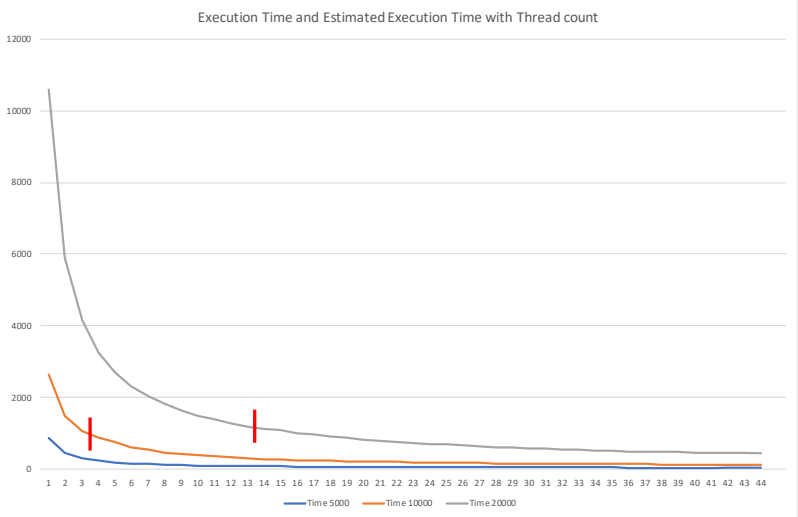

Figure 3: Graph showing the estimated execution times in the context of real data

$$T_{10,000} \approx 2639t^{-0.83} \tag{1}$$

$$T_{20,000} \approx 10597t^{-0.85} \tag{2}$$

These models (1, 2) are used to estimate the execution times for a single thread.

$$T_{10,000} = 2639, \quad t = 1 \qquad (1) \tag{3}$$

$$T_{20,000} = 10597, \quad t = 1 \qquad (2) \tag{4}$$

These models were found by fitting a power function to the real data. Figure 3 shows the estimated values in the context of the real values. The data to the left of the red bars is fabricated.

While this is definitely not ideal, I have still taken the liberty to include this data so that the shapes of the graphs can still be examined. I believe these values are smaller than what the true value would be, as the trend shown in Figure 4 imply that they could be larger. However, as I have been trying to increase the speedup since I started writing the code, I believe these numbers should remain conservative.

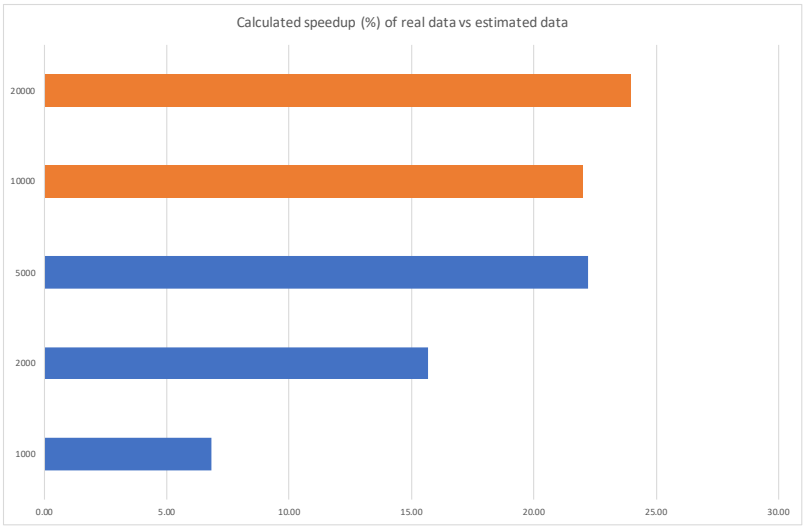

Figure 4: Graph showing the calculated speedup of the real data compared to estimated data

Throughout the report, we will mainly be exploring the (true) results of the 5,000x5,000 board, and it will be clearly stated if these estimated values have been used to calculate a value.

4.2 Execution Time

Figure 1 shows the decrease in execution time as we use more threads for the larger problems. You can see the two largest problems have been cut off, as the time it took to solve the problem with fewer threads exceeded the 20 minute (1200 seconds) job time-limit. Figure 2 shows only the smaller problems, and it shows that for the smaller problems, execution time **increases** as we add more threads. This is likely because of the overhead of orchestrating the many threads takes longer than the time it saves, compared to solving the problems with a single thread.

The "blocky" lines for the smaller problems are because I used a resolution of 1 second for the measurements. In retrospect, it would have been better to use milliseconds (like I did for the Amdahl calculations), but using seconds for the larger problems is fine.

5 Scalability

5.1 Speedup

Figure 5 shows a clear speedup of the problem of board-size 5000. This shows that the engineering of my code was successful, and using more threads is decreasing the time that this problem takes. For the other problems, we can see clear plateaus, and even decreases in speedup as increase the number of threads. I believe we would see similar parabolas for all problems, it is just that we have "exhausted" the two smallest problems shown here, and given enough threads, we would see it appear in the larger two.

All speedup values are less than linear speedup, meaning that at no point did my code obtain super-linear speedup. This could suggest that my uni processor solution for the problem is "fair" to compare to my parallel solution. It could also suggest that cache wasn't used effectively, and I have missed an opportunity to improve the performance of my parallel program.

Figure 6 exploits the values we estimated for single-thread times for the two larger problems. We can see a similar shape to the 5000x5000 problem, which has a true single-thread time. Although the actual speedup values are inaccurate, we can still appreciate the upwards trends. No linear speedup guide line has been included in this graph as it would be meaningless to compare estimated values to a perfect speedup.

5.2 Efficiency

While we observe an increase in speedup as we use more threads, we notice a decrease in gradient too. This is an example of diminishing returns, and we

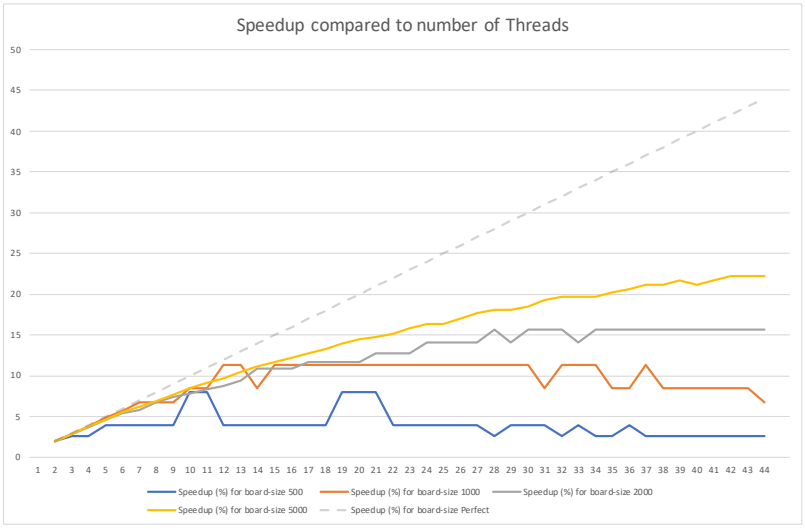

Figure 5: Graph showing the change in speedup as thread count increases. The dashed line signifies linear speedup.

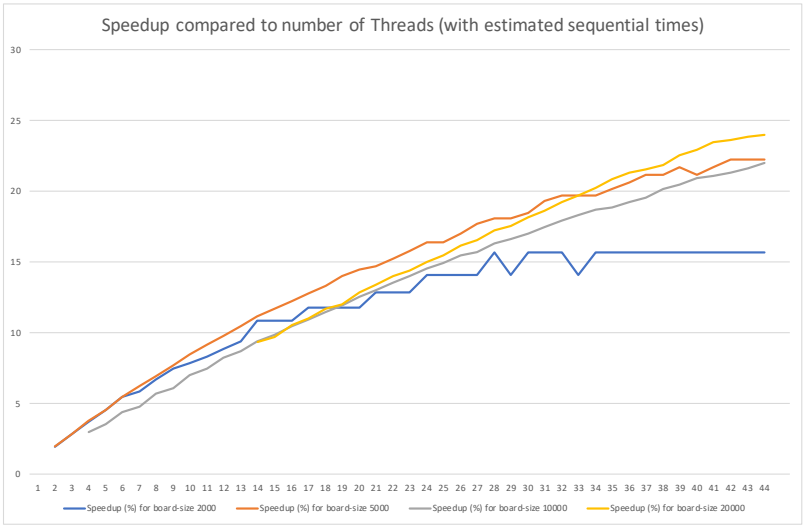

Figure 6: Graph showing the change in speedup as thread count increases, using estimated sequential times

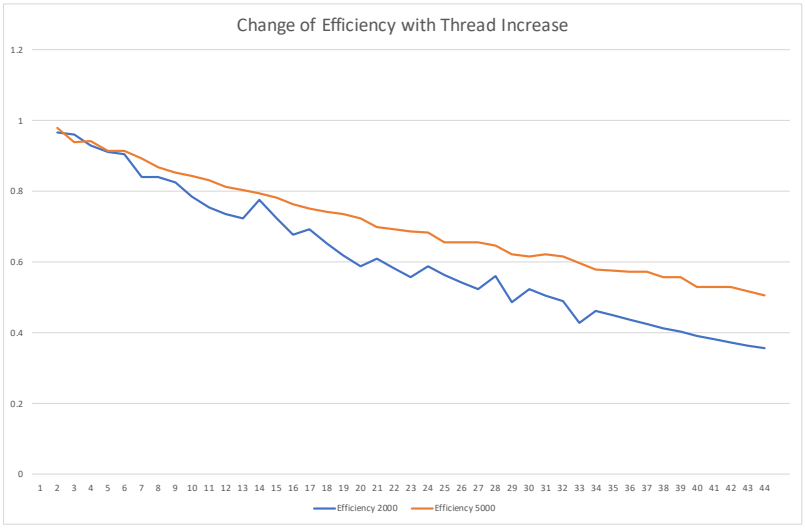

Figure 7: Graph showing the change of efficiency as thread count increases

can measure this as "efficiency". It would be unrealistic to expect x speedup from x more threads, and figure 7 shows this. A clear downwards trend can be seen, meaning that the improvement in speedup is diminishing as we add more threads.

We can see that the smaller problem is clearly less efficient than the larger, which is expected: as the problem size n gets larger, $e \to 1$, as we are closer to using all the cores, all the time.

Figure 8 uses the fabricated single-threaded execution times, and shows similar efficiency performance on the larger problems. We can see similar trends as in figure 7 for the two larger problems. If these values are accurate, I would suggest that the largest problem has the most "stable" efficiency as the sequential overhead is dwarfed by the parallel processing that this large problem requires.

5.3 Karp-Flatt Metric

$$e = \frac{\frac{1}{S_p} - \frac{1}{p}}{1 - \frac{1}{p}} \tag{5}$$

Using the (true) speedup values, we can calculate the Karp-Flatt metrics for each problem size. The Karp-Flatt metric (5) measures how much of the processing time is sequential: a higher value indicates a larger sequential part.

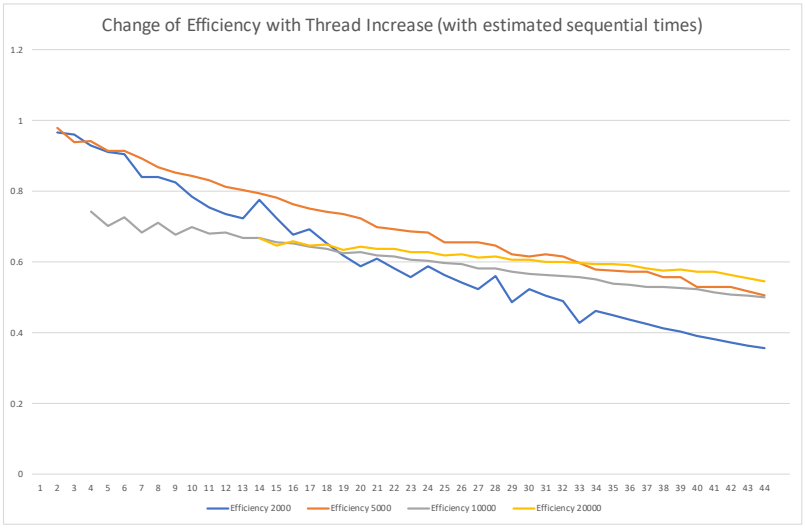

Figure 8: Graph showing the change of efficiency as thread count increases, using estimated sequential times

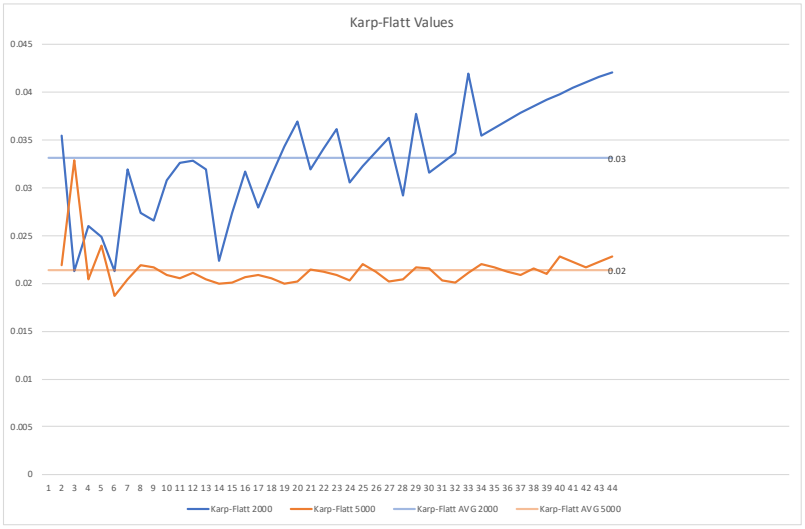

Figure 9: Graph showing the calculated Karp-Flatt value for each speedup datapoint, with average value for each problem size.

Figure 9 shows the calculated Karp-Flatt metric for board-sizes 2000 and 5000, along with the average values: 0.03 and 0.02.

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

To maintain efficiency, as we increase the number of processors p, we will also need to increase the problem size n by some amount. We can show this relationship by expressing n in terms of p.

$$T_s = cT_o \tag{6}$$

$$T_o = pT_p - T_s \tag{7}$$

The time it takes for the parallel solution to solve the problem of size n is $T_p = \frac{n}{p} + \sqrt{n}$, as we split the number of cells evenly amongst the processors ($\frac{n}{p}$), and then the remaining $p >$ rows are calculated in parallel ($\sqrt{n}$).

The sequential solution will simply step through each cell, calculating the value: $T_s = n$.

Using equation (7):

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

Using equation (6), and T_o:

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

In big-O notation...

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

This means that if the problem size is increased by a factor x, we have to increase the number of processors by $\sqrt{x}$ to maintain efficiency.

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 $n = 25e^6, p = 17$ is 0.75, an error of 0.09. While this proposed iso-efficiency could be good for giving an estimate, it doesn't appear to be extremely accurate. The real relationship between n and p will require p to be increased more to maintain a constant efficiency.

6 Conclusion

Overall, I believe this project to be a success: I have successfully written a parallel program that demonstrates the expected characteristics, including decrease in execution time, increase in speedup, and decrease in efficiency as we add more core to the same problem, and an increase in efficiency as we provide a larger problem to the same amount of cores. I'm happy with the low calculated Karp-Flatt Metrics.

However, this report isn't without flaws, and I believe I could improve it in the following ways.

6.1 Improvements

6.1.1 More large problems with uni-processor times

My plan for this project was write the code, gather results, and then write this report. However, I realised after gathering the (many) data points, that without single-threaded values, the rest of the results for the board size are close to worthless - as most metrics are comparisons to the performance of the program with no parallel calculations. Luckily, I had board-sizes 2000 and 5000 which demonstrated the characteristics of a parallel program with sufficient work to do, but I would have preferred to have more data-series to discuss in this report.

Perhaps better uni-processor times could have been estimated by modelling the speedup from the other problems, and calculating the estimated execution time from that. However, an estimated value is an estimated value, and they are all similarly useless in comparison to a truly measured value.

This problem could have been avoided with better planning / consideration from me.

6.1.2 More precise timings for smaller problem sizes

Similar to 6.1.1, it was only after generating graphs using the timing data of the smaller problems that I realised that the smaller problems needed timing at

a higher resolution.

I could have run the tests again but I wanted to be courteous to the other students using the shared computer, and didn't want to compare values measured in milliseconds with those measured in seconds.

6.1.3 A better predicted iso-efficiency relationship

I believe my inexperience with parallel computing has limited my ability to analyse my implementation, and this is shown by the iso-efficiency not giving an accurate prediction for the needed number of processors to maintain efficiency. I would like to spend more time analysing my solution to be able to deduce a more accurate iso-efficiency relationship.

6.1.4 An accurate measurement of sequential vs parallel time

Having reliable values for the time spent in parallel and sequential (T_{par}, T_{seq}) would allow me to calculate both **work efficiency** and the **Amdahl limit**.

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.

$$S_\infty \leq \frac{T_{seq} + T_{par}}{T_{seq}} = \frac{70.39634 + 0.239875}{0.239875} = \frac{70.636215}{0.239875} = 294.470932777 \quad (8)$$

The measurements I took were $T_{seq} = 0.239875$ and $T_{par} = 70.39634$ for 16 threads on board size 5000. These give an Amdahl limit of 294.47 as shown in (8). As the measured T_{seq} is probably less than the true value, in reality the maximum possible speedup is probably smaller than the calculated value:

$$S_\infty \ll 294.5 \quad (9)$$

6.2 Source code

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

A Execution Times Table

Threads	50	100	200	500	1000	2000	5000	10000	20000
1	0	1	1	8	34	141	867	2638.8	10597
2	0	0	1	4	17	73	443		
3	0	0	1	3	12	49	308		
4	0	1	0	3	9	38	230	889	
5	0	0	1	2	7	31	190	753	
6	1	0	1	2	6	26	158	605	
7	0	0	1	2	5	24	139	551	
8	0	0	1	2	5	21	125	464	
9	0	1	0	2	5	19	113	433	
10	0	1	1	1	4	18	103	378	
11	0	1	1	1	4	17	95	353	
12	0	1	1	2	3	16	89	322	
13	0	1	1	2	3	15	83	304	
14	1	0	1	2	4	13	78	282	1135
15	0	1	1	2	3	13	74	269	1092
16	1	1	1	2	3	13	71	253	1006
17	0	2	1	2	3	12	68	242	967
18	1	1	1	2	3	12	65	230	906
19	1	1	2	1	3	12	62	222	881
20	1	1	1	1	3	12	60	210	823
21	0	2	2	1	3	11	59	203	794
22	0	2	1	2	3	11	57	195	758
23	1	2	1	2	3	11	55	189	736
24	1	1	2	2	3	10	53	182	705
25	1	2	2	2	3	10	53	177	685
26	1	2	2	2	3	10	51	171	657
27	1	2	2	2	3	10	49	168	641
28	1	2	2	3	3	9	48	162	614
29	1	2	3	2	3	10	48	159	604
30	1	2	3	2	3	9	47	155	584
31	2	2	2	2	4	9	45	151	570
32	1	3	2	3	3	9	44	147	552
33	1	3	3	2	3	10	44	144	539
34	1	2	3	3	3	9	44	141	524
35	2	2	3	3	4	9	43	140	509
36	1	3	3	2	4	9	42	137	498
37	1	3	3	3	3	9	41	135	492
38	1	2	3	3	4	9	41	131	485
39	1	3	3	3	4	9	40	129	470
40	1	3	3	3	4	9	41	126	463
41	1	3	4	3	4	9	40	125	452
42	1	3	4	3	4	9	39	124	449
43	1	3	4	3	4	9	39	122	445
44	2	3	4	3	5	9	39	120	442

B Speedup Table

Threads	50	100	200	500	1000	2000	5000	10000	20000
2	#DIV/0!	#DIV/0!	1.00	2.00	2.00	1.93	1.96		
3	#DIV/0!	#DIV/0!	1.00	2.67	2.83	2.88	2.81		
4	#DIV/0!	1.00	#DIV/0!	2.67	3.78	3.71	3.77	2.97	
5	#DIV/0!	#DIV/0!	1.00	4.00	4.86	4.55	4.56	3.50	
6	0.00	#DIV/0!	1.00	4.00	5.67	5.42	5.49	4.36	
7	#DIV/0!	#DIV/0!	1.00	4.00	6.80	5.88	6.24	4.79	
8	#DIV/0!	#DIV/0!	1.00	4.00	6.80	6.71	6.94	5.69	
9	#DIV/0!	1.00	#DIV/0!	4.00	6.80	7.42	7.67	6.09	
10	#DIV/0!	1.00	1.00	8.00	8.50	7.83	8.42	6.98	
11	#DIV/0!	1.00	1.00	8.00	8.50	8.29	9.13	7.48	
12	#DIV/0!	1.00	1.00	4.00	11.33	8.81	9.74	8.20	
13	#DIV/0!	1.00	1.00	4.00	11.33	9.40	10.45	8.68	
14	0.00	#DIV/0!	1.00	4.00	8.50	10.85	11.12	9.36	9.34
15	#DIV/0!	1.00	1.00	4.00	11.33	10.85	11.72	9.81	9.70
16	0.00	1.00	1.00	4.00	11.33	10.85	12.21	10.43	10.53
17	#DIV/0!	0.50	1.00	4.00	11.33	11.75	12.75	10.90	10.96
18	0.00	1.00	1.00	4.00	11.33	11.75	13.34	11.47	11.70
19	0.00	1.00	0.50	8.00	11.33	11.75	13.98	11.89	12.03
20	0.00	1.00	1.00	8.00	11.33	11.75	14.45	12.57	12.88
21	#DIV/0!	0.50	0.50	8.00	11.33	12.82	14.69	13.00	13.35
22	#DIV/0!	0.50	1.00	4.00	11.33	12.82	15.21	13.53	13.98
23	0.00	0.50	1.00	4.00	11.33	12.82	15.76	13.96	14.40
24	0.00	1.00	0.50	4.00	11.33	14.10	16.36	14.50	15.03
25	0.00	0.50	0.50	4.00	11.33	14.10	16.36	14.91	15.47
26	0.00	0.50	0.50	4.00	11.33	14.10	17.00	15.43	16.13
27	0.00	0.50	0.50	4.00	11.33	14.10	17.69	15.71	16.53
28	0.00	0.50	0.50	2.67	11.33	15.67	18.06	16.29	17.26
29	0.00	0.50	0.33	4.00	11.33	14.10	18.06	16.60	17.54
30	0.00	0.50	0.33	4.00	11.33	15.67	18.45	17.02	18.15
31	0.00	0.50	0.50	4.00	8.50	15.67	19.27	17.48	18.59
32	0.00	0.33	0.50	2.67	11.33	15.67	19.70	17.95	19.20
33	0.00	0.33	0.33	4.00	11.33	14.10	19.70	18.33	19.66
34	0.00	0.50	0.33	2.67	11.33	15.67	19.70	18.71	20.22
35	0.00	0.50	0.33	2.67	8.50	15.67	20.16	18.85	20.82
36	0.00	0.33	0.33	4.00	8.50	15.67	20.64	19.26	21.28
37	0.00	0.33	0.33	2.67	11.33	15.67	21.15	19.55	21.54
38	0.00	0.50	0.33	2.67	8.50	15.67	21.15	20.14	21.85
39	0.00	0.33	0.33	2.67	8.50	15.67	21.68	20.46	22.55
40	0.00	0.33	0.33	2.67	8.50	15.67	21.15	20.94	22.89
41	0.00	0.33	0.25	2.67	8.50	15.67	21.68	21.11	23.44
42	0.00	0.33	0.25	2.67	8.50	15.67	22.23	21.28	23.60
43	0.00	0.33	0.25	2.67	8.50	15.67	22.23	21.63	23.81
44	0.00	0.33	0.25	2.67	6.80	15.67	22.23	21.99	23.98

C Efficiency Table

Threads	50	100	200	500	1000	2000	5000	10000	20000
2	#DIV/0!	#DIV/0!	0.50	1.00	1.00	0.97	0.98		
3	#DIV/0!	#DIV/0!	0.33	0.89	0.94	0.96	0.94		
4	#DIV/0!	0.25	#DIV/0!	0.67	0.94	0.93	0.94	0.74	
5	#DIV/0!	#DIV/0!	0.20	0.80	0.97	0.91	0.91	0.70	
6	0.00	#DIV/0!	0.17	0.67	0.94	0.90	0.91	0.73	
7	#DIV/0!	#DIV/0!	0.14	0.57	0.97	0.84	0.89	0.68	
8	#DIV/0!	#DIV/0!	0.13	0.50	0.85	0.84	0.87	0.71	
9	#DIV/0!	0.11	#DIV/0!	0.44	0.76	0.82	0.85	0.68	
10	#DIV/0!	0.10	0.10	0.80	0.85	0.78	0.84	0.70	
11	#DIV/0!	0.09	0.09	0.73	0.77	0.75	0.83	0.68	
12	#DIV/0!	0.08	0.08	0.33	0.94	0.73	0.81	0.68	
13	#DIV/0!	0.08	0.08	0.31	0.87	0.72	0.80	0.67	
14	0.00	#DIV/0!	0.07	0.29	0.61	0.77	0.79	0.67	0.67
15	#DIV/0!	0.07	0.07	0.27	0.76	0.72	0.78	0.65	0.65
16	0.00	0.06	0.06	0.25	0.71	0.68	0.76	0.65	0.66
17	#DIV/0!	0.03	0.06	0.24	0.67	0.69	0.75	0.64	0.64
18	0.00	0.06	0.06	0.22	0.63	0.65	0.74	0.64	0.65
19	0.00	0.05	0.03	0.42	0.60	0.62	0.74	0.63	0.63
20	0.00	0.05	0.05	0.40	0.57	0.59	0.72	0.63	0.64
21	#DIV/0!	0.02	0.02	0.38	0.54	0.61	0.70	0.62	0.64
22	#DIV/0!	0.02	0.05	0.18	0.52	0.58	0.69	0.62	0.64
23	0.00	0.02	0.04	0.17	0.49	0.56	0.69	0.61	0.63
24	0.00	0.04	0.02	0.17	0.47	0.59	0.68	0.60	0.63
25	0.00	0.02	0.02	0.16	0.45	0.56	0.65	0.60	0.62
26	0.00	0.02	0.02	0.15	0.44	0.54	0.65	0.59	0.62
27	0.00	0.02	0.02	0.15	0.42	0.52	0.66	0.58	0.61
28	0.00	0.02	0.02	0.10	0.40	0.56	0.65	0.58	0.62
29	0.00	0.02	0.01	0.14	0.39	0.49	0.62	0.57	0.60
30	0.00	0.02	0.01	0.13	0.38	0.52	0.61	0.57	0.60
31	0.00	0.02	0.02	0.13	0.27	0.51	0.62	0.56	0.60
32	0.00	0.01	0.02	0.08	0.35	0.49	0.62	0.56	0.60
33	0.00	0.01	0.01	0.12	0.34	0.43	0.60	0.56	0.60
34	0.00	0.01	0.01	0.08	0.33	0.46	0.58	0.55	0.59
35	0.00	0.01	0.01	0.08	0.24	0.45	0.58	0.54	0.59
36	0.00	0.01	0.01	0.11	0.24	0.44	0.57	0.54	0.59
37	0.00	0.01	0.01	0.07	0.31	0.42	0.57	0.53	0.58
38	0.00	0.01	0.01	0.07	0.22	0.41	0.56	0.53	0.57
39	0.00	0.01	0.01	0.07	0.22	0.40	0.56	0.52	0.58
40	0.00	0.01	0.01	0.07	0.21	0.39	0.53	0.52	0.57
41	0.00	0.01	0.01	0.07	0.21	0.38	0.53	0.51	0.57
42	0.00	0.01	0.01	0.06	0.20	0.37	0.53	0.51	0.56
43	0.00	0.01	0.01	0.06	0.20	0.36	0.52	0.50	0.55
44	0.00	0.01	0.01	0.06	0.15	0.36	0.51	0.50	0.54

References

[1] R Bradford. *CM30225 Parallel Computing Assessed Coursework Assignment 1*. URL: https://people.bath.ac.uk/masrjb/CourseNotes/Assign.bpo/assign30225-22a.pdf.