Parallel Computing Assignment 2
Distributed Memory

December 23, 2016

Contents

1 Approach to Parallelisation **3**

1.1 Introduction 3

1.2 Partitioning the Matrix 3

1.3 MPI Strategy 5

2 Correctness Testing **6**

3 Scalability Investigation **8**

3.1 Fixed Problem Size 8

3.2 Variable Problem Size 9

3.3 Precision 9

4 Cycle Accounting **10**

A Full Testing Results **13**

A.1 Time and Speedup for 1-16 threads, $d = 1000$, $p = 0.5$ 13

A.2 Time and Speedup for 2-32 threads, $d = 1000$, $p = 0.5$ 13

A.3 Dimensions against time, $p = 0.5$, $t = \{1, 8, 16\}$ 14

A.4 Precision against iterations and time, $d = 500$, $t = 16$ 14

List of Figures

1 Additional rows required (blue) to relax elements in a single row (yellow). . . 4

2 Worker thread control flow 6

3 Main thread control flow 6

4 Time for 1-16 threads to relax the same matrix, $d = 1000$, $p = 0.5$ 8

5 Speedup and efficiency achieved by 1-16 threads relaxing the matrix, $d = 1000$, $p = 0.5$ 8

6 Time for 2-32 threads to relax the same matrix, $d = 1000$, $p = 0.5$ 8

7 Speedup achieved by 2-32 threads relaxing the matrix, $d = 1000$, $p = 0.5$ . . 8

8 Matrix dimensions against time for 1 (light grey), 8 (medium grey) and 16 (dark grey) threads. 9

9 Precision values against iterations and time 9

10 The top ten functions by self-cycles 10

11 Call graph showing `relax_cell` and `relax_section` 10

1 Approach to Parallelisation

1.1 Introduction

The assignment was to implement matrix relaxation for a distributed memory architecture using MPI, on a matrix $\mathcal{M}$ of variable square dimension d, using c MPI processes each running on its own processor on one of n nodes, working to a floating point precision of p. Since Balena doesn't permit oversubscription, t will always be $\leq c$.

My solution uses `MPI_Scatterv` and `MPI_Gatherv` to distribute and reassemble the matrix among processors, and `MPI_Allreduce` to broadcast and reduce a global exit condition status each iteration.

1.2 Partitioning the Matrix

The primary goal in partitioning the matrix for relaxation across multiple processors was to minimise the amount of data which needs to be passed between nodes, as network I/O is orders of magnitude slower than accessing data from memory on the same node, or CPU cache.

To avoid overly complicating my solution for negligible fairness gains, I chose not to split rows of the matrix between different processes and only assigned processes a whole number of rows to work on. Relaxing a square matrix of size $n \times n$ with c processes requires $(n-2)$ rows to be relaxed each iteration minus the first and last values of each row, as the array boundary values remain fixed.

Every process relaxes a minimum of $(n-2)$ div c rows, where div is the integer division operation. This leaves $(n-2)$ mod c cells remaining, and as MPI is SPMD and therefore not run in lockstep there is no way of predicting which processor will finish first, for the sake of simplicity this leaves the first $((n-2) \bmod c)$ processes assigned to do $((n-2) \operatorname{div} c) + 1$ rows each.

Since each process only works on a subset of the complete matrix and there is a 1:1 ratio of MPI processes to real processors, there is no need to generate, store, or update the entire matrix on every core. This massively reduces the overhead of updating data after each iteration and the overhead of allocation.

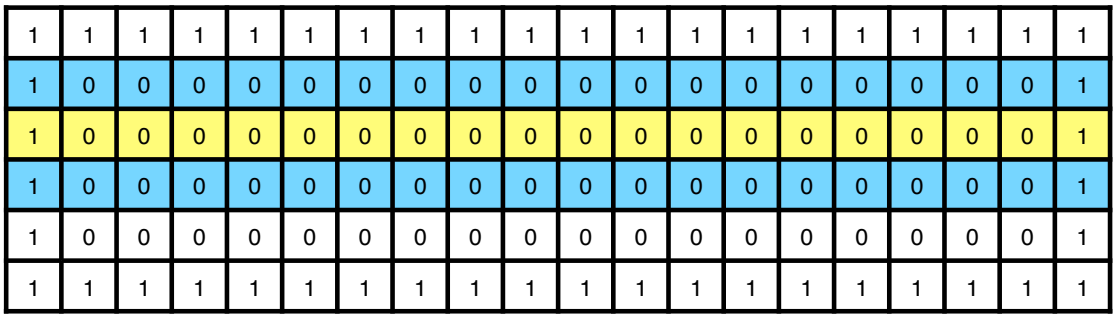

Figure 1: Additional rows required (blue) to relax elements in a single row (yellow).

As Figure 1 illustrates, irrespective of how many partitions the matrix is split into, each CPU will require the rows immediately before and after its partition in a read-only capacity. This overhead is significant in Figure 1 (200%), and it can be seen that for an $n \times n$ matrix, it is inefficient (i.e. has an overhead $\geq 100\%$) to have more than $\frac{n-2}{2}$ processors relaxing. From a practical parallelism standpoint, relaxing a matrix with dimensions only double that of the number of available processors is poor exploration of data parallelism and likely to be slower than a serial equivalent.

The complete $n \times n$ matrix is allocated and initialised from the root process and every processor (including the root) allocates space for $2n + \frac{(n-2)(n)}{c} + \frac{(n-2)(n)}{c}$ elements. The additional array of size $+\frac{(n-2)(n)}{c}$ is to preserve the consistency of data being read by performing the *relax* operation; no cell can be overwritten with its new value until the new value every other cell has been calculated. This is the case regardless of whether the relaxation is performed sequentially or in parallel, and necessitates using a second matrix of the same dimensions, into which the relaxed values are written. These dimensions however, do not have to include the $2n$ read-only values from the rows above and below.

This allocation on each of c processors, with an additional $n \times n$ on the root, gives a space complexity of $\mathcal{O}(3n^2 - 4n + 2nc)$. When calculating space complexity, c is constant as it is simply the number of processors so the space complexity is still asymptotic to $\mathcal{O}(n^2)$; the same space complexity as the shared memory equivalent solution which involved swapping two $n \times n$ arrays between iterations.

After each iteration, each process sends the $\frac{n-2}{c}$ rows which it has relaxed back to the root process. In actuality, this is not necessary because only the first and last relaxed rows contain values which are needed by any other process, so only they need to be sent. I decided not to attempt to exploit this because it introduced additional complexity, however if I were

attempting to scale my solution to much larger matrices, this would be the first optimisation I would make.

An additional optimisation I could have made was in the calculation of precision. My implementation avoids iterating over cells twice by both relaxing and calculating whether the precision has been reached in the same loop, rather than calculating precision once all cells have been relaxed within a process or from the root process. However, once a process encounters a cell whose new value differs from the old by more than the specified precision, it does not have to calculate the precision for any other values. I kept calculating every value however, as some of my testing used small precision values and it was useful that the values of `local_continue` and `global_continue` were exactly the number of cells where precision had yet to be reached.

1.3 MPI Strategy

There are three types of communication which need to happen during the execution of the program, all of them every iteration:

1. Sending $2 + \frac{n-2}{c}$ rows of the full matrix to each process.

   As only a subset of the matrix is required by each process, `MPI_Scatter` was the logical solution. However, Scatter requires all chunks to be the same size, which is only the case when $(n-2) \bmod c = 0$. The equivalent for unequal partitions is `MPI_Scatterv`, so this is how I send data from the root process at the start of each iteration.

2. Sending $\frac{n-2}{c}$ relaxed rows from each process back to the root.

   This communication is essentially symmetric to the previous, so I used `MPI_Scatterv`'s counterpart `MPI_Gatherv` to communicate the relaxed cells back to the root process.

3. Communication between all processes as to whether another iteration is required.

   I considered adding an extra cell to each array sent back to the root process in the previous communication to piggyback the completion status of each processor with its data, but ultimately this seemed an untidy solution to a problem which `MPI_Allreduce` could solve. I therefore reduce the `MPI_SUM` operation over the values of `local_continue` on every processor (where any positive value indicates a cell which has not yet reached the required precision.) If any processor needs to continue, they must all continue.

Figure 2: Worker thread control flow

Figure 3: Main thread control flow

None of my worker threads allocate thread-local variables on the heap, as this is unnecessary and also requires allocator and deallocator synchronisation across every thread [2].

2 Correctness Testing

To verify that my implementation of relaxation was correct, I manually calculated every step in relaxing a matrix and verified not only that both solutions terminated after the same number of iterations, but also that after each iteration, my matrix was identical to the computed one. Both my hand-computed relaxation (below) and my single-threaded implementation required four iterations to relax the 5×5 matrix to a precision of 0.75.

1.000000	1.000000	1.000000	1.000000	1.000000	**Initial Matrix**
1.000000	3.000000	7.000000	2.000000	1.000000	
1.000000	8.000000	6.000000	5.000000	1.000000	
1.000000	9.000000	0.000000	4.000000	1.000000	
1.000000	1.000000	1.000000	1.000000	1.000000	Max Δ: abs(9-2.5) = 6.5
					6.5 $\geq$ 0.75 so continue.
1.000000	1.000000	1.000000	1.000000	1.000000	
1.000000	4.250000	3.000000	3.500000	1.000000	
1.000000	4.750000	5.000000	3.250000	1.000000	
1.000000	2.500000	5.000000	1.750000	1.000000	
1.000000	1.000000	1.000000	1.000000	1.000000	Max Δ: abs(5-2.5625) = 2.4375
					2.4375 $\geq$ 0.75 so continue.
1.000000	1.000000	1.000000	1.000000	1.000000	
1.000000	2.437500	3.437500	2.062500	1.000000	
1.000000	3.187500	4.000000	2.812500	1.000000	
1.000000	2.937500	2.562500	2.562500	1.000000	
1.000000	1.000000	1.000000	1.000000	1.000000	Max Δ: abs(3.4375-2.375) = 1.0625
					1.0625 $\geq$ 0.75 so continue.
1.000000	1.000000	1.000000	1.000000	1.000000	
1.000000	2.156250	2.375000	2.062500	1.000000	
1.000000	2.593750	3.000000	2.406250	1.000000	
1.000000	1.937500	2.625000	1.843750	1.000000	
1.000000	1.000000	1.000000	1.000000	1.000000	Max Δ: abs(2.625-1.9453125) = 0.6796875
					0.6796875 $<$ 0.75 (done).
1.000000	1.000000	1.000000	1.000000	1.000000	
1.000000	1.742188	2.054688	1.695312	1.000000	
1.000000	2.023438	2.500000	1.976562	1.000000	
1.000000	1.804688	1.945312	1.757812	1.000000	
1.000000	1.000000	1.000000	1.000000	1.000000	**Relaxed matrix (after 4 iterations)**

The hand-computed results show my checking of the maximum delta for each array after relaxing it to determine whether another iteration is required. Once I verified that my implementation was correct with one thread, I relaxed under the same conditions with 2-9 threads, printing all intermediate values and the iteration count at termination. I then used the Unix `diff` utility to check all 9 outputs were identical. These files have been included in the `correctness` directory, for verification.

3 Scalability Investigation

3.1 Fixed Problem Size

Once I had verified the correctness of my relaxation implementation, I chose a fixed array size and precision, and ran my program relaxing the same array with 1-16 threads. The time, speedup and efficiency (speedup achieved by n processors divided by n [3]) for this can be seen in Figures 4 and 5.

Figure 4: Time for 1-16 threads to relax the same matrix, $d = 1000$, $p = 0.5$

Figure 5: Speedup and efficiency achieved by 1-16 threads relaxing the matrix, $d = 1000$, $p = 0.5$

Each thread[1] saw efficiency reduce, as the speedup had a progressively smaller impact on the time reduction (diminishing returns; [1]). This is consistent with Amdahl's law, a corollary of which states that for P processors and a computation whose sequential proportion is denoted f, predicted speedup is bound by $\frac{1}{f}$ [4]. Working backwards from this upper bound and assuming a maximum speedup of 14.5, this would imply that the sequential proportion of my solution is ≤ 0.069, or 6.9% of the total computation for this fixed problem size. It is important to note that the sequential proportion is not fixed, as changing the number of threads, precision or dimensions would alter the overhead needed to allocate, assign thread work, and wait at barriers.

The speedup I achieved for this problem size was sub-linear, but close to linear. The sequential proportion of the computation appears to become the limiting factor above 16 threads, preventing further speedup through parallelisation.[2] In order to verify this, I repeated the experiment with even numbers of threads between 2 and 32 on the same matrix. The resulting time, speedup and efficiency values can be seen in Figures 6 and 7, with the values for 1 thread (i.e. the sequential computation) left in for comparison.[3]

Figure 6: Time for 2-32 threads to relax the same matrix, $d = 1000$, $p = 0.5$

Figure 7: Speedup achieved by 2-32 threads relaxing the matrix, $d = 1000$, $p = 0.5$

[1] The thread counts specified in this section refer to worker threads, i.e. the main thread is not included.
[2] Full data in Appendix A.1.
[3] Full data in Appendix A.2.

3.2 Variable Problem Size

The second factor I tested was the effect of varying the matrix dimensions on the time, for various thread counts. The results of this can be seen in Figure 8[4]. The x-axis in the graph is scaled with respect to dimensions but not size, since the actual computation required for each dimension is proportional to the square of the dimension, not the dimension itself.

Figure 8: Matrix dimensions against time for 1 (light grey), 8 (medium grey) and 16 (dark grey) threads.

Consider the values which intersect the dotted line in Figure 8. In one minute, one thread can relax a 3000^2 matrix, with 9,000,000 elements. In the same amount of time, eight threads can relax a 7000^2 matrix (49,000,000 elements, and a speedup of 7) and sixteen threads can relax approximately 8500^2 (72,250,000 elements, and a speedup of 8.03). The relative drop in speedup from 1-8 and 8-16 threads again demonstrates the effect of diminishing returns and efficiency; the efficiency of eight threads is $7 \div 8 = 87.5\%$; for sixteen threads this becomes $8.03 \div 16 = 50.1\%$. It is important to note however that the most efficient number of processors is not the number which will yield the greatest absolute speedup. Mathematically, peak efficiency is reached by maximising $\frac{\text{speedup}}{n}$, which is equivalent to $\frac{T_{\text{seq}}}{n \times T_n}$. T_{seq} is fixed, so maximum efficiency is found minimising $n \times T_n$; the solution to which will be $n = 1$ unless speedup is superlinear.

3.3 Precision

I conducted a single experiment into the effects of lowering the precision threshold on the time and iterations required for my solution to terminate. Below are the results for a fixed array with $d = 500$ and 16 threads[5], showing iterations and time increasing at the same rate as the precision threshold is lowered.

Figure 9: Precision values against iterations and time

I tried to run my solution on a precision of 0.0000001, but it timed out on Balena, due to either exceeding the 15 minute limit while relaxing, or reaching the point at which applying

[4]Full data in Appendix A.3
[5]Full data in Appendix A.4

the relax operation left every cell unchanged. Not every matrix when relaxed to under a given precision will terminate. My implementation does not attempt to identify these cases and would therefore loop infinitely for precision values which aren't reachable for a given matrix.

4 Cycle Accounting

To determine whether there were any simple optimisations I could make to my implementation to improve speeds, I ran it on Balena under Valgrind, using the `callgrind` and `cachegrind` tools. I then exported the output file and analysed it with qcachegrind[6].

In `callgrind`, I ordered functions by cycle count (ignoring cycle counts of their callees). Although cycle count is not perfectly analogous to latency, it is the closest metric callgrind provides. This allowed me to see the functions which were dominating execution cycles, as shown in Figure 10. At the lower end of the list are functions which can be expected such as

Figure 10: The top ten functions by self-cycles

`main`, functions prefixed `dl_` which are the dynamic linker loading pthread library functions, and functions responsible for populating the array with random numbers. None of these are associated with the actual relaxation logic.

Figure 11: Call graph showing `relax_cell` and `relax_section`

The top two functions; `relax_cell` and its caller, my thread function `relax_section` account for the vast majority of cycles. Figure 11 shows that the 4,980,020 calls (over 5 iterations) to `relax_cell` account for 79.02% of the cycles accross the 16 (one for each thread) calls to `relax_section`.

There was no obvious improvement which could be made to the logic in `relax_cell`, as Figure **??** shows the average cycles to be fairly similar for all source lines which need to access the array (implying the cycles are due to load latency of the data), and for the assignment of the variable `new` (where the cycles would be due to latency of the store).

[6] https://kcachegrind.github.io/html/Home.html

To confirm this, I used the `cachegrind` tool on the same input and looked at the L1 misses for reads and writes in `relax_cell` (See Figure **??**.)

There is no easy fix for cache misses, as this is clearly a physical constraint and therefore machine-specific. Consequentially, as I had identified `relax_cell` as the single critical performance hotspot in my application, I did not do any further performance tuning as it would have yielded inconsequential improvements.

I have included the output from valgrind in both `callgrind` and `cachegrind` mode in the `valgrind` directory, from which all of the figures in this section can be reconstructed and explored. The options used for this run of my program were $d = 1000$, $p = 5.0$ and $t = 16$.

References

[1] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM. doi: 10.1145/1465482.1465560. URL `http://doi.acm.org/10.1145/1465482.1465560`.

[2] Andrew Binstock. Memory constraints on thread performance. `http://www.drdobbs.com/tools/memory-constraints-on-thread-performance/231300494`, 2011. Accessed: 10/11/2016.

[3] Derek L Eager, John Zahorjan, and Edward D Lazowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, 38(3):408–423, 1989.

[4] John L. Gustafson. *Amdahl's Law*, pages 53–60. Springer US, Boston, MA, 2011. ISBN 978-0-387-09766-4. doi: 10.1007/978-0-387-09766-4_77. URL `http://dx.doi.org/10.1007/978-0-387-09766-4_77`.

A Full Testing Results

A.1 Time and Speedup for 1-16 threads, $d = 1000$, $p = 0.5$

Threads	Time	Speedup
1	9s 187ms	1
2	4s 626ms	1.986
3	3s 109ms	2.955
4	2s 343ms	3.921
5	1s 892ms	4.856
6	1s 585ms	5.796
7	1s 369ms	6.711
8	1s 205ms	7.624
9	1s 81ms	8.499
10	979ms	9.384
11	902ms	10.185
12	834ms	11.016
13	774ms	11.87
14	731ms	12.568
15	686ms	13.392
16	647ms	14.199

A.2 Time and Speedup for 2-32 threads, $d = 1000$, $p = 0.5$

Threads	Time	Speedup
2	4s 624ms	1.987
4	2s 342ms	3.923
6	1s 591ms	5.774
8	1s 214ms	7.568
10	982ms	9.355
12	833ms	11.029
14	726ms	12.654
16	653ms	14.069
18	1s 103ms	8.329
20	1s 60ms	8.667
22	1s 126ms	8.159
24	1s 129ms	8.137
26	1s 103ms	8.329
28	1s 71ms	8.578
30	1s 53ms	8.725
32	1s 30ms	8.919

A.3 Dimensions against time, $p = 0.5$, $t = \{1, 8, 16\}$

Dimension	1 Thread	8 Threads	16 Threads
1000	9s 231ms	1s 207ms	646ms
2000	28s 13ms	3s 616ms	2s 7ms
3000	1m 1s 257ms	7s 896ms	4s 291ms
4000	2m 31s 143ms	19s 331ms	10s 843ms
5000	4m 29s 167ms	34s 413ms	19s 845ms
6000	5m 23s 100ms	41s 339ms	24s 290ms
7000	7m 29s 558ms	57s 778ms	32s 389ms
8000	(timed out)	1m 21s 480ms	47s 624ms
9000	(timed out)	2m 0s 608ms	1m 9s 36ms
10000	(timed out)	2m 16s 340ms	1m 15s 692ms

A.4 Precision against iterations and time, $d = 500$, $t = 16$

Precision	Time	Iterations
1	38ms	78
0.1	872ms	2160
0.01	25s 239ms	57755
0.001	1m 19s 879ms	172235
0.0001	2m 9s 480ms	288416
0.00001	3m 22s 586ms	404599
0.000001	4m 18s 886ms	520783
0.0000001	(timed out)	(timed out)