1 Code

1.1 Design

Designing the program to work sequentially would be a simple task, so the challenge quickly became to find a good way to let multiple threads work on the data simultaneously, such that the program could have a good parallel speedup. My first idea was to let each thread work on equal sections of the array, that is, if there are p threads and n^2 elements in the array, to let each thread handle $(n^2)/p$ elements. As the different elements in a C double array are stored in different memory locations, each thread can then work completely independently of the others within one iteration of the algorithm.

One of the first issues I noticed with this idea was that some threads would still be doing much more work than others. For example, if one thread is only handling elements on the edges of the array, all it has to do it copy them from the old array (1 read and 1 write) which is a trivial task compared to the averaging of adjacent elements (4 reads, some computation and 1 write). This will mean that the workload of the task is not being split evenly between the threads. However, the speed at which the threads run will vary from one machine to another, so there isn't a good way to predict the workload of each task and allocate elements to each thread accordingly. The only other solution would be to have threads dynamically change their workload - that is, threads that are free would search for unprocessed elements in the array during runtime and perform the appropriate calculations. However, this leads to more problems, mainly race conditions if two threads pick the same element to process, which could only be fixed with a large amount of locks (one for each element of the array). This may in fact lead to more parallel overhead than the original idea, depending on how fast the lock implementation is. In the end, I decided to stick with the original idea of giving each thread an equal number of elements, sacrificing some potential speedup for a much simpler and more portable program.

Another potential data race arises when we consider where the threads read from in each iteration. Threads must read values from the previous generation of the array in order to calculate the next generation, but after this, we need to update the pointer to the 'previous' array to such that the threads may start to work on the next iteration. This update must be done sequentially while all other threads wait, otherwise threads will read from different generations, causing incorrect results. Creating a separate 2D array for each generation is also not an option as there is no way to quickly calculate the number of iterations there will be before the program starts, not to mention the extreme memory requirements. It is clear that some degree of synchronisation is required, in particular the 'superstep' style of parallel programming described by the BSP model is a good fit, as each iteration can be seen as one superstep. To this end, I decided to use barriers to synchronise the threads; one at the start of each iteration and one at the end, with the sequential part of the program taking place between the end of one iteration and the start of the next.

1.2 Usage

The 'solve' function takes all the necessary parameters, and by default the program will generate an array of random elements (mod 20) to process. To compile the code, use:

gcc main.c -o main -lpthread

... and to run it, use:

./main [array dimension (int)] [thread count (int)] [precision (double)] [print (1 = yes, 0 = no]

2 Correctness Testing

2.1 Algorithm

Firstly, I tested the relaxation method algorithm on one thread, to ensure that there were no logical errors with how the result array was calculated. To do this, I started by printing the first two iterations (as well as the initial input) of a smalll 4x4 array, and also doing the calculations by hand in order to verify that they are correct. I decided to check two iterations in order to see that the result of the first is correctly used as the input of the second.

$$\begin{pmatrix} 12 & 0 & 2 & 10 \\ 1 & 18 & 3 & 12 \\ 4 & 8 & 13 & 3 \\ 5 & 18 & 14 & 9 \end{pmatrix} \begin{pmatrix} 12 & 0 & 2 & 10 \\ 1 & 3 & 11.25 & 12 \\ 4 & 13.25 & 7 & 3 \\ 5 & 18 & 14 & 9 \end{pmatrix} \begin{pmatrix} 12 & 0 & 2 & 10 \\ 1 & 6.375 & 6 & 12 \\ 4 & 8 & 10.375 & 3 \\ 5 & 18 & 14 & 9 \end{pmatrix}$$

Figure 1: In order from left to right, first three iterations

From left to right above, we have the input array, the first iteration, and the second iteration. We see that the elements along the edges stay constants, and the middle values line up with the correct calculations. To see that the algorithm stops at the right time, I looked at the final three iterations:

$$\begin{pmatrix} 12 & 0 & 2 & 10 \\ 1 & 4.009277 & 6.498047 & 12 \\ 4 & 8.498047 & 8.009277 & 3 \\ 5 & 18 & 14 & 9 \end{pmatrix} \begin{pmatrix} 12 & 0 & 2 & 10 \\ 1 & 3.999023 & 6.504639 & 12 \\ 4 & 8.504639 & 7.999023 & 3 \\ 5 & 18 & 14 & 9 \end{pmatrix} \begin{pmatrix} 12 & 0 & 2 & 10 \\ 1 & 3.999023 & 6.504639 & 12 \\ 4 & 8.504639 & 7.999023 & 3 \\ 5 & 18 & 14 & 9 \end{pmatrix}$$

Figure 2: In order from left to right, last three iterations

In this case, the precision was 0.01, and we can see that the algorithm terminated in the right place. At first, 4.009277 is changed to 3.99023, which is a difference of 0.010254 (more than the precision) and the algorithm correctly decides to do another iteration. This final

iteration ends up identical (truncated to six decimal places) so no change more than 0.01 occurs, and the algorithm terminates.

I repeated this process for a number of randomly generated arrays, of different sizes and with different precision values to test that the logic of the algorithm works in a sequential setting.

2.2 Parallel Testing

I also performed tests to ensure that the algorithm works when ran in parallel. To do this, I compared the final results of program when ran on one thread and when ran on many threads. Also, to increase the chance of catching logical errors occuring due to sequential assumptions I may have made while programming, I made each thread sleep for a random duration before each iteration. As the threads work on their own disjoint sections of the array, and barriers are used to synchronize the threads at the start and end of each iteration, these sleeps were expected to make no difference to the result. In addition, I performed some tests on my PC and some on the Azure HPC system. With varying array sizes and precision values, tests were passed each time (see Appendix).

```
void* threadMain(void* arg){
        //...some code...

        do{

                //DEBUG: Add random sleep
                sleep(rand() % 10);

                //...rest of algorithm...

        }(while !finished);

        pthread_exit(0);
}
```

Figure 3: Abbreviated code, showing random sleep in each thread

2.3 Data Races

Race conditions can still be present despite all tests being positive, and are difficult to debug. I decided to look for these by statically analysing the code to see which variables could potentially cause critical regions, and check that the implemented solutions are sufficient. To find critical regions, all shared resources used by threads in the program should be considered.

The main loop of the program is split into a sequential section, where only the main thread is running, and a parallel section, where every other thread is processing a section

of the array. These sections are formed using two barrier waits: the threads all wait once before starting their iteration, and once when they're finished, which match up to the main thread's two waits, meaning that when the main thread is running the sequential code, all other threads are waiting, and when the other threads are running, the main thread is waiting.

```
do{
        //This part is ran sequentially
        resultArray = malloc(size*size*sizeof(double));
        if(resultArray == NULL){
                printf("ERROR: Couldn't allocate memory");
                return;
        }
        changed = false;

        pthread_barrier_wait(&barrier);

        //Other threads do one iteration

        pthread_barrier_wait(&barrier);

        //Back to sequential
        free(startArray);
        startArray = resultArray;

}while(changed);

finished = true;
pthread_barrier_wait(&barrier);
```

Figure 4: Sequential and parallel sections in main thread's loop

To avoid data races, we need to ensure that if a shared resource needs to be read and updated, the update takes place in the sequential section. This is true for the **startArray** and **finished** variables, which leaves two other variables to consider.

changed This bool variable is updated by the threads - however, it may only be updated one way: from false to true. This means that it is safe to let the threads update the variable in parallel (if a data race occurs where two threads update the value at the same time, it won't matter as they're both trying to set it to the same thing). The variable also needs to be read at the end of each iteration to decide whether to keep going, and this is done in the sequential part, such that no other threads may update it while it is being checked.

resultArray The program ensures that each thread writes to disjoint sections of the array (e.g. Thread 1 processes elements 1-8, Thread 2 processes 9-16). In C, different elements in a double array are stored in different memory locations, so no two threads will attempt to write to the same memory location, and no races will occur.

Thus, none of the variables of concern are both written to and read by threads in the parallel section. When both a read and write is required, for example in the case of the

'changed' variable, one of these is done in the sequential section of the code, protected by two barrier waits.

3 Scalability Investigation

Compared to the simple computational task of calculating averages, the overhead from creating, synchornising and joining threads should be very expensive, so it is expected to dominate for smaller problem sizes. To see this, I measured the average time of 3 runs for a 20x20 array with precision 0.0001 (ignoring the first result to account for VM spinup time):

Threads	Time	Speedup	Efficiency
1	0.028	-	-
4	0.051	0.549	0.137
40	0.334	0.084	0.002

We see that the small precision forces the program to go through many iterations, and because each time the averaging process takes as negligible amount of time, we are practically measuring the thread synchornisation process, which actually takes longer the more threads we have, leading to a dramatic slowdown.

This is obviously undesirable, but it is an effect that is unavoidable for the type of program I have written. To see this, assume that the time to synchronise the threads scales with the amount of threads, and call the total synchronisation time f. Then, for an nxn array, p threads, and a constant precision, we would expect a time scaling function to look something like:

$$Time \propto ((n^2)/p) + f(p)$$

As p tends to infinity, the first term approaches 0, while the second increases, meaning that no matter the size of the problem, there will always be a point where adding another processor increases the time taken by the program.

(Note that we are making a lot of simplifications here – in reality, some threads may be only copying values that are on the edge of the array, taking a lot less time than those which have to calculate averages.)

Gustafson's Law tells us that in order to get a better speedup, we should increase the problem size. For example, here are the measurements for a 500x500 array with the same precision (Measured parallel overhead is calculated by: $T_o = pT_p - T_s$):

Threads	Time (s)	Speedup	Efficiency	Measured Parallel Overhead (s)
1	9.069	-	-	-
2	4.825	1.880	0.940	0.581
4	2.669	3.398	0.849	1.607
8	1.916	4.733	0.592	6.259
12	1.708	5.310	0.442	11.427
16	1.874	4.839	0.302	20.483
20	1.920	4.723	0.236	29.331
40	3.159	3.001	0.075	117.291

As expected, we get some much better results, especially for smaller amounts of threads. Somewhere between 12 and 16 threads we start to see slowdown again; this is the point where the f(p) term overtakes $(n^2)/p$ as discussed earlier. To investigate this term a bit more deeply, we can look at the measured parallel overhead column, and see how it scales with the number of threads.

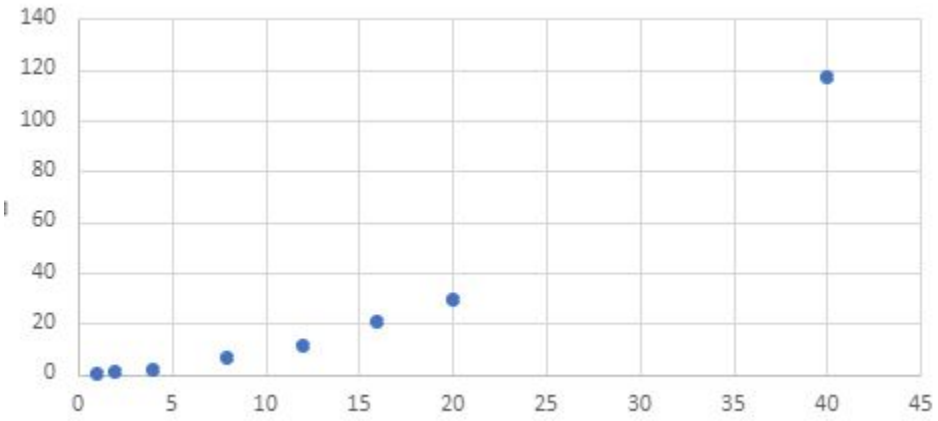

Figure 5: A scatter chart showing thread count on the x-axis, and parallel overhead on the y-axis.

The graph above seems to imply that the parallel overhead has a square correlation to the number of threads, and indeed we can see this by making some calculations: for example, $117.291/20.331 = 3.999 \approx 4 = (40/20)^2$. This coincides with the intuition that each thread is checking the status of each other thread while waiting at a barrier, so we should get something that scales with at least p^2. If we assume this relationship holds for other problem sizes and thread amounts, we can model $f(p) = kp^2$. Using this, we can get an estimate for the isoefficiency:

$$T_s = cT_o$$

$$n^2 = ckp^2$$

$$\text{n} = \text{o(p)}$$

These assumptions end up inducing a linear relationship between the dimension of the array and the number of threads, but note that this is not a fair measure of isoefficiency, as a better descriptor of the 'size of the problem' here would be n^2, the number of elements, so this would actually suggest a square isoefficiency, i.e. $n^2 = s \propto p^2$. To see if this is accurate, here are some more measurements:

Array Dimension	Threads	Time (s)	Efficiency
250	1	2.355	-
250	2	1.341	**0.878**
500	1	9.069	-
500	4	2.669	**0.849**
1000	1	36.141	-
1000	8	5.410	**0.835**
1500	1	82.919	-
1500	12	9.091	**0.760**

This appears to be roughly the case for the smaller problem sizes, however as we get to much higher numbers, the efficiency drops more than expected. This may be due to the assumptions simplifying the problem too much, and that there are other overheads which must be taken into account. It also could be due to the hardware architecture - as more threads are added, shared resources are being read more often, causing a memory bottleneck. Also, a 1500x1500 array of doubles will take up a lot of memory, and some of it may need to be stored further from the chip than in the smaller cases.

Lastly, we should also consider changing the precision value. For a 500x500 array, here is the data for various precisions:

Threads	Precision	Time (s)	Efficiency
1	0.0001	9.069	-
4	0.0001	2.669	0.849
16	0.0001	1.874	0.302
1	0.00001	89.170	-
4	0.00001	31.564	0.706
16	0.00001	28.431	0.196
1	0.000001	354.191	-
4	0.000001	224.576	0.394
16	0.000001	178.519	0.124

We see that decreasing the precision has a dramatic negative effect on the speedup and efficiency. This is to be expected (to some extent) as the lower the precision, the more iterations needed to copmplete the problem, and the threads are synchronised twice per iteration. This gives the program many more 'chances' to slow down, for example it is more likely that there will be a notably slow thread at some point, which will block the whole program for some time.

4 Appendix

4.1 Correctness Testing Evidence

To recreate these tests, change the random seed in main.c:main and run the executable with the given command line parameters (and with print=1 to show each iteration).

Seed	Threads	Dimension	Precision	System	Sleeps	Correct Avg.	Correct Stop
11037	1	4	0.01	PC	N	Y	Y
101121	1	6	0.001	PC	N	Y	Y
11037	1	4	0.01	PC	Y	Y	Y
101121	1	6	0.001	PC	Y	Y	Y
11037	4	4	0.01	PC	Y	Y	Y
101121	4	6	0.001	PC	Y	Y	Y
11037	1	4	0.01	Azure	N	Y	Y
101121	1	6	0.001	Azure	N	Y	Y
11037	40	4	0.01	Azure	Y	Y	Y
101121	40	6	0.001	Azure	Y	Y	Y

Example printed result - for second test:

[17.000000 9.000000 16.000000 1.000000 14.000000 4.000000
3.000000 8.926496 11.125012 8.613605 12.035614 13.000000
19.000000 12.581073 10.959798 10.293960 12.528757 18.000000
12.000000 11.437848 9.839414 9.073435 9.785619 12.000000
7.000000 11.331068 7.886333 6.375012 5.540132 5.000000
11.000000 19.000000 4.000000 3.000000 1.000000 1.000000
]