Shared Memory Parallel Coursework

Kyle Bignell - kb405@bath.ac.uk

November 24, 2014

Contents

1 Problem description

The problem requires the implementation of the relaxation technique on a two dimensional array of floating point values. This process is to be continued until the current values of the array differ from the values of the previous iteration by less than a defined precision. This is to be achieved with the use of shared memory parallelisation techniques.

In this problem space the relaxation technique involves replacing the value of each array cell with the average value of it's four adjacent neighbours.

2 Algorithm

My algorithm for the solution utilises the superstep paradigm. It alternates between sequential comparison of array values in the main thread and parallel processing of the relaxation technique on sections of the array.

At the beginning of each iteration of the main thread loop the main thread waits for the worker threads to finish the relaxation processing on their sections of the arrays. Once they are complete the main thread sequentially compares the previous and current states of the array. If the values differ by less than the specified precision then the program is complete. Otherwise a direction flag is flipped to indicate to the worker threads that the roles of the arrays should be swapped. The program then continues to the next iteration.

In order to evaluate the effectiveness of my algorithm I created both a sequential and parallel version. This report focuses on the parallel implementation. The sequential version is implemented using the same algorithm but without the use of threads.

2.1 Initialisation

In order to support the parallelisation of my solution I first initiliase the required data structures and parallel primitives along with data required for my algorithm.

2.1.1 Thread data structure

In order to facilitate communicate between the main and worker threads I have created a data structure which is passed to a worker thread upon it's creation. The structure is populated with the data required to communicate and synchronise between threads. The majority of the data within this structure consists of pointers which allows threads to communicate using shared memory. However, some values are specific to each thread and as such a structure needs to be generated for each thread.

The data in this structures includes:

- Pointer to first data array
- Pointer to second data array
- Pointer to direction flag
- Pointer to complete flag
- Pointer to serial complete barrier
- Pointer to parallel complete barrier
- Array dimension
- Lower x array bound
- Upper x array bound
- Lower y array bound
- Upper y array bound

The bounds in the data structure indicate which parts of the arrays a thread is responsible for working on.

2.1.2 Data arrays

Two square data arrays are intiliased with dimensions equal to the dimension specified at run time. These arrays are used to store the previous and current states of the answer. The cells in the top row and left column are set to a value of one with all other cells being set to a value of zero.

Two data arrays are utilised in order to simulate the relaxation technique being applied to all values in the array simultaneously. This is achieved by alternating between reading from one array and writing to the other. Once an iteration is complete the roles of the arrays are reversed. This allows the threads to operate on the arrays independently of each other as there will be no writing to the array that the threads are reading from. Threads are also assigned sections of the arrays to work on which ensures that two threads will never need to write to the same place in memory.

2.1.3 Direction flag

The direction flag is an integer that is used to indicate which of the two data arrays should be read from and which should be written to, i.e. which represents the previous and which represents the current state of the answer. This flag is flipped by the main thread if the desired precision has not been reached during an iteration. A pointer to this flag is passed to the worker threads in order to allow them to check it's value during their processsing.

2.1.4 Complete flag

The complete flag is an integer that is used to indicate whether the algorithm is complete, meaning that the difference between the two data arrays is less than the desired precision. A pointer to this is passed to the worker threads in order to ensure that they are aware when the process is complete and that they can therefore exit.

2.1.5 Serial complete barrier

This barrier is initiliased to wait for all worker threads and the main thread before breaking. This allows the worker threads to wait at this barrier until the main thread has completed it's sequential.

2.1.6 Parallel complete barrier

This barrier is initiliased to wait for all worker threads and the main thread before breaking. This allows the main thead to wait at this barrier until the worker theads have completed their parallel work.

2.1.7 Bound calculation

To simplify the process by which lower and upper bounds are assigned to a thread only the rows of the arrays are considered. This means that the lower and upper x bounds for a thread are always the same.

To begin, the standard row range allocation for each thread is calculated by dividing the the size of the active area of the array (two less than the dimension of the arrays) by the number of threads. The overflow, the number of rows left unassigned if only the standard range allocation is used for each thread, is calculated by using the modulo operation on the active are of the array by the number of threads.

During the initiliasation of each thread if there is any overflow remaining it is consumed by decrementing it's value by one and incrementing the upper y bound of the current thread by one. Due to the nature of the modulo operation we can be certain that the number of overflow rows will always be smaller than the number of threads. This ensures that all rows of the arrays are assigned to the threads and that the load is balanced as fairly as possible between them.

The only situation in which threads could be left idle would be with an array dimension that is less than the number of threads, not counting the inacitve edge regions. As it is unlikely that any jobs would be submitted on dimensions this small, or that they would be worthwhile, this solution represents a good trade off between complexity and results.

2.2 Main thread

During each iteration the main thread waits at the parallel complete barrier until all worker threads have completed the relaxation technique on their section of the arrays. The two arrays are then compared. A value is returned from this function and it is used to decide whether or not to continue iterating.

If the arrays differ by more than the desired precision the direction flag is flipped and the main thread then waits at the serial complete barrier. However, if the arrays do not differ by more than the precision then the complete flag is set to 1 and the main thread waits at the serial complete barrier. Upon passing this barrier if the complete flag is set to 0 then the program continues to the next iteration. However, if the complete flag has been set to 1 the main thread waits for the worker threads to finish and then exits.

2.3 Worker threads

Each worker thread loops while the the complete flag is equal to 0. In this loop the direction flag is used to decide which arrrays represent the previous and current states. The thread then uses the relaxation technique to calculate new values for the cells within it's lower and upper bounds. Once this is complete it waits at the parallel complete barrier so that the parallel threads can synchronise. Once this barrier has been passed the main thread completes it's sequential work to decide if the desired precision has been reached. The worker threads wait at the serial complete barrier until this is complete. After passing this barrier the worker threads check if the complete flag has been set to 1. If it has then they exit, otherwise they continue to the next iteration.

2.4 Array comparison

In order to compare the values in both arrays they are looped through and the values of corresponding cells are compared. If the values differ by more than the defined precision a value of 0 is returned. However, if no differences greater than the precision are found then a value of 1 is returned to indicate that the array values have settled within an acceptable precision.

3 Correctness testing

In order to test the corectness of my solution I have utilised a variety of techniques to help identify any errors. I can be confident that my solution is correct, however proving it's correctness is outside the scope of this report.

3.1 Algorithm walkthrough

My initial testing involved walking through my algorithm step by step. At each stage I would consider the state of the program and it's possible behaviours, trying to identify any areas in which errors could occur. This technique was essential in helping me to debug race conditions during development of the program.

3.2 Output comparison

After initial development was complete I moved onto testing my program in a range of configurations in order to expose any bugs or logical errors that were still present. After runs had been completed in these different configurations I would compare the output in order to ensure that they were the same. This technique helped me to understand how reliable my solution was. In theory the same results should be produced no matter the configuration, the only thing that changes should be the speed at which the results are produced.

I ran my program on LCPU (in both single core and multi core configurations), the Aquila master node, and in parallel on Aquila in order to expose it to a variety of different environments. I used a range of input arguments for the array dimension size, number of threads to use, and precision. The results of runs with matching input arguments, in different environments, were compared in order to ensure that there were no differences. I also included the results produced by the sequential version of my algorithm in order to provide more evidence that the same results were produced by the parallel version.

This technique helped to reassure me that the program was producing the same results across different environments. As I am confident about the correctness of the sequential version this comparison also increased my confidence in the correctness of the parallel version.

4 Results

My sequential and parallel solutions were run on the University of Bath computing cluster: Aquila. These are executed on a shared memory node with 8 cores. Note: 0 threads in my results indicates the sequential version of my solution. I have decided to use this value in order to make comparisons and the plotting of graphs straightforward.

These results were produced by executing the sequential and parallel versions to a precision of 0.01 on a square array with a dimension of 10,000 by 10,000.

4.1 Execution time

Threads	Total Time	Serial Time	Parallel Time
0	55.513606	55.513606	0
1	57.277079	2.139745	55.137334
2	29.650145	2.139429	27.510716
3	20.710814	2.136334	18.57448
4	16.11912	2.139989	13.979131
5	13.655371	2.14294	11.512431
6	12.015017	2.142486	9.872531
7	11.290612	2.142082	9.14853
8	10.22912	2.139137	8.089983

Table 1: Execution time in seconds

In the parallel version you can see that the time spent in the sequential section of the code is almost equal across all number threads. This is due to comparison of the arrays that always takes place sequentially. As this process is identical no matter how many threads that are being used it makes sense that it should always take approximately the same amount of time.

You can also see that the benefits of increasing the number of threads becomes smaller as the number of threads increases. The largest reduction in execution time occurs between the transition from 1 to 2 threads, with a reduction of approximately 18 seconds. However, when increasing from 7 to 8 threads we only see a reduction of approximately 1 second.

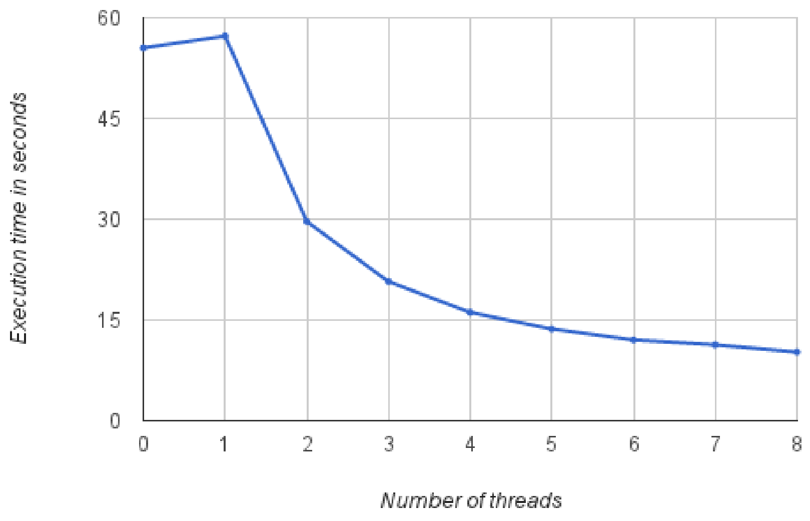

Figure 1: Execution time in seconds

The graph of this data makes it clear to see that the reduction of execution time using parallelism is greatly reduced after 4 threads. The increase in execution time between the sequential version (0 threads) and the parallel version using 1 thread is also evident. This is because both programs are doing all the work sequentially but the parallel version has the added overhead of creating threads and the required synchronisation between them.

I believe that the execution time of the parallel version would begin to increase again past any number of threads greater than 8. This is due to the architecture of the Aquila nodes; as they only have 8 cores available any number of threads greater than the number of cores would require additonal scheduling by the operating system to switch between them. As such the execution time of the program would increase with the number of threads past 8 due to the scheduling required to execute them on the limited number of cores.

4.2 Speedup

$$\text{Speedup} = \frac{\text{Time on a sequential processor}}{\text{Time on n parallel processors}}$$

Speedup is a measure of how much faster the program is running than the sequential version. A value of 1 means that the program is running at a speed equal to that of the sequential version. A value less than 1 means that it is running slower and a value greater than 1 means that it is running faster.

Threads	Speedup
0	1
1	0.969211541
2	1.872287842
3	2.680416424
4	3.443960092
5	4.065331217
6	4.620351848
7	4.91679335
8	5.427016791

Table 2: Speedup

The use of execution time data in the calculation of speedup allows us to spot similar patterns to those seen in the execution time results. The speedup value for parallel version with 1 thread is lower than 1 due to the sequential execution and additional overhead compared to the purely sequential version. This is clearly linked to the slower execution time of the single thread parallel version seen in the execution time data.

As to be expected the speedup value increases as the number of threads increases but does not exceed the number of threads in use. However, I expect that the speedup value will decrease relative to the number of threads past the number of available cores. This is similar to the expected increase in execution time for any number of threads past the number of available cores, as the additonal overhead of scheduling threads will result in longer execution times and therefore lower speedup values.

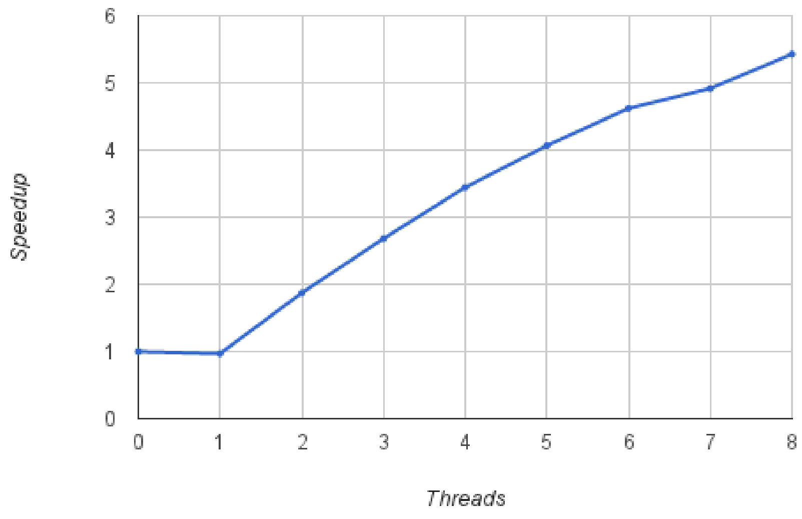

Figure 2: Speedup

4.3 Efficiency

$$\text{Efficiency} = \frac{\text{Speedup}}{\text{Number of threads}}$$

Efficiency is a measure of how many of the available clock cycles our program is using. A value of 1 indicates that all available cycles are in use whereas 0 indicates that none are being used.

Threads	Efficiency
0	1
1	0.969211541
2	0.936143921
3	0.8934721413
4	0.860990023
5	0.8130662433
6	0.7700586413
7	0.70239905
8	0.6783770989

Table 3: Efficiency

The data produced by the efficiency calculations fits the expected downward curve for an increasing number of threads. The reduced efficiency in this program is mainly due to the superstep paradigm. As we increase the number of threads that the program uses we are also increasing the number of threads that are not doing anything while waiting for the main thread to finish it's sequential comparison and flag setting. This results in increasing number of wasted CPU cycles for each additional thread.

The synchronisation between threads also plays a role in the reduced efficiency of the program. As threads must wait at the parallel complete barrier until all threads have completed their work it is possible for some of the threads to be waiting on others to finish. Coupled with the fact that threads can have small differences in the number of rows they are assigned to work on, due to the total number of rows not being cleanly divisible by the number of threads, it is possible that these threads will be holding up the others while they complete their work on the additional rows.

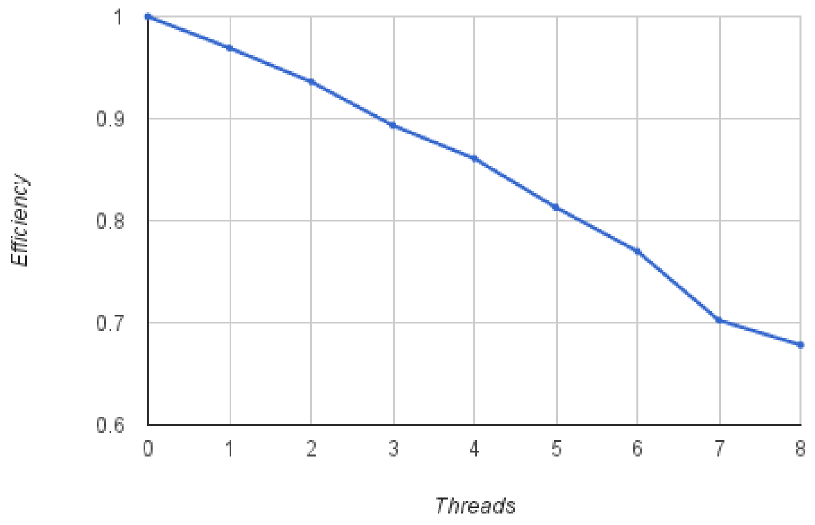

Figure 3: Efficiency

4.4 Sequential ratio

$$\text{Sequential ratio} = \frac{\frac{1}{\text{Speedup}} - \frac{1}{\text{Number of threads}}}{1 - \frac{1}{\text{Number of threads}}}$$

The sequential ratio is a measure of what ratio of the program is being executed sequentially. Ideally this ratio will be as small as possible as that means there will be a larger section of the program that we can execute faster with the use of parallelisation techniques.

Threads	Sequential Ratio
0	1
1	1
2	0.0682118182
3	0.05961453846
4	0.05381788145
5	0.05747802169
6	0.05972048006
7	0.07061535509
8	0.06772948085

Table 4: Sequential ratio

The parallel version executed with a single worker thread has a sequential ratio of 1 due to the fact that it is executed in a sequential fashion. Taking an average of the other values gives us an approximate sequential ratio of approximately 6%, meaning that the program is approximately 94% parallel.

These values are plausible due to the majority of the program's work being in the relaxation technique which is executed in parallel. The majority of the sequential work is in the comparison of the previous and current arrays. We are able to tell quite early on in comparison function if the arrays differ by values greater than the precision due to the fact that largest concentration of changing values are in the areas we check first, i.e. the first rows and columns. This means that the program is able to spend the minimum amount of time in the sequential sections and return control back to the parallel threads.

4.5 Maximum speedup

$$\text{Amdahl's law} = \frac{1}{\text{Sequential ratio} + \frac{\text{1 - Sequential ratio}}{\text{Number of threads}}}$$

Amdahl's law is used to find the maximum theoretically possible speedup for an algorithm using a given number of threads/cores. In order to calculate this you need to know the sequential ratio of the algorithm, which I calculated in the previous section to be approximately 6%.

For example the maximum possible speedup for my solution using 8 threads can be calculated using this equation:

$$\text{Speedup} = \frac{1}{0.06 + \frac{0.94}{8}}$$

This produces an answer of 5.63380802817. The actual speedup achieved by my algorithm on 8 threads was 5.427016791, which is close to the theoretically maximum speedup. This is most likely lower due to the scheduling and synchronisation overheads required to support the parallelism in the solution.

We can also calculate the maximum theoretically possible speedup for this algorithm by setting the number of cores/threads to an infinite value. This would be equivalent to us using an inifinite amount of computing power to run the solution. With the number of cores/threads set to inifinity in the previous formula we arrive at a new formula that will provide us with the maximum possible speedup:

$$\text{Maximum speedup} = \frac{1}{\text{Sequential ratio}}$$

Solving this with a sequential ratio of 0.06 results in a mximum theoretical speedup of 16.66. This will never likely be achievable in the real world however due to additonal overheads that come with synchronising an inifinte amount of threads and the technology required to allow concurrent shared memory access for an infinite amount of cores.

4.6 Parallel overhead

Parallel overhead = (Number of threads * Time in parallel) - Time in sequential

Parallel overhead is a measure of how much time is spent in work that is required to support parallelism. That is, work that does not directly contribute to calculating the final answer.

Threads	Parallel overhead
0	0
1	1.7634731
2	3.786684
3	6.618836
4	8.962874
5	12.763249
6	16.576496
7	23.520678
8	26.319354

Table 5: Parallel overhead

As you can see from the data the parallel overhead has a greater than linear growth as the number of threads increases. This is most likely due to the overheads of creating the additional threads and the synchronisation between an increasing number of threads that is required to achieve the superstep design.

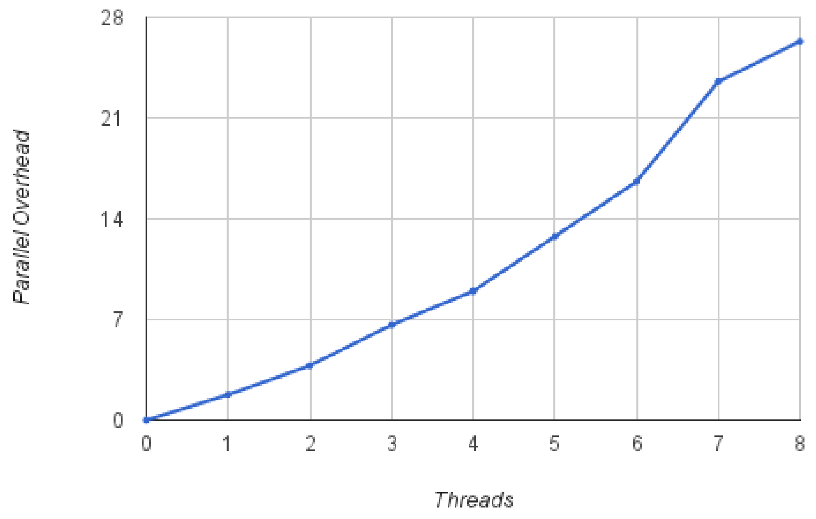

Figure 4: Parallel overhead

5 Conclusion

In summary, I believe my algorithm provides a valuable solution to the parallelisation of the relaxation technique with the use of the superstep paradigm. It achieves almost maximum theoretical speedup at 8 threads calculated from Amdahl's law. I believe the simplicity of my algorithm contributes to the small parallel overheads and therefore this close to maximum speedup. If I were to improve the algorithm I would focus on parallelising the comparison function so that each thread can complete the comparison of it's section of the array. This should lead to further speedup as the sequential ratio of the resulting program should be greatly reduced.

My algorothm does suffer from reduced efficiency as the number of threads increases. This is due to the use of the superstep paradigm. When the main thread is doing it's sequential comparison of the two arrays the other cores are sitting idle, wasting clock cycles. Again, updating the algorithm to do the array comparison in parallel would go a long way to increaing the overall efficiency of the algorithm.

Parallel overhead increases with the number of threads as expected. While this time is relatively small per thread it does have greater than linear growth. I believe this would be the biggest issue when considering the scalability of my solution. As the number of threads increased the solution would pass a point where it would spend more time dealing with the parallelisation overheads than doing useful work. My suggested improvements to the algorithm should help to reduce the parallel overhead as the threads would not have to hand control back to the main thread,. They would instead always be working in parallel until all threads have reached a desireable level of precision. This should greatly reduced the amount of parallel overhead.

I decided to use up to 8 worker threads instead of 7 alongside the main thread, for a total of 9 threads. This was to ensure that during the parallel section of the program all 8 cores were being utilised. While this resulted in a slight increase in scheduling overhead, due to the swap required to put one of the worker threads onto the core that had the main thread on, it did result in a higher efficiency as seen in the efficiency graph. This is because all 8 were being utilised during the parallel section of the program instead of having a core sitting idle in the main thread waiting for the parallel work to complete.

Overall, my algorithm in it's current state clearly demonstrates the advantages of parallelising the relaxation technique as seen in the speedup results. However, it does suffer from decreaing efficiency and increasing parallel overhead as the number of threads increases. Any number of threads greater than 8 on Aquila will not result in a greater speedup due to the number of cores available per node. My suggested improvements to the algorithm should help alleviate these problems and produce an algorithm that is more scalable.