CM30225 – Distributed Memory Coursework

Approach

I initially started this piece of coursework by creating a sequential version of the program in C. This was done in order to gain a deeper understanding of the problem and how it may be solved. This code would become useful later in two ways: for one, I could reuse key parts of this code with minimal changes (such as the function for finding the greatest difference between 2 arrays, and the main relaxation function which average elements); and I was able to compare it against my parallelised code.

The central idea behind the distributed version of the program was to split the main relaxation array into a number of segments, each of which could be sent to one CPU, where the algorithm could be carried out individually, averaging all elements. So, for example, running the code with 11 processors and a 100 x 100 array would lead to 10 processors responsible for averaging values in a 10 x 100 array. Of course, there is one unused processor here, and that's because I used a master-slave approach, whereby one processor (the master) is responsible for dividing up the work, and allocating it to each other (slave) processor. Every slave processor would send the greatest change in its array segment to the master node every iteration, and once the maximum change was below the given precision, each slave processor would send the full array to the master node, and this could be rejoined to produce the complete, relaxed array.

This way I could fairly naturally use a divide and conquer approach, which in theory should be able to drastically speed up runtime. However, a naïve approach would run into a problem, as by simply splitting the array into segments and averaging the elements repeatedly, the elements in the border arrays (first and last array in each segment) wouldn't have their average calculated properly, as one of the necessary values would be in a different segment on a different processor. To make it work properly, every segment would have one extra array at the top and bottom (if there is an extra array in this direction), and every iteration would send the arrays second from the top and second from the bottom (as these are the outermost arrays that have their averages calculated properly) to the master processor, which would then send new border arrays to each processor. Through this method, I was able to attain correct results.

In distributed code, it is important to minimise the amount of communication between processors, as this can be very expensive, so by sending only the correct border arrays every iteration, I aimed to solve the problem using the minimum amount of communication possible, even if this resulted in more computation from each processor, working out which arrays to send.

I aimed to prevent deadlocks and race conditions by using more sophisticated MPI structures, for example using MPI_Gather and MPI_Reduce, rather than simply using MPI_Send and MPI_Recv. This way, I can simply rely on the implementation of these functions to be correct, rather than using my (potentially incorrect) logic. On top of this, I was likely to see an increase in performance, due to the fact that the more complex functions likely have more efficient communication than using many simple ones.

Testing

To test the code, I could initially simply hand calculate the results of individual elements after each iteration, and compare this against the output my code gave. However, as the size of the arrays I was using increased, this very quickly became infeasible. So instead, I compared the final results against the results of my sequential code. My sequential code was much simpler than my parallel code, and I was able to test many individual elements, finding no

issues with the results of any iterations. On top of this, it inherently avoided parallel specific issues, such as race conditions.

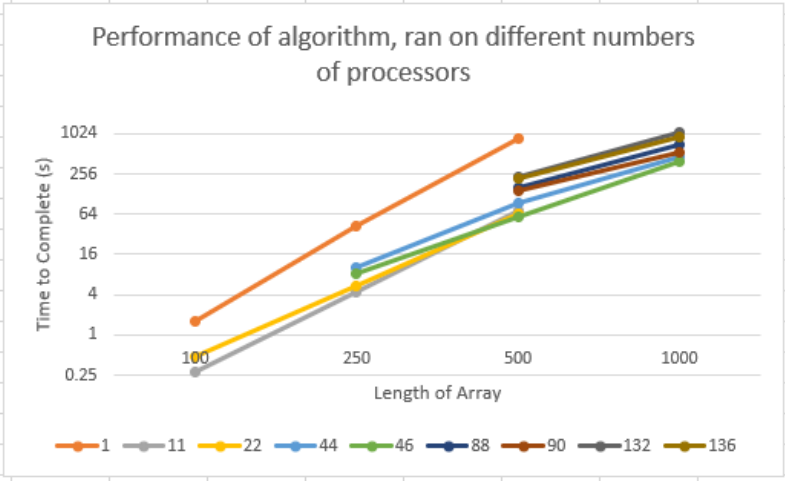
By comparing these final results against the results of my parallel code, and also checking the number of iterations that each piece of code went through to generate these results, I was left confident that both variations of the program were producing correct results. I checked this against multiple array sizes and number of processors, to ensure that there is no issue with the central logic.

Analysis

To test the scalability of the program, I added a timer to the various versions I made, and ran them each on the cluster provided. All code was run for a precision of 0.01, and there were no significant outliers in my results.

Note that smaller arrays are omitted for larger numbers of processors, as it is impossible to divide a 2D array so that less than three arrays are assigned to each processor under my implementation. Any missing values for larger arrays simply mean that the code was unable to complete processing in under the 20 minutes allowed for a job on the cluster we could use.

Number of Processors	1	11	22	44	46	88	90	132	136	172
Array size (x by x)										
100	1.64	0.28	0.47							
250	41.96	4.3	5.39	10.36	8.1					
500	864.51	72.33	63.37	93.27	59.36	162.67	144.18	236.09	219.88	
1000				454.92	395.63	704.45	549.92	1078.84	904.07	



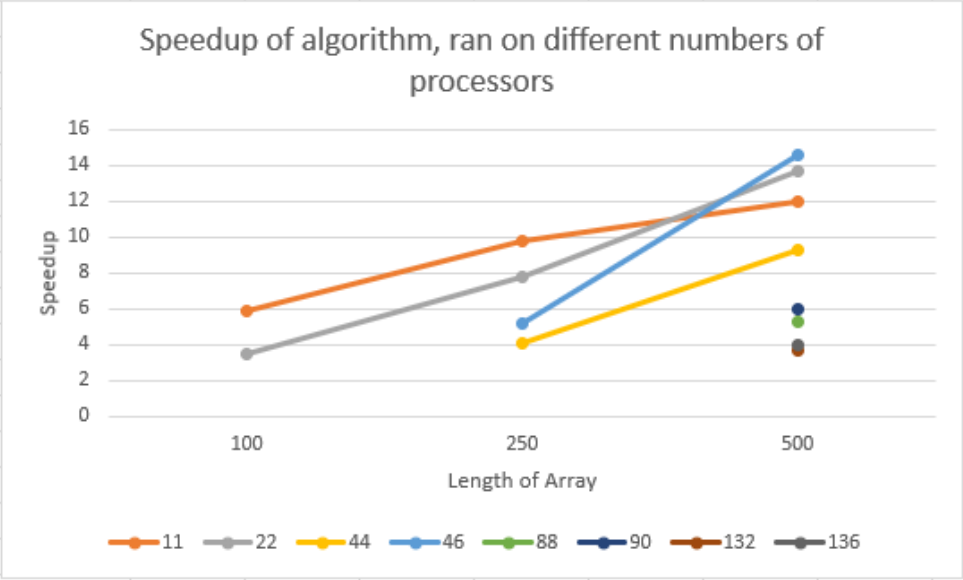
As you can see here, the parallel version of the code performed better than the sequential code for every number of processors, showing that my solution has given speedup of over 1 for every number of processors. (Although 172 processors could only run on a 1000 x 1000 array here, and even then, couldn't finish it in under 1200 seconds. Although extrapolating from the existing data, it is very likely that it would still be much faster than the sequential code.) One thing that you can see is that for each array size, adding more processors seems to make the code run faster to a point, after which more processors produced slower results. This is likely due to the communications overhead becoming more significant than the increased computational power, as you add more processors, there are more segments, and they each become smaller, so less processing takes place in each segment, but more communication does.

One thing that's interesting to note is that increasing from the maximum number of processors in one node to using one more node with only a small number of processors more (e.g. 44 processors on one node vs 46 processors over two nodes) seems to increase the speed of the code. This goes against my expectations that increasing the number of nodes would lead to slower communication due to more inter-node communication. This may be due to the fact that adding another node assigns the job more memory, and inter-node communication must not be as slow as I thought. It's also worth considering that my implementation assigns any arrays that do not divide evenly by the number of processors to the final processor, so numbers that divide with a smaller remainder should be faster, as the

code is limited by the slowest processor (due to this reason, often the last one). I could improve efficiency in the future, by changing this to assigning the segments more evenly.

I then calculated the speedup of each configuration, in order to look at the scalability of the algorithm. I calculated the speedup by dividing the time it took to run the code sequentially, vs the time it took to run the code on a given number of processors.

	Number of Processors	11	22	44	46	88	90	132	136
Array size (x by x)									
100		5.857143	3.489362						
250		9.75814	7.784787	4.050193	5.180247				
500		11.9523	13.64226	9.268897	14.56385	5.314502	5.996047	3.661782	3.931735

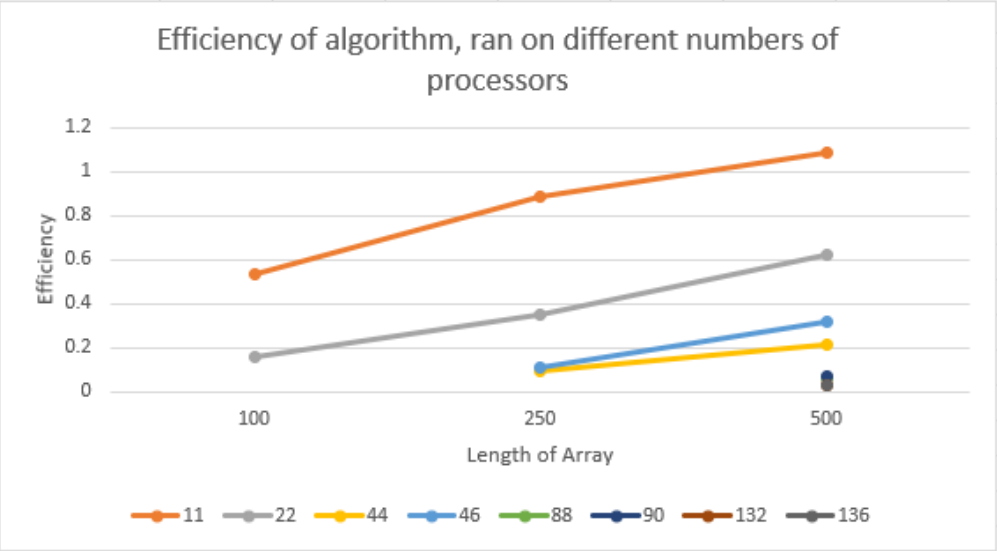


Here it becomes increasingly clear that in order to achieve fast performance, you need to find the best number of processors to use depending on the size of your problem. Too few and they won't be able to process the data quickly enough, too many and the cost of communication will overshadow the gains in computational power.

In one instance, in the case of an 11-processor configuration on a 500 x 500 array, we seem to achieve superlinear speedup, violating Amdahl's law (which states that every program has a maximum speedup, and it cannot exceed the number of processors being used)! However, as is mostly the case, this is most likely due to minor differences in the algorithms, the sequential code used dynamic memory allocation instead of working statically, which may cause slightly slower performance. On the other hand, we do seem to show Gustafson's law (which states that for larger problems, potential speedup is larger) in action. The greatest speedup we get for a 500 x 500 array (14.56) is greater than the greatest speedup for a 250 x 250 one (9.76), which is in turn greater than the greatest speedup for a 100 x 100 one (5.86).

Another interesting metric to consider is the efficiency of each configuration. I calculated this by dividing the speedup by the number of processors being used.

	Number of Processors	11	22	44	46	88	90	132	136
Array size (x by x)									
100		0.532468	0.158607						
250		0.887104	0.353854	0.09205	0.112614				
500		1.086573	0.620103	0.210657	0.316605	0.060392	0.066623	0.027741	0.02891



From this we can see the increasing number of processors generally leads to a lower efficiency. This is in fact very common in parallel systems, due to the fact that increasing the number of processors will likely not increase efficiency proportionally, but will likely increase the communication overhead. Unfortunately, it is unclear at what point the efficiency caps out for each configuration, as none of them show a decrease in efficiency when moving to a larger array size, to investigate this further we would need to run the code on larger arrays, where we would likely see better results, especially for higher numbers of processors.

Conclusion

In conclusion, I believe that I have been successful in creating a distributed memory solution to the problem provided, and in using these results to observe the scalability of the algorithm. It would be interesting in future to run the code on a larger scale, using more processors and being able to run the code for longer, as I'm sure the results would show further efficiency.