Distributed Memory Parallel Coursework

Kyle Bignell - kb405@bath.ac.uk

December 31, 2014

Contents

1 Problem description

This work requires the implementation of the relaxation technique on a two dimensional array of floating point values. This relaxation is to be iterated until the current values of the array differ from the values of the previous state by less than a defined precision. The solution is to be developed with the use of distributed memory and message passing interface (MPI) parallelisation techniques.

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

2 Algorithm

The algorithm I designed utilises the parallel processes available in distributed memory systems by dividing the array into overlapping chunks that can be worked on by one of these processes. It has similarities to the superstep paradigm due to the alternating message passing and relaxation steps. The messages are responsible for transferring the overlapping boundary values of the sub-arrays to neighbouring processes in order to ensure that each sub-array is working with correct values.

At the start of each iteration, relaxation takes place on the sub-array that a process is responsible for. Each process then sends the second lowest row of its array to the process with the rank one lower than its own, this becomes the highest row of the process that receives it. The second highest row of the array is then sent to the process with the rank one higher than its own, this row becomes the lowest row of the process that recieves it. This achieves the overlapping effect that is required to ensure that each sub-array is working with the correct boundary values during each iteration. Each process then compares the current state of its array with the previous state in order to check if they have differed by more than the defined precision. If they have not the process is complete and the program exits, otherwise all processes continues to the next iteration.

2.1 Sub-dividing the array

Each process is responsible for calculating which rows of the array it has to work on. This is done in order to eliminate the need for a coordinating process and the required messaging.

To begin, the standard row allocation for each process is calculated by dividing the the size of the active area of the array by the number of processes. The active area of the array being two less than the dimension to account for the edge rows that never change. The row overflow, the number of rows left

unassigned if only the standard row allocation is used, is calculated by using the modulo operation on the active area of the array by the number of processes.

If there is any overflow during the assigningment of rows to each process it is consumed by decrementing the value by one and incrementing the upper row of the current process 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 processes. This ensures that all rows of the array are assigned to the processes and that the load is balanced as evenly as possible between them. When assigning the rows to a process the upper and lower rows are incemented and decremented by one respectively in order to achieve the overlapping required to ensure that the sub arrays are able to affect one another.

The only situation in which processes could be left idle would be with an array dimension that is less than the number of processes, not counting the inacitve edge regions. This solution represents a good trade off between complexity and correctness as it is unlikely that any jobs would be submitted with dimensions that small.

2.2 Data arrays

Each process creates two square arrays with dimensions equal to the number of rows determined in the row allocation step. These arrays are used to store the previous and current states of the answer. Once an iteration is complete the roles of the arrays are reversed. This allows the processes to easily compare the current and previous states to determine if the desired precision has been met.

2.3 Relaxation

In order to complete an iteration of relaxation the program loops through each cell in the previous state array. The values of the four adjacent cells are averaged, that value is then stored in the cell corresponding to the currently cell in the current state array. This repeats until all cells in the current state array have been updated with their new values.

2.4 Message passing

After the relaxation step has been completed message passing is used to send the updated values of a sub-array to processes with neighbouring sub-arrays. Basic send and receive functions are used to send rows of the current state array of a process to another process. This ensures that each process has up to date boundary values that correctly reflect the current state of the neighbouring sub-arrays.

2.5 Array comparison

In order to compare the values of both arrays they are looped through and the values of corresponding cells are compared. If the difference is greater than the defined precision then the comparison fails. However, if no differences greater than the precision is found then the comparison succeeds. The result of this comparison is used to decide whether the algorithm should continue iterating or not.

2.6 Completion checking

After each process has completed its comparison the process with rank zero calculates whether all processes have finished by using the MPI reduce function. It uses the 'logical and' argument to reduce the finished flag of all processes into a single value. This value is then broadcast back to all processes in order to let them know if they should continue iterating or not.

3 Correctness testing

In order to evaluate the effectiveness of my algorithm I created a sequential version in order to conduct comparisons. The sequential version is implemented using the same relaxation technique but it runs on a single process and as such does not use message passing.

3.1 Algorithm walkthrough

My initial testing involved walking through my algorithm sequentially. At each stage I would consider the state of the program and it's possible behaviour, trying to identify any areas in which errors could occur. This technique was essential as it was my initial debugging proccess for problems I faced during development.

3.2 Output comparison

After development was complete I tested my program in a range of configurations in an attempt expose any bugs or logical errors that were still present. I used a range of input arguments for the array dimensions, number of processes to use, and the desired precision. After these tests were complete I compared the results in order to check for differences. This technique helped me to determine how reliable my solution was. In theory the same results should always be produced no matter the configuration or if it was the serial or parallel version that was run. The only difference should be the speed at which the results were produced. As the results were consistent across all my tests I am confident that my solution is correct for all standard configurations on the Aquila cluster.

This testing helped to reassure me that the program was producing the correct results no matter the environment, configuration, or program version. My confidence in the parallel version was greatly increased when its results matched those of the sequential version.

4 Results

I ran my sequential and parallel programs on the University of Bath computing cluster: Aquila. The parallel version was run using distributed memory techniques, MPI in this case, across four cluster nodes. Each of these nodes has 8 cores meaning there were up to 32 processes executing in parallel. Note: 0 processes 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 more straightforward.

4.1 Execution time

Processes	1000	5000	7500	10000
0	5.267	130.623	294.422	523.031
1	5.363	130.785	294.941	522.869
2	4.146	77.251	170.505	301.895
4	2.573	40.855	90.292	158.831
8	1.925	27.454	60.27	105.643
12	1.816	19	41.015	71.187
16	1.69	14.898	31.691	54.164
24	1.629	10.871	22.46	37.354
32	1.714	9.154	17.882	29.261

Table 1: Execution time in seconds on multiple square arrays of different dimensions to a precision of 0.001

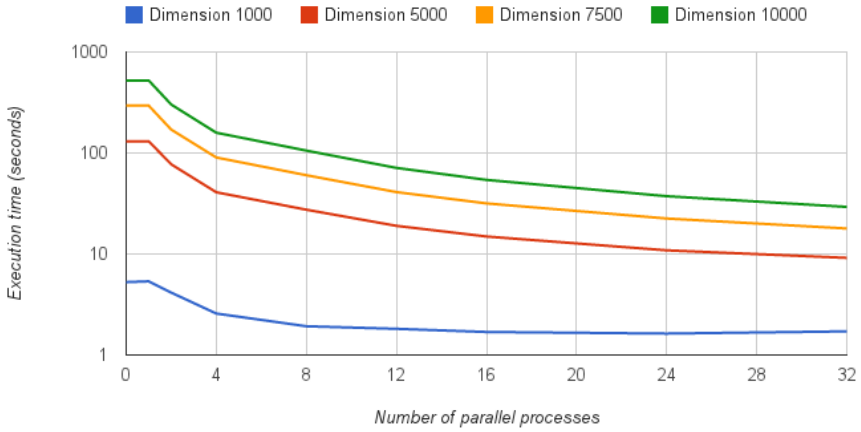

Figure 1: Execution time in seconds on multiple square arrays of different dimensions to a precision of 0.001

Processes	0.01	0.005	0.001	0.0005
0	13.971	26.977	131.044	245.966
1	14.042	27.04	130.782	245.637
2	9.334	16.846	76.495	155.494
4	5.424	9.332	40.819	83.108
8	4.107	6.666	27.417	55.413
12	3.234	4.972	19.02	38.681
16	2.850	4.138	14.842	30.34
24	2.417	3.359	10.872	22.592
32	3.382	3.007	9.075	18.6

Table 2: Execution time in seconds on a square array with dimensions of 5000 to varying degrees of precision

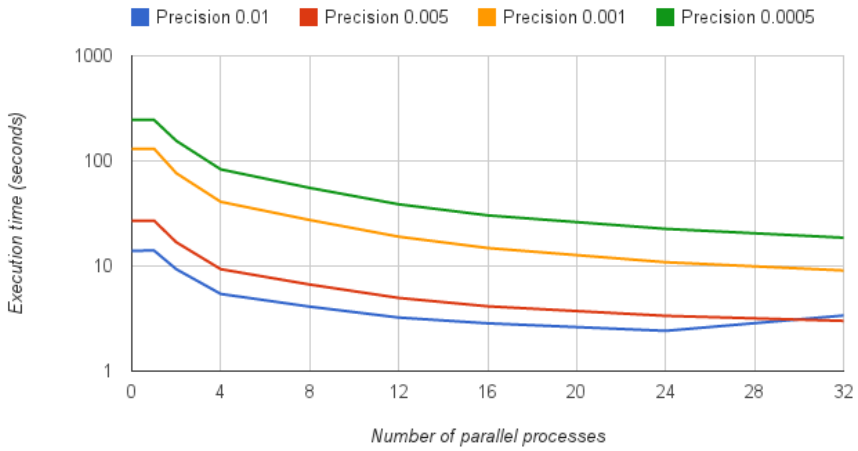

Figure 2: Execution time in seconds on a square array with dimensions of 5000 to varying degrees of precision

You can see from the data that the sequential and parallel version running with only a single process have almost identical execution time. This is due to the fact that no message passing takes place when there is only a single process. As such the programs behave almost identically apart from the initialisation and termination of the MPI library.

With 32 processes you can see an increase in execution time compared to 24 processes in both the dimension 1000 and precision 0.01 data. This is clearly evident in the precision scaling graph. I believe this is due to the relative simplicity of the problems and as such a greater percentage of time is spent in the messaging and coordination of the processes which results in slower execution times. This supports Gustafson's theory that more computing power attracts larger problems, you can see that with larger array dimensions and higher precisions we continue to see improved execution times with 32 processes.

It is clear to see that the returns of using more processes becomes smaller as the number of processes increases. We see the biggest performance improvements per process happen when moving from 1 to 2 processes. This is because when we move from 1 process to 2 we see the biggest change in problem size. We still see performance improvements but they are not as large as the initial change.

You can see from the graphs that the decrease in execution time becomes very small as the number of processes becomes large. As such it is clear to reason that the execution time will never drop below a lower bound. This supports Amdahl's law that a problem has an upper limit to speedup no matter how

much computing power you use.

Increasing the dimension of the problem results in much greater execution times than an increase in precision. This is due to an increase in dimension size adding both rows and columns to the problem. This increases the time it takes to both relax and compare the arrays. Increasing the precision on the other hand means that more iterations are required before the desired precision is met.

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.

Processes	1000	5000	7500	10000
0	1	1	1	1
1	0.982	0.999	0.999	0.997
2	1.270	1.691	1.727	1.727
4	2.047	3.197	3.261	3.283
8	2.736	4.758	4.885	4.936
12	2.900	6.875	7.178	7.325
16	3.117	8.768	9.290	9.627
24	3.233	12.016	13.109	13.960
32	3.073	14.269	16.465	17.821

Table 3: Algorithm speedup on multiple square arrays of different dimensions to a precision of 0.001

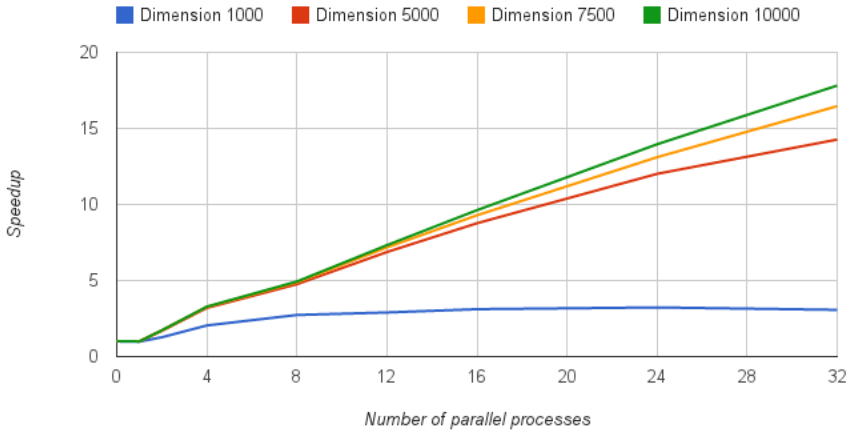

Figure 3: Algorithm speedup on multiple square arrays of different dimensions to a precision of 0.001

Processes	0.01	0.05	0.001	0.005
0	1	1	1	1
1	0.995	0.998	0.999	0.999
2	1.497	1.601	1.708	1.578
4	2.576	2.891	3.200	2.952
8	3.402	4.047	4.764	4.428
12	4.320	5.426	6.868	6.343
16	4.902	6.519	8.801	8.087
24	5.780	8.031	12.015	10.860
32	4.131	8.971	14.394	13.191

Table 4: Algorithm speedup on a square array with dimensions of 5000 to varying degrees of precision

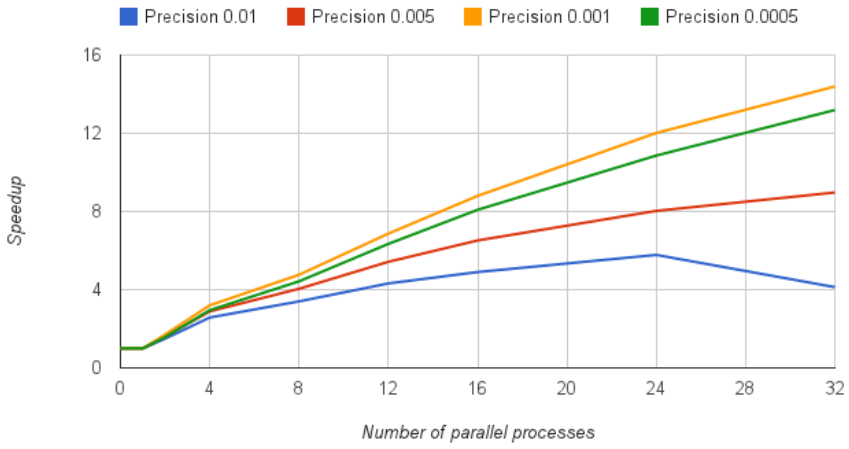

Figure 4: Algorithm speedup on a square array with dimensions of 5000 to varying degrees of precision

As mentioned previously we can see a reduction in speedup when using 32 processes on the dimension 1000 and precision of 0.01 problems. This is clearly evident in the precision scaling graph with the sharp decline in speedup after 24 processes. As speedup is calculated from execution times it is obvious that this pattern would appear in this data as well.

It is interesting that we see very little speedup for the dimension 1000 problem. This is further support of Gustafson's theory as we see better speedup when running larger problems.

We see a speedup value lower than 1 when using a single process for the parallel version. This is due to only using the same number of processes as the sequential version but having the additional overhead of initialising the MPI libraries and the setup code that is designed to the parallelisation of the problem.

We see the typical increase in speedup as the number of processes increase, apart from the previously mentioned cases. I believe that we can achieve even greater speedup with the use of more processes as the majority of messaging between processes happens in parallel. It is only the messaging between the processes that puts a limit on the speedup of the solution and the majority of it occurs between a process and its neighbours. It is only the reduce and broadcast messaging that must communicate with all the processes simultaneously but as it is only 2 messaging functions that are designed for communicating with a large amount of processes it should not have too much of an effect on the scalability of the solution. You can see from the dimension scaling data and graphs that the larger problems have an almost linear increase in speedup as

the number of processes increases.

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.

Processes	1000	5000	7500	10000
0	1	1	1	1
1	0.982	0.999	0.999	0.997
2	0.635	0.845	0.863	0.864
4	0.512	0.799	0.815	0.821
8	0.342	0.595	0.611	0.617
12	0.242	0.573	0.598	0.610
16	0.195	0.548	0.581	0.602
24	0.135	0.501	0.546	0.582
32	0.096	0.446	0.515	0.557

Table 5: Algorithm efficiency on multiple square arrays of different dimensions to a precision of 0.001

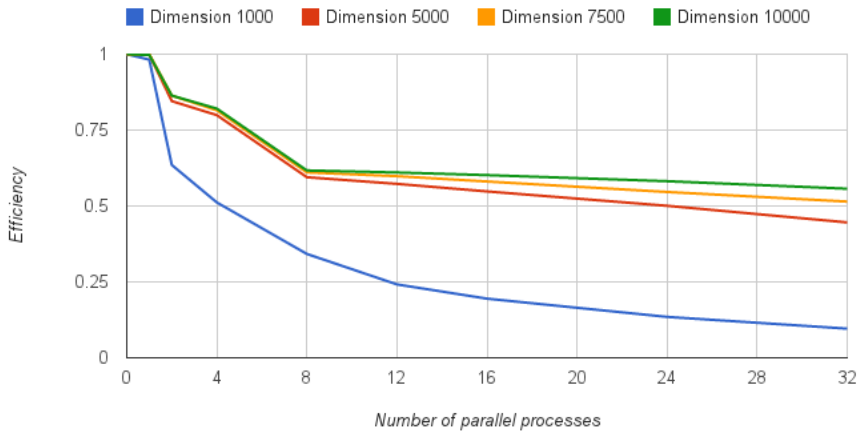

Figure 5: Algorithm efficiency on multiple square arrays of different dimensions to a precision of 0.001

Processes	0.01	0.05	0.001	0.005
0	1	1	1	1
1	0.995	0.998	0.999	0.999
2	0.748	0.801	0.854	0.789
4	0.644	0.723	0.800	0.738
8	0.425	0.506	0.596	0.553
12	0.360	0.452	0.572	0.529
16	0.306	0.408	0.550	0.505
24	0.241	0.335	0.501	0.453
32	0.129	0.280	0.450	0.412

Table 6: Algorithm efficiency on a square array with dimensions of 5000 to varying degrees of precision

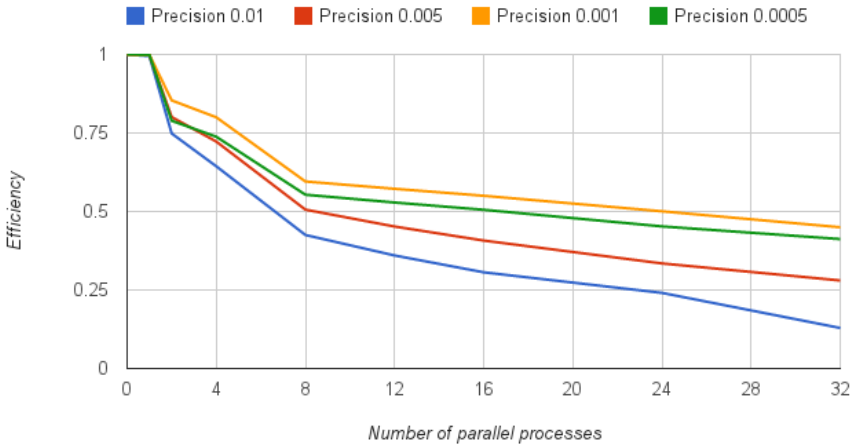

Figure 6: Algorithm efficiency on a square array with dimensions of 5000 to varying degrees of precision

We see the expected reduction in efficiency as the number of processes increases. However there are some interesting patterns. We again see the poor performance for the dimension 1000 problem with efficiency sharply dropping to low values. The other dimensions on the other hand have relatively drastic drops in efficiency up to 8 processes before a more gradual decline in efficiency up to 32 processes.

The precision scaling results on the other hand have much closer results which I believe is due to different ways that dimension and precision scaling affect the results as discussed previously. We again see the quick drops in efficiency up to 8 processes before a more gradual decrease begins.

A sharp change in efficiency can be seen for the 0.01 precision problem after 24 processes. This reflects the speedup dropoffs seen earlier. As mentioned previously this is most likely due to the relative simplicity of the problem which results in the coordination of the increased number of processes having negative effects on the execution time.

There is a reduction in efficiency as the number of processes increases as the reduce and broadcast messages that take place at the end of each iteration must communicate with all processes. The increased number of processes means that there will be more cores idling while this single process work is completed. I believe the sharp changes in efficiency for the lower number of processes is due to the messaging that takes place in each iteration. As the number of processes increases the less of an effect that the messaging has on the execution time.

4.4 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 result.

Processes	1000	5000	7500	10000
0	0	0	0	0
1	0.096	0.162	0.069	1.413
2	3.025	23.879	46.588	82.334
4	5.025	32.797	66.746	113.868
8	10.133	89.009	187.738	323.688
12	16.525	97.377	197.758	332.788
16	21.773	107.745	212.634	345.168
24	33.829	130.281	244.618	375.04
32	49.581	162.305	277.802	414.896

Table 7: Parallel overhead of algorithm on multiple square arrays of different dimensions to a precision of 0.001

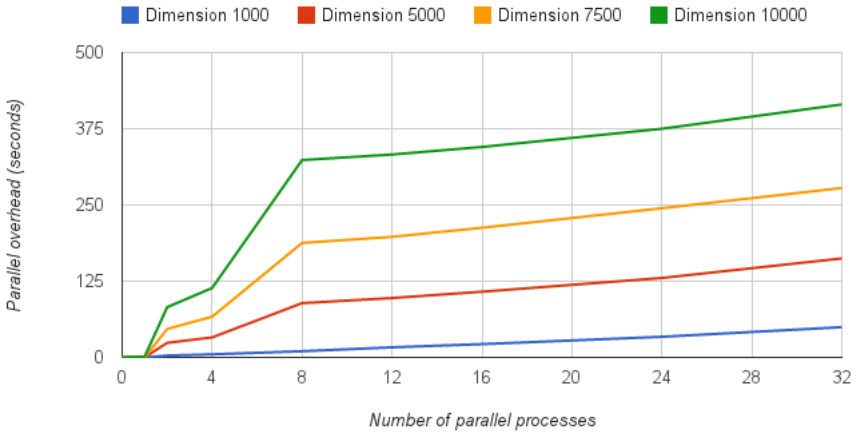

Figure 7: Parallel overhead of algorithm on multiple square arrays of different dimensions to a precision of 0.001

Processes	0.01	0.05	0.001	0.005
0	0	0	0	0
1	0.071	0.063	0.159	0.279
2	4.697	6.715	22.367	65.63
4	7.725	10.351	32.653	87.074
8	18.885	26.351	88.713	197.946
12	24.837	32.687	97.617	218.814
16	31.629	39.231	106.849	240.082
24	44.037	53.639	130.305	296.85
32	94.253	69.247	159.777	349.842

Table 8: Parallel overhead of algorithm on a square array with dimensions of 5000 to varying degrees of precision

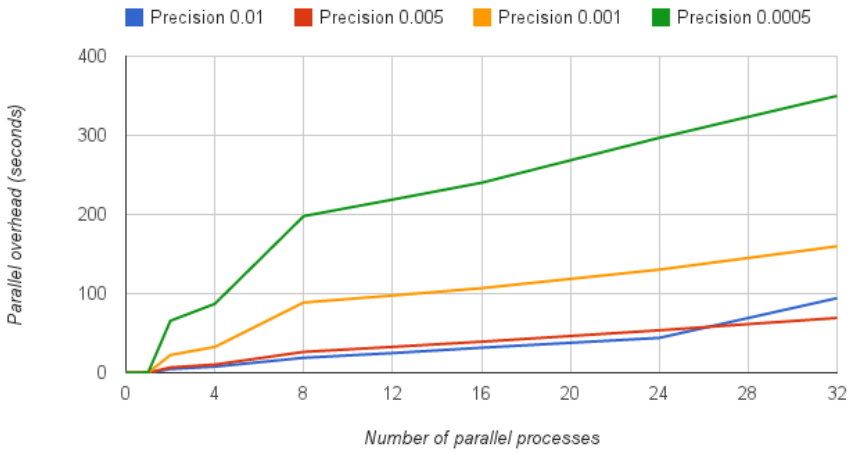

Figure 8: Parallel overhead of algorithm on a square array with dimensions of 5000 to varying degrees of precision

It is clear to see the jumps in parallel overhead that reflect the decreases in efficiency discussed previously. It is the introduction of messaging that I believe causes these jumps. However, as the messaging between neighbouring processes takes place in parallel the overhead it causes is not cumulative. It therefore has a reduced effect after the initial jumps.

In the dimension scaling graph the parallel overhead increases as the dimensions increase. This is due to the increase in message size as the rows become longer and therefore more data must be sent in each message.

The precision scaling data displays the same patterns as the dimension scaling. There is a jump in overhead for the lowest precision when using 32 processes though which reflects the reduced efficiency and increased execution time seen previously.

5 Conclusion

I believe that the results of my correctness testing and scalability investigation are evidence that my algorithm provides a valuable solution to the parallelisation of the relaxation technique using distributed memory. The influence of the superstep paradigm in its design made the implementation very straightforward, with clear alternations between work and parallelisation support. Worthwhile speedup results can clearly be seen in the data along with potential for continued speedup for with even more computing power.

Poor results have been seen for the simpler problems, notably in the dimension 1000 and precision 0.01 data, which indicates that this algorithm and architecture is most efficient when tackling more sophisticated problems. The reduced speedup and increased execution times are only seen when using 32 parallel processes which I believe supports Gustafson's theory that more computing power attracts larger problems. This is reinforced by the valuable speedup seen for the more complex problems when using 32 processes.

I was reluctant to label the messaging that takes place between the process as sequential work as a lot of it takes place in parallel, with processes only having to communicate sequentially with their neighbours. I believe that the reduce and broadcast messages at the end of each iteration that are used to communicate the state of each of the processes could be considered a sequential part of the algorithm. However, it does still involve all the processes communicating at the same time which implies parallel work. It is clear though that no matter how these sections are labelled the messaging will always have to take place in some order and as such will always present a form of minimal execution time, which is a clear example of Amdahl's law.

There are some interesting jumps in the data between 1 and 8 processes which do not appear when using more processes. I believe this is due to performance improvements when running on a single node. With there being a maximum of 8 parallel processes per node we see great performance improvements until the 8 processes are in use, which is due to us taking full advantage of the quick communication available within a node. After this the slow inter-node communication must be used for the messaging which leaves us with much smaller performance improvements as the number of processes increases.

This solution does utilise a lot of parallel overhead but this is due to the distributed nature of the system in use. The use of messaging to communicate state between processes is relatively slow compared to the work that can take place on a single core. However, as the majority of this messaging can take place in parallel the overhead has a reduced effect on the overall execution time. The fact that this messaging can take place in parallel actually makes the algorithm very scalable, in terms of the number of parallel processes, as the messaging should take approximately the same amount of time no matter how many processes are in use.

Overall, my algorithm provides a valid solution that favors more complicated problems. Valuable speedup can be seen in the scalability testing even as the complexity of the problems increase. The correctness testing has also shown that reliable results are produced no matter what configuration has been used. I believe that the solution could continue to produce even greater speedup with an increased number of parallel processes for the more complex problems.