HPC Report 2 - Gregory Sims - gs15687

Starting configuration

The starting configuration from the serial coursework achieves the following times, added here as a reference:

Size	Solver Time (s)
2000	4.078155
4000	45.478557

Parallelise loop over rows

I first used OpenMP to parallelise the for loop which iterates over the rows of the matrix. This approach is typical of a loop level, or fine grain, parallel pattern; However, if this code were extended to work across nodes, then I suspect a task farm pattern would be more effective.

A comment should be made about the variance of the recorded times. Taking 20 samples of the time of this code; the range of the results was on the scale of a second and the variance was 0.377585. Because of this unreliability, recorded times will be averages with their variance given. I would suspect this inconsistency is due to load on blue crystal or kernel throttling on some of the cores.

Size	Solver Time (s)	Variance
2000	0.510614	0.042909
4000	17.653278	0.377585

This change causes an increase in performance because each row in the jacobi iteration is independent; meaning that the computation on any given row does not affect any other row. This in turn means that each row can be computed at the same time in parallel, instead of waiting for the previous row to complete as in the serial version.

Parallelising for loop over columns

The inner for loop, which loops over the columns, cannot be parallelised because the row variable cannot exist in more than one OpenMP scope.

Reducing the convergence check

The loop which checks for convergence is also independent with respect to the rows, so it could be merged into the initial loop above. However, doing this naively will cause errors due to multiple threads entering the critical region at once when trying to change the sqdiff variable. To counter this, the OpenMP reduction pragma can be used. This pragma collects all of the results from the loop and applies a reduction to safely perform the addition.

Size	Solver Time (s)	Variance
2000	0.400712	0.008934
4000	18.957253	7.670672

The increase in average time and variance for the 4000x4000 matrix is difficult to explain. I would suspect that this is caused by poor balancing of the reduction function in some cases. The high variance suggests this method can sometimes be fast and would fit that sometimes poor balancing of the reduction is causing threads to wait for longer than is necessary before reduction can take place.

Generating the data in parallel

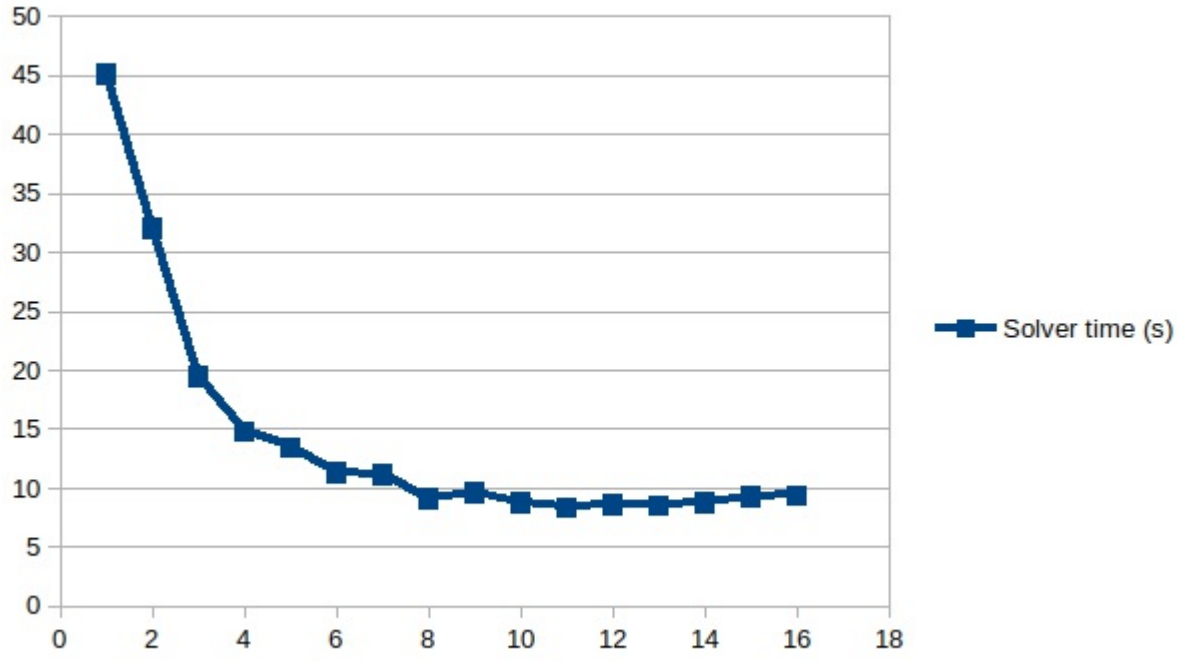
Generating the data, which the jacobi solver is to be run on, in parallel should increase performance. This has the primary effect of speeding up the data generation. However, this does not reduce the solver runtime. However, since the data is generated across the cores, the time spent sending data between cores will be reduced as the data is likely, for any one core, to be in a cache closer to that core.

Size	Solver Time (s)	Variance
2000	0.392383	0.019678
4000	9.403560	1.655810

The reductions in speed shown above would suggest that the creation of threads and transfer of data takes up a significant time.

Scaling across cores

For testing how the code scales over the number of cores used, a 4000x4000 matrix will be used. The average solver time is plotted against the number of cores used below.



This graph shows that, initially, increasing the number of cores the code runs on will dramatically increase performance. However, after more than 4 cores are being used, the cost to communicate and synchronize between the cores is beginning to become the limiting factor of the speed of the code. Once around 12 cores are being used, the code appears to have reached a plateau, where increasing cores is no longer increasing performance, as the performance gains are being negated by the cost of communication. After about 14 cores, the speed of the code appears to decline. I suspect this is due to the code fitting to a sublinear declining form when scaled. There is the possibility that this is caused by some error from the timing and averaging, but I would suspect that if there were more than 16 cores available, the sublinear decline would continue.