COMP3320 – Project 2

Timothy Crundall - U5018130

Sequential Improvements

An investigation of the code with gprof showed that the computationally heavy section is the eval_pef() function. Compiling kernel_main and cloth_code with gprof and running with default values revealed that 85% of time was spent in eval_pef(). Based of this information I decided to focus my optimisation efforts on the eval_per() function.

There were various aspects of the code which I suspected could be responsible for inefficiencies, however editing these aspects out did not always yield performance improvements. Sometimes they even resulted in performance degradations. One explanation for the lack of improvement is that the compiler was clever enough to remove this inefficiency at the assembly level. As for performance degradation, I offer no explanation. Perhaps there was statistical error or error on my part. Further investigation would be required.

My first edit was the removal of explicit division by *vmag*. I introduced a new variable *vmag_i* which I set to be the inverse of *vmag* and replaced all further division by *vmag* by multiplication by *vmag_i*. The result was ~5% reduction in the time per call of eval_pef() with 10 nodes per dimension, and a ~15% reduction in the time per call of eval_pef() with 50 nodes per dimension. The fact that there was any improvement at all was surprising. I had assumed the compiler would notice the recurring division by the same value which stays constant within the loop, and replace the division with multiplication. However this did not seem to be the case. The differences that the different node count had on the amount of improvement was also surprising. There was substantial deviation from the mean with the recorded times for 10 nodes per dimension but fairly consistent times for 50 nodes per dimension. The discrepancy of performance improvement could be simply due to statistical noise.

The next aspect of the code to be modified was the conditional branch used to ignore a node's interaction with itself. It would seem that the branch would be reasonably simple to predict. It was only ever not taken at the precise mid-point of the nested loops. It would seem that there are limitations to how well branch prediction can work. I removed the branch by unrolling the two loops which iterated over the nodes within interaction distance of the node (X,Y): [xmin, xmax] and [ymin, ymax] say.

Loop A dealt with [xmin, xmax] and [ymin, Y).
Loop B dealt with [xmin, xmax] and (Y, ymax].
Loop C dealt with [xmin, X) and Y.
Loop D dealt with (X, xmax] and Y.

Removal of the branch increased performance by a further 5% with 50 nodes per dimension. It also significantly reduced the deviation of the measured times for eval_pef() per call.

I then tried to remove a couple of lines which were calculating the same factor. I calculated this factor and stored it in a variable. This worsened performance however. I suspect the compiler already had a hand in removing repeated calculations but in a way that required less overhead. For example, maybe the required value was pre-calculated and carefully left in a register instead of being stored as a whole variable.

The final change I made was to adjust the indices so as to remove striding. This made no significant difference in timings. I am forced to assume the inefficient array accesses were already being compiled out.

One enhancement that I didn't look at was adding the interaction forces from a pair of nodes to both

of their force totals. This would potentially half the computation time since as it stands the code is effectively calculating each interaction twice.

Here is a table of the measured time means and standard deviations:

	Mean (ms)	StDev (ms)
Original	2.04	0.03
Replacing division	1.74	0.05
Removing branch	1.67	0.01
Removing repeated multiplication	1.80	0.05
Removing Striding	1.65	0.04

SSE Vectorisation

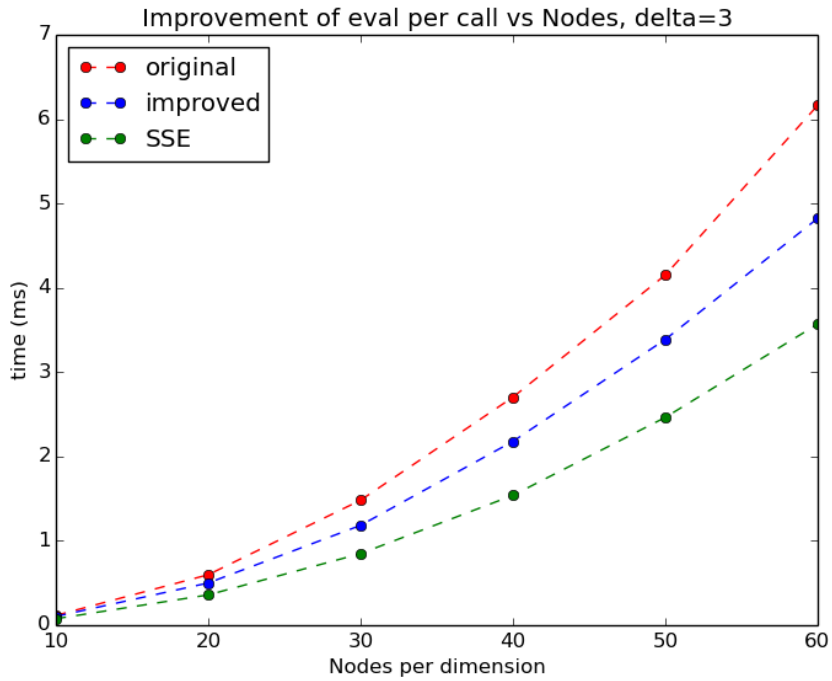
I did not have high hopes of achieving much performance improvement from the introduction of SSE intrinsic functions. My main doubt was based on the fact that only two doubles would be able to fit in each 128 bit register. This put an absolute upper bound of a speed up of two. Even then this upper bound is a large over-estimation. In eval_pef() not all of the computation done within the loop can take advantage of SSE intrinsic functions. There is also the overhead involved of initialising the SSE vectors.

That being said, a speed up of about 20% was observed.

Unfortunately I couldn't get the SSE code to produce correct results, however I am confident that the SSE code I developed differs in performance to the correct SSE code I would have eventually developed by a negligible amount. Because of this I am content using my incorrect SSE code for performance investigation.

Here is a plot of the incremental improvements:

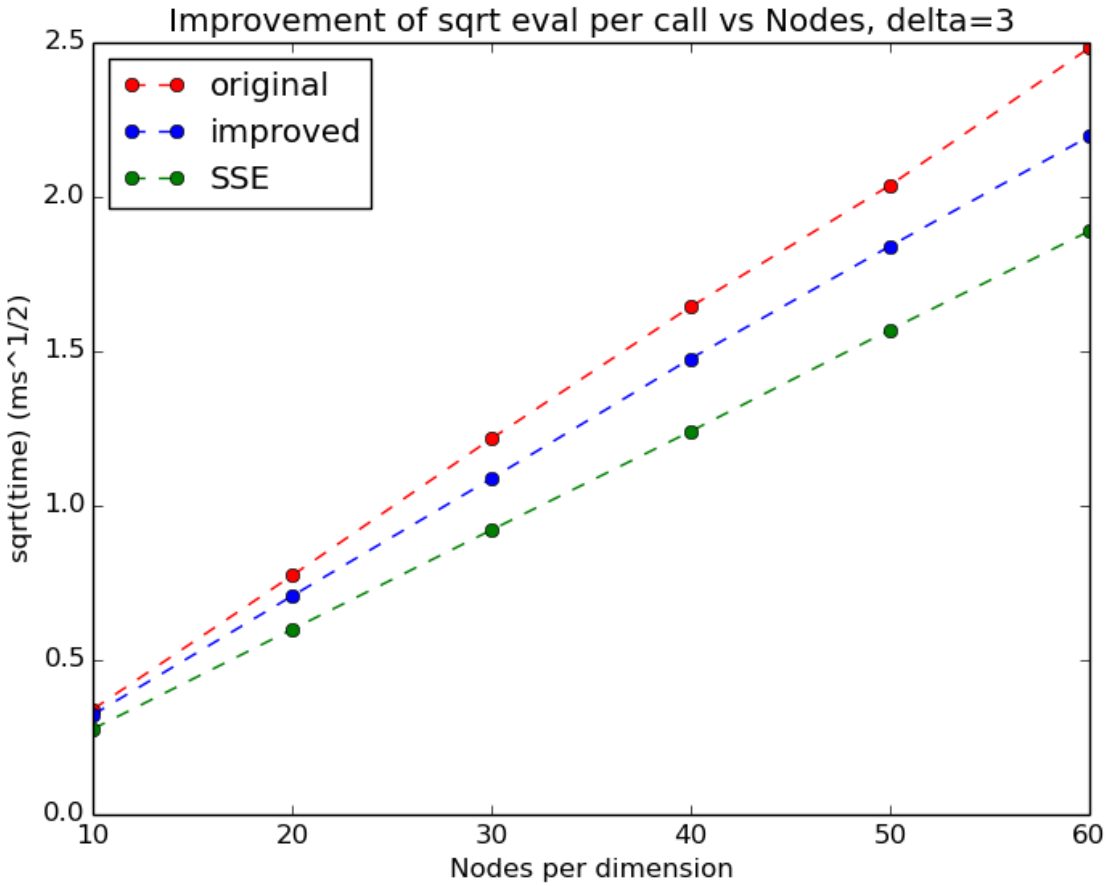
As can be seen there is steady



improvement with each phase.

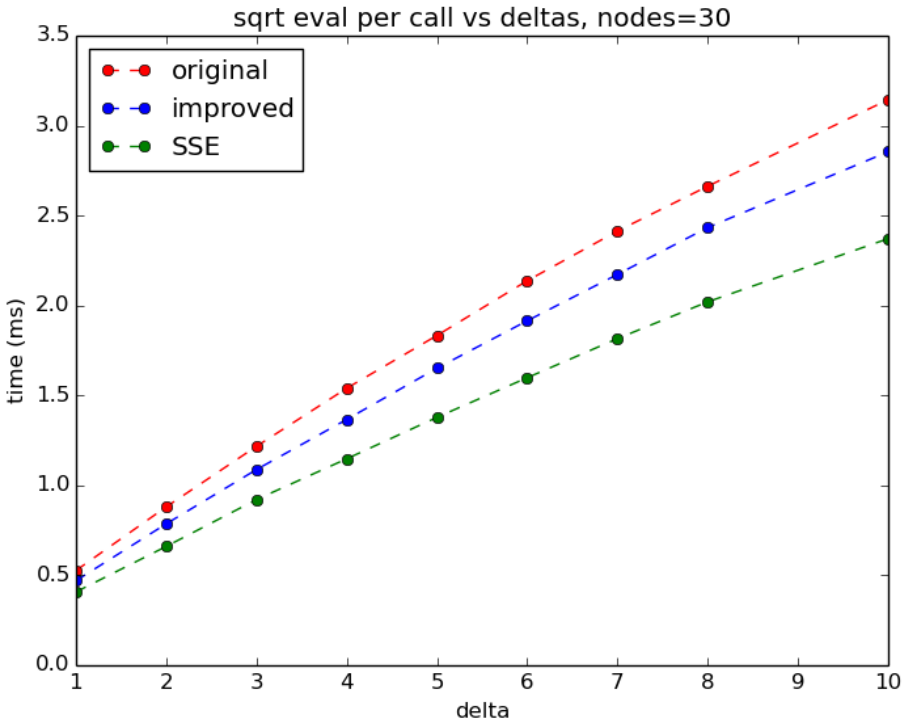
Complexity

An interesting question to look at is how does the computation time scale with problem size. Since the computation per node is capped by the size of delta, it is expected that the computation will scale with the number of nodes or with N^2 where N is number of nodes per dimension. Here is a plot of the square root of the time plotted against nodes per dimension:



As can be seen the lines are almost perfectly linear. So from this we can conclude that across all implementations the complexity is $O(N^2)$.

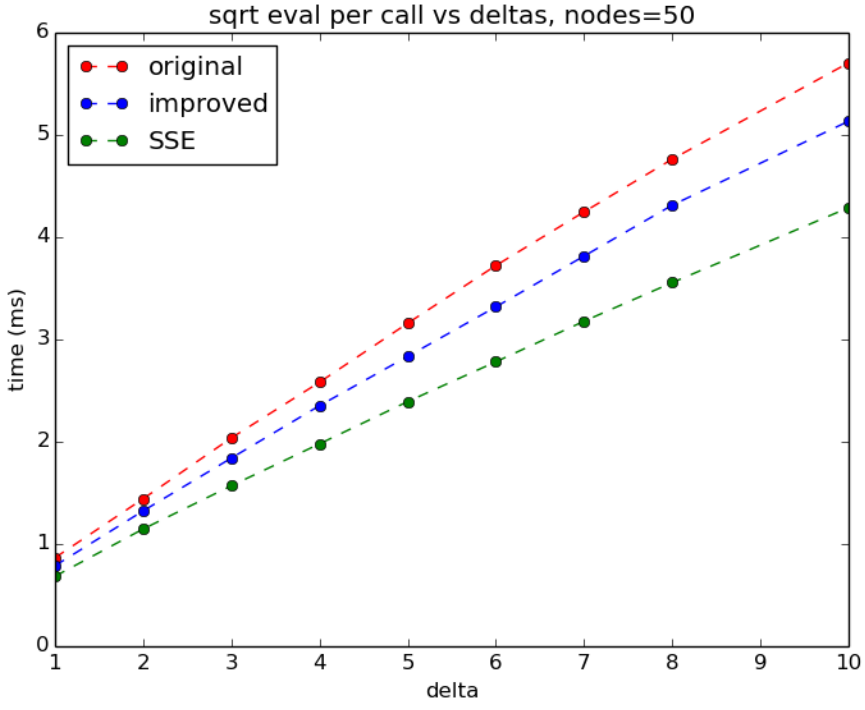
How does the problem scale with delta though?



It can be seen that when the square root of the time is plotted against delta the lines come out as approximately linear. The slope seems to be flattening out for higher values for delta, implying that at sufficiently high deltas any change in the delta will result in no change in computation time. This is sensible, since for a piece of cloth with 30 nodes per dimension, once delta reaches 30, any increase to delta will have no affect on the problem size, since every node will already be interacting with every other node.

This is further supported by looking at a plot against delta with a larger number of nodes per dimension:

It can be seen that the flattening out of the curve is less pronounced here.



OpenMP

I was unable to implement OpenMP successfully.