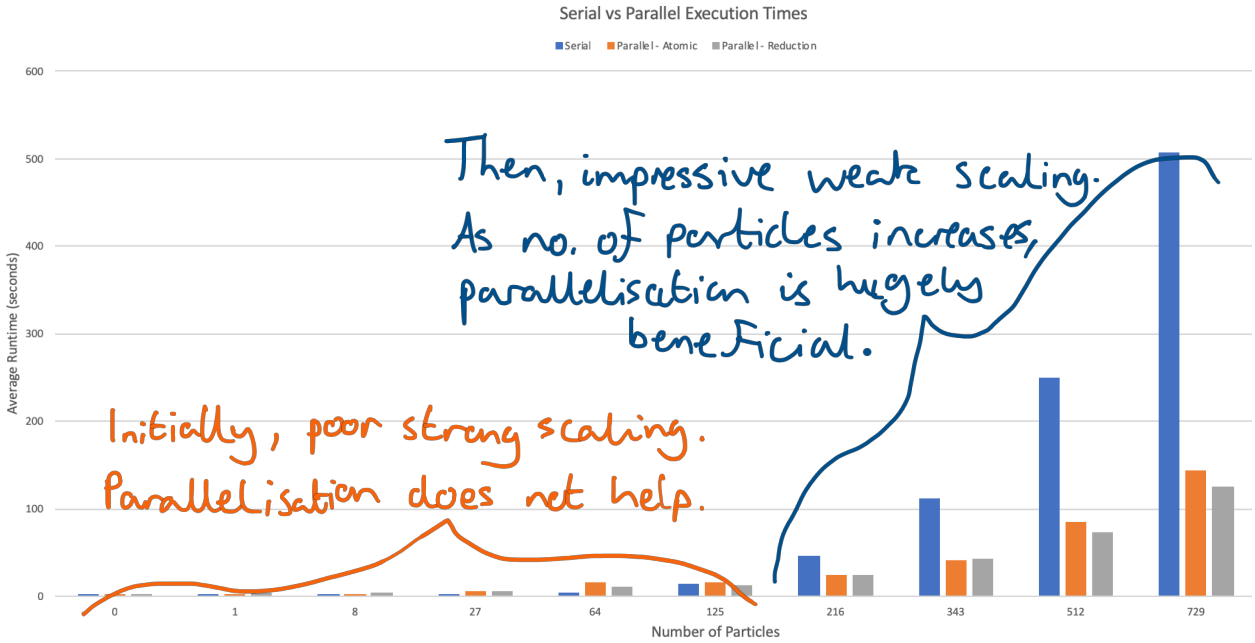
Loop Unravelling and Alternative Parallelisation Implementation

Having implemented my ideas in Step 5, I methodologically tested my code. Before beginning the proper experimentation, I wanted to quickly test the effect of manually unrolling the loop which zeros the force array. I hypothesised that the nested loop overheard, for such a small loop body, would negatively impact performance when called thousands of times. Furthermore, I hoped that manually unravelling the loop would prompt the compiler to make use of SSE SIMD instructions, further increasing performance. However, once I implemented this and tested it, it transpired that it was actually quicker to use the loop. Therefore, for the remainder of my experiments this was the version I used. The second experimentation area that I analysed was the effect of having multiple $\#pragma\ omp\ atomic$ clauses within my parallel code. I re-implemented my code, using a reduction clause on the *force* array, to remove the necessity of the $\#pragma\ omp\ atomic$ clauses, whilst still preventing race-conditions. Whilst this removed the serial accesses to the $force$ array, it potentially added further overheard as each thread now held a private copy of the *force* array, which were binary reduced at the end, and so may not lead to any speedup. For the remainder of the experimentation I tested both versions and compared the results to that of the serial version.

Strong and Weak Scaling

I then analysed the effect of strong scaling on my code. For a small number of particles, we see that increasing the number of processors without significantly increasing the problem size – strong scaling – does not lead to a speedup. This is likely due to the parallelisation overheard that I discussed in Section 5. However, once the number of particles is increased above a certain threshold (around 150 in this instance), the true benefit of parallelisation can be seen:



We can see that if we increase the problem size further, whilst keeping the number of threads fixed – weak scaling – a significant speedup is observed. We can also see that for larger problem sizes, despite the added overhead of copying the force array to each thread, a binary reduction of the entire force array yields better performance than atomically accessing its individual elements. I used a Python script to automate these tests, which arranged the particles evenly in an $n\ \times n \times n$ lattice with $5 \times 10^{-9} m$ between any two particles, and with probability $\frac{1}{5}$ either gave or did not give each component of every particle's velocity an initial velocity of 1×10^{-12}. I had to use a relatively large time step of 0.1s for these experiments due to timing constraints, which likely effected the stability/convergence of the code. However, as I was analysing the parallelisation in the section, I didn't believe this to be of too much concern.

Optimisation Levels

The final area that I experimented with was the optimisation level of the compiler. I found that changing from full optimisation (-O3) to no optimisation (-O0) had a staggering effect on the run time, particularly for the serial code, which had a factor of 4 slow down after this change was made. Whilst this is obviously not desirable, a high optimisation level gives the compiler free reign to reorder instructions, which can cause unexpected results in floating-point intensive calculations such as these, and so should be used with caution. If I were to continue my analysis, I would look at the effects of applying the -fassociative-math and -ffasst-math compiler options, on both the speedup and the accuracy of the code.