Solution Step 5 – Parallelism Report

Matt Ingram

Data Analysis

I began my data analysis by considering the data dependencies in the *updateBody* function, and then drawing a Directed Augmented Graph. I identified the following sequential dependencies in my code:

$$\begin{tikzcd}\boxed{\text{Zero Force Array Loop}} \arrow[r] & \boxed{\text{Calculate Force Loop}} \arrow[r] & \boxed{\text{Update Position and Velocity Loop}}\end{tikzcd}$$

Throughout my parallelism, I shall use a grain size of 1 (i.e the unit of execution allocated across the threads is one iteration of the loop). As each iteration should have an (approximately) homogeneous computation time, the default static parallelisation should be sufficient to achieve good load balancing. Based on the DAG above, when implementing the parallelism, I shall have one *omp parallel* region, within which I shall declare the variables that need to be private to each thread. Within this *parallel* region, I shall have three *omp for* regions, each of which will implement Bulk Synchronous Parallelism. The details of the parallelism within these three regions will be as follows:

- **Zeroing the Force Array:** I will use $\#pragma\ omp\ for\ collapse(2)\ private(i, k)$ on this loop, to collapse the nested loop iterations into one loop space, of size $No.Bodies\ \times\ No.Components$. For a small $No.Bodies$, we should achieve complete single-iteration parallelisation (one iteration per thread).
- **Calculating the Force:** I will use $\#pragma\ omp\ for\ collapse(2)\ private(i, j)\ reduction($ $min{:}minDx)$ on this loop, again to collapse the nested loop iterations into one loop space. I will insert a $\#pragma\ omp\ atomic$ statement above each access to $force[i][k]$, to prevent race conditions. Each thread will also compute a local minimum distance, and then at the implicit barrier synchronisation at the end of the $omp\ for$, the local *minDx* variables will be reduced efficiently using OMP's *min* function, and OMP's built-in binary-tree reduction scheme, to find the global minimum across all threads. Using this binary reduction should hopefully reduce the sequential overhead compared to using an *atomic* or *critical* section, as placing $\#pragma\ omp\ atomic$ above the *minDx = std::min(minDx, r);* line would cause every access (across all threads) to the *minDx* variable to occur sequentially. To increase loop parallelism and thus hopefully scalability, I removed the 'optimisations' included in Step 2. Furthermore, I unravelled the inner component loop, in the hope that the compiler might use SIMD instructions when updating the force array.
- **Updating the Position and Velocity:** I will use $\#pragma\ omp\ for\ collapse(2)\ private(i, k)$ $reduction(max{:}\,maxV)$, to again collapse the nested loop iterations, and use a binary reduction to efficiently keep track of the maximum velocity.

Concurrency Analysis

Despite the high amount of parallelisation available within the *updateBody* function, I do not expect to achieve any speedup until a significant amount of particles are used. This is because the contents of sequential calls of the *updateBody* function are parallelised, rather than parallelisation of the actual calls to the function. The overhead of the $\#pragma\ omp\ parallel$ will compound this, as creating a new group of threads every time *updateBody* is called is a relatively expensive operation. If it were possible to parallelise the iterative calls to the *updateBody* function, we would be able to achieve a much higher speedup. This is a problem many iterative numeric solvers suffer from.
However, for larger numbers of bodies, the increased speed due to parallel computation will hopefully outweigh the thread overhead, and a speedup will be observed. If we fix the problem size ($No.Bodies$) to a sufficiently large amount to negate the thread overhead (such that $No.Bodies \gg No.Threads$), and turn off all slow output functionality, we can begin to analyse the expected speedup using the speedup laws discussed in the lectures. As we are fixing the problem size, and increasing the number of processors (from 1 to up to 56 on Hamilton), this is an example of strong scaling. Therefore, the correct law to use is Amdahl's Law: $t(p) = f \cdot (1) + (1 - f) \cdot \frac{t(1)}{p}$, from which the speedup due to parallelisation can be derived (with *p = 56*):

$$S(p) = \frac{t(1)}{t(p)} = \frac{t(1)}{f \cdot t(1) + (1 - f) \cdot \frac{t(1)}{p}} = \frac{56}{55f + 1}.$$

Finally, we can use this formula for the speedup to calculate the efficiency of our parallelism:

$$E(p) = \frac{S(p)}{p} = \frac{\frac{56}{55f + 1}}{56} = \frac{1}{55f + 1}$$

Whilst the individual calls to the update body function occur sequentially, within the update body function itself there is a very high level of parallelism, so if we take f to be approximately 0.1, we see we have the *possibility* of a speedup of $\frac{56}{55 \times 0.1 + 1} \approx 8.6$ times, and an efficiency of $\frac{1}{55 \times 0.1 + 1} \approx 0.15$. In closing, I would like to emphasise again, that such a high speedup is likely to be unattainable in reality, due to the significant thread creation overhead.