When considering how well suited our earlier (Step 1) implementation is to parallelisation we will first consider a basic DAG. Figure 1 vaguely describes the dependencies in one iteration of the for loop. The *updateBody*() function simply consists of a nested for loop.

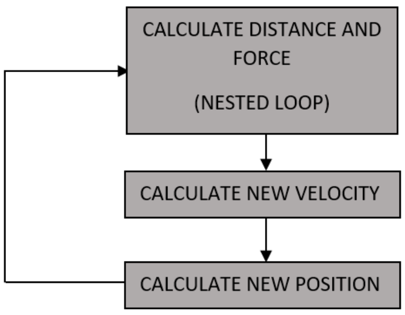

Figure 1: Simplified DAG for the *updateBody*() function. Here an arrow indicates a dependency.

In Figure 1 we see that the inner loop (nested loop) calculates the distance between each pair of particles. The distance is in turn used to calculate the forces acting on each particle. Following this, the forces are utilised to compute the velocity of each particle, which is use in order to obtain each particles new position/coordinates. The dependencies within the function are clear.

Parallelising either the inner (nested) loop (which calculates distances and forces) or the outer loop (which updates velocity and coordinates) alone is relatively simple, as the order that the iterations are performed does not matter in either case. The inner loop depends on the masses of each particle - which remain constant throughout the simulation. It also depends on the distance between particles, which naturally is dependent on the position of each particle (which is calculated in the outer loop). Provided positions (or coordinates) are not updated whilst the inner loop is executed then no issues will arise. Similarly, when parallelising the outer loop alone it is simple as the provided the inner loop still executes sequentially, the forces calculated will be accurate for each calculation of velocity and consequently position.

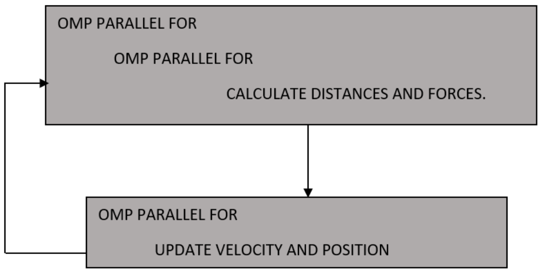

Figure 2: Our proposed OMP solution.

We wish to parallelise as much of our original code as possible, however it is important that we do not introduce any error when doing so. Figure 2 displays a vague outline of the proposed parallelisation in OMP. This way we can parallelise the calculation of distances and forces, and once each of the forces have been calculated we can update the velocity and position based on the forces that have just been calculated. As any dependency will already have been calculated the update velocity and position can also be done in parallel. The parallelisation paradigm that we will be adhering to is SPMD. This is because our algorithm is independent of the data it is working on, as the *updateBody*() function has no selection or opportunity to vary at any point. Each thread manage its own work and any thread-specific behaviour, so no scheduling is required.

When predicting the efficiency of the parallelisation it will depend significantly on the number of particles in the simulation, as this is what is being iterated over in the for loops that we are parallelising. The sequential code involves n iterations of the outer loop and $\frac{n(n-1)}{2}$ iteration of the inner (nested) loop. Assuming that it takes time t to execute the inner loop and the remainder of the outer loop (as they're likely to be very similar). We can approximate the run-time of our sequential code as $T_{sequential} = \frac{n(n-1)}{2}t + nt = \frac{n^2+n}{2}t$. We can approximate the run-time of our proposed parallel program as $T_{parallel} = t + \frac{n-1}{n}t + t \approx 3t$ (the time to run each loop once). From this we observe a speedup of $S(n) = \frac{n^2+n}{6}$. The number of processors we would require to do this would be roughly $p = \frac{(n-1)(n)}{2}$, giving us a theoretical efficiency of $E(p) = \frac{n^2+n}{3(n^2-n)}$. These very approximate calculations suggest that as you increase the number of processors/threads the speedup will increase (which is obviously to be expected) and the efficiency will decrease. The efficiency and for 12 processors (fix $n = 12$) are $E(p) \approx 0.3939 \equiv 39.39\%$ and $S(p) = 26$.