Data-parallel programming Coursework

William Stapleton - NKFN77

1 The N-Body Problem

1.1 What is the N-Body Problem?

The N-Body problem is a common physics and maths problem, where the motion of N bodies is predicted while being influenced by forces like gravity. Pair-wise interactions for all N-bodies must be modelled, which can prove to be computationally expensive. Trivially, there are $O(N^2)$ force calculations that need to be computed.

1.2 The base solution: step-0

Step-0 uses a serialised approach to model the motion of stellar bodies by computing the force of gravity between each body in the **force_calculation** function and then using these forces to update the positions and velocities of each body in the **updateBody** method.

1.3 Experimental set-up

For all the tests carried out, the initial parameters were: final time = 60 seconds, snapshot time unit = 1, time step size = 1, and the minimum and maximum mass of the particles was 1 and 3 respectively. The simulations work with bodies ranging from 10^3 to 24^3. To ensure data consistency, and to account for varying load on Hamilton, 5 simulations were run for each number of bodies in order to take a meaningful average.

1.4 ISA Extensions and Number of Cores

Hamilton uses multiple ISA extensions which benefit the solution:

- avx2: Advanced Vector Extensions which allows for better optimisation for vectorisation
- fma: Fused Multiply-Add Instructions, which allow for increased performance for numerical calculations
- ht: Hyper-Threading is extremely beneficial in section 2 due to the parallelisation from multi-threading.

For the serialised solutions in step-0 and step-1, a core count of 50 was assigned to each batch job. For parallelised jobs in step-2, a range of 10 to 50 cores was used to show the scaling of parallelisation with number of assigned cores.

2 Step-1: Vectorised solution

2.1 Approach

Step-0 used two-dimensional arrays to store the position and velocities of the bodies in the simulation. Step-1 on the other hand was optimised to use one-dimensional arrays to store each direction of the velocity (x,y, and z) and also each coordinate for the position (x,y, and z). The use of these arrays granted performance increase from memory access speed, and also allowed for SIMD vectorisation to be utilised more effectively.

The **setUp** method was adjusted to implement the new array structure mentioned in 2.1. Following this, the **updateBody** method was updated to remove repetitive calls to **force_calculation**, instead storing the intermediate values as variables. In addition to these, OpenMP SIMD pragmas were implemented on the main loop. An additional parameter (collapse(2)) ensured the nested loops were unrolled to implement better vectorisation. The position and velocity update loops were merged together as the calculations were related to the same array indexes, and SIMD vectorisation was applied to them.

2.2 Comparison to Step-0

Figure 1 shows the time elapsed for different numbers of bodies for both step-0 and step-1. For smaller numbers of bodies, there was little to no performance increase due to vectorising loops with

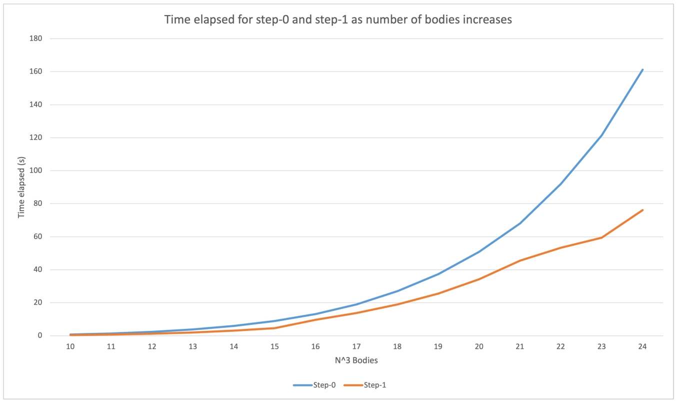

Fig. 1. Time elapsed for step-0 and step-1 as number of bodies increases

smaller bounds having no benefit. For extremely small N, where N is less than N^3, the OpenMP overhead actually slowed down the computation compared to step-0.

3 Step-2: Parallelised solution

3.1 Approach

In the class initialisation for **NBodySimulationParallelised**, the number of cores used is set to the maximum available to the batch file (ranging from 10 to 50). Nested parallelism is also disabled to reduce the run-time overhead for each thread. The **setUp** method follows a structure similar to the vectorised solution in step 1. To take advantage of parallelisation, **updateBody** had the force calculation loop parallelised with OpenMP pragmas. Reductions were applied to this for the **minDx** global variable, and the force arrays. Parallelising the $O(N^2)$ loop allowed for optimisation, especially when dealing with large numbers of bodies. Following the force calculations, the particle position and velocity update loop was parallelised, using static scheduling due to the equal load on each iteration. A reduction operation was applied to the loop to find the maximum velocity across all threads. These optimisations allowed for the execution time to be rapidly speed up, especially for large numbers of N and when more cores are assigned to the executable then the speed-up is even more profound as shown in section 3.3.

3.2 Comparison to Step-0 and Step-1

Figure 2 shows the time elapsed for varying numbers of bodies for step-0, step-1 and step-2. Step-2 utilises 50 cores for all values of N. While step-1 was just over two times faster than step-0 for 24^3 bodies,

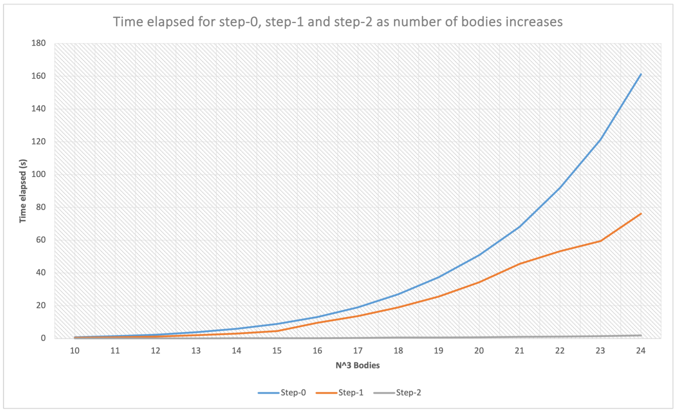

Fig. 2. Time elapsed for step-0, step-1 and step-2 as number of bodies increases

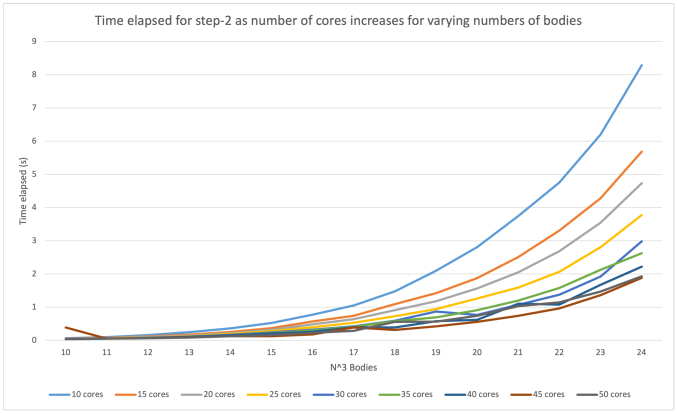

Fig. 3. Time elapsed for step-2 as the number of cores increases for varying numbers of bodies

step-2 was over 84 times faster than step-0 and over 40 times faster than step-1. Similar to the vectorised approach in step-1 compared to step-0, parallelising the solution did not improve performance for small numbers of bodies due to the extra thread overhead applied.

3.3 Efficiency compared to number of cores

Assigning more cores to the executable on Hamilton lead to execution speed-up, especially for larger numbers of bodies. Figure 3 shows the run-time for varying numbers of cores as the number of bodies increases. A notable improvement in performance is for 24^3 bodies where 10 cores took 8.2 seconds whereas when 50 cores was assigned to the executable it took 1.93 seconds, improving the performance by a factor of 4.2.