SET10108 Concurrent and Parallel Systems Coursework

Paulius Bieksa

October 2018

.

Abstract

This project aims to investigate and improve the performance of an n-body simulation algorithm by parallelising it. Two approaches were attempted - OpenMP and CUDA. GPU implementation with CUDA was found to be successful, with speedup of S = 5.78, with higher speedup expected when calculating more bodies.

1. Introduction and Background

This coursework attempts to speed up an n-body simulation algorithm by parallelising it. The n-body problem is the problem of predicting the individual motions of a group of celestial objects interacting with each other gravitationally. The problem requires calculating how each body affects each other body being simulated, and has complexity of $O(n^2)$. The simulation considers each body as a point particle (a body with no spacial extension), and uses Newton's laws of motion to calculate the movement of the bodies in 2D space.

The implementation of the problem, used in this project, runs over a given period of simulated time, with a given physics step time. Each physics step the algorithm has to calculate the forces for each particle and then perform an integration to move the particles based on the forces calculated. The application also renders a graphical representation of the particles in a gif format.

2. Initial Analysis

The graphical output is only required to show that the bodies are simulated correctly, and therefore are not included in the algorithm analysis. The graphical output seems correct, and is usually rendered once for every configuration when the program is run.

As identified in the previous section, the algorithm has two main steps: the force calculation and the integration, which have complexities of $O(n^2)$ and $O(n)$ respectively. Both of these can be classified as data parallel problems, but since the force calculation has a higher complexity, it presents itself as the main target for parallelisation. Running the Visual Studio performance profiler tool supports this observation, as simulating only 10 particles, the force calculation uses over 60% of CPU time, and any configuration with more than 50 particles uses more than 98% of CPU time. Given that this is the bottleneck for the application, and that the algorithm for calculating forces is structured as a nested loop, all attempts to parallelise this algorithm will focus on this specific part.

The initial measurements were taken by running the algorithm in different configurations, 50 times in every configuration. Changing the duration of the simulation or the distribution of mass or position of the bodies does not change the per-frame performance of the algorithm. Changing the number of bodies, however, significantly impacts execution time.

	Seq	**Seq + step**
1000 bodies	0.02405	0.02407
1750 bodies	0.07504	0.07405
2500 bodies	0.15105	0.15044
3250 bodies	0.25275	0.25400
4000 bodies	0.38469	0.38369
4750 bodies	0.54197	0.54947

Table 1: *average time per frame calculating forces in seconds.*
Seq - sequential algorithm, forces only,
Seq + step - sequential algorithm, forces and integration.

3. Methodology

There were two parallelisation approaches used in this project: OpenMP and CUDA. Both had to modify the algorithm to support parallelism, due to using different hardware architecture. To make sure these changes did not affect the calculations, final positions of the bodies were saved to file, in addition to the visual output. All final positions were exactly the same in all implementations.

The OpenMP approach did not require much change to the algorithm. The sequential implementation calculates forces in a nested loop, which makes it a perfect target for OMP directives. OMP has nested loop support, but it was deemed that this would add additional overhead due to having a lot more context switches. The algorithm calculates forces towards each body separately, and then adds them to a cumulative force vector. This means that a data race is likely in this part of the algorithm. To make sure no data races occur mutual exclusion was used in the form of OMP locks. An array of locks size equal to number of bodies simulated was created. Before writing to the cumulative force vector, corresponding locks are activated, and after the adding to the force, deactivated.

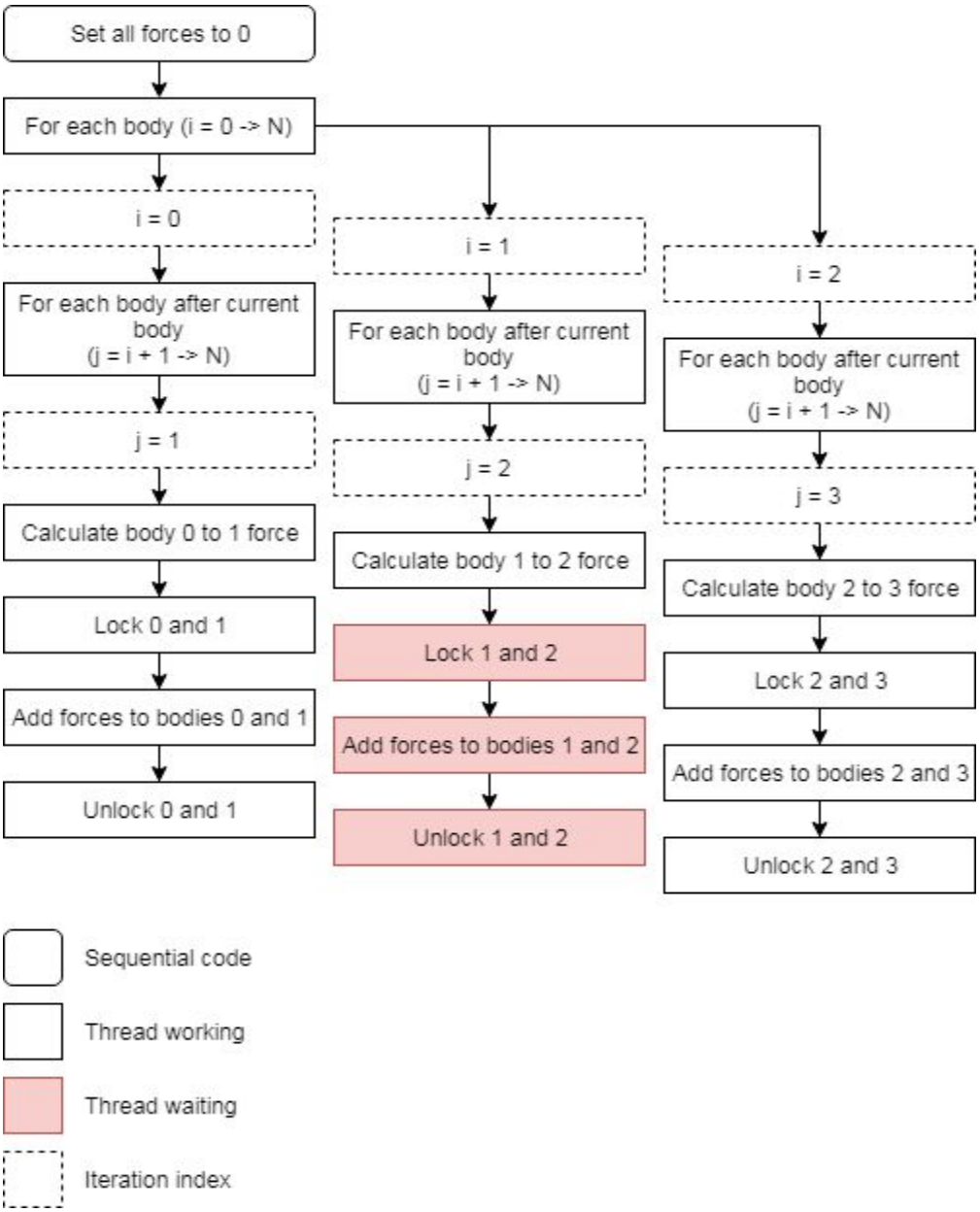

Diagram 1: *Example of OMP parallel execution. Thread 1 has to wait, because lock 1 is activated by thread 0.*

The GPU parallelisation using CUDA needed more alterations to the algorithm due to radically different architecture. The SIMD architecture, however, provides a much higher potential for speedup. It also avoids data races, as each body in the

simulation can have it's own output data. The GPU algorithm performs integration in addition to calculating forces, as this has to be done for each body anyways, and SIMD architecture makes this very easy. There might be a tradeoff here due to this returning more data than just returning the force (2 doubles) and calculating the integration sequentially. This was not tested, however, due to time constraints.

Before running the kernel, data has to be set up in a way that can be copied to the GPU efficiently (e.g. single copy operation). The algorithm requires position, velocity and mass to calculate each frame, which can be converted into 5 double precision floating point numbers per body. This conversion is done before sending the data to the GPU. The data is also returned from the GPU in the same format. The conversion to and from the double array is treated as part of the algorithm when measuring execution time. GPUs only support a certain number of threads per block (1024 is common), which means that any number of bodies higher than that have to be split into blocks. The algorithm uses the minimum number of blocks required.

The kernel can only run a limited number of blocks concurrently, due to the way the thread block scheduler works. Because of this, simulating large number of bodies can cause a slowdown. As a possible solution to this problem the kernel can simulate multiple bodies per thread. The index is calculated so all the bodies simulated in a single thread are in a continuous region in device memory. After getting the correct index the algorithm for calculating the single interaction force and integration is functionally the same as in the sequential algorithm. Main difference being that the opposite force is not applied to the other body in the interaction. The algorithm requires some branching to function correctly. Each body has to make sure is is not at exactly the same position as it's interaction partner to avoid division by zero. Also, any index beyond the last body index is discarded to avoid modifying memory that is not allocated by the program. In both of these cases one of the branches does no work, therefore this should not impact the performance of the kernel in any real way.

```
calculate chunk index
index = chunk index
for (0 -> bodies_per_thread)
    if (index < body_count * 5)
        cumulative_force = 0
        for (i = 0 -> body_count * 5; i += 5)
            calculate vector to body[i]
            if (||vector to body[i]|| is 0)
                discard this iteration
            calculate interaction force
            cumulative_force += interaction force
        write to output array
    index += 5
```

Listing 1: *pseudocode for cuda kernel.*

Each approach has been run 50 times in each configuration. The execution times have been recorded to calculate speedup. All configurations use delta time of 0.01. The machines used for testing have the same configuration (Intel(R) Core™ i7-8700 CPU @ 3.20GHz. GeForce GTX 1070).

4. Results and Discussion

The CUDA implementation achieved some very good results. All configurations for CUDA achieved some amount of speedup, with the highest being S = 5.78. There was a clear trend of higher speedup when simulating more bodies. The smallest number of bodies simulated was 1000, so it is reasonable to believe that with lower number of bodies the GPU implementation could be outperformed by the sequential one. Simulating multiple bodies per GPU thread performed worse in every test, though it still provided consistent, although small speedup. It is very likely that the number of bodies used for testing was not high enough to reach the point where not all blocks could be run concurrently, as the largest number of bodies simulated (4000) only used 4 blocks, with 1024 threads each.

	Time per frame			Speedup	
	Seq + step	**Share 1**	**Share 5**	**Share 1**	**Share 5**
1000 bodies	0.02407	0.01687	0.02189	*1.42630*	*1.09953*
1750 bodies	0.07405	0.02932	0.05475	*2.52550*	*1.37071*
2500 bodies	0.15044	0.04167	0.10351	*3.61055*	*1.45918*
3250 bodies	0.25400	0.05390	0.18434	*4.71251*	*1.37113*
4000 bodies	0.38369	0.06629	0.26779	*5.78802*	*1.43655*

Table 2: *Average time per frame and speedup for CUDA implementation. Seq + step - sequential algorithm, forces and integration, Share 1 - 1 body per thread, Share 5 - 5 bodies per thread.*

The OpenMP implementation achieved mixed results. Almost all of the configurations were slower than sequential on the University machines. Some speedup was expected here; especially running the program on computers with 12 logical cores. Testing mostly proved otherwise, however. The locking, required to avoid data races, was likely too frequent to make use of the parallel hardware. Doing calculations in larger chunks had no effect.

Some strange behavior was noticed when testing the OpenMP implementation. Rendering graphical output sped up the algorithm, even though rendering is separated from calculating forces, and should not influence it in any way. It was later found that sleeping the main thread for a short period of time each frame, outside of the measured region would achieve the same result. For a thousand bodies the optimal amount of time to wait was found to be roughly 100 ms. Testing this behaviour on a personal laptop (Tables 5, 6), has shown that waiting can speed up the simulation overal. It is important to note that tests on this machine without the 100 ms wait, when testing with more than 3250 bodies were slower than the ones with the process stopping and waiting. The other tested machines showed similar behaviour, but less

pronounced and at higher body counts. All cores were being used on all configurations. It is unknown what caused this effect. MSI afterburner was used as a diagnostics tool to try to track down the root of this behaviour. The first likely cause was believed to be CPU declocking due to elevated temperature, however after investigating this it was found not to be the case. The temperature does increase by roughly 4 - 5 degrees celsius, but CPU speed remains the same. Another suspect is the operating system having more time to schedule outside processes when gaps are left waiting, and thus not interfering with force calculations. It is unknown how likely this is.

	Seq	**OMP**		**SpeedUP**	
		Chunk 1	**Chunk 8**	**Chunk 1**	**Chunk 8**
1000 bodies	0.02405	0.06378	0.06378	*0.38*	*0.38*
1750 bodies	0.07504	0.15384	0.15384	*0.49*	*0.49*
2500 bodies	0.15105	0.26103	0.26103	*0.58*	*0.58*
3250 bodies	0.25275	0.40392	0.40392	*0.63*	*0.63*
4000 bodies	0.38469	0.58451	0.58451	*0.66*	*0.66*
4750 bodies	0.54197	0.80258	0.80258	*0.68*	*0.68*

Table 3: *Average time per frame and speedup for OMP implementation. Run on University machine. Seq - sequential algorithm, forces only, Chunk 1 - 1 body per thread, Chunk 8 - 8 bodies per thread.*

	Seq	OMP no wait		OMP wait 50 ms		OMP wait 100 ms	
		Chunk1	Chunk 8	Chunk1	Chunk 8	Chunk1	Chunk 8
1000 bodies	0.02405	0.06378	0.06378	0.06613	0.06590	0.01780	0.01668
1750 bodies	0.07504	0.15384	0.15384	0.15377	0.15323	0.12653	0.12239
2500 bodies	0.15105	0.26103	0.26103	0.26174	0.26115	0.23039	0.23218
3250 bodies	0.25275	0.40392	0.40392	0.40572	0.40627	0.34791	0.34809
4000 bodies	0.38469	0.58451	0.58451	0.58804	0.58790	0.49990	0.50006
4750 bodies	0.54197	0.80258	0.80258	0.81933	0.82597	0.62711	0.64550
1000 bodies	*Speedup ->*	*0.37709*	*0.37709*	*0.36368*	*0.36498*	*1.35086*	*1.44227*
1750 bodies	*Speedup ->*	*0.48777*	*0.48777*	*0.48799*	*0.48973*	*0.59307*	*0.61314*
2500 bodies	*Speedup ->*	*0.57866*	*0.57866*	*0.57708*	*0.57838*	*0.65560*	*0.65056*
3250 bodies	*Speedup ->*	*0.62576*	*0.62576*	*0.62297*	*0.62214*	*0.72649*	*0.72612*
4000 bodies	*Speedup ->*	*0.65815*	*0.65815*	*0.65420*	*0.65435*	*0.76955*	*0.76930*
4750 bodies	*Speedup ->*	*0.67528*	*0.67528*	*0.66148*	*0.65616*	*0.86423*	*0.83961*

Table 4: *OMP algorithm performance with various wait times. Wait times not included in recorded times, as it is not part of force calculation.*

	Seq	OMP wat 100 ms		SpeedUP	
		Chunk1	Chunk 8	Chunk1	Chunk 8
1000 bodies	0.03842	0.02130	0.02630	*1.8*	*1.46*
1750 bodies	0.11806	0.08670	0.08597	*1.36*	*1.37*
2500 bodies	0.24086	0.16197	0.15886	*1.49*	*1.52*
3250 bodies	0.40685	0.28117	0.24674	*1.45*	*1.65*
4000 bodies	0.61770	0.41319	0.36886	*1.49*	*1.67*
4750 bodies	0.86085	0.57073	0.53218	*1.51*	*1.62*

Table 5: *OMP algorithm performance on an Intel(R) Core™ i5-5200U @ 2.20GHz laptop. Waiting time **not** included in measurement.*

	Seq	OMP		SpeedUP	
		Chunk1	Chunk 8	Chunk1	Chunk 8
1000 bodies	0.03842	0.12130	0.12630	*0.32*	*0.3*
1750 bodies	0.11806	0.18670	0.18597	*0.63*	*0.63*
2500 bodies	0.24086	0.26197	0.25886	*0.92*	*0.93*
3250 bodies	0.40685	0.38117	0.34674	*1.07*	*1.17*
4000 bodies	0.61770	0.51319	0.46886	*1.2*	*1.32*
4750 bodies	0.86085	0.67073	0.63218	*1.28*	*1.36*

Table 6: *OMP algorithm performance on an Intel(R) Core™ i5-5200U @ 2.20GHz laptop. Waiting time included in measurement.*

5. Conclusion

The project succeeded in finding a parallel implementation that speeds up the algorithm. The GPU implementation excels at calculating large numbers of bodies (n > 1000). The SIMD architecture seems to be a perfect fit for the n-body problem. The OpenMp implementation has shown some interesting and unexpected behaviour. The computations performed with this algorithm were correct, but the execution time was inconsistent across different machines. Most configurations were slower, but having a wait of 100 ms each frame improved the overall performance of simulations with the highest numbers of bodies tested.

6. References

1. Coursework materials. URL: https://github.com/kevin-chalmers/set10108
2. Fermi architecture whitepaper. URL: https://www.nvidia.co.uk/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf