SET10108 Concurrent and Parallel Systems Coursework 2 N-Body Simulation

Adam Blance 40161070

December 3, 2017

Abstract

In this course-work the task was to create an N-Body simulation, then solve the N-Body problem using several paralleiziation techniques. To complete this task the two approaches used, these were OpenMP and CUDA. This allowed for a direct comparison between the GPU and CPU. These were measured in two different ways, the first being time taken to complete the simulation and the second being the frame rate the application achieved during its run time. These results were then displayed in tables and discussed. Proving that paralleiziation is overall faster on the GPU, although paralleiziation on the CPU is simpler and quicker with smaller data sets.

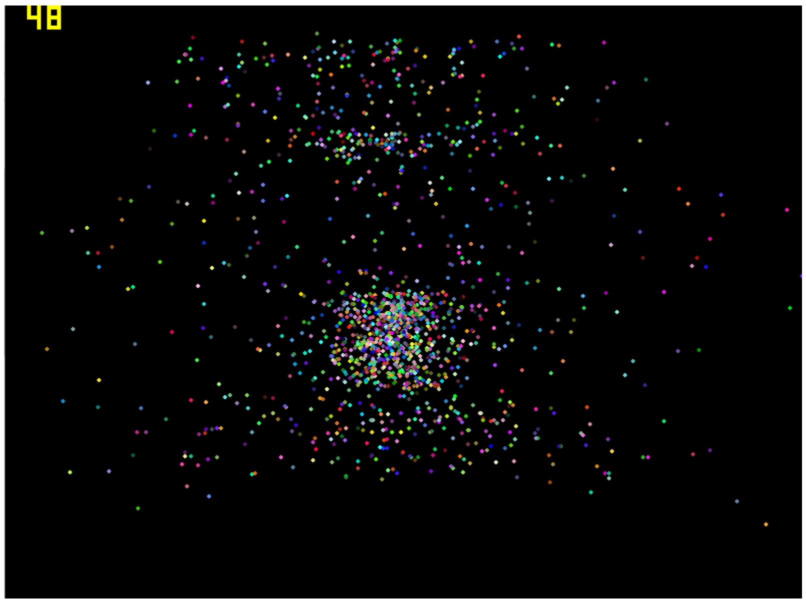


Figure 1: An N-Body simulation.

1 Introduction

As more and more of our world becomes heavily reliant on data found online, it is becoming of higher importance to learn how to parallelize this data, allowing data to be processed as quickly as possible. Learning how to parallelize sequential systems has become a very necessary skill to possess in your repertoire.
In this course-work the task was to parallelize an N-body algorithm. An N-Body algorithm is a simulation where a particle is affected by the all other particles according to the laws of physics. Issues occur when large amounts of particles are rendered on the screen, as all particles must be effected by all other particles causing the number calculations to increase exponentially. This causes any program running an N-Body simulation sequentially to struggle. This is known as the N-Body problem. In this report several parallelization techniques were used to solve the N-Body problem and then compared.

2 Inital Analysis

The first task in the course work was choosing which of the three tasks posed to us would be undertaken. It was decided that N-Body simulation would be the task that would be selected. The reason for deciding on N-Body simulation was because research into JPEG compression found little results and while prime numbers search was a possibility, N-body was chosen down to personal preference.
Once this was decided on research was conducted into N-Body simulation and which algorithm found online would be used to parallelise. During the search many possible candidates were found but the best one was harrism/mini-nbody . The reason for this was it because of its relative simplicity allowing for graphical output to be easily added into the code. After that small tweaks were made to the N-Body simulation and using Allegro 5, a graphical interface was created. From this point the initial analysis was conducted.

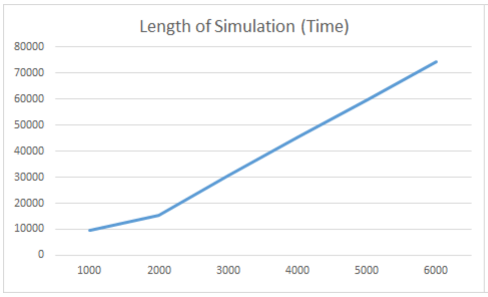
2.1 First Results

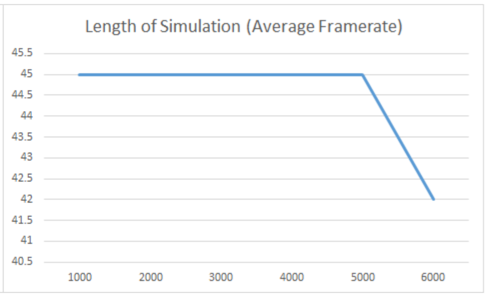
It was decided that three things would be tested to gather the baseline data of the application, to look for any bottlenecks. These were:

- Length of Simulation.
- Size of Times Step.
- Number of Particles.

These three factors were measured by the execution time, this was calculated by using the system clock that came in the chrono standard library. Then printing to the console, also taken was the average frame-rate, this was measured using Fraps. The tables for the measurements are shown below.

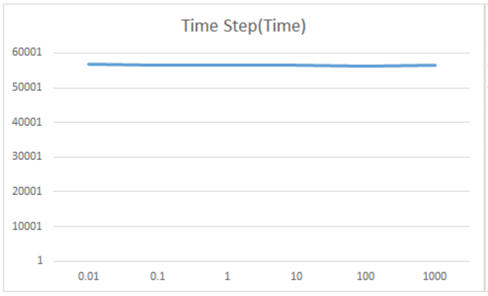
2.1.1 Length of Simulation

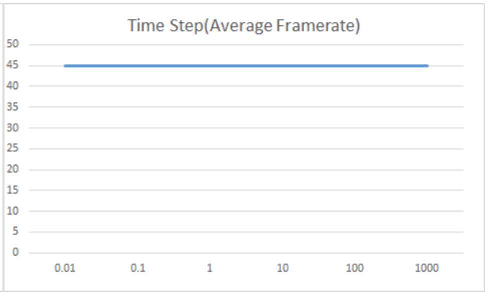




The reason for measuring the length of the simulation was to see if a drop off occurred at any point and performance suffered because of this. This was measured in number of iterations of the simulation. As shown in the results no bottleneck occurred at any point as the time taken increased at steady amount as was expected. the frame-rate also did not suffer and stayed steady. An anomaly that occurred during this test was that the frame-rate dropped to 42 fps at 6000 iterations rather the holding at 45 fps like the other tests this probably had nothing to do with code and with assumed to be an anomaly.

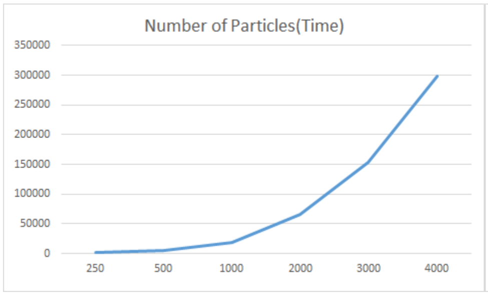
2.1.2 Size of Times Step

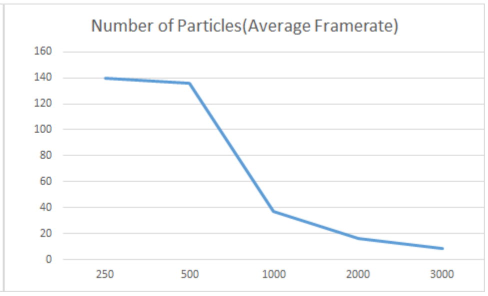




The second test that was taken was to check if changing the time step effected the results. This could occur if drawing objects on screen caused a bottleneck and the speed of moving the on screen could cause the frame-rate to drop. As shown in the results, this is not the case. The frame rate and speed both stayed the same the entire time the programme was run.

2.1.3 Number of Particles





The final measurement that was taken was Number of Particles the graph show clearly that a bottleneck occurs as the number of particles increases, this is because increasing as the number of particles increases, the number of calculations the PC must perform increases exponentially. The aforementioned test clearly shows the place that must be parallelized. The code that will be parallelized is shown below.

```
for (int i = 0; i < n; i++)
{
        //code for calculating the n-body simulation
        float Fx = 0.0f, Fy = 0.0f;
        for (int j = 0; j < n; j++)
        {
                float dx = p[j]->x - p[i]->x;
                float dy = p[j]->y - p[i]->y;
                float distSqr = dx*dx + dy*dy;
                float invDist = 1.0f / sqrtf(distSqr + SOFTENING);
                float invDist3 = invDist * invDist * invDist;
                Fx += (p[i]->mass) * dx * invDist3;
                Fy += /*(p[i]->mass )*/  dy * invDist3;
        }
        p[i]->vx += dt* Fx;
        p[i]->vy += dt* Fy;
}
```

3 Methodology

To solve the bottle-neck that was defined in the initial analysis several parallelization techniques were used. These were OpenMP and CUDA. The idea behind using these approaches was to be able to compare parralelization on the GPU against parralelization on the CPU.They would be measured by increasing the number of particles on screen and their results would be presented below.
To keep the values constant each test was run 11 times and an average value was calculated with the first value being ignored to allow the program to warm up. Another step that was taken to keep consistency between results was to keep the hardware constant on all tests. The hardware was:

- CPU- i7-4790k @ 4.00 GHz 4 hardware cores.
- RAM- 16.0 GB
- GPU- NVIDA GeForce GTX 980
- OS- Windows 10 64-bit

To evaluate these techniques two values were quantified, these being the time taken to complete the simulation and the average frame-rate. To calculate the time taken this system clock within Visual Studio was used. This was important to present in the results as it is the most common and easiest way to compare with a sequential program. To measure the Frame-rate Fraps was used. The reason for this is because in most simulations the frame-rate is more important then the time taken. The frame-rate was capped at 136 frames per second, this was because anything above 90 fps becomes unnoticeable. Finally speed-up would be calculated as this was deemed to be the best way to compare the two approaches.

3.0.1 OpenMP

The first approach was to use OpenMP, OpenMP is a library to support shared memory concurrency that works in the CPU, allowing it to be portable and added easily to the program. It can be easily used in conjunction with other approaches allowing more threading to be added at a later stage.
To implement OpenMP was fairly straight forward as the bottleneck had already been identified in the initial analysis. As OpenMP deals very well with for loops the code was very simple to apply to the application.

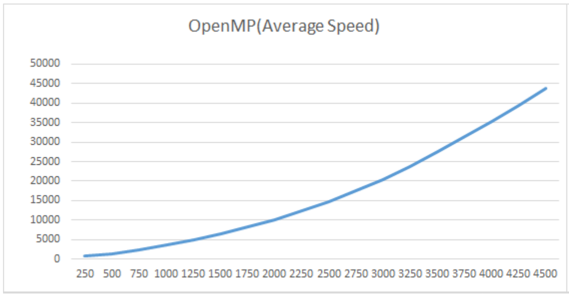
3.0.2 CUDA

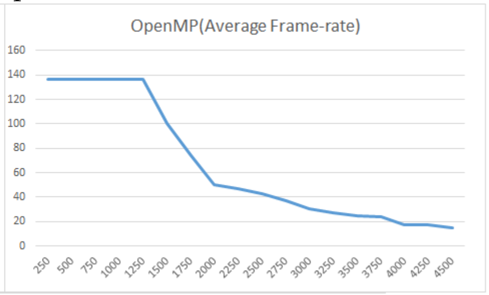
CUDA is a parallel computing platform and application programming interface model created by Nvidia. It works inside the GPU to allow for great speed-up potential. CUDA was designed to work with programming languages such as C and C++ which makes it easier to use as a product compared to other APIs. Its use of shared and unified memory, which allows multiple programs to use the same memory.
To carry out the task of adding CUDA to the application, a different approach was taken as CUDA is more complicated to add then OpenMP. Once CUDA was installed in Visual Studio it was discovered that the code would have to be completely reworked to add CUDA into the programme. This may not have been the case but a lack of overall knowledge in the background of CUDA restricted the outcome of the project.
The place that the parrelization was to be added would be the same place as OpenMP however before this could be implemented the code needed to be converted into a kernal. After this memory was allocated for CUDA this involved calculating how much data would be needed to be used. This was done by multiplying the size of the particle struct by the number of particles.

4 Results

4.1 OpenMP

The first results that are shown are the OpenMP results.



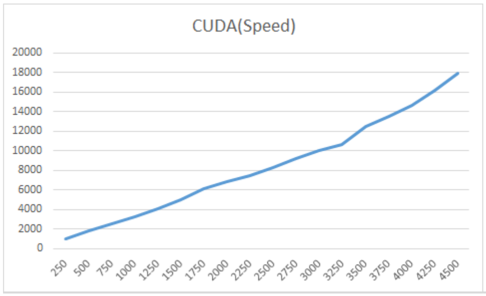


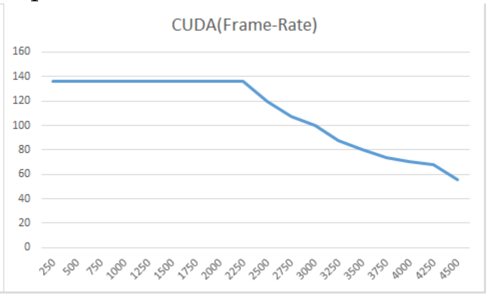
Particles	250	500	1000	2000	3000	4000
Speed Up	1.808103	3.978991	5.094877	6.555513	7.544375	8.464232

In the average speed the results show a smooth curve in the upward direction, this is probably not a good thing and shows at large values, bottlenecking would still occur.
The frame-rate has a sharp decline starting at 1250 particles, this was in keeping with the idea that a bottleneck was still occurring albeit at a smaller rate.

4.2 CUDA

The next set of results shows the output from the CUDA tests.

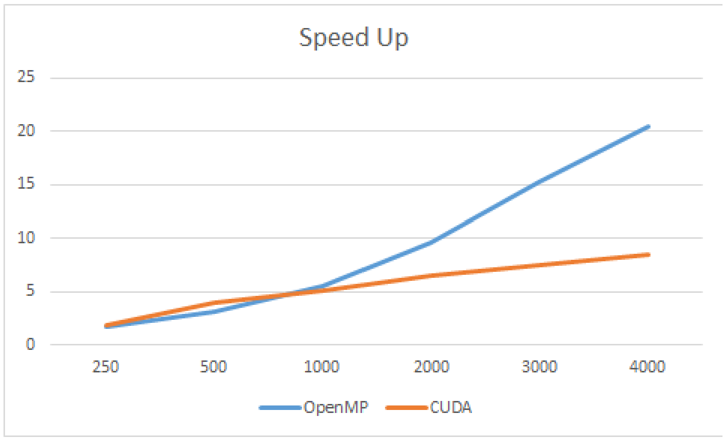




Particles	250	500	1000	2000	3000	4000
CUDA	1.67042	3.126797	5.57574	9.662448	15.30934	20.41299

From this no obvious bottleneck occurs with the line being nearly straight, and gradient of the line is very gradual with the simulation remaining under 100,000 seconds up until around 3000 particles.
The frame rate of CUDA tests stays constant up until 2250 particles,after this the drop off is rather small with 4250 particles still being above 60 frames per second.

4.3 Comparison



The speed-up calculation shows that at small amounts of particles, OpenMP is the better parralelization technique however above 1000 particles CUDA is the more superior of the two with the gap only growing between the two at extreme sizes of data.
The frame-rate of CUDA also stays stable to a larger amount of particles then the OpenMP this keeps the idea that CUDA is better at large amounts of data constant.

5 Conclusion

In this project two of the most well-known parralelization techniques were correlated against each other and while CUDA had the highest speed up of the two for large amounts of data. OpenMP on the other had is marginally better at smaller data sets. It is also important to note that OpenMP is more useful to a weak programmer because of its easiness to set up and simplicity to use well.

5.1 Limitations

The main limitation of this project was the hardware. While the computers that these tests were performed was very powerful. It still became a limiting factor when running large amounts of particles.

5.2 Future Work

Several optimizations could be made to CUDA system including using parallel compilers which come include in the CUDA library. BEcause of OpenMPs simplicity it is harder to optimize later on however combining it with CUDA could be investigated to see if it increased speed up even more. Another idea to improve the overall performance of the program could be to make some assumptions about N-Body simulation, ie group particles mass together if their in a certain direction.

5.3 Final Thoughts

During this project I enjoyed creating and implementing an N-Body simulation. While the biggest challenge in this project was implementing CUDA it was still very interesting to properly implement it into my own programme. Working with an N-Body simulation allowed for a swift discovery of the bottleneck and allowed more focus to be placed parralelizing the algorithm rather then the initial analysis.