An analysis and comparison of techniques of parallelisation of an n-body simulation

40203970

Concurrent and Parallel Systems (SET10108)

Abstract

This report presents a comparison and analysis of different parallelisation methods of a brute force n-body simulation: CPU-based (Multithreading with OpenMP), Distributed (MPI), and GPU-based (CUDA).

1 Introduction and Background

The initial sequential code was mostly written from scratch, except for some constants and the code for generation of bodies which were copied from an open-source project found on GitHub [1]. The library `SFML` [2] was used to display the bodies. The program consists of a class which defines a body and a class which carries out the simulation. When a `Simulation` object is created, all the bodies are initialised (the number of the bodies can be changed by changing a constant value). A random position is generated for each body, so that the bodies are arranged in a disk. Additionally, all bodies have equal mass (calculated by dividing predefined total system mass by the number of bodies), random velocity and their acceleration is initially set to 0. Lastly, a very heavy body with no acceleration and velocity is put in the centre of the "universe" to simulate a sun. When the simulation is started, an `SFML` window is opened and a loop is started which runs until the window is closed, checking for `SFML events`, updating the bodies and rendering them. The update method calculates the forces resulting in the interaction of bodies. This implementation uses a naïve brute force method which iterates through the list of bodies and calculates the results of the interaction of each body with every other body. Lastly, the position of each body is updated based on the forces acting on it, and the forces are reset.

The force acting on a body (b_i), resulting from its interaction with another body (b_j) is calculated with the following formula:

$$F_i = \Delta t \times G \times \frac{m_i \times m_j}{d(d^2+s^2)}$$

where F_i is the *resulting force*, Δt is the *time step* (fixed in this case), G is the *gravitational constant*, m_i is the *mass* of b_i, m_j is the mass of b_j, s is a softening constant and d is the length of the vector calculated by substracting the position of b_j from the position of b_i. The sum of the forces acting on body b_i is stored and after it has interacted with all other bodies, its new position is calculated.

All of the tests described in the next sections were run on the same computer with specifications listed in **Table 1**.

Table 1: *PC specifications*

CPU	Intel Core i5-8300H
GPU	NVidia GTX 1050 Ti
RAM	8 GB
OS	Windows 10 Pro x64

Additionally, all test programs were compiled in Release mode with compiler optimisations using the *MSVC* compiler.

2 Initial Analysis

The Visual Studio Profiler was used to analyse the program and find the most CPU-intensive functions. The results showed that that the call to `Simulation::start()` in `main.cpp` was consuming the most CPU time: 98.85%. This, however, is not surprising as this function runs the whole simulation. The second slowest part of the program was the call to `Simulation::update()` (96.02%) which is run every frame. This is also to be expected as this function uses nested loops to iterate through all bodies and calculate the forces acting on them. Since there are no dependencies between the elements of the list, this function can be parallelised. Additionally, the call to `Body::interact()` in the update method was also found to be consuming a lot of CPU time: 91.42%. The methods itself is not suitable for parallelisation, however, it can be run in parallel in the `Simulation::update()` method. The rest of the program is much less CPU-intensive, however, the function `Body::generate_bodies()` can also be parallelised, even though it is not shown to be CPU-intensive. It simply generates a set amount of bodies and adds them to an `std::vector`. Since each body is independent, the work can be split across multiple threads, however the resulting vector will have to be protected.

Additionally, after experimenting with the value for number of bodies, it became obvious that the performance of the program is greatly reduced as the number of bodies is increased. This is due to the update of bodies using a nested for loop, and both the inner and outer loop run NUM_BODIES times. This means that the simulation takes

$O(n^2)$ time to compute body forces as it is based on the number of bodies. However, the number of iterations can be reduced as Newton's Third Law applies to the simulation. That is, if a body *A* exerts a force on body *B*, then body *B* must exert a force of equal magnitude and opposite direction back on body *A*. Therefore, when calculating the force acting on body b_i, the force of body b_j can also be calculated: $F_j = -F_i$. Additionally, this means that the inner `for loop` can be optimised. Instead of iterating over all bodies, the loop can start at $i + 1$ and stop at number of bodies. Making this change decreases the number of iterations required in half and decreases the time taken to calculate the forces for all bodies.

3 Methodology

3.1 Multithreading with OpenMP

The first attempt at parallelisation was using *OpenMP*. The reason this method was the first choice was that it would be simple to implement and would show roughly what results from parallelisation could be expected. As explained above, the `Simulation::update()` function was found to be suitable for parallelisation. The outer `for loop` was parallelised with a `#pragma omp parallel for` statement, using the native hardware concurrency of the system (8 threads). That meant that the bodies (2048 at the time of the experiment) were split evenly across 8 threads and their force calculations were run in parallel. Attempting to parallelise the inner loop, however, did not yield any performance gains. Furthermore, the other loop in this method which updates the bodies' positions based on the accumulated forces, was also suitable for parallelisation. Unfortunately, the performance gains were minimal. This did not come as a surprise as the loop was not CPU-intensive to begin with. Lastly, after experimenting with loop scheduling, it was discovered that the default settings (`schedule(static, 1)`) resulted in the best performance.

3.2 GPGPU with CUDA

The second attempt at parallelisation was using GPGPU techniques. *CUDA* was selected as the GPU of the PC used for testing was an NVidia GTX 1050 Ti and it is known that CUDA performs better than *OpenCL* on NVidia hardware.

The same function was parallelised (`Simulation::update()`) so that the results could be compared against CPU parallelism. It was expected that the GPU version would outperform the OpenMP implementation as n-body is a data-driven problem. GPUs are known to be extremely suitable for such problems due to their `SIMD` architecture.

A number of changes were required to the sequential program. All the body force calculations had to be carried out on the GPU. To achieve this, a kernel was written. **Algorithm 1** shows a simplified version of the algorithm for the kernel.

Algorithm 1: Algorithm for CUDA kernel to calculate forces

```
calculate global position $i$
for $j \leftarrow i + 1$ to $num\_bodies$ do
    calculate forces resulting from interaction of body $b_i$ with body $b_j$
end
```

Additionally, a second kernel was made which calculated the positions of the bodies based on the forces calculated from the first kernel. Initially, the kernels were run with 1024 threads per block. It was experimented with the values for block size, however, and the results are presented in **Section 4**.

3.3 Distributed parallelism with MPI

The implementation is based on a tutorial by the University of Saskatchewan [3]. A lot of changes had to be made to the sequential program. Firstly, object orientation seemed unsuitable for OpenMP. Additionally, as suggested in [3], using a custom class or struct to store all data about bodies would require a derived data type which is generally slower than native MPI types. For this reason, the bodies' data (mass, position, velocity, force) was stored in separate arrays.

A communication construct called *ring pass* was used to achieve communication between processes. Processes are connected in a ring and process p communicates only with process $p \pm 1$. The general case is: process p sends data to process $(p - 1 + comm\_sz)\ \%\ comm\_sz$ and receives data from process $(p + 1)\ \%\ comm\_sz$ (where $\%$ is the modulo operator).

The basic algorithm is shown in **Algorithm 2**.

All the MPI tests were run locally due to the lack of a second PC. It is expected that the

Algorithm 2: Body force and position calculation algorithm with MPI

```
Broadcast masses
Scatter velocities and positions
while program is running do
    calculate forces resulting from body interactions in this process
    calculate forces resulting from body interactions with the other processes
    for each local body $b_i$ do
        calculate body index $i$ in the array
        calculate velocity $vel_i$ and position $pos_i$
    end
    MPI_Gather() positions and velocities into master process
    foreach position $pos_i \in positions$ do
        render circle at $pos_i$
    end
end
```

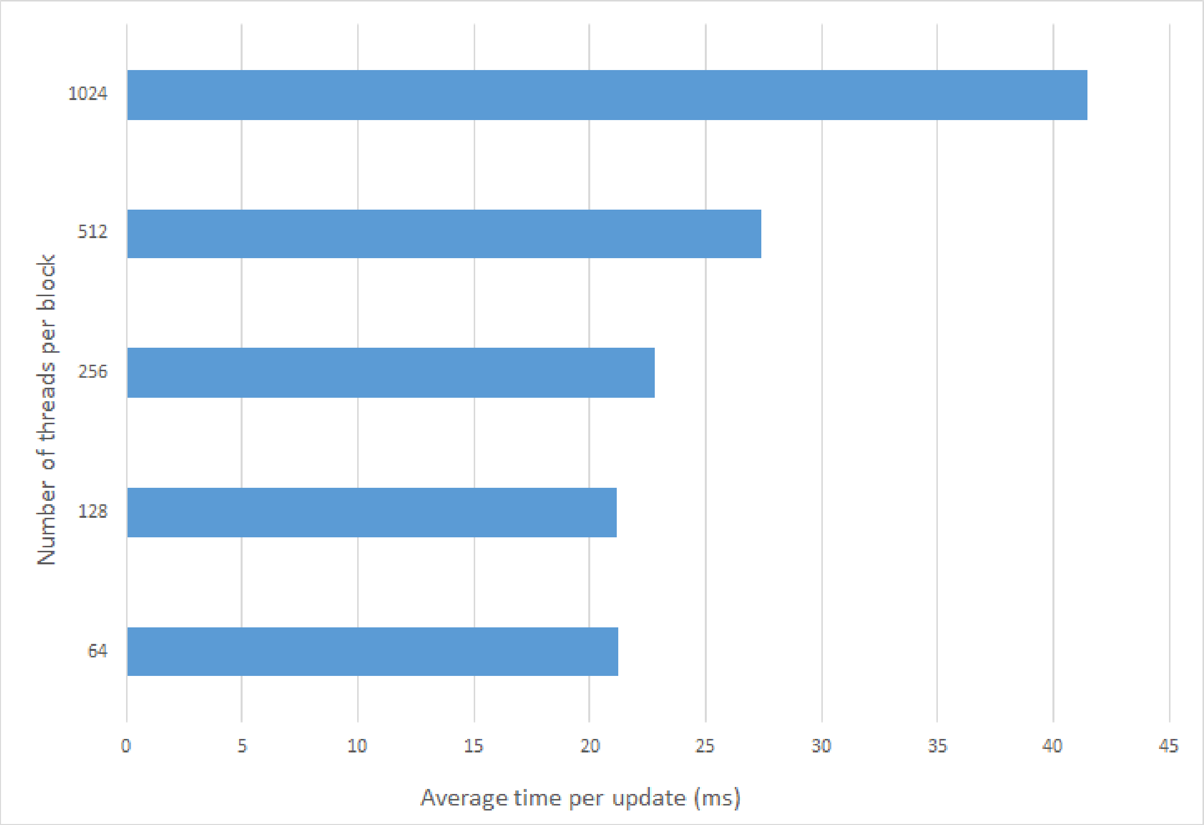
results would have been worse had the program been run in a distributed environment as latency and bandwidth-related issues might have occurred. Additionally, all test were run with 8 processes.

4 Results and Discussions

Firstly, it was tested if changing the block size for the kernel would affect performance of the CUDA version. **Fig. 1** shows the results. As it can be seen, performance definitely improves the lower the number of threads per block is. The most increase in performance is with 128 threads, and it seems that using less does not increase performance any further. As a result, the time for 1 update was almost halved from the initial tests.

What is more, it was soon realised that launching 2 kernels was inefficient as it required transferring the body data from the host to device and vice versa twice. Data transfer is known to be a very slow operation. To fix this problem, the 2 kernels which calculate forces and then update the bodies' positions were changed to `__device__` functions

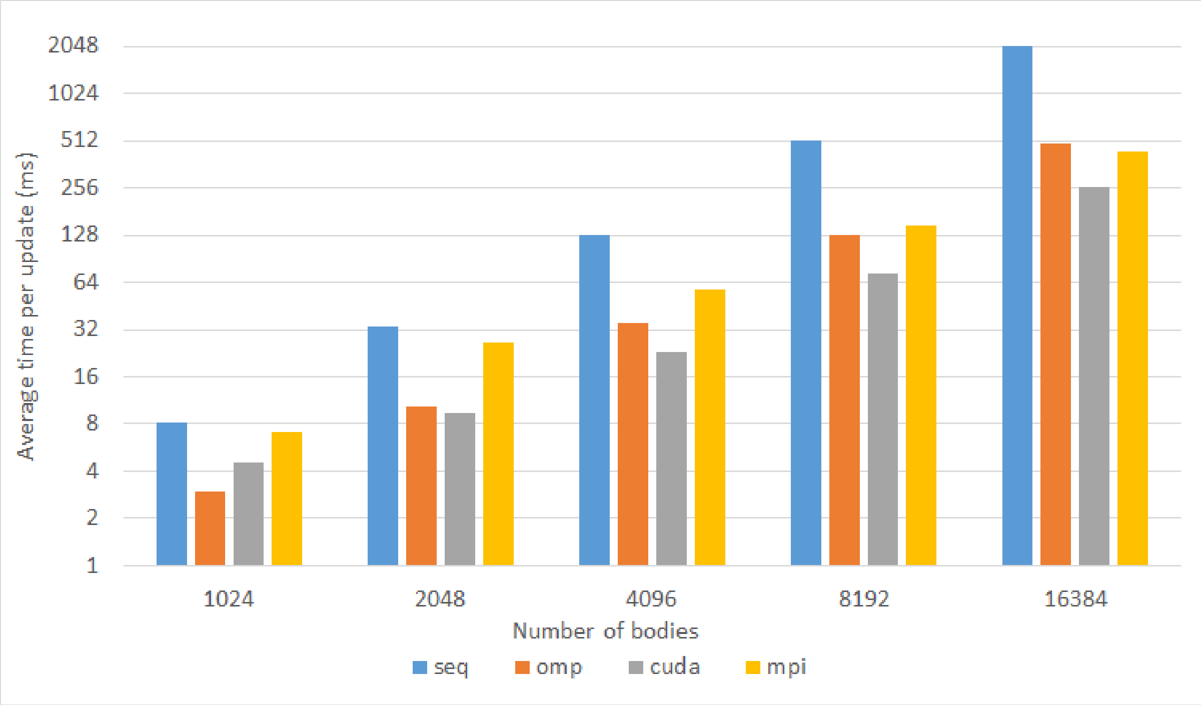
Figure 1: *A comparison of the time taken to run 1 update (average of 1000 runs) for different block sizes.*



and a new kernel was added which executed both functions. This meant that only 1 transfer of data from the host to the device was required. As a result, performance was improved, however it was only about 5ms faster in general. Lastly, some optimisations based on *GPU Gems* [4] were attempted. As suggested in the book, the body positions were stored in shared memory when the kernel is launched and then they can be accessed faster from threads. Another optimisation suggested was *loop unrolling*. Both techniques were implemented. As a result the output of the simulation was changed. There were small performance gains (the kernel would execute about 1ms faster on average), however the results were not considered as the output was different.

To compare the different parallelisation techniques of the program, the `Simulation::update()` function of each version was timed using the `std::chrono` functions. **Fig. 2** shows a comparison the time it takes to run 1 update for each program (average of 1000 updates). As it can be seen, when the number of bodies is low, OpenMP is the fastest version. This is probably due to optimisations it does in the background. With 2048 bodies, the CUDA version (with 256 threads per block) is slightly faster. When the number of bodies is greater than 2048, CUDA becomes consistently faster. The MPI version, however is the slowest regardless of the number of bodies in the simulation.

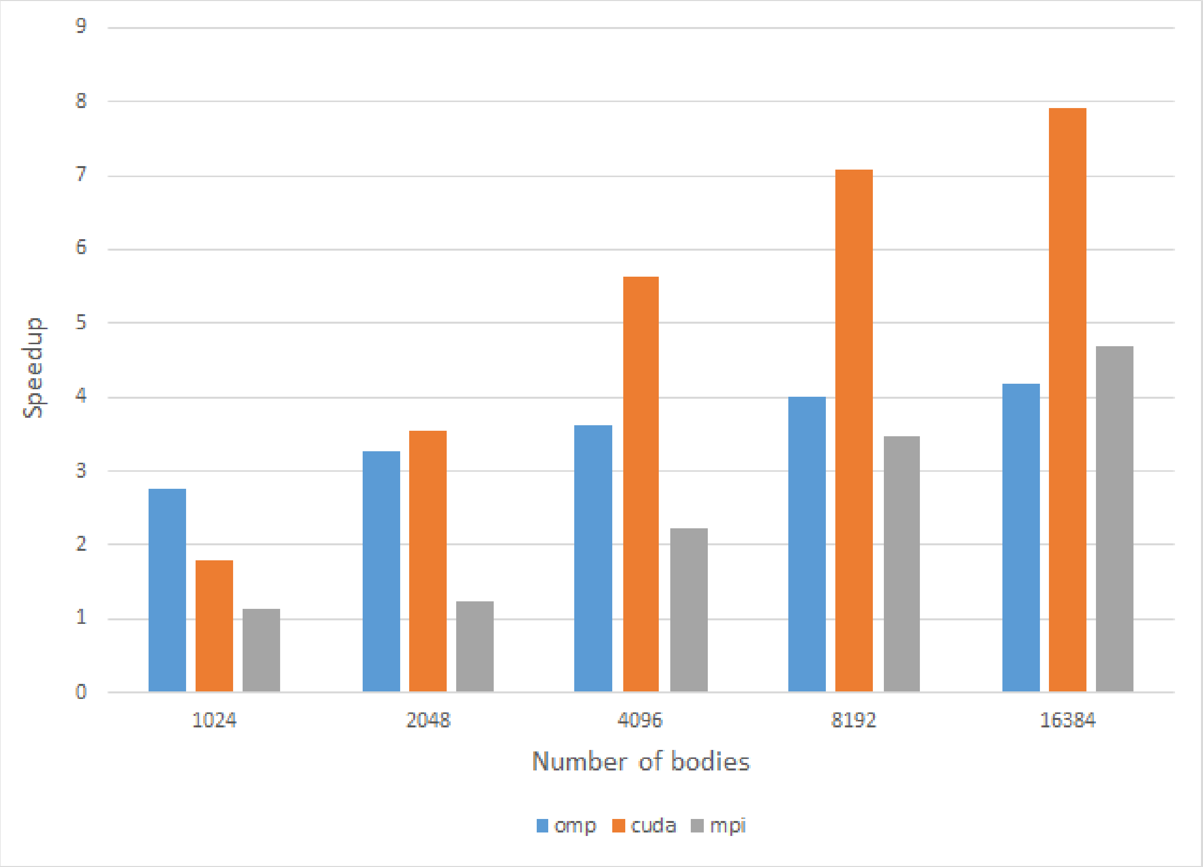
Figure 2: *A comparison of the time taken per update (average of 1000 updates) for each version (log base 2 scale).*



A comparison of the speedup of each implementation for different values of number of bodies is shown in **Fig. 3**. As it can be seen, CUDA achieved the best speedup values, except for when the number of bodies is less than 2048. This is probably due to low utilisation of the GPU. The speedup values for the MPI version are the lowest. This is probably caused by the fact that the program has to transfer a lot of data between processes, scattering and gathering arrays every update.

Fig. 4 shows a comparison of the efficiency of each implementation. However, calculating the efficiency of the CUDA program is tricky as the number of cores GPUs have is very large. Additionally, rarely does a program utilise all cores of a GPU. Due to this, the efficiency results for CUDA were very low, meaning that the comparison might be unfair.

Figure 3: *A comparison of the speedup achieved by each version at different values for number of bodies.*

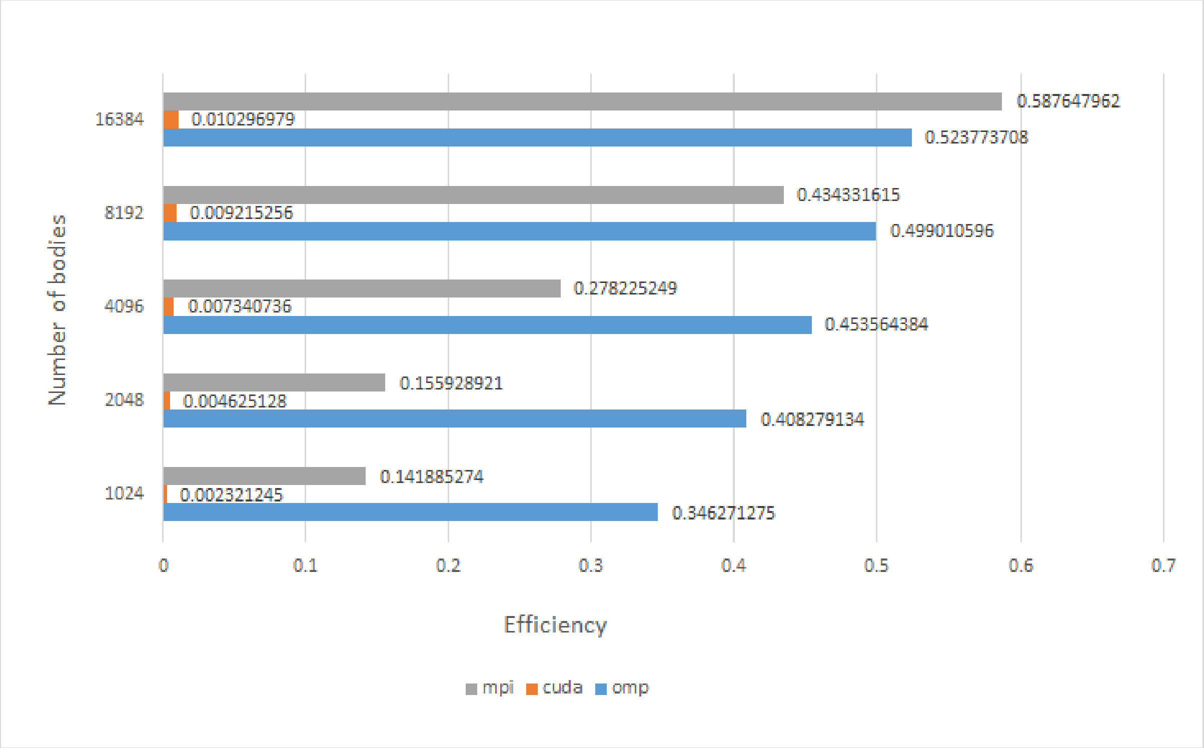


OpenMP had the best efficiency values as it performed well consistently, although it was outperformed by CUDA at more than 4096 bodies. Lastly, it can be seen that the efficiency of the MPI version is the worst.

5 Conclusion

In conclusion, it can be said that the n-body problem can definitely benefit from parallelisation. While CPU parallelisation yields great results (second best in the experiment), GPU parallelisation is the most suitable and consistently outperforms the other versions. This is due to the n-body problem being data-driven and GPUs being extremely suitable for such problems. Lastly, the slowest optimisation is the MPI version. Even though it provides some speedup, the cost of constant data transfers, scatters and gathers is probably too high to result in any satisfactory results. Furthermore, the speedup is

Figure 4: *A comparison of the efficiency of each version vs. the number of bodies.*



probably going to be even lower if the MPI version was run in a network due to latency and bandwidth issues.

While the CUDA version yielded best speedup, for a lower amount of bodies, an OpenMP implementation could be better. For more than 4096 bodies, however, the performance gains of CUDA are noticeable and it should be preferred.

Future work would include improving the sequential program by using a better algorithm (e.g. Barnes-Hut) and parallelising it to gain more performance. Additionally, as mentioned above, the function which generates bodies can also be parallelised, however the performance gains are expected to be minimal. Another possible experiment could be comparing the CUDA version with an OpenCL implementation.

References

[1] P. Whidden, “PWhiddy/Nbody-Gravity: A small but powerful nbody gravity simulator with a built-in renderer.” https://github.com/PWhiddy/Nbody-Gravity. (Accessed on 1/12/2018).

[2] L. Gomila, “SFML.” https://www.sfml-dev.org/. (Accessed on 2/12/2018).

[3] U. of Saskatchewan, “The n-body problem.” https://www.cs.usask.ca/~spiteri/CMPT851/notes/nBody.pdf. (Accessed on 16/12/2018).

[4] H. Nguyen, *Gpu Gems 3*. Addison-Wesley Professional, first ed., 2007.