

# SET10108 Concurrent and Parallel Systems

## Report for Coursework Part 2

Beej Persson, 40183743@live.napier.ac.uk

School of Computing, Edinburgh Napier University, Edinburgh

For part 2 of the coursework required for the SET10108 Concurrent and Parallel Systems module at Edinburgh Napier University an  $n$ -body simulation application's performance was to be evaluated and improved by utilising parallel techniques. This report documents one such investigation where the algorithm was parallelised and the difference in its performance was measured.

*Index Terms*—parallel,  $n$ -body, OpenMP, CUDA, C++11, performance, speedup, efficiency.

### I. INTRODUCTION AND BACKGROUND

**T**HE aim of this report is to evaluate the performance of an  $n$ -body application and attempt to improve this performance using parallel techniques. The  $n$ -body algorithm used initially processes sequentially on a single core of the CPU, but by investigating different parallel techniques the algorithm was changed to run on either multiple CPU cores or the GPU in an attempt to increase the performance.

#### A. $N$ -body Problem

The  $N$ -body problem is the problem of attempting to predict the positions and velocities of a group of bodies whilst they interact with each other via gravity. Finding a solution to this problem is generally done by calculating the sum of the forces acting on a each body in the system and using this to estimate its velocity and position. Given a body's mass,  $m_i$ , and

position,  $p_i$ , the force acting on it by another body,  $m_j$  and  $p_j$  is given by Equation 1 below (Reference [1]):

$$F_{ij} = \frac{Gm_i m_j (p_j - p_i)}{\|p_j - p_i\|^3} \quad (1)$$

Where  $F$  is the force and  $G$  is the gravitational constant. Using this equation, and knowing that  $F = ma$ , the acceleration of the body can be determined. Thus the new velocity and position can be found by multiplying the acceleration by a chosen timestep. To produce an  $n$ -body simulation this calculation can be done multiple times with a small enough timestep to accurately model the movement of the bodies in a space.

#### B. $N$ -body Simulation

The application used to generate an  $n$ -body simulation was written in C++ using a combination of two  $n$ -body algorithms available online. The structure of the application was based

on Mark Harris' [2], whilst much of the maths used to calculate forces was based on Mark Lewis' [3]. To ensure the algorithm operated as intended, the simulation was visualised by generating a data file of the body's positions and radii at each timestep and running a python script that converted that data into a video file.

## II. INITIAL ANALYSIS

Upon running the application a few times and changing the number of bodies and the number of iterations of the simulation, an idea of its baseline performance was gathered. Below, in Tables I and II, the results of this initial testing on the sequential algorithm can be seen.

TABLE I  
1000 ITERATION SEQUENTIAL ALGORITHM PERFORMANCE

Simulation Iterations = 1000	
Number of Bodies	Average Time / ms
64	20.3
128	80.87
256	322.96
512	1289.07
1024	5119.45
2048	20196.89
4096	80797.32

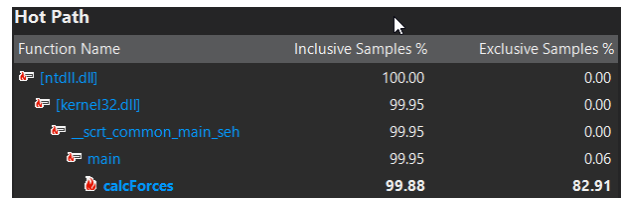
The application's *calcForces* method had complexity of  $O(n^2)$ : it contained a nested forloop, where each body in the system was compared against every other body. Given this, increasing the number of bodies results in the time taken to run 1000 iterations of the simulation to increase at an  $n^2$  rate, as seen in Table I. However, running more simulation iterations resulted in a linear increase in the time taken

TABLE II  
1024 BODIES SEQUENTIAL ALGORITHM PERFORMANCE

Number of Bodies = 1024	
Simulation Iterations	Average Time / ms
250	1337.40
500	2699.15
750	3884.60
1000	5214.85
1250	6523.30
1500	7799.55

to simulate 1024 bodies, with almost perfect positive correlation, as can be seen in Table II.

Further to this, a performance profiler was run to identify the possible bottlenecks and determine the best areas to attempt parallelisation to improve the application's performance. As can be seen in the code's hot path in Figure 1, the *calcForces* method, discussed earlier, was what took up the majority of processing time when there was a significant number of bodies.



Hot Path		
Function Name	Inclusive Samples %	Exclusive Samples %
[ntdll.dll]	100.00	0.00
[kernel32.dll]	99.95	0.00
_scrt_common_main_seh	99.95	0.00
main	99.95	0.06
calcForces	99.88	82.91

Fig. 1. A image showing the results of running Visual Studio's Performance Profiler, where the "Hot Path" is displayed.

This method, therefore, was the area that was parallelised in attempt to improve performance. However, when there was only 2 bodies the method executed very quickly showing that each individual comparison between bodies required little processing time. The high processing time when there was a large number of bodies was simply that there were so many comparisons to be made. This opened up GPU

parallelisation as a potential solution given the large number of lower clock-speed cores.

### III. METHODOLOGY

A systematic approach was undertaken to evaluate the performance of the algorithm and to measure any improvements in performance gained by parallelising the algorithm. The parallelising technologies were chosen based on the results of the initial analysis and on their ability to maximise the performance of the application.

The first step was to run a series of tests on the application to determine likely areas that could be parallelised and which technologies would be suitable, and to provide a baseline that the performance of the different parallel implementations could be compared to. These tests were all done on the same hardware, the relevant specifications of which are shown in table III. The details of the tests are shown in the testing subsection below.

TABLE III  
HARDWARE SPECIFICATIONS

CPU	i7-4790K 4 Core HT @ 4.00GHz
GPU	NVIDIA GeForce GTX 980
RAM	16.0GB
OS	Windows 10 Pro 64-bit

#### A. Parallelisation Techniques

After these benchmarks for the sequential algorithm were recorded, the chosen parallelising techniques were applied to the algorithm and some preliminary simulations were run, each time checking the visualised output of the

application. The intention here was twofold; to ensure that the techniques had been implemented correctly, that the parallelised algorithm was still producing the same simulation as the sequential application, and to gain an idea of their relative performance. The techniques used were OpenMP and CUDA, and the parallelisation was only applied to the *calcForces* method as it took the majority of processing time and in an attempt to reduce accidental speedup to the application beyond simply parallelising the sequential algorithm itself.

#### 1) OpenMP

OpenMP is an API that supports shared-memory parallel programming and allows some additional manipulations in the scheduling that were used in an attempt to increase performance. The pre-processor argument shown in Listing 1 was used to parallelise the outer for-loop, allowing the force calculation algorithm to be run across multiple threads.

```

1 void calcForces(Body *p, int numBodies) {
2 #pragma omp parallel for schedule(static)
3   for (int i = 0; i < numBodies; ++i)
4   {
5     /* nested forloop force calculations */
6   }

```

Listing 1. The OpenMP parallel for used to parallelise the shown forloop across the number of threads desired.

OpenMP's parallel for function comes with a *schedule* clause, seen in Listing 1, that was set to static as each iteration of the loop took the same amount of time to process. OpenMP was chosen as a parallel technique as the initial analysis showed that the *calcForces* method was taking a long time to compute when run sequentially, and parallelising this algorithm so that it could run on multiple threads simultaneously would improve the performance of the application. OpenMP was chosen as it pro-

vides this parallelisation simply and effectively across the available CPU cores.

## 2) CUDA

CUDA is an API and parallel computing platform created by NVIDIA that allows software to utilise a CUDA-enabled GPU's virtual instruction set and execute the application in parallel using compute kernels. CUDA allows the user to determine the number of *blocks* and *threads per block*, showing in Listing 2, to be used for the kernel method and some testing was done to determine the optimal ratios. Below is an excerpt from the CUDA application.

```

1 __global__ void calcForces(Body *p, int n) {
2   int i = (blockDim.x * blockIdx.x) + threadIdx.x;
3   if (i < n) {
4     /* nested forloop force calculations */
5   }
6 int main() {
7   ...
8   calcForces<<<BLOCKS,THREADS_PER_BLOCK<<<
9   >>>(d_p, numBodies);
10  ...
11 }
```

Listing 2. The *calcForces* method with CUDA adjustments made and how the kernel method is called within the *main*.

CUDA was chosen as a notable contrast to OpenMP as it runs on the GPU and would enable a comparison between their respective performance. Further to this, given the results of the initial analysis, executing the *calcForces* method in parallel on the GPU would provide significant speedup, especially for larger numbers of particles, due to the large number of available cores.

## B. Testing

The same series of tests that were run on the sequential algorithm in the initial analysis were then undertaken for each implemented parallelisation. These tests were done under the same conditions and on the same hardware

to eliminate discrepancies. For the majority of the OpenMP tests, the algorithm was run on the maximum number of threads available. CUDA allowed effective cores in use to be determined as attributes in the kernel method, therefore some initial testing was done on the performance of the application with the number of *threads per block* manipulated to identify the number of *blocks* and *threads per block* that would be used in the later tests.

The testing parameters used to evaluate the performance of the algorithms and the tests themselves are listed below.

For all tests the dependent variable being measured was the amount of time it took for the applications to produce the required number of iterations of the simulation. For each change in the independent variables, 100 tests were run and the time it took for the algorithm to produce the necessary values recorded, before the average run times were calculated.

### 1) CUDA's Threads Per Block

- Independent variables: threads per block (from 1 to 32 incremented by powers of 2).
- Constants: simulation iterations (1000), number of bodies (1024).

### 2) CUDA and OpenMP Comparisons

- Independent variables: number of bodies (from 64 to 4096 incremented by powers of 2)
- Shared constants: simulation iterations (1000).
- CUDA constants: threads per block (8), blocks (number of bodies divided by threads per block).
- OpenMP constants: threads (8).

### 3) Single Thread Tests

- Independent variables: each algorithm (sequential, CUDA, OpenMP).
- Constants: simulation iterations (1000), number of bodies (512), threads (1).

### C. Evaluation

The results of these tests were then collated and compared to the results from the sequential algorithm's testing and used as the basis for the evaluation of their respective performance.

To represent the improved performance, the speedup,  $S$ , and efficiency,  $E$ , of the algorithms was calculated using the formula shown in Equation 2 below:

$$S = \frac{T_{serial}}{T_{parallel}}, \quad E = \frac{S}{p} \quad (2)$$

Where  $T_{serial}$  and  $T_{parallel}$  are the sequential and parallel computational times respectively, and  $p$  is the number of processor cores. When calculating the efficiency of the OpenMP application,  $p$  was set to 4, which was the number of available CPU cores on the test hardware. However, determining  $p$  for the CUDA application was more complex. Whilst the GPU used to test the application was listed as having 2048 cores, that wasn't necessarily the number of hardware cores assigned when running the kernel method. Instead,  $p$  was calculated as the number of *blocks* multiplied by the number of *threads per block* set as attributes in the kernel method, as this results in the total number of threads doing work on the GPU.

## IV. RESULTS AND DISCUSSION

The results of the initial test on the CUDA application can be found in Table IV below.

TABLE IV  
CUDA PERFORMANCE: THREADS PER BLOCK

Number of Bodies = 1024, Simulation Iterations = 1000	
Threads Per Block	Average Time / ms
1	3126.17
2	1183.69
4	1051.98
8	1058.97
16	1117.77
32	1256.76

When the kernel method was called with only a single *thread per block*, the application took significantly longer to generate a 1000 iterations of the simulation of 1024 bodies than when using a higher number of *threads per block*. Whilst there was a noticeable increase in performance when run using 2 *threads per block*, the average time seemed to reach its lowest at around 4 or 8 *threads per block*, before again increasing at 16 or more *threads per block*. These results are visualised in Figure 2 below. The errors bars on this graph were determined by calculating the standard deviations of each set of 100 test results, which, as can be seen, were very small.

Given these results, 8 *threads per block* were used for all CUDA performance tests undertaken afterwards, and *blocks* were set to the number of bodies being tested divided by *threads per block*, to ensure each body would receive its own thread, maximising use of the available hardware.

The results from the performance testing done on the applications can be seen summarised in Table V below.

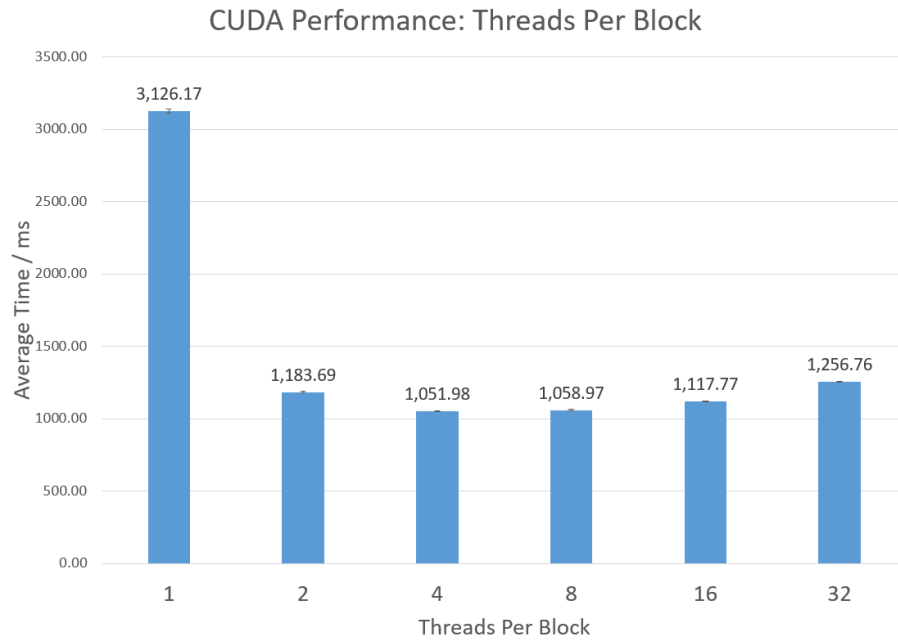


Fig. 2. A graph showing the average time it took the CUDA application to generate the simulation data when changing the allocated number of threads per block on the GPU.

TABLE V  
1000 ITERATIONS N-BODY SIMULATION

Number of Iterations = 1000			
Application	Sequential	OpenMP	CUDA
Number of Bodies	Average Time / ms		
64	20.39	15.46	105.75
128	80.87	45.81	156.79
256	322.96	148.66	285.48
512	1289.07	533.66	528.34
1024	5119.45	2007.81	1063.36
2048	20196.89	7820.01	2398.78
4096	80797.32	30628.95	6822.16

As discussed in the initial analysis there is an  $n^2$  positive correlation between the average time and the increasing number of bodies for the sequential application, but this also holds true for the OpenMP application. For the CUDA application, however, the relationship is more complicated. For a small number of bodies, the average time taken to generate an

$n$ -body simulation was higher than both the OpenMP and sequential applications. But as the number of bodies became increasingly large, the average times became significantly lower than the other applications. For both parallel applications, whenever there was a more significant number of bodies being simulated, the average times in which the applications produced the data were much lower than the sequential application.

Below, in Figure 3, is the graph of these results. The errors bars on this graph were once again determined by calculating the standard deviations of each set of 100 test results and are, again, barely visible. As the number of bodies to be simulated were incremented in powers of 2, the data is discrete and the  $x$ -axis has been displayed using a base-2 logarithmic scale. As a result of this the  $y$ -axis is also

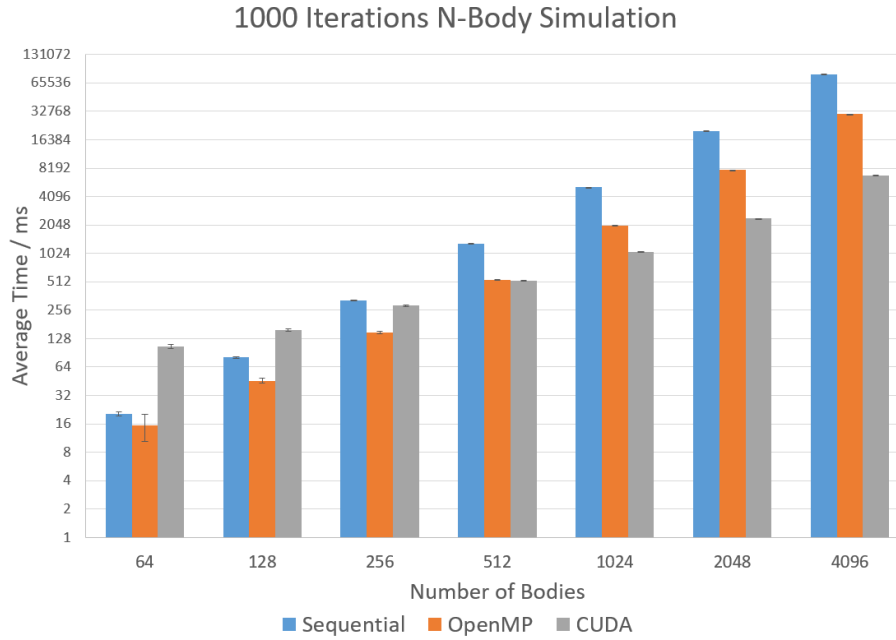


Fig. 3. A graph showing the average time it took each application to generate 1000 iterations of a simulation with differing numbers of bodies.

displayed using a base-2 logarithmic scale so that data does not appear skewed.

By running these averaged times through the formula shown in Equation 2, the speedup and efficiencies of these applications compared against the sequential application were calculated. The below table (Table VI) shows these results.

TABLE VI  
ALGORITHMIC SPEEDUP AND EFFICIENCY COMPARISON

Number of Iterations = 1000				
Application	OpenMP	CUDA	OpenMP	CUDA
Number of Bodies	Speedup		Efficiency	
64	1.319	0.193	0.330	0.024
128	1.765	0.516	0.441	0.032
256	2.172	1.131	0.543	0.035
512	2.416	2.440	0.604	0.038
1024	2.550	4.814	0.637	0.038
2048	2.583	8.420	0.646	0.033
4096	2.638	11.843	0.659	0.023

This table and its accompanying graphs,

Figures 4 and 5, show that for low numbers of bodies, OpenMP bested CUDA in terms of both speedup and efficiency when compared to the sequential application. When generating a simulation with 512 bodies or more, however, CUDA significantly outperformed OpenMP in terms of speedup, and was only improving as more and more bodies were simulated. In spite of these speedups, CUDA remained incredibly inefficient throughout, whilst OpenMP was nearly 3 times more efficient than the sequential application.

As discussed in the testing section earlier, a single threaded performance test was also performed to help contextualise these efficiencies.

Even when ran on a single thread the parallel algorithms generated the required data in less time than the sequential algorithm. In particular, the attempt at manual multi-threading produced significantly faster results. Therefore

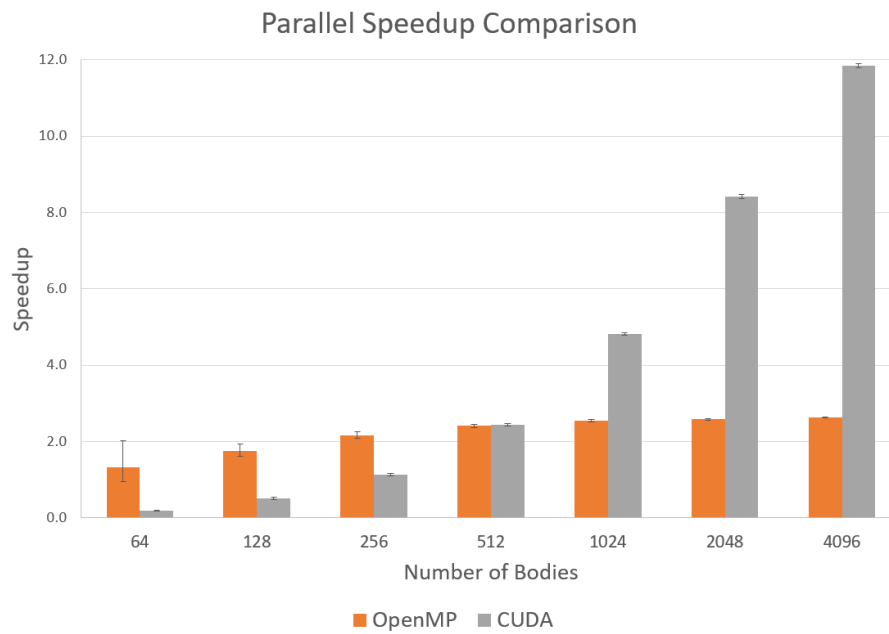


Fig. 4. A graph showing the speedup of the different parallel applications compared to the sequential application when generating 1000 iterations of a simulation of differing numbers of bodies.

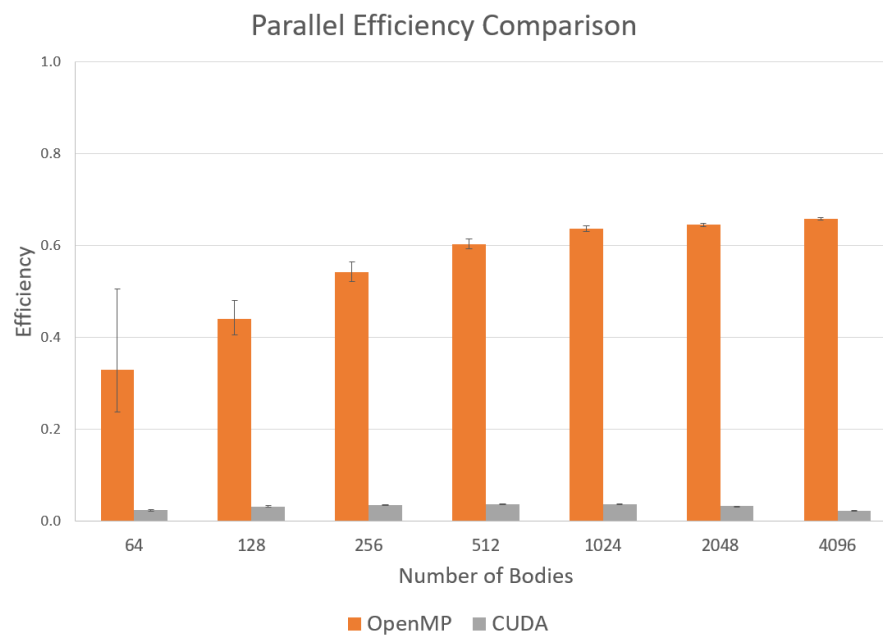


Fig. 5. A graph showing the efficiency of the different parallel applications compared to the sequential application when generating 1000 iterations of a simulation of differing numbers of bodies.

there were some incidental optimisations of the sequential algorithm when it was parallelised.

Further to this, Figure 5 shows that there was significant speedup for all the parallelised



algorithms when ran on 8 threads instead of 4. The hardware used had 4 cores but allowed for hyper-threading to 8 threads and was potentially a reason for why the efficiencies of more than 1 arose.

## V. CONCLUSION

### A. Explanation of Results and Evaluation of Performance

Across all the tested algorithms, when either the dimensions of the image or the number of samples per pixel were increased, the time it took to produce the data in the pixel vector used to generate the image would increase with near perfect positive correlation. Furthermore, as shown in Figures 2 and 3, independent of whether the image dimensions or the samples per pixels were changed, each of the parallelised algorithms significantly outperformed the sequential algorithm. Compared against each other, OpenMP with static scheduling was always outperformed by both OpenMP with dynamic scheduling and the manually multi-threaded algorithm. The manually multi-threading algorithm was generally equal to, or very slightly faster than OpenMP with dynamic scheduling.

Each of the parallelised algorithms had a speedup of 4 or more when compared to the sequential algorithm, and efficiency ratings more than 1. The reasons that the algorithm's efficiency ratings were able to be higher than 1 are likely twofold. Firstly, as shown in Figure 4, the parallelised algorithms were able to generate images quicker than the sequential algorithm even when operating on a single thread. This shows that there were optimisa-

tions implemented when parallelising the algorithm, this is especially true for the manually multi-threaded algorithm. Secondly, as shown in Figure 5, there was noticeable speedup from all the algorithms when going from running on 4 threads to running on 8. Given that there were only 4 physical cores and that the equation used to calculate efficiency uses this, a significant proportion of the speedup of the parallelised algorithms is due to the hyper-threading done by the CPU.

Combined, these two effects make a significant effect on the parallelised algorithm's efficiency ratings. Whilst both parallel algorithms that used OpenMP were only very slightly faster on a single thread than the sequential algorithm, the manually multi-threaded algorithm had a speedup of 1.42 whilst still operating on a single thread. This shows that the performance gained by manually multi-threading the algorithm had much less to do with running the algorithm over multiple threads than the performance gained by the OpenMP parallelised algorithms. The speedup from the hyper-threading, however was more even across the parallelised algorithms.

### B. Final Thoughts

The provided sequential algorithm originally took a considerable amount of time to produce large images of quality. The attempts at parallelising the algorithm were largely successful. They all performed significantly faster than the sequential algorithm and made the time to generate large images of high detail more manageable. However, whilst running on multiple threads did provide speedup, an important part of the increased performance of the

algorithms was due to external optimisations. The attempt at manually multi-threading the algorithm benefited the most from this and as such ended up outperforming both versions of parallelisation using OpenMP. In terms of increased performance purely due to running the algorithm across multiple threads the algorithm that was dynamically scheduled using OpenMP was the top performer.

## REFERENCES

- [1] K. R. Meyer, *Introduction to Hamiltonian Dynamical Systems and the n-body Problem*. Springer Science and Business Media, 2009.
- [2] M. Harris, *mini-nbody*.  
<https://github.com/harrism/mini-nbody>
- [3] M. Lewis, *NBodyMutable*.  
<https://gist.github.com/MarkCLewis>

## APPENDIX A

### RAYTRACER.CPP

---

```

1 // Required for manual threading to allow forLoopAlgorithm method to update pixel vector
2 vector<vec> pixels;
3
4 // Nested for loop method, to be manually multi-threaded
5 void forLoopAlgorithm(unsigned int threads, size_t dimension, unsigned int num_threads, size_t samples, _Binder<_Unforced, ←
    uniform_real_distribution<double>&, default_random_engine&> get_random_number, vec r, vec cx, vec cy, vector<sphere> ←
    spheres, ray camera)
6 {
7     for (size_t y = threads; y < dimension; y += num_threads)
8     {
9         for (size_t x = 0; x < dimension; ++x)
10        {
11            for (size_t sy = 0, i = (dimension - y - 1) * dimension + x; sy < 2; ++sy)
12            {
13                for (size_t sx = 0; sx < 2; ++sx)
14                {
15                    r = vec();
16                    for (size_t s = 0; s < samples; ++s)
17                    {
18                        double r1 = 2 * get_random_number(), dx = r1 < 1 ? sqrt(r1) - 1 : 1 - sqrt(2 - r1);
19                        double r2 = 2 * get_random_number(), dy = r2 < 1 ? sqrt(r2) - 1 : 1 - sqrt(2 - r2);
20                        vec direction = cx * static_cast<double>(((sx + 0.5 + dx) / 2 + x) / dimension - 0.5) + cy * static_cast<double>(((sy ←
+ 0.5 + dy) / 2 + y) / dimension - 0.5) + camera.direction;
21                        r = r + radiance(spheres, ray(camera.origin + direction * 140, direction.normal()), 0) * (1.0 / samples);
22                    }
23                    pixels[i] = pixels[i] + vec(clamp(r.x, 0.0, 1.0), clamp(r.y, 0.0, 1.0), clamp(r.z, 0.0, 1.0)) * 0.25;
24                }
25            }
26        }
27    }
28 }
29
30 int main(int argc, char **argv)
31 {
32     random_device rd;
33     default_random_engine generator(rd());
34     uniform_real_distribution<double> distribution;
35     auto get_random_number = bind(distribution, generator);
36
37     // *** These parameters can be manipulated in the algorithm to modify work undertaken ***
38     constexpr size_t dimension = 256;
39     constexpr size_t samples = 16; // Algorithm performs 4 * samples per pixel.
40     vector<sphere> spheres
41     {
42         sphere(1e5, vec(1e5 + 1, 40.8, 81.6), vec(), vec(0.75, 0.25, 0.25), reflection_type::DIFFUSE),
43         sphere(1e5, vec(-1e5 + 99, 40.8, 81.6), vec(), vec(0.25, 0.25, 0.75), reflection_type::DIFFUSE),
44         sphere(1e5, vec(50, 40.8, 1e5), vec(), vec(0.75, 0.75, 0.75), reflection_type::DIFFUSE),
45         sphere(1e5, vec(50, 40.8, -1e5 + 170), vec(), vec(), reflection_type::DIFFUSE),
46         sphere(1e5, vec(50, 1e5, 81.6), vec(), vec(0.75, 0.75, 0.75), reflection_type::DIFFUSE),
47         sphere(1e5, vec(50, -1e5 + 81.6, 81.6), vec(), vec(0.75, 0.75, 0.75), reflection_type::DIFFUSE),
48         sphere(16.5, vec(27, 16.5, 47), vec(), vec(1, 1, 1) * 0.999, reflection_type::SPECULAR),
49         sphere(16.5, vec(73, 16.5, 78), vec(), vec(1, 1, 1) * 0.999, reflection_type::REFRACTIVE),
50         sphere(600, vec(50, 681.6 - 0.27, 81.6), vec(12, 12, 12), vec(), reflection_type::DIFFUSE)
51     };
52     // *****
53
54     // Create results file
55     ofstream results("data.csv", ofstream::out);
56
57     // Output headers to results file
58     results << "Test, Image Dimensions, Samples Per Pixel, Time, " << endl;
59
60     // Run test iterations
61     for (unsigned int j = 0; j < 100; ++j)
62     {

```

```

63 ray camera(vec(50, 52, 295.6), vec(0, -0.042612, -1).normal());
64 vec cx = vec(0.5135);
65 vec cy = (cx.cross(camera.direction)).normal() * 0.5135;
66 vec r;
67
68 // Required for manual threading
69 pixels.resize(dimension * dimension);
70
71 // Required for OpenMP
72 //vector<vec> pixels(dimension * dimension);
73 //int y;
74
75 // * TIME FROM HERE... *
76 auto start = system_clock::now();
77
78 // *** MANUAL MULTITHREADING ***
79 vector<thread> threads;
80
81 for (unsigned int t = 0; t < 4; ++t)
82 {
83     threads.push_back(thread(forLoopAlgorithm, t, dimension, 4, samples, get_random_number, r, cx, cy, spheres, camera));
84 }
85 for (auto &t : threads)
86     t.join();
87
88 // *** OPENMP *** (change scheduling to "dynamic" or "static")
89 // #pragma omp parallel for num_threads(4) private(y, r) schedule(dynamic)
90 // for (y = 0; y < dimension; ++y)
91 // {
92 //     for (size_t x = 0; x < dimension; ++x)
93 //     {
94 //         for (size_t sy = 0, i = (dimension - y - 1) * dimension + x; sy < 2; ++sy)
95 //         {
96 //             for (size_t sx = 0; sx < 2; ++sx)
97 //             {
98 //                 r = vec();
99 //                 for (size_t s = 0; s < samples; ++s)
100 //                 {
101 //                     double r1 = 2 * get_random_number(), dx = r1 < 1 ? sqrt(r1) - 1 : 1 - sqrt(2 - r1);
102 //                     double r2 = 2 * get_random_number(), dy = r2 < 1 ? sqrt(r2) - 1 : 1 - sqrt(2 - r2);
103 //                     vec direction = cx * static_cast<double>(((sx + 0.5 + dx) / 2 + x) / dimension - 0.5) + cy * static_cast<double>(((sy + 0.5 + dy) / 2 + y) / dimension - 0.5) + camera.direction;
104 //                     r = r + radiance(spheres, ray(camera.origin + direction * 140, direction.normal()), 0) * (1.0 / samples);
105 //                 }
106 //                 pixels[i] = pixels[i] + vec(clamp(r.x, 0.0, 1.0), clamp(r.y, 0.0, 1.0), clamp(r.z, 0.0, 1.0)) * 0.25;
107 //             }
108 //         }
109 //     }
110 // }
111
112 // * ...TO HERE *
113 auto end = system_clock::now();
114 auto total = duration_cast<milliseconds>(end - start).count();
115
116 // Output test no., variables and total time to results file
117 results << j + 1 << ", " << dimension << ", " << samples * 4 << ", " << total << endl;
118
119 // Output test information to console outside of the timings to not slow algorithm
120 cout << "Test " << j + 1 << " complete. Time = " << total << ". " << num_threads << endl;
121 array2bmp("img.bmp", pixels, dimension, dimension);
122
123 // Required for manual threading
124 pixels.clear();
125 }
126
127 return 0;
128 }

```

Listing 3. The source code for the raytracer.cpp including all the added parallelisation techniques.