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

THE 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_i is given by Equation 1 below (Reference [1]):

$$F_{ij} = \frac{Gm_im_j(p_j - p_i)}{||p_j - p_i||^3} \tag{1}$$

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	
Number of 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.

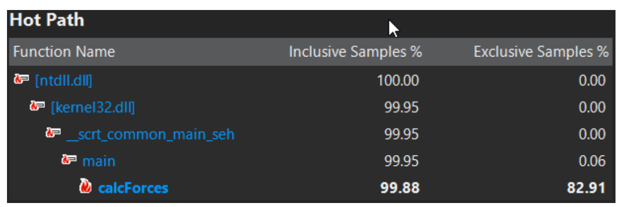

Fig. 1. A image showing the results of running Visual Studio's Performance Profiler, where the "Hot Path" is displayed.

This method, therefore, was the area that was parallelised in attempt to improve performance.

III. Methodology

The general approach is well defined, and the technological approach is likewise defined. Suitable justification based on initial analysis for why this approach was taken also made.

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 3 was used to parallelise the outer for loop, allowing the algorithm to be run across multiple threads.

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

Listing 1. The OpenMP parallel for used to parallelise the shown for loop across the number of threads desired. The removed nested for loops can be seen in Listing 1.

OpenMP's parallel for function comes with a *schedule* clause, seen in Listing 3, that can be used to change the way it spreads the workload across the threads. By default, OpenMP statically assigns each for loop iteration to a thread. However, if each iteration takes a different amount of time, it can be beneficial to use dynamic scheduling. When scheduled dynamically the threads can request work when ready and be assigned the next iteration that hasn't been executed yet. Given that this may further improve the performance of the ray tracing algorithm, both types of scheduling were tested.

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 was chosen as the initial analysis showed that the vast majority of processing time was spent calculating the forces on each particle. Executing the $calcForces$ method in parallel on the GPU would therefore provide significant speedup, especially for larger numbers of particles. CUDA allows the user to determine the number of *blocks* and *threads per block*, showing in Listing 1, 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.

```
__global__ void calcForces(Body *p, int n) {
  int i = (blockDim.x * blockIdx.x) + threadIdx.x;
  if (i < n) {
    /* nested forloop force calculations */
  }
int main() {
  ...
    calcForces<<<BLOCKS,THREADS_PER_BLOCK>>>(d_p, numBodies);
  ...
}
```

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

B. Testing

The same series of tests that were run on the sequential algorithm were then undertaken for each implemented parallelisation. These tests were done under the same conditions and on the same hardware to eliminate discrepancies. The testing parameters used to evaluate the performance of the algorithms is outlined below.

For the majority of the tests, the algorithms were run on the maximum number of threads available (the test hardware allowed up to 8). The dependent variable being measured was the amount of time it took for the algorithms to produce all the data in the pixel vector, which is used to generate the final image. The independent variables were the dimensions of the image, the number of samples per pixel. First the image size was kept constant at 256x256 whilst the number of samples per pixel was incremented, by powers of 2, from 4 up to 512. After this the samples per pixel was kept constant at 16 whilst the dimensions of the image were incremented, again by powers of 2, from 128x128 up to 1024x1024. For each change in the independent variables, 100 tests were run and the time it took for the algorithm to produce the data recorded, before the average run times were calculated. Further to this, 2 tests were run with a large image size of 1024x1024 and 1024 samples per pixel. This was only done twice due to how long it took for the sequential algorithm to generate the image, but was still useful for comparison. Additionally a few tests were done on each parallel algorithm where the number of threads they ran on were controlled. This was done to help contextualise whether the potential performance increase came from the number of additional threads or from changes to the algorithm itself.

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 efficiency, E, of the algorithms was calculated using the formula shown in Equation 2 below:

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

Where S is speedup, p is the number of processor cores (the test hardware had 4), and T_{serial} and $T_{parallel}$ are the sequential and parallel computational times respectively.

IV. Results and Discussion

The results from the performance testing done on the algorithms can be seen summarised in tables IV and V below. As discussed in the initial analysis there is an almost perfect positive correlation between the average time and the increasing samples per pixel and image dimensions for the sequential algorithm, but this also holds true for parallel algorithms. However, the times in which the parallel algorithms produced the data were much lower than the sequential algorithm, even at extreme values (Table VI).

Table IV and its accompanying graph (Figure 2) comparing the average time taken for the algorithms to generate the data required to produce a 256x256 image at different numbers of samples per pixel are shown below.

As the number of samples per pixel 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 displayed using a base-2 logarithmic scale so that data does not appear skewed. Here we can see that all the parallel algorithms show significant speedup over the sequential algorithm, with the manual multi-threading often being the fastest, if only by a small margin.

TABLE IV
256x256 Image Generation Performance Comparison

Image Dimensions = 256x256				
Algorithm	Sequential	OMP Static	OMP Dynamic	Manual
SamplesPerPixel	Average Time / ms			
4	762.57	187.13	163.98	168.87
8	1536.27	372.96	328.74	333.76
16	3059.91	728.10	655.39	649.57
32	6223.33	1427.95	1308.69	1283.35
64	12510.68	2804.45	2625.94	2570.45
128	24544.85	5589.91	5206.20	5100.70
256	48129.36	11119.07	10354.04	10108.84
512	95256.72	22138.99	20634.72	20348.64

The results of the next tests are shown in Table V. This table shows the average time it took the algorithms to produce the data for images of varying sizes at 16 samples per pixel. This data is visualised in Figure 3 below.

TABLE V
16 Spp Image Generation Performance Comparison

Samples Per Pixel = 16				
Algorithm	Sequential	OMP Static	OMP Dynamic	Manual
Image Dimensions	Average Time / ms			
128	746.34	187.23	165.81	168.42
256	3012.15	728.84	654.27	645.84
512	11881.24	2817.64	2582.84	2581.12
1024	47855.70	11098.78	10282.72	10190.96

Once again, the dimensions used for the size of the image were incremented in powers of 2, resulting in a similarly scaled x- and y-axis, as before. The data from these tests are in line with the previous results. Again it can be seen that the parallel algorithms outperform the sequential algorithm.

There were also a smaller number of tests done at extreme values to compare the algorithm's performances when generating large, high detail images. This is shown in Table VI

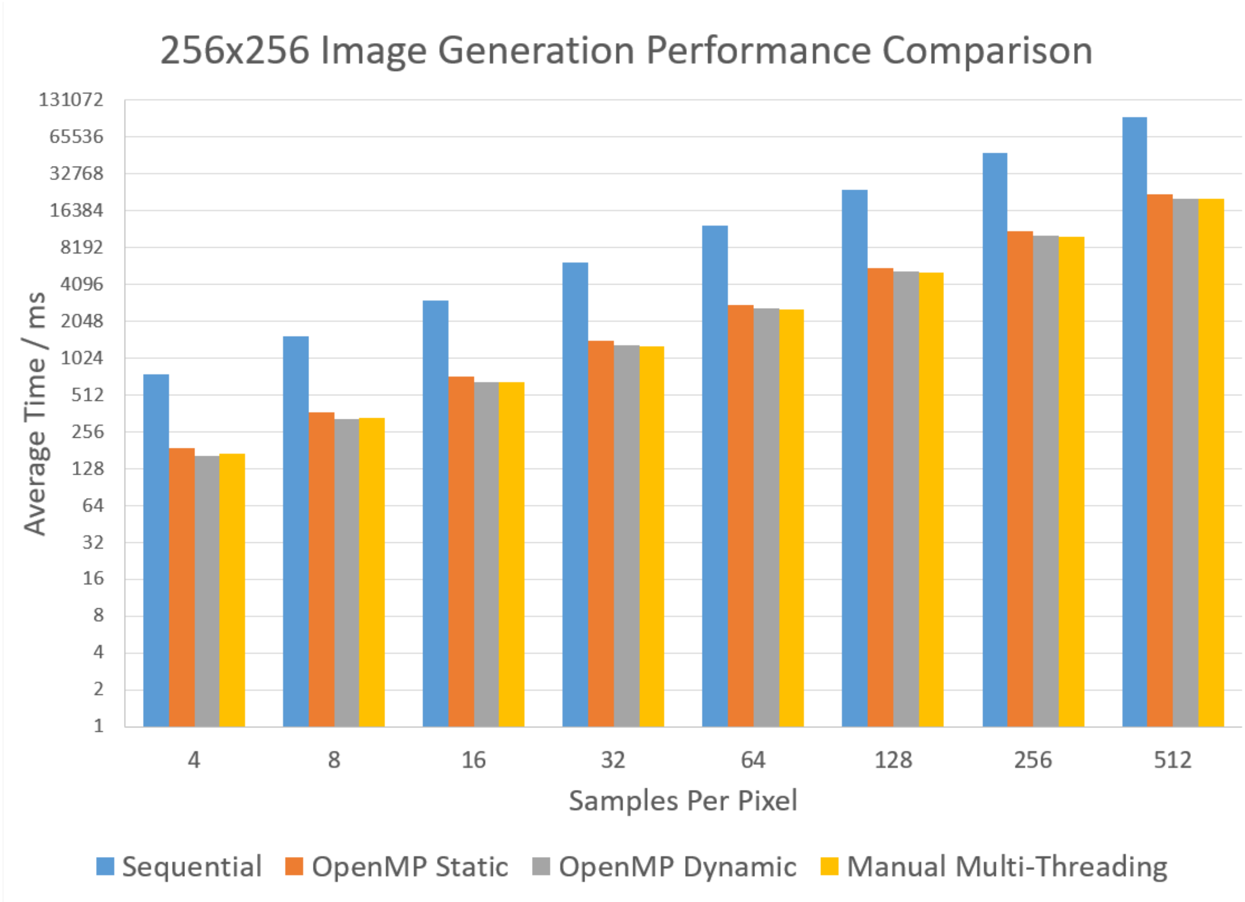

Fig. 2. A graph showing the average time it took each algorithm to generate a 256x256 image at different numbers of samples per pixel.

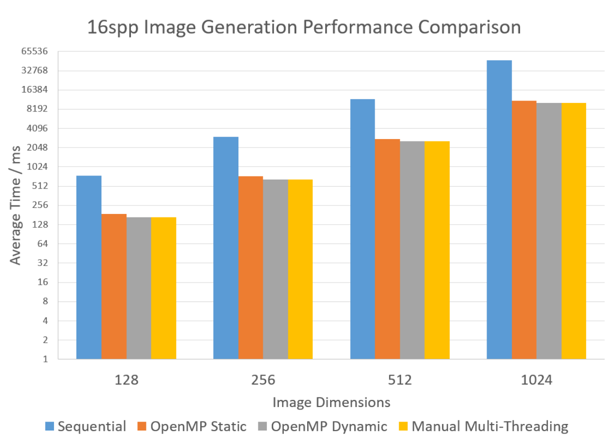

Fig. 3. A graph showing the average time it took each algorithm to generate images of varying dimensions at 16 samples per pixel.

below. The data was in agreement with the rest and showed that even at the extreme end the parallelised algorithms generated the images much faster.

TABLE VI
EXTREME SPP AND IMAGE SIZE PERFORMANCE COMPARISON

Image Dimensions = 1024, Samples Per Pixel = 1024				
Algorithm	Sequential	OMP Static	OMP Dynamic	Manual
Test No.	Time / ms			
1	3002631	709916	659067	651456
2	3001927	708573	658709	647313

By running these averaged times through the formula shown in Equation 2, the efficiency of these algorithms compared against the sequential algorithm was calculated. The below table (Table VII) shows these efficiencies for the first set of test results.

TABLE VII
ALGORITHMIC EFFICIENCY COMPARISON

Samples Per Pixel = 16			
Algorithm	OMP Static	OMP Dynamic	Manual
Image Dimensions	Efficiency		
128	0.997	1.125	1.108
256	1.033	1.151	1.166
512	1.054	1.150	1.151
1024	1.078	1.163	1.174

This table shows that some of the algorithms tested had efficiency ratings of more than 1, implying that they ran faster per thread than the sequential algorithm. As stated in the methodology, a few additional tests were done on the parallel algorithms where the number of threads they ran on was controlled. This allowed for a comparison to be made between each algorithm's single threaded performance in order to better contextualise the efficiency ratings of more than 1. Below are two graphs (Figures 4 and 5) which show some of the results from these tests.

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 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.

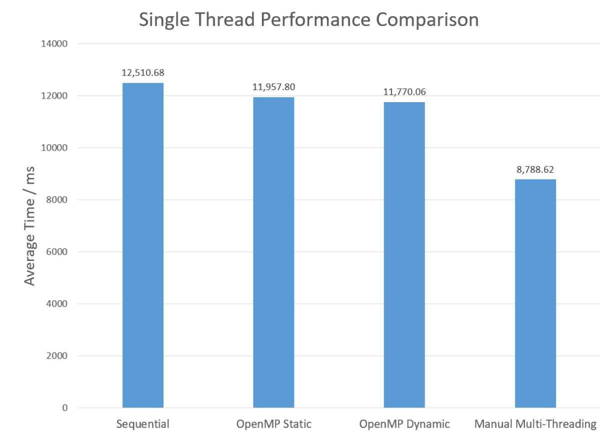

Fig. 4. A graph showing the average time it took each algorithm to generate a 256x256 image at 16 samples per pixel whilst limited to a single thread.

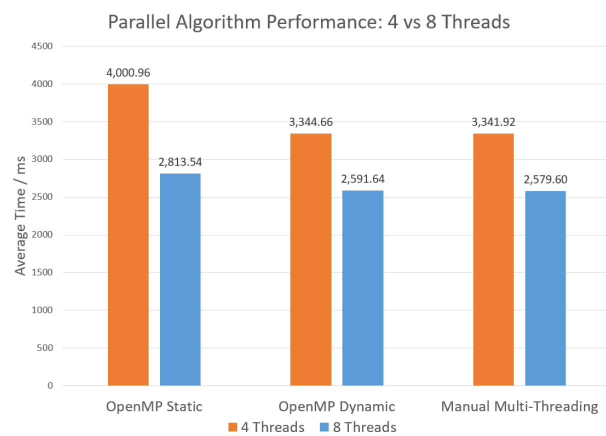

Fig. 5. A graph showing the average time it took each algorithm to generate a 256x256 image at 16 samples per pixel on both 4 and 8 threads.

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 optimisations 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

```
// Required for manual threading to allow forLoopAlgorithm method to update pixel vector
vector<vec> pixels;

// Nested for loop method, to be manually multi-threaded
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)
{
    for (size_t y = threads; y < dimension; y += num_threads)
    {
        for (size_t x = 0; x < dimension; ++x)
        {
            for (size_t sy = 0, i = (dimension - y - 1) * dimension + x; sy < 2; ++sy)
            {
                for (size_t sx = 0; sx < 2; ++sx)
                {
                    r = vec();
                    for (size_t s = 0; s < samples; ++s)
                    {
                        double r1 = 2 * get_random_number(), dx = r1 < 1 ? sqrt(r1) - 1 : 1 - sqrt(2 - r1);
                        double r2 = 2 * get_random_number(), dy = r2 < 1 ? sqrt(r2) - 1 : 1 - sqrt(2 - r2);
                        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;
                        r = r + radiance(spheres, ray(camera.origin + direction * 140, direction.normal()), 0) * (1.0 / samples);
                    }
                    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;
                }
            }
        }
    }
}

int main(int argc, char **argv)
{
    random_device rd;
    default_random_engine generator(rd());
    uniform_real_distribution<double> distribution;
    auto get_random_number = bind(distribution, generator);

    // *** These parameters can be manipulated in the algorithm to modify work undertaken ***
    constexpr size_t dimension = 256;
    constexpr size_t samples = 16; // Algorithm performs 4 * samples per pixel.
    vector<sphere> spheres
    {
        sphere(1e5, vec(1e5 + 1, 40.8, 81.6), vec(), vec(0.75, 0.25, 0.25), reflection_type::DIFFUSE),
        sphere(1e5, vec(-1e5 + 99, 40.8, 81.6), vec(), vec(0.25, 0.25, 0.75), reflection_type::DIFFUSE),
        sphere(1e5, vec(50, 40.8, 1e5), vec(), vec(0.75, 0.75, 0.75), reflection_type::DIFFUSE),
        sphere(1e5, vec(50, 40.8, -1e5 + 170), vec(), vec(), reflection_type::DIFFUSE),
        sphere(1e5, vec(50, 1e5, 81.6), vec(), vec(0.75, 0.75, 0.75), reflection_type::DIFFUSE),
        sphere(1e5, vec(50, -1e5 + 81.6, 81.6), vec(), vec(0.75, 0.75, 0.75), reflection_type::DIFFUSE),
        sphere(16.5, vec(27, 16.5, 47), vec(), vec(1, 1, 1) * 0.999, reflection_type::SPECULAR),
        sphere(16.5, vec(73, 16.5, 78), vec(), vec(1, 1, 1) * 0.999, reflection_type::REFRACTIVE),
        sphere(600, vec(50, 681.6 - 0.27, 81.6), vec(12, 12, 12), vec(), reflection_type::DIFFUSE)
    };
    // ************************************************************************************

    // Create results file
    ofstream results("data.csv", ofstream::out);

    // Output headers to results file
    results << "Test, Image Dimensions, Samples Per Pixel, Time, " << endl;

    // Run test iterations
    for (unsigned int j = 0; j < 100; ++j)
    {
```

```cpp
        ray camera(vec(50, 52, 295.6), vec(0, -0.042612, -1).normal());
        vec cx = vec(0.5135);
        vec cy = (cx.cross(camera.direction)).normal() * 0.5135;
        vec r;

        // Required for manual threading
        pixels.resize(dimension * dimension);

        // Required for OpenMP
        //vector<vec> pixels(dimension * dimension);
        //int y;

        // * TIME FROM HERE... *
        auto start = system_clock::now();

        // *** MANUAL MULTITHREADING ***
        vector<thread> threads;

        for (unsigned int t = 0; t < 4; ++t)
        {
            threads.push_back(thread(forLoopAlgorithm, t, dimension, 4, samples, get_random_number, r, cx, cy, spheres, camera));
        }
        for (auto &t : threads)
            t.join();

//      // *** OPENMP *** (change scheduling to "dynamic" or "static")
//#pragma omp parallel for num_threads(4) private(y, r) schedule(dynamic)
//      for (y = 0; y < dimension; ++y)
//      {
//          for (size_t x = 0; x < dimension; ++x)
//          {
//              for (size_t sy = 0, i = (dimension - y - 1) * dimension + x; sy < 2; ++sy)
//              {
//                  for (size_t sx = 0; sx < 2; ++sx)
//                  {
//                      r = vec();
//                      for (size_t s = 0; s < samples; ++s)
//                      {
//                          double r1 = 2 * get_random_number(), dx = r1 < 1 ? sqrt(r1) - 1 : 1 - sqrt(2 - r1);
//                          double r2 = 2 * get_random_number(), dy = r2 < 1 ? sqrt(r2) - 1 : 1 - sqrt(2 - r2);
//                          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;
//                          r = r + radiance(spheres, ray(camera.origin + direction * 140, direction.normal()), 0) * (1.0 / samples);
//                      }
//                      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;
//                  }
//              }
//          }
//      }

        // * ...TO HERE *
        auto end = system_clock::now();
        auto total = duration_cast<milliseconds>(end - start).count();

        // Output test no., variables and total time to results file
        results << j + 1 << ", " << dimension << ", " << samples * 4 << ", " << total << endl;

        // Output test information to consolde outside of the timings to not slow algorithm
        cout << "Test " << j + 1 << " complete. Time = " << total << "." << num_threads << endl;
        array2bmp("img.bmp", pixels, dimension, dimension);

        // Required for manual threading
        pixels.clear();
    }

    return 0;
}
```

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