SET10108 Concurrent and Parallel Systems Report for Coursework Part 1

Beej Persson, 40183743@live.napier.ac.uk
School of Computing, Edinburgh Napier University, Edinburgh

For part 1 of the coursework required for the SET10108 Concurrent and Parallel Systems module at Edinburgh Napier University a ray tracing algorithm'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, ray tracer, OPENMP, C++11, performance, speedup.**

I. Introduction and Background

THE aim of this report is to evaluate the performance of a ray tracing algorithm and attempt to improve this performance using parallel techniques. The ray tracer initially processes sequentially on a single core of the CPU, but by investigating different parallel techniques the algorithm was changed to run on multiple threads in an attempt to increase the performance.

A. Ray Tracer

Ray tracing is a technique used to render an image where "a ray is cast through every pixel of the image, tracing the light coming from that direction" [1]. The path is traced from an imaginary eye through each pixel on a virtual image plane and the colour of the object visible through it is calculated. It is typically capable of producing visually realistic images of high quality but at a greater computational cost compared to typical rendering techniques. Therefore it tends to be used when an image can be generated slowly ahead of time, but isn't so well suited to the real-time rendering requirements of video games.

II. Initial Analysis

The provided algorithm generates the image by iterating through each pixel using nested for loops. The ray tracer can also sample each pixel multiple times to produce a more accurate and detailed image but at the cost of processing time. Upon running the program a few times and changing the dimensions of the image produced and the number of samples per pixel an idea of its base-line performance was gathered. By its nature of currently operating sequentially, increasing either the dimensions of the image produced or the number of samples per pixel increases the time it takes to produce the image with an almost perfect positive correlation. That is to say: doubling the size of the image produced or doubling the samples per pixel doubles the time taken.

```
1 for (size_t y = 0; y < dimension; ++y)
2 {
3   for (size_t x = 0; x < dimension; ++x)
4   {
5     for (size_t sy = 0, i = (dimension − y − 1) ∗ dimension + x; sy <↩
        2; ++sy)
6     {
7       for (size_t sx = 0; sx < 2; ++sx)
8       {
9         for (size_t s = 0; s < samples; ++s)
10        { ... }
11      }
12    }
13  }
14 }
```

Listing 1. The many nested for loops of the ray tracer algorithm with the operations within the loops removed for clarity.

For smaller images at a low number of samples per pixel this results in reasonable times to produce an image, but when producing large images at a high number of samples per pixel the time to produce them was bottlenecked by only running sequentially on a single thread. Most of the time running the program is spent iterating through the nested for loops (seen in Listing 1) and so a clear solution to improving the performance of the algorithm is to run these for loops concurrently on multiple threads.

III. Methodology

A systematic approach was undertaken to evaluate the performance of the algorithm and attempt to measure any improvements in performance gained by parallelising the algorithm.

The first step was to run a series of tests on the provided sequential algorithm to provide a base-line that the performance of the different parallel techniques could be compared to. These tests were all done on the same hardware, the relevant specifications of which are shown in table I. The details of the tests are shown in the testing subsection below.

A. Parallelisation Techniques

After these benchmarks for the sequential algorithm were recorded, a few different parallelising techniques were applied to the algorithm and some preliminary tests were run. The intention here was to ensure that the techniques

TABLE I
HARDWARE SPECIFICATIONS

Processor	i7-4790K 4 Core HT @ 4.00GHz
RAM	16.0GB
OS	Windows 10 Pro 64-bit

had been implemented correctly and to gain an idea of their relative performance. The techniques used were manual multi-threading and OpenMP with both static and dynamic scheduling.

1) Manual Multi-Threading

To parallelise the algorithm using manual multi-threading the set of for loops seen in Listing 1 were added to a single method which could then be run on multiple threads, as shown in Listing 2.

```
1 for (unsigned int t = 0; t < num_threads; ++t)
2 {
3    threads.push_back(thread(forLoopAlgorithm, ..., /*variables*/, ...));
4 }
5 for (auto &t : threads)
6    t.join();
```

Listing 2. The for loop used to run the forLoopAlgorithm method accross the required number of threads. The variables that are passed to the method are removed for clarity.

2) 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.

```
1 #pragma omp parallel for num_threads(num_threads) private(y, r) ↩
        schedule(dynamic)
2    for (y = 0; y < dimension; ++y)
3    {
4       .../*nested for loops*/
5    }
```

Listing 3. 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.

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 1 below:

$$E = \frac{S}{p} = \frac{(\frac{T_{serial}}{T_{parallel}})}{p} \tag{1}$$

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 II and III 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 IV).

Table II and its accompanying graph (Figure 1) 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.

TABLE II
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

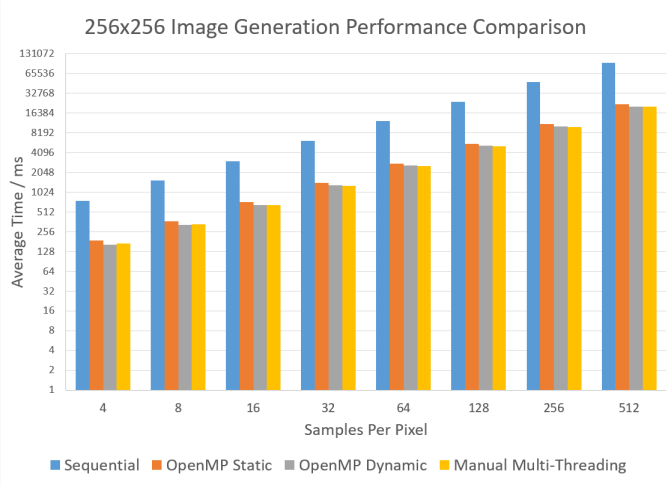

Fig. 1. A graph showing the average time it took each algorithm to generate a 256x256 image at different numbers of samples per pixel.

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.

The results of the next tests are shown in Table III. 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 2 below.

TABLE III
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

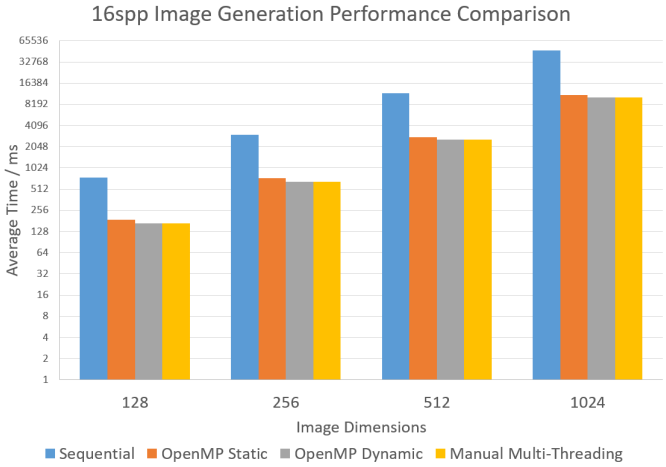

Fig. 2. A graph showing the average time it took each algorithm to generate images of varying dimensions at 16 samples per pixel.

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 IV 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 IV
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 1, the efficiency of these algorithms compared against the sequential algorithm was calculated. The below table (Table V) shows these efficiencies for the first set of test results.

TABLE V
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 3 and 4) which show some of the results from these tests.

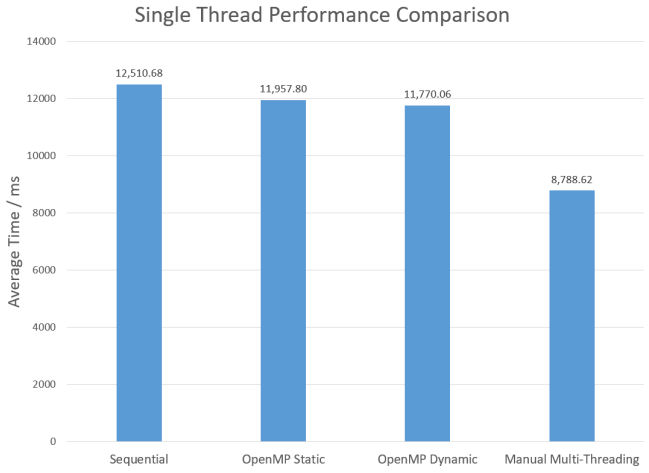

Fig. 3. A graph showing the average time it took each algorithm to generate a 256x256 image at 16 samples per pixel whilst limited to a single thread.

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.

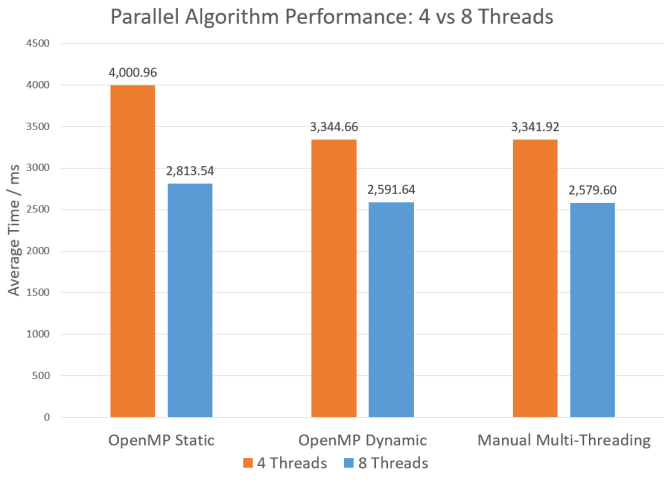

Fig. 4. A graph showing the average time it took each algorithm to generate a 256x256 image at 16 samples per pixel on both 4 and 8 threads.

Further to this, Figure 4 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

Level of discussion and appropriateness of the conclusions drawn based on the results gathered. [10]

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 1 and 2, 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 3, 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 4, 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] T. Nikodym, "Ray tracing algorithm for interactive applications," 2010. [Online]. Available: https://dip.felk.cvut.cz/browse/pdfcache/nikodtom_2010bach.pdf

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)
  {
    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;
```

```
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 consolde 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 4. The source code for the raytracer.cpp including all the added parallelisation techniques.