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.

```
for (size_t 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)
      {
        for (size_t s = 0; s < samples; ++s)
        { ... }
      }
    }
  }
}
```

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.

```
for (unsigned int t = 0; t < num_threads; ++t)
{
    threads.push_back(thread(forLoopAlgorithm, ..., /*variables*/, ...));
}
for (auto &t : threads)
    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.

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

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.

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 and 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. Further to this, a few additional 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:

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

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

IV. RESULTS AND DISCUSSION

The results from the testing done on the algorithms can be seen summarised in tables II 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.

A graph 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 is shown below in Figure 1.

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 speed up over the

TABLE II
256x256 Image Generation Performance Comparison

Image Dimensions 256x256				
Algorithm	Sequential	OpenMP Static	OpenMP Dynamic	Manual Multi-Threading
Samples Per Pixel	Average Time / ms			
4	1.0	1.0	1.7	0.989
8	1.0	1.0	2.5	0.920
16	1.0	1.0	3.5	0.919
32	1.0	1.0	3.4	0.901
64	1.0	1.0	25.5	0.962
128	1.0	1.0	82.9	0.949
256	1.0	1.0	183.6	0.968
512	1.0	1.0	335.1	0.968

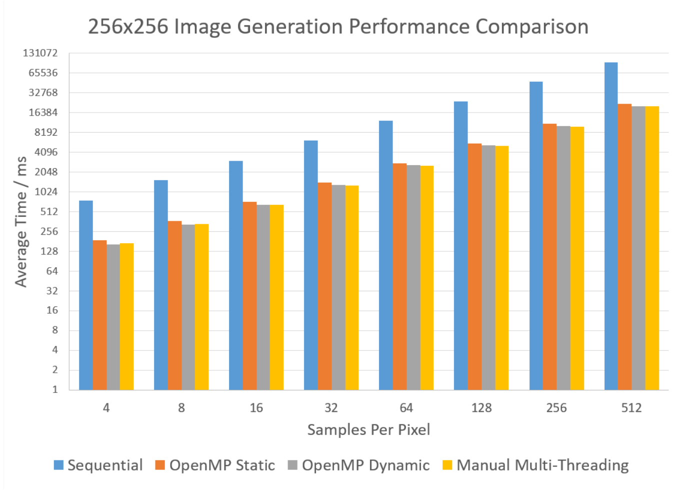

Fig. 1. A graph showing the average time it took each algorithm to generate a 256x256 image at different numbers of samples per pixel.

sequential algorithm, with the manual multi-threading being the fastest.

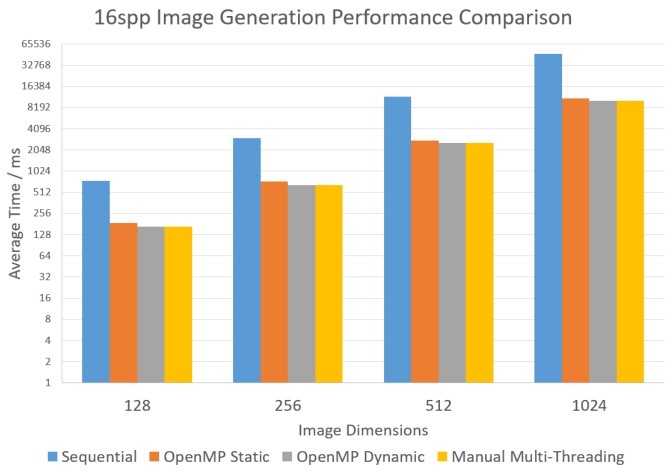

Fig. 2. A graph showing the average time it took each algorithm to generate images of varying dimensions at 16 samples per pixel.

Suitable performance analysis and testing documentation for the problem, including quality of presentation of the results. [10]

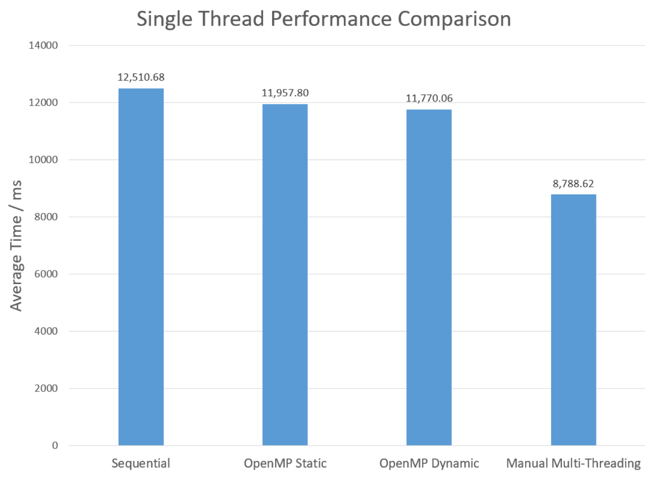

Fig. 3. A graph showing the average time it took each algorithm to generate a 256x256 image at 16 samples per pixel whilst limited to a single thread.

V. Conclusion

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

Appendix A
Proof of fucking something

Appendix one text goes fucking here.

```
#include <iostream>

int main(){
  std::cout << "Hello World!" << std::endl;
  return 0;
}
```

Listing 4. A fucking code listing.

Appendix B

Fucking appendix two text goes here.

References

[1] T. Nikodym, “Ray tracing algorithm for interactive applications,” 2010. [Online]. Available: https://dip.felk.cvut.cz/browse/pdfcache/nikodtom_2010bach.pdf