**Parallelization of a Ray Tracer**

Ryan O’Flaherty – 40168766

**Abstract**

As part of a university assignment, we were given a task to parallelize a given sequential ray tracing algorithm, analysing the results of different concurrency and parallelism techniques. This report discusses the methods and results of that investigation.

# **Introduction and Background**

**What is a Ray Tracer?**

Ray tracing is a technique used in the rendering of 3D computer graphics(1). As rays of light reflect and ‘bounce’ around the scene, their paths are traced to determine what colour each light ray should be(2).

# **Initial Analysis**

The initial program is very slow, and contains a series of nested for loops. These are proving to be a big bottleneck, and a large part of why the program takes such a long time to run. These nested loops are therefore going to be the targeted area for parallelization.

An average of the time taken for the initial code to run with 9 objects in the scene at 256 samples per pixel with 400x400 and 1024x1024 aspect ratios can be seen below.

|  |  |  |
| --- | --- | --- |
|  | Home | Games Lab |
| 400x400 | 328751.8 | 152698.4 |
| 1024x1024 | 2159342 | 999672 |

We can see here that the games lab PCs are far superior to the one at home.

Regardless, our initial sequential code is very slow and needs to be improved.

# **Methodology**

**Techniques:**

**Multi-Threading**

Multi-threading is when an operating system executes more than a single part of a program simultaneously, by assigning what are known as ‘threads’ to each particular section of the code(3). A thread is the smallest unit of processing in an operating system(4).

In this example, we are going to use all of the available threads on each machine, using the following line of code:

auto threadCount = thread::hardware\_concurrency();

**Mutex**

Mutex stands for mutual exclusion. It is used in conjunction with multi-threading to protect

sections of code to ensure that only one thread can make alterations to it at any one time.

So, if we have a variable that is going to be changed by each thread, we must ensure that

each thread is accessing it separately.

To use it, we’re simply putting lock and unlock statements either side of the protected

variable’s calculation:

muteX.lock();

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;

muteX.unlock();

**OpenMP**

OpenMP is an API that supports shared-memory programming in parallel, compatible with multi-core CPUs(workbook). In particular, we are using the Parallel For feature of OpenMP.

**Parallel For**

With parallel for, OpenMP automatically creates threads for us and uses them in the

execution of a for loop(workbook), with each iteration being assigned a thread.

In our implementation, the code for this is the following single line at the very top of

our nested loops:

#pragma omp parallel for num\_threads(threadCount) private (r)

**Variables considered in Testing:**

**Image Resolution**

Resultions were tested at 400x400px and 1024x1024px.

**Sample Rate**

Sample rates of 4, 16, 64 and 256 samples per pixel were tested.

**Number of Objects**

The default 9 spheres were tested along with sets of 12 and 15.

The same tests were performed both at home and in the games lab at university.

|  |  |
| --- | --- |
| **Home Spec** | **Games Lab Spec** |
| CPU – AMD FX™-8350 @ 4.00GHz  4 hardware and 4 logical => 8 cores | CPU – Intel i7-4790K @ 4.00GHz  4 hardware and 4 logical => 8 cores |
| GPU – Nvidia GTX 750 | GPU – Nvidia GeForce GTX 980 |
| OS – Windows 7 Pro N 64-bit | OS – Windows 10 Pro N 64-bit |

# **Results and Discussion**

Seen here in the lowest of the tested conditions, increasing the number of spheres

has a more noticeable effect on the sequential code than both threads and parallel

for, which both handle it well.

In all 3 cases, going from 9 spheres to 12 actually speeds up the process, however

the jump to 15 pushes the times higher than where they started with 9 spheres. The

parallel methods keep this to increase to a decent minimum, but the sequential

implementation drastically increases in the time taken.

As the rate of sampling increases, so does the time taken to complete the task. Completing the task entirely in series naturally takes a long time, exceedingly so when using higher sampling rates.

In the experiments, OpenMP’s parallel for and manual multi-threading the use of a mutex are almost identical, as the orange ‘Threads w/Mutex’ line in the graph is almost entirely covered by the line representing parallel for. This is a theme that sticks around despite the variable changes.

Increasing the dimensions of the image unsurprisingly increases the gap between the parallel implementations and the sequential one, however manual threads and parallel for remain neck and neck.

Having already scaled the resolution up to 1024x1024, the next step is to increase the amount of spheres. However, this had the same effect as it did in the earlier stages when working with a 400x400 aspect ratio.

Using results from the games lab as they are superior, let’s look at the efficiency and speedup comparisons at each sample rate using 9 spheres, with a 1024x1024 image resolution:

|  |  |  |
| --- | --- | --- |
| *Speed up* | **Parallel For** | **Threads** |
| **4 Samples** | 3.260799585 | 3.438716566 |
| **16 Samples** | 3.392064459 | 3.44719531 |
| **64 Samples** | 3.323747681 | 3.457526068 |
| **256 Samples** | 3.356931445 | 3.452154533 |

Speedup is calculated by simply dividing the original sequential result by the result of the technique in question. This can then be divided by the number of physical cores your computer has - which in both of our cases was 4 – to determine the efficiency.

|  |  |  |
| --- | --- | --- |
| *Efficiency* | **Parallel For** | **Threads** |
| **4 Samples** | 0.815199896 | 0.859679141 |
| **16 Samples** | 0.848016115 | 0.861798827 |
| **64 Samples** | 0.83093692 | 0.864381517 |
| **256 Samples** | 0.839232861 | 0.863038633 |

Something else we could look at is the difference between the results obtained at home against the results from experiments conducted in the games lab.

This time, we will again look at using 9 spheres, but this time with a 400x400 image resolution.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Home | | Games Lab | |
| **Threads** | **Parallel For** | **Threads** | **Parallel For** |
| **4 Samples** | 1567.8 | 1680 | 561.9 | 618.2 |
| **16 Samples** | 8175.8 | 8607.7 | 2201.8 | 2277 |
| **64 Samples** | 32501.5 | 34919.7 | 8953.4 | 9134.2 |
| **256 Samples** | 125720.2 | 135913 | 35140.3 | 35962.4 |

It’s pretty clear to see why the games lab results have been used predominantly throughout this report – they’re much better. Looking at the 256 sample rate example, there’s almost 100,000 milliseconds of a difference. That’s ridiculous!

Disclaimer:

The home PC has been performing very poorly for quite some time now, so these results are not indicative of the PC’s spec.

# **Conclusion**

It is hard to draw a conclusion from the results of this investigation, as both of the implemented parallelization techniques produced almost identical results. However, what is clear to see, is that both implementations have vastly improved the speed and efficiency of the program.

In both cases of parallelization, we get efficiency results of 0.8 and above. This is very good as the highest *possible* efficiency rating is 1.

**References**

*[1] Explanation of a ray tracer from -* [*https://www.cs.unc.edu/~rademach/xroads-RT/RTarticle.html*](https://www.cs.unc.edu/~rademach/xroads-RT/RTarticle.html)

*[2] Another explanation of ray tracers -* [*https://www.cs.unc.edu/~rademach/xroads-RT/RTarticle.html*](https://www.cs.unc.edu/~rademach/xroads-RT/RTarticle.html)

*[3] Multi-threading -*[*https://www.webopedia.com/TERM/M/multithreading.html*](https://www.webopedia.com/TERM/M/multithreading.html)

*[4] What is a thread -* [*https://www.techopedia.com/definition/27857/thread-operating-systems*](https://www.techopedia.com/definition/27857/thread-operating-systems)