Optimising a ray tracer algorithm using different Parallelization techniques

Valentina Scarfi – 40182166

Concurrent and Parallel Systems (SET10108)

Abstract

This document will analyse the baseline performance of a ray tracer algorithm to identify possible parallelization techniques to be applied. The chosen techniques, which are C++11 threads and OpenMP with different scheduling, have consistently improved algorithm performance spreading the workload across the available threads. Different conditions have been tested, such as different machines, image dimensions, objects and samples per pixel. It is concluded that the difference between the techniques performances is not enough relevant to prefer an implementation over another.

1. Introduction and Background

The Ray Tracer Algorithm A ray tracer algorithm computes the “visibility between points” [1], generating an image based on the source light path and how this interacts with surfaces of the objects composing the scene. As the image shows (*Figure 1*) the result of the algorithm is a realistic scene reproducing lights, reflections and refractions accurately that can deviate the light and hit other surfaces, being reflected or refracted again or even absorbed.

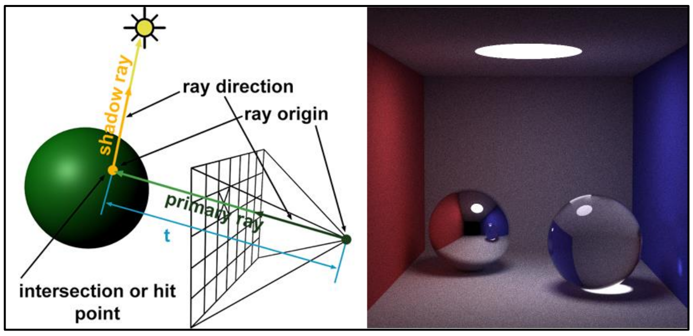


Figure 1: *Image showing principle of ray tracer algorithm [1] and the resulting image*

Project Objectives The purpose of this project is to optimize a given ray tracer algorithm using different parallelization techniques, in order to improve performances.

The results of each technique, applied to different setting and conditions, will be compared and evaluated in terms of speed up and efficiency. Correlation coefficient and standard deviations of algorithm timings will be also considered.

Techniques Two parallelization techniques have been implemented: C++11 threads and OpenMP library. They both allow to parallelize a serial algorithm in different ways. These techniques will be described and discussed later in this document

2. Initial Analysis

Hardware The algorithm has been tested on two different machines at every condition of each technique: PC at University (Games Lab) and Home PC.

Games Lab

- CPU – Intel i7-4790K @ 4.00GHz 4 hardware and 4 logical = 8 cores
- GPU – Nvidia GeForce GTX 980
- OS – Windows 10 Pro N 64 bit

Home

- CPU – Intel i5 760 @ 2.80GHz 4 hardware cores
- GPU – Nvidia GeForce GTX 960
- OS – Windows 10 Pro 64 bit

First Analysis A deep initial analysis has been performed in order to evaluate the starting point for the algorithm optimization. Several tests have been run to gather enough data. The runs have been limited to a number of 10 after evaluating the Standard Deviation of more runs.

The algorithm has been tested and compared using different conditions:

- Sample per pixel – 4, 16, 64, 256
- Image dimension – 400x400, 1024x1024
- Sphere number in the scene – 9, 14, 20
- Machine – Games Lab and Home Pcs

More image dimension and sphere numbers have been tested, only for the initial analysis, to calculate a correlation coefficient.

The main loop has been monitored and time registered accordingly. Console output statements have been kept outside the time measuring, as they badly affected the results.

The strongest correlation is between samples per pixel and the time taken (*Table 3*), as the coefficient is 1 in every condition – machine, image dimension and sphere number -. Image dimension and sphere number in relation with time have a strong correlation very close to 1 (*Table 1, Table 2*) and they have only been tested on one machine.

Sample per pixel	Time taken(s) Games Lab	Time Taken(s) Home
4	1,904	4,070
16	7,534	15,215
64	30,171	60,190
256	121,398	241,436
Correlation Coefficient	1.000	1.000

Table 3: table showing sample per pixel – time taken correlation

Image dimension	Time (s)
128	0,2072
400	1,910
700	6,0108
1024	12,662
Correlation Coefficient	0,975

Table 1: table showing image dimension – time taken correlation based on a 400x400 image with 4 samples per pixel (Games Lab only)

Spheres	9	14	20	25	30
Time (s)	1,904	2,137	2,398	2,847	3,065
Correlation	0,992				

Table 2: table showing sphere number – time taken correlation, based on a 400x400 image with 4 samples per pixel (Games Lab only)

The Standard Deviation has been calculated to establish the variation of the values throughout the different runs performed. Even with few exceptions, the SD of the time taken increases with the increase of the sample per pixel value (*Table 4*).

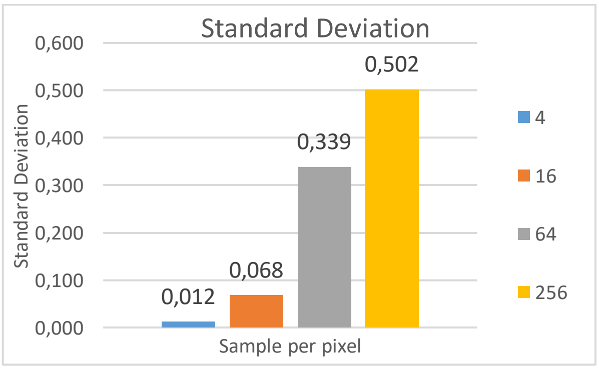
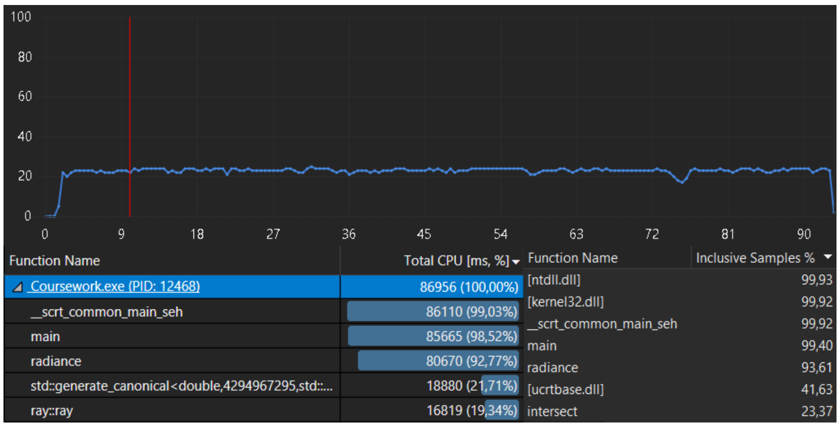


Table 4: graph showing Standard Deviation tendency, based on a 400x400 image with 4 sample per pixels (Games Lab).

Parallelization Analysis From the initial analysis it is clear that any of the conditions analysed will heavily influence the algorithm performances. The most interesting data comes from the Performance Profiler. This shows how the CPU usage for the process doesn't go higher than 25% on average (at home), meaning that the CPU is using only 1 thread out of 4 at 100%. As it can be seen in *Figure 2*, CPU spent 98,52% of the time executing functions within the *main* method, where the loop calculating *radiance* and *intersect* is located. This is the core of the algorithm and this is where the parallelization approach will focus on.



Function Name	Total CPU [ms, %]
Coursework.exe (PID: 12468)	86956 (100,00%)
__scrt_common_main_seh	86110 (99,03%)
main	85665 (98,52%)
radiance	80670 (92,77%)
std::generate_canonical<double,4294967295,std::...	18880 (21,71%)
ray::ray	16819 (19,34%)

Function Name	Inclusive Samples %
[ntdll.dll]	99,93
[kernel32.dll]	99,92
__scrt_common_main_seh	99,92
main	99,40
radiance	93,61
[ucrtbase.dll]	41,63
intersect	23,37

Figure 2: Data from Visual Studio 2017 Performance Profiler

3. Methodology

Multi-threading The first technique that has been applied is C++11 thread feature that allows to retrieve the available threads, which are limited by the machine hardware. Using mutexes and lock guards combined with threads is necessary to prevent that more than one thread operates with the resource at the same time.

Threads have been created assigning a start and an end for each, which is the dimension of the image divided by the number of available threads (*Algorithm 1*). This has been implemented to guarantee an even workload to each thread.

```
//Retrieving available threads
auto num_threads = thread::hardware_concurrency();
//Splitting dimension of the image by available threads
auto range = dimension / num_threads;
```

Algorithm 1: algorithm showing threads retrieval and workload splitting

Two approaches have been tried when creating threads: using all available threads to perform the required calculations and using all the available threads minus the last one which has been replaced by the main thread.

No difference in timings has been found, thus the second approach has been used for next tests.

OpenMP The second technique that has been implemented to improve algorithm performance is the OpenMP API. This implementation has been chosen especially for its feature that allows to manipulate the scheduling of work amongst the threads. This introduces a new and different concept, as the schedule can be static (iterations evenly spread across threads) or dynamic, where “OpenMP assigns one iteration to each thread. When the thread finishes, it will be assigned the next iteration that hasn’t been executed yet” [2].

OpenMP is also simple and quick to implement, having a performance improvement with just a line of code.

```
    //Using OpenMP declaring r as private and scheduling the work statically
    #pragma omp parallel for private(r) schedule(static)
```

Algorithm 2: OpenMP usage

As *Algorithm 2* shows, *vec r* has been declared as private. This is because, by default, all variables within the loop are shared. In this case, having *vec r* as shared, causes an incorrect output image: a shared *vec r* can be overwritten by other threads that are using the variable.

4. Results and Discussion

Tests Several tests have been run using each technique. As *Table 5* shows, the improvement in terms of average time of execution is significant.

Speedup and hardware efficiency, based on the number of spheres in the image, have been calculated to evaluate the effective result of each implemented feature (*Table 6*).

Games Lab Results From the table presented (*Table 5*, *Table 6*), a good speedup and efficiency has been achieved with all the implementations. Efficiency is slightly lower than 1, which is a good result as 1 would be a perfect efficiency. Speedup achieved is also good, since a value equal to 4 (number of cores) would be a linear speedup, which is rare to obtain. The best result has been achieved by OpenMP with a dynamic scheduling. An effective speedup of 72,51% has been reached by this implementation, followed by manual threads with a speedup of 71,94% and OpenMP with static scheduling with 71,54%.

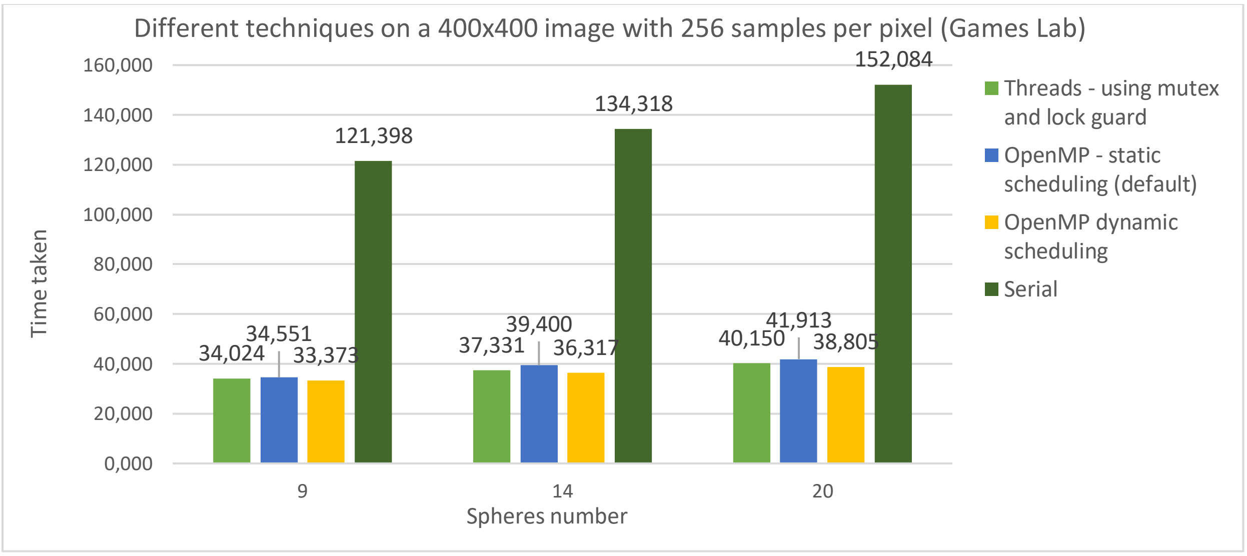


Table 5: graph showing difference of timings between original serial algorithm and improved algorithm using different techniques (Games Lab)

Speedup of each technique					**Efficiency of each technique (p = 4)**		
	Spheres				**Spheres**		
Technique		**9**	**14**	**20**	**9**	**14**	**20**
	Threads	3,568	3,598	3,788	0,892	0,900	0,947
	OMP Static	3,514	3,409	3,629	0,878	0,852	0,907
	OMP Dynamic	3,638	3,698	3,919	0,909	0,925	0,980

Table 6: speedup and hardware efficiency for each technique, considering different numbers of spheres. A 400x400 image with 256 samples per pixel has been processed in the Games Lab to obtain these results.

Home Results The results for speedup and efficiency obtained from home machine (*Table 7*) are very different compared to Games Lab's ones. This is because of the very different hardware, as one is a 1st generation and the second is a 4th generation Intel processor. The speedup and efficiency obtained at home are positive, but not as good as the ones obtained in the Games Lab. Also, manual threads results are better than OpenMP with a dynamic schedule. The effective speedup achieved at home by the best technique, which is C++11 multi-threading, is of a 60,64%. This can still be considered a good result, as the gain in time, especially considering the samples per pixel (*Table 8*), is positive and still significant.

Speedup of each technique					Efficiency of each technique (p = 4)		
		Spheres			Spheres		
		9	14	20	9	14	20
Technique	Threads	2,541	2,618	2,682	0,635	0,654	0,670
	OMP Static	2,373	2,476	2,531	0,593	0,619	0,633
	OMP Dynamic	2,349	2,535	2,611	0,587	0,634	0,653

Table 7: speedup and hardware efficiency for each technique, considering different numbers of spheres. A 400x400 image with 256 samples per pixel has been processed at Home to obtain these results.

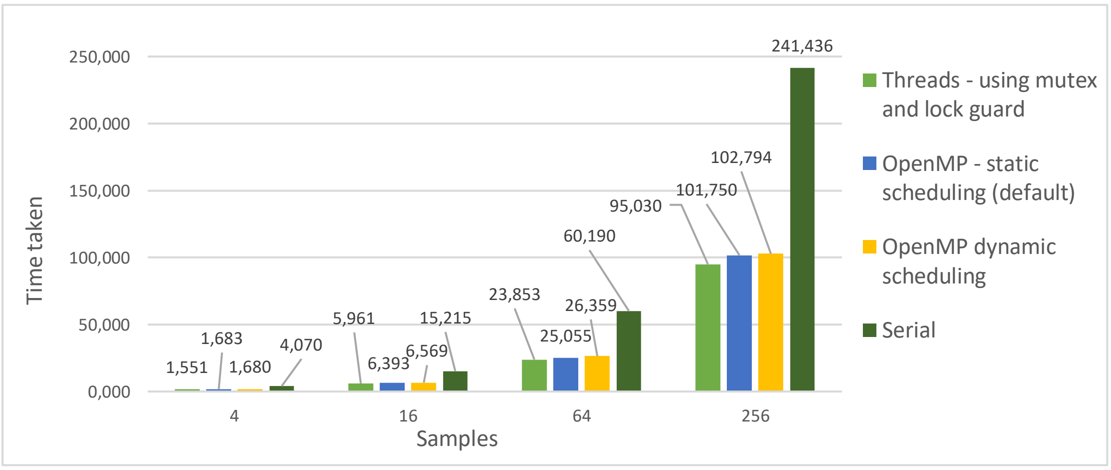


Table 8: graph showing difference of timings between original serial algorithm and improved algorithm using different techniques with different samples per pixel on an image 400x400 at Home.

The CPU usage, for all the different techniques, hits an average of 75% with peaks near 80%. From the Performance Profiler this can be clearly seen, although the CPU usage in this case seems to oscillate more than the serial algorithm usage (*Figure 3*)

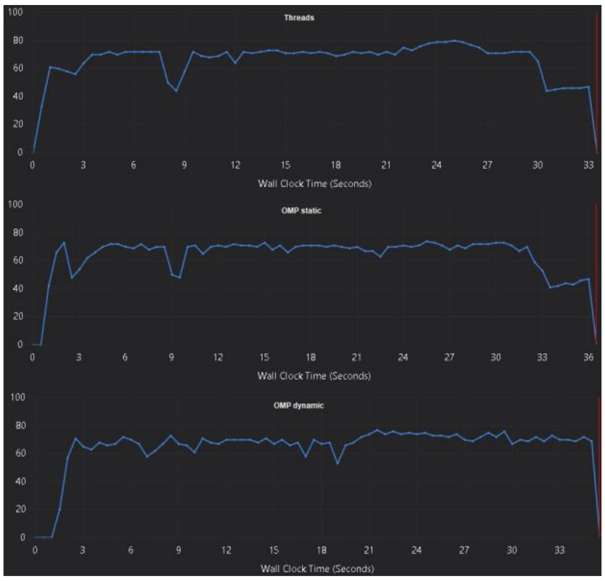


Figure 3: threads, OMP static and OMP dynamic CPU usage at Home

5. Conclusion

Threads and dynamic OpenMP As the results show, Games Lab and Home performances, with the techniques implemented, are quite different. Especially when it comes to compare threads and OpenMP with a dynamic schedule: the second is the best one in the Games Lab, while the first has better results at Home. This might happen because of the hardware differences already discussed: dynamic schedule in OpenMP could cause some overhead at Home because of the fewer cores available compared to the ones in the Games Lab.

Real improvement Analysing again *Table 5* and *Table 8* it is evident how all the techniques used have levelled up the huge difference in performance between the smallest sample per pixel and sphere number analysed, 4 and 9 respectively, and the greatest number of these considered, 256 and 20. While in the serial algorithm an image with 20 spheres was taking almost 30 seconds to be created more than the one with only 9 spheres, the parallelized algorithm has a difference of 5 to 8 seconds to perform the same calculation.

Final considerations The results obtained by the techniques implemented can be considered satisfactory. Even though there are differences in the results of each implementation, these are not enough significant to prefer a technique over the others in terms of performance obtained.

On the other hand, OpenMP is the way easier to implement than threads, as the workload is split and managed directly by the API, while using manual threads implies the use of mutexes and guards to guarantee a proper functioning of the whole system.

References

[1] *An Overview of the Ray-Tracing Rendering Technique* from https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-overview

[2] University of Mary Washington. *OpenMP Scheduling* from http://cs.umw.edu/~finlayson/class/fall16/cpsc425/notes/12-scheduling.html