An Investigation into the Parallelisation of a Ray-Tracer via the use of Threads and Distribution Methods

Sam Dixon
40056761@live.napier.ac.uk
SET10108 - Concurrent and Parallel Systems
School of Computing, Edinburgh Napier University, Edinburgh

***Index Terms*—C++11, Ray-Tracer, Parallel, OpenMP,Open-MPI, Thread, Distribution, Speed Up.**

I. Introduction

THE aim of this report is to document, analyse and compare the benefits of a number of concurrency and parallelisation techniques, and their effect upon the execution time of a C++11 implementation of a genetic string matching algorithm.

A. Ray-Tracing

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

B. OpenMP

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

C. Open-MPI

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Project available at: github.com/neaop/SET10108Coursework_2

II. Methodology

A. Profiling

Prior to implementing any methods that will save time, the sequential code must first be analysed. By using the Visual Studio Performance Profiler, it is possible to evaluate the sequential code and locate the functions or methods that use up the most CPU time. Once the potentially problematic areas have been identified, a suitable parallelisation method can be implemented to reduce the impact of those areas on the execution time. It should be noted that all code presented in the report was run with all compiler optimisation turned off.

B. Data Collection

To ensure fair comparison and accurate results, each implementation was tested using the same parameters. Each solution was run for two-thousand and forty generations and the execution time was recorded. This was then repeated one-hundred times for each application and the results were then averaged. The various techniques were all tested using four threads, then again using eight threads to investigate if the use of virtual cores would have an impact upon the speed of execution. All code was benchmarked on the same PC, the specifications of which can be seen in Table I, Page 2. As well as the average execution time, speed-up and efficiency are calculated for each technique. Speed-up is defined as:

$$S = \frac{s_t}{p_t}$$

With s_t being sequential time and p_t being parallel time. Efficiency is calculated as

$$E = \frac{S}{P}$$

TABLE I
PC SPECIFICATIONS

CPU	i7-4790k 4 Core HT @ 4.00 ghz
RAM	16gb Dual Channel DDR3
GPU	Nvidia GeForce GTX 980
OS	Windows 7 64 Bit

S being speed-up from the previous formula and P is the number of physical cores on the CPU.

C. OpenMP

OpenMP (Open Multi-Processing) is an open source API that allows for the implementation of shared memory multiprocessing with minimal developmental effort. OpenMP makes use of the C++ #pragma directive and the pre-processor to allow developers to flag sectors of code to be parallelised. A number of different parallelisation and scheduling options can be implemented to alter the way OpenMP effects code. The OpenMP options for parallelising the code include: parallel for, static scheduling, dynamic scheduling and guided scheduling.

D. Parallel For

The parallel for flag instructs OpenMP to split a sequential for loops workload over multiple threads. There are a number of caveats a developer must be aware of to implement parallel for. Firstly, the number of loop iterations must be determined at compilation time. Secondly, developers must also take care to protect any data shared across multiple threads, this can be implemented easily using OpenMPs shared flag. The parallel for flag will be used in conjunction with scheduling flags listed below.

E. Static Scheduling

Static Scheduling is where the workload of a for loop is split into predetermined chunks which are then assigned to threads in a round-robin style. Typically, the workload of the loop is distributed as: $NumberOfIterations/NumberOfThreads$, However, the developer is free to change this configuration should they see fit.

F. Dynamic Scheduling

Dynamic scheduling is, once again, where a loop is split into chunks of predetermined size: one by default (although the developer can opt for their own chunk size) which are then distributed to any available thread. However, the difference from static scheduling is that when a thread completes a chunk of work, it will request a new chunk work to process i.e. First in, first out.

III. RESULTS

A. Sequential

The only changes made to the provided sequential code were as follows: a for loop to allow multiple executions and a data collection function allowing the time stamps to be collected in a .csv file. It should be noted that there are several non-parallelisation techniques that could be implemented to speed up the application. These are, namely, removing console output and allowing automated compiler optimization.

TABLE II
TABLE DEPICTING EACH TECHNIQUE, THE NUMBER OF THREADS AND THE AVERAGE EXECUTION TIME, SPEED-UP AND EFFICIENCY

Method	**Threads**	**Time(ms)**	**Speed-up**	**Efficency**
Dynamic	8	27077.73	1.427588	0.356897
Guided	8	27250.29	1.418548	0.354637
Static Interleaved	8	27545.24	1.403358	0.350840
Static	8	27675.49	1.396754	0.349188
Guided	4	27963.20	1.382383	0.345596
Dynamic	4	28016.61	1.379747	0.344937
Static Interleaved	4	28537.99	1.354540	0.338635
Static	4	29107.61	1.328032	0.332008
Threads	4	29493.40	1.310661	0.327665
Threads	8	29574.82	1.307052	0.326763
Sequential	1	38655.84	NA	NA

1) Profiling

The profiling of the sequential code (Fig **??**, Page **??**) revealed that the majority of the CPUs time was spent within the update_epoch function. Within update_epoch CPU time was spent mainly in the epoch function this function became the prime candidate for parallelisation. All the techniques documented hereafter were applied to epoch function, allowing a fair comparison of all methods used.
It should be noted that a significant amount of CPU time was spent on the generation of random numbers and a significant speed up could be gained, should the generation be altered. This was not investigated any further during the project, however, as it would consist of a considerable amount of code refactoring, which this project aimed to avoid.

2) Execution Time

As stated in the methodology, the sequential solution was run one-hundred times, and the execution time recorded. Once all executions were complete these results were averaged to give us a runtime of 38655.84 milliseconds. See Table II, Page 2.
It was crucial to benchmark the sequential algorithm, as it will allow for the calculation of speed-up and efficiency of the parallelisation techniques being implemented.

B. Parallel For

As determined by the profiling of the sequential code, the prime function for parallelisation is epoch. This was achieved by simply inserting the line: **#pragma omp parallel for num_threads(MAX_THREADS) shared(babies) schedule(schedule)** above the for loop present in the function. As can be seen, there are a number of variable parameters in the parallel for line that a developer can use to augment how OpenMP affects the code:
num_threads() allows the developer to define how many threads the for loop should be split across.
schedule() is where a developer define which schedualing method should be used. By default OpenMP will use static.
As can be seen, it takes very little effort to parallelise a for loop using OpenMP. Once the location of the parallel for had

been determined, benchmarking of each scheduling flag could be performed.

C. Static Schedule

Static schedule is the parallel for default, for this reason it was one of the first techniques to be tested. Two configurations of Static Scheduling were tested, each of four and eight threads. The first configuration (Static in Table II.) divides the total work load of the for loop between the available threads i.e. each thread runs one chunk of work. The other configuration (Static Interleaved in Table II.) has each thread run a single iteration, then wait until they are assigned another iteration to process.

1) Findings

As seen from the results table (Table II, Page 2), offers a times speed-up of at least 1.32, regardless of the number of threads, or forced interleaving. It should be noted that the interwoven Static schedule is consistently faster then the default Static schedule. This is likely an artefact of how OpenMP unfolds and distributes a for loop amongst threads, and is a potential area for future investigation.The thread operations for the default Static schedule can be seen in fig **??**, and the interwoven in fig **??**.

D. Dynamic Schedule

The Dynamic Schedule has a larger overhead than Static scheduling, but has a greater potential for speed-up. After each thread has executed a chunk of iterations, one by default, it will retrieve a new chunk to process. In theory, dynamic scheduling should reduce the amount of time threads are asleep, as each thread can retrieve more work when it is ready, rather than having to wait for its turn.

1) Findings

The results show that Dynamic Scheduling with eight threads offered the most speed-up of all the methods used, with the four threaded version being the second best of all the four-threaded configurations. These results are as we expected and prove that the overall speed-up of the Dynamic schedule far outweigh the overhead required to initiate it. Should time allow, further research into optimal chunk size should be carried out, to see if any further improvement could be gained. Visuals of the four thread and eight thread implementation are visible in figs **??** and **??** respectively. As can be seen the eight thread version does include some sleep time due to there only being four physical cores on the CPU.

E. Guided Schedule

Guided scheduling operates in a very similar manner to Dynamic, bar the size of the work chunks. The chunks initially start quite large, but reduce in size every iteration, down to a minimum size - one by default. Typically Guided scheduling operates well on workloads that cannot be evenly dived, or that may not be finished, neither of which are the case in this scenario.

1) Findings

Interestingly, Guided Scheduling performed very well, coming in at around 170ms slower than Dynamic Scheduling with eight threads. It showed also be noted that Guided scheduling was the superior out of all the four thread configurations.

F. C++11 Threads

The traditional method of parallelising code is via the use of a threading library. Threading requires significantly more effort that utilising OpenMP, a it demands the developer to protect any shared data and extract code in to chunks that can be run on individual threads. It is not expected for the hand threaded solution to perform better than the OpenMP solutions, as it is likely that OpenMP implements its own optimisation methods.

1) Findings

As expected, the threaded solutions did offer speed up, but not quite of the same calibre as the OpenMP solutions. That is not to say that hand threaded solutions should not be implemented, only that OpenMP will offer a bigger speed-up at lower effort to the developer. A visual representation of the threads is visible in fig **??**, as can be seen, while worker threads spend their entire duration working, the main thread has a considerable amount of synchronization to carry out.

IV. Conclusion

As can be seen from the results presented in this paper, most programs can be parallelised with minimal effort from the developer, and still reduce the overall execution time of the system. Out of all the methods implemented OpenMP Parallel For with Dynamic Scheduling offered the best results in terms of speed-up and requires little effort to implement.

1) Future Work

Areas that could be investigated further in the future include; superior random number generation, parallelisation of multiple for loops, SIMD and Futures.

Bibliography

[1] A. Williams, *C++ concurrency in action: practical multithreading*. Shelter Island, NY: Manning Publ., 2012.