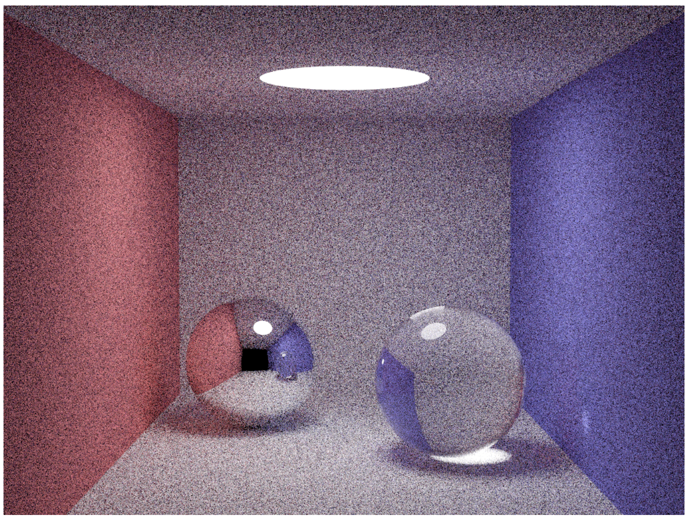
Programming and Architecture of Computing Systems

Laboratory 4

Two Building Blocks of Parallel Programming: queues and thread pools



Authors:

Juan Plo Andrés (795105)
Darío Marcos Casalé (795306)

Escuela de Ingeniería y Arquitectura
Universidad Zaragoza

1 Introduction

In this lab, we will use synchronization tools to implement two blocks of parallel programming. We will implement a thread-safe queue that allows threads to perform atomic pushes and pops, and we will implement a thread pool to decouple the creation and execution of the task, which are integrated into a renderer. We will also analyze the potential of the division of the task in many subtasks, in order to speed up the execution of a simple raytracer and an image processing pipeline.

2 Implementation details

Firstly, we introduce the details of the threadpool and the queue that supports it. Furthermore, a non-blocking implementation of a safe queue will be briefly described.

2.1 Thread-safe queue (Q1)

Since reading and adding elements to a queue are non-atomic, we need to ensure that those operations execute in mutual exclusion between threads. With the `std::mutex` tool we ensure that the push and pop are executed atomically, therefore we evade race conditions that alter the expected performance of the program. With the `std::condition_variable` we can block the threads that want to execute push and pop operations when another thread is performing an operation on the queue.

2.2 Thread Pool (Q2)

Implementing the wait in the destructor of the thread pool allows to decouple the task creation and waiting for the completion. It also serves as a way of ensuring that all tasks are executed, just in case the programmer forgets to `wait()`.

2.3 Non-blocking concurrent queue (Qβ)

We implemented the non-blocking queue proposed by Michael and Scott (1996) [1] using the atomic primitives found in the `<atomic>` standard library. This queue allows to perform thread-safe queue operations without having to block during the whole operation. A small benchmark that runs a fixed number of threads was also implemented, where each of them pulls a value from the queue and then appends it back. Since the execution is tightly coupled to the queue usage (because the queue operations are executed very frequently), the blocking queue implementation will be much slower than the non-blocking one because the atomic operations ensure correctness even without locking the whole operation.

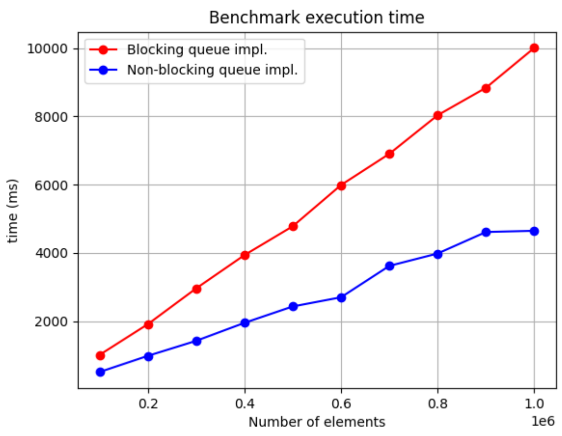


Figure 1: Benchmark metrics for both implementations

Figure 1 shows the trend on execution time for this benchmark, which clearly shows that the non-blocking queue implementation outperforms the blocking one.

3 Performance analysis (Q3)

The experiments were performed on an Intel i7-10750-H (Comet Lake architecture), with a base frequency of 2.6 GHz and 12 cores. We chose 3 parallelization strategies. Let w, h be the width and height of the image in pixels. Considering that the workload on a single thread scales with the number of pixels:

- **Split the image in r rows**: Since the image has w pixels per row, each task will render up to $w\lfloor\frac{h}{r}\rfloor$ pixels, so the execution time scales with the inverse of the number of splits.
- **Split the image in c cols**: This case is analogous to the row split strategy, the time scales with the inverse of the number of column splits.
- **Split the image in r rows and c cols**: Every cell has up to $\left(\frac{w}{c}\cdot\frac{h}{r}\right)^2$ pixels, which means that it both scales with the inverse of the number of rows and columns, respectively.

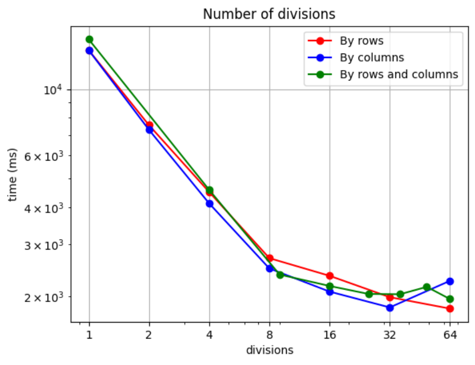


Figure 2: Number of divisions and execution time with the 3 strategies

We have to take into account that the processor doesn't have an infinite amount of cores, so if the number of concurrent tasks reaches the number of cores, the performance will be limited.

We observed the following:

- The parallelization strategy is independent of the execution time. However the number of cells (and therefore, the number of pixels per split) directly affects the execution time.
- We can observe that the 3 lines reach a critical point when the number of division surpass the number of cores. and in that case, the execution time stays the same.

3.1 Parallelization of a simple image processing pipeline (Qα)

A simple image processing pipeline shown in Figure 3 has been implemented in order to detect the soft and hard edges of an image with a Canny edge detector[1]. Since the workload has a variable size, the task is suitable for a threadpool acceleration approach.

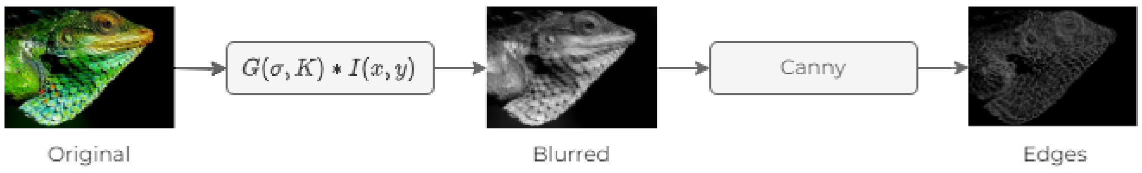


Figure 3: Simple Canny edge detector

The task has been parallelized by splitting the image in tiles, and a single threadpool will be in charge of applying both stages of the pipeline. A similar behaviour to the raytracer example was found here, where the execution time decreases linearly with the number of threads until the number of subtasks reach the number of cores of the machine. At that point, the linear trend starts to slow down, up to the point where the gain in parallel execution is shadowed by the overhead of popping tasks from the queue. Figure 4 shows this trend, and also exhibits the irregularity in the execution of each subtask, because the execution time trend is not perfectly linear.

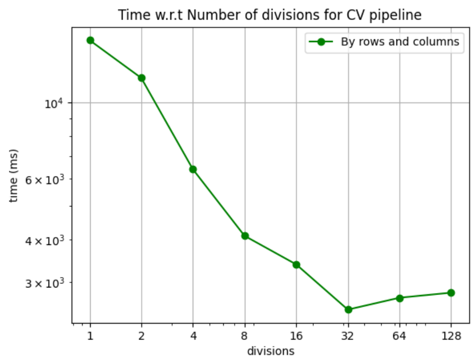


Figure 4: Canny edge detector execution time with respect to the number of tiles.

[1]No maximum-suppression nor hysteresis has been implemented for simplicity.

4 Conclusions

We can conclude that the parallelization potential comes from the number of splits done to the image, but performing too many divisions will add an extra overhead due to the time invested in popping tasks from the queue, even though this overhead is very small compared to the rendering time. It is also apparent that tasks with some degree of irregularity are better suited to use a threadpool as a parallelization technique.

Bibliography

[1] Maged M. Michael and Michael L. Scott. "Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms". In: *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*. PODC '96. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1996, pp. 267–275. ISBN: 0897918002. DOI: `10.1145/248052.248106`. URL: `https://doi.org/10.1145/248052.248106`.

[2] *Canny edge detector*. `https://en.wikipedia.org/wiki/Canny_edge_detector`. Accessed: 2023-11-26.