Lab 4 Programming and Architecture of Computing Systems

Diego Bielsa Monterde (776564), Alejandro Báscones Gállego (801469)

November 17, 2023

1 Introduction

The objective of this lab session is to advance knowledge in parallel programming, more specifically in two essentials of this programming architectures: a multi-producer multi-consumer queue and a thread pool.

2 Thread-safe Queue

The first part of the session consist on implementing a thread-safe queue, that provides an element to store and feed tasks, to the working threads that continuously trying to catch them to execute.

In order to guarantee correctness between elements in a multi-thread environment we must use functions like lock, mutex or condition variables, to ensure avoiding bad behaviours due to simultaneous actions on the queue. With this functions we achieve mutual exclusion, mutex allows one thread to lock a shared resource and unlock it when it is finished. We can combine this with a condition variable, which allows us to let threads efficiently wait for a condition to be satisfied, this happens due to the notification of another thread which has already finished using the shared resource.

3 Thread Pool

3.1 Thread pool destructor for completion of all tasks

There are two possible ways that let the programmer take advantage of the destructor:

- **Dynamic memory usage**: this is the most common one. The usage of dynamic memory allows the programmer to explicitly delete the memory allocation referenced, causing the destructor to be called as soon as the deletion is performed. In the following lines a code example is shown:

```
thread_pool* pool = new thread_pool
    (std::thread::
    hardware_concurrency());
...
delete pool; // destructor called
```

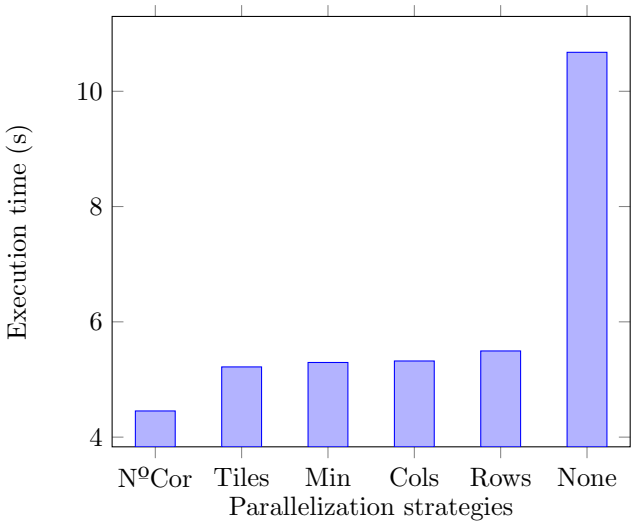
- **Scope usage**: There is also another option which, instead of taking advantage of dynamic memory (usually dangerous in c++), leverages scopes to manage the lifetime of objects. When a scope is established, all objects inside the scope itself only "live" in there, causing their destruction at the end of the scope. In the following lines a code example is shown:

```
{
thread_pool pool(
    std::thread::
    hardware_concurrency());
    ...
} // destructor called
```

4 Measuring Performance

In this section, several parallelization strategies have been used in order to measure the performance of each of them and getting the best strategy (regarding the chosen ones) that could be used to render an image. The tested strategies are: columns division (as $h/h_div >= 4$, the minimum number of columns per thread is 4), rows division (as $w/w_div >= 4$, the minimum number of rows per thread is 4), 4x4 tiles (minimum division), 128x128 tiles, amount of tiles equal to the number of cores and no parallelization. Note that a laptop with 4 cores has been used for these measurements.

4.1 Parallelization strategies measurements



In the graph displayed above this paragraph, a graph that relates the different parallelization strategies and execution times is shown. In the following list, three main clusters of execution times are explained:

- **Optimal**: the optimal execution time that can be obtained in this kind of task parallelization architecture is given by the amount of cores. Assuming that every task can be performed in the same amount of work, using exactly the amount of cores given by the machine leverages the maximum amount of resources resulting in no extra computation of any of them and no context-switches, which results in the optimal execution time. Regarding the graph, the column with the label *NºCor* is the one just described.
- **Sub-optimal**: in order to get any sub-optimal execution time the task must be divided in a higher amount of sub-tasks than the number of cores. This approach makes all cores work in a sub-optimal way as context-switches appears making the execution slower. The more sub-tasks, the more context-switches, the slower our program is. Regarding the graph, the columns with *Tiles* (48 tiles), *Min* (49.152 tiles), *Cols* (192 tiles), *Rows* (256 tiles) labels are described in this paragraph. (Given the low execution time, the amount of context switches does not get the importance it would with higher tasks).
- **Worst**: the worst parallelization strategy is for sure the no parallelization. This is really obvious, as with no parallelization of any work, there is no optimization of the execution time. According to the graph, the column named *None* is the one just described.