PACS LABORATORY 4

by

Daniel Ariza Antón (775545) and Marcos Ilarraza Sarto (797720)

QUESTION 1

Do we need to protect the function members with mutual exclusion to guarantee correctness?
Yes. We have to protect the functions by using mutual exclusion (or one/multiple mutex) to guarantee that multiple functions don't access critical data concurrently which could lead to an inconsistency or failure of the program we may try to run.

How std::mutex and std::condition_variable help with this task?
The std::mutex data type allows the programmer to craft a code section which can only be accessed by a thread at the same time to execute in mutual exclusion, allowing to handle shared variables without worrying about race conditions. The std::condition_variable data type makes use of std::mutex, through the std::unique_lock data type and a gamma function, that allows the programmer to stop a thread until the condition specified within the gamma function is achieved.

QUESTION 2

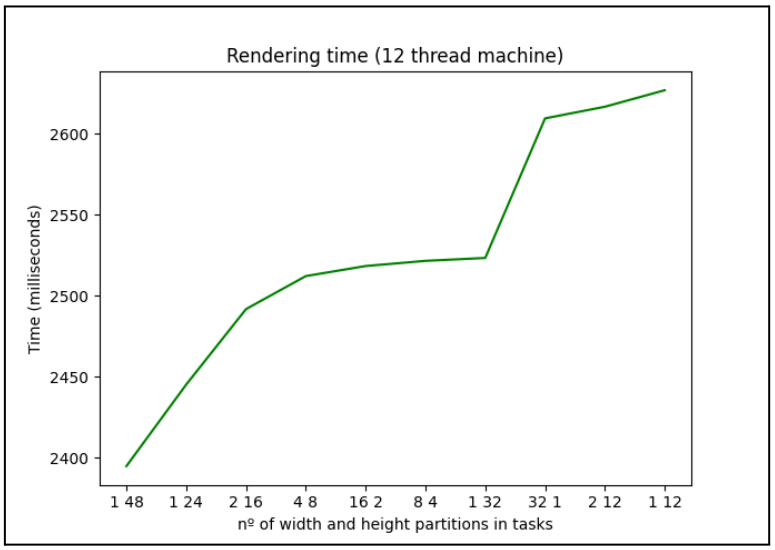
Is it possible to implement the thread pool, so that it waits for the completion of all the tasks with the help of thread_pool destructor?
Yes, it is. By calling the *wait* function in the thread_pool destructor we guarantee that the structure won't be destroyed until the threads have finished all the tasks. This will also stop the code from advancing until the call to the destructor is over.

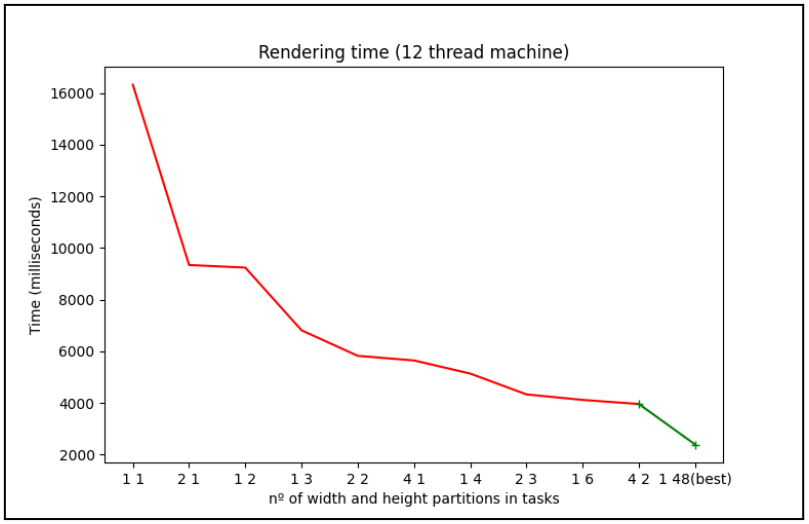
QUESTION 3

Plot the execution times (with the tested configurations on the X axe) and try to justify your findings. For example, does working with large regions provide more or less opportunities for speeding up the applications? Why?
For this question, we have identified the width (1024) and the height (768) of the image and then we have created a script that tested and printed the time used for each different combination of the divisors of each of the numbers, in other words, we have tested every single type of square, rectangle, column, row or tile that was possible. For this test, we used 4 samples per pixel.

Divisors of 1024: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024
Divisors of 768: 1, 2, 3, 4, 6, 8, 12, 16, 24, 32, 48, 64, 96, 128, 192, 256, 384, 768

After that, we processed the obtained times and decided to find the best and worst 10 times from the obtained means to analyze the correlation between the chosen divisions.





The X-axis represents the entry arguments of the program. For example, “1 48” which is the best time obtained in the testing, means that the program divided the rendering in a total of 1*48 = 48 tasks; and each task’s tile dimensions were such as, with **1024** pixels **wide** and **768** pixels **tall**, ***width*** = *1024/1 = 1024*, ***height*** = *768 / 48 = 16*. So, the best result was obtained by dividing the image into 48 tasks of 16 full rows each.
We did the tests on a 12-thread machine, and so we observed that the best times were achieved when the number of tasks matched a multiple of 12. We also tested it on another 16-thread machine and the results were similar, the best-performing executions were those with a nº of tasks equal to a multiple of 16.
In conclusion, to run the rendering optimally we will have to **take into account the number of threads of the machine** and match them with the number of tasks, as well as **not inputting an extremely big nº of tasks** because the tests revealed they take way more time than bigger divisions probably due to the overhead by queueing the task and compiling the results being too time consuming to get a better result (in this case) with more than 4 times the number of threads. For example, rendering pixel by pixel (argument entry 1024 768) gave a good result (2887.8 ms) but nothing over the top, as well as “1 96” which is the next multiple of 12 available(3025.6 ms).