Laboratory 4 - Report

Group Members:
Javier Almario, NIP: 962449
Alvaro Provencio, NIP: 960625

The program has been executed in an Intel Core i5-8300H CPU @ 2.30GHz.

1 Question 1: Do we need to protect the function members with mutual exclusion to guarantee correctness? How *std* :: *mutex* and *std* :: *condition_variable* help with this task?

Yes, We might want to protect our queue from race conditions and locks that lets only one thread access a critical section at a time. Then we use *std* :: *mutex* to guarantee unique operations over the queue and *std* :: *condition_variable* to wait until a value has been pushed to the queue, so it does not try to obtain any when the queue is empty in *wait_and_pop*().

2 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 the thread_pool destructor?

Yes, in fact, the current implementation already waits for threads to finish, and it waits for all tasks in the queue to complete. To achieve that, we use condition variables and mutex, in the wait() function we add the following condition (*active_tasks* == 0)&&*work_queue.empty*(), because the empty queue is not the same as to all the tasks being done.

3 Question 3: Does working with large regions provide more or less opportunities for speeding up the applications? Why?

The parallelization strategy used was to divide the image into a grid of rectangular regions (tiles); each region is rendered independently by a render function and submitted as a task to a thread pool, whose worker threads pull tasks from a shared queue and compute different tiles in parallel, while a completion mechanism (wait()) blocks the main thread until all tiles have finished so the full image buffer can be safely written to disk.

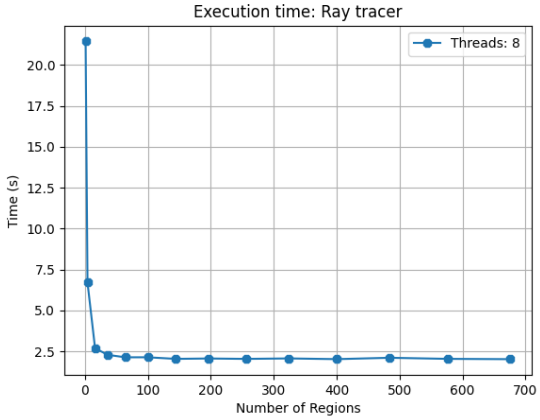

Figure 1: Execution time

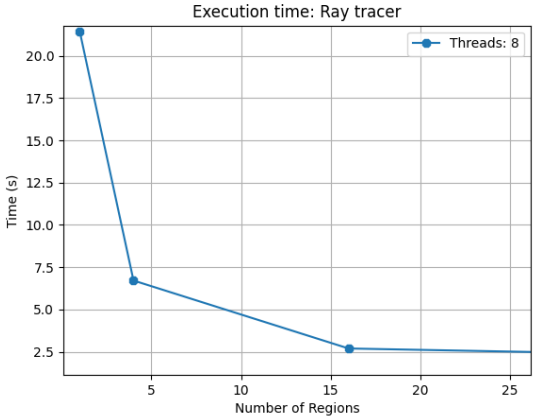

Figure 2: Zoom execution time

From the plot, we can observe that with the implementation of thread pool on Ray tracer program, it is roughly $\sim$ 8x times faster than when running on a single thread. As well, we can see from the graph that after about 15 regions, there is no further gain in execution time. Smaller regions means there are more tasks which improves load balancing and keeps all cores busy, giving more speedup—up to the point where the number of tasks is much larger than the number of threads,saturating the CPU.

4 Question 4: What are the advantages of removing de sleep_for from the thread pool? Should the new code reduces the amount of wasted execution cycles?

For the sleep we tested with this code in the wait() function:

```
void wait()
{
   while (active_tasks.load() != 0) {
       std::this_thread::sleep_for(std::chrono::milliseconds(1));
    }
}
```

Removing sleep_for and using a condition_variable instead avoids active waiting. With sleep_for, the waiting thread wakes up periodically just to poll shared state, which wastes CPU cycles and may delay detecting task completion by up to the sleep interval. With the new implementation, the thread blocks until it is explicitly notified that the last task has finished, so the CPU doesn't do useless work and the waiting thread resumes almost immediately. Yes, this reduces wasted execution cycles and gives a more efficient and scalable thread pool.

5 Question 5: What are the differences in total energy and average power? Does a reduction in execution time improve energy consumption?

Since our machine can obtain the energy measurements, we continued this last question on the same machine:

Config	# Regions	Avg exec time	Energy (power/energy-cores)	Average Power
1×1 regions	1	21 340.8 ms	515.38 J	24.15 W
2×2 regions	4	5888.2 ms	179.12 J	30.42 W
4×4 regions	16	2601.4 ms	122.26 J	47 W

Table 1: Energy consumption comparison between 1, 4 and 16 regions (parallelized)

We see that our fastest configuration (16 regions) needs more power to complete the tasks than the 4 region execution, but since it takes less time, at the end it consumes less energy, so reduction in execution time can improve the energy consumption.

References

[1] Anthony Williams (2019), *Concurrency in action* Manning Publications, 2nd ed.