Assignment 3

Introduction to Parallel Programming.

David Padilla Orenga
Ignacio Pastore Benaim

Virtual Machine Specifications

The laboratory was executed in a Virtual Machine (VM) with the following specifications:

Specification	**Details**
Architecture	ARM64 (aarch64)
Machine Type	QEMU 7.2 ARM Virtual Machine
Memory	4 GB
Disk Size	21.38 GB

Question 1: Mutual Exclusion for Correctness

Mutual exclusion is critical for ensuring the correctness of multithreaded programs. In the absence of mutual exclusion mechanisms, shared resources accessed by multiple threads simultaneously can lead to race conditions. Race conditions occur when the behavior of a program depends on the relative timing of thread execution, often producing unpredictable results.

Role of `std::mutex`: A `std::mutex` ensures that only one thread at a time can access a critical section of code or a shared resource. For example, in a producer-consumer problem where both producer and consumer threads access a shared buffer, without a `std::mutex`, multiple threads might modify the buffer simultaneously, leading to data corruption.

Role of `std::condition_variable`: While `std::mutex` prevents race conditions, `std::condition_variable` ensures efficient coordination between threads. For instance, a consumer thread can wait for a signal from the producer thread, avoiding busy-waiting and conserving CPU cycles.

The combination of `std::mutex` and `std::condition_variable` allows for safe, efficient interaction between threads, ensuring correctness and optimal resource usage.

Question 2: Thread Pool Destructor

To ensure the thread pool waits for all tasks to complete before destroying the thread pool object, it is indeed possible to implement this logic in the destructor of the thread pool class. In our implementation, we follow these steps:

- The `wait()` function is called to ensure that the task queue is empty before proceeding with destruction.
- A flag (e.g., `_done`) is used to signal all worker threads to stop accepting new tasks.

Destructor Code Implementation:

```
~thread_pool() {
    wait();
    _done = true;
}

void wait() {
    while (!_task_queue.empty()) std::this_thread::yield();
}
```

This destructor implementation ensures that:

- The `wait()` function ensures all tasks in the queue are completed before setting the `_done` flag.

- No task is left incomplete, and the thread pool only proceeds with destruction once the task queue is empty.

In this implementation, the `wait()` function continuously checks if the task queue is empty. By yielding the thread (`std::this_thread::yield()`), it ensures that other threads can continue processing tasks until all are completed. Once the queue is empty, the destructor sets `_done` to true, which signals the threads to stop. This approach guarantees that all tasks are completed before the thread pool begins the destruction process.

Question 3: Performance Analysis

The `smallpt_thread_pool` binary was evaluated using different parallelization strategies, partitioning work into grids defined by rows and columns. The goal was to assess the impact of task size on execution time and overall performance, with reported execution times being an average of 3 runs.

This is a toy rendering application that parallelizes rendering by selecting different grid configurations. Increasing the number of rows and columns results in smaller tiles and higher levels of parallelization.

Performance Results

Figure 1 illustrates the execution times for these configurations. Smaller tiles increased parallelism but introduced significant overhead due to frequent task scheduling, while larger tiles reduced overhead but limited concurrency.

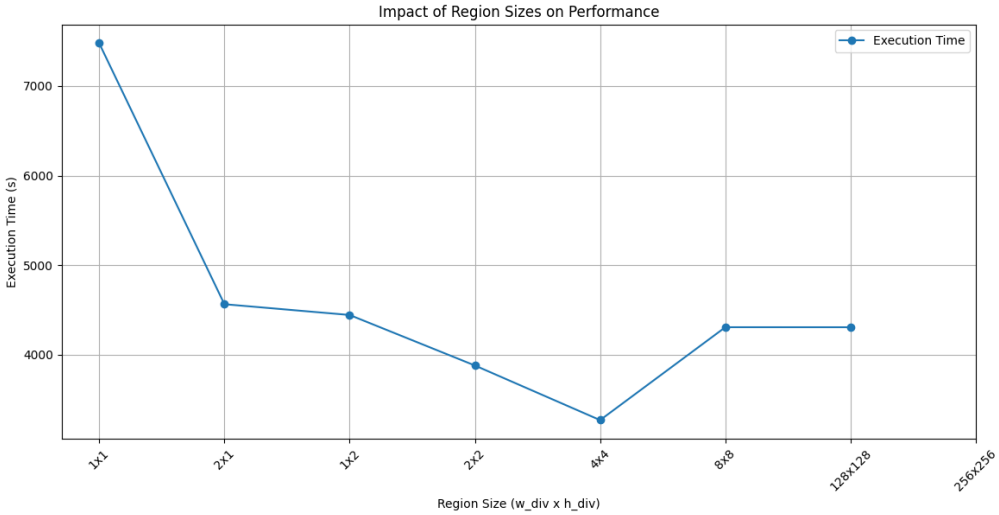

Figure 1: Execution times for different parallelization strategies.

- (1×1)**:** No upscaling of performance was observed due to the lack of parallelism.
- $(1 \times 2$ **;** $2 \times 1)$**:** Different performance results were observed between these configurations, likely due to variations in load balancing and the effect of splitting tasks into rows versus columns.
- $(2 \times 2$ **;** $4 \times 4)$**:** These configurations showed a steady improvement in performance due to increased parallelization, with 4×4 being the optimal setup for the virtual machine.
- (8×8) **;** $128 \times 128)$**:** While small tiles offer higher parallelism by creating numerous independent tasks, the overhead of task management outweighs the benefits. This results in suboptimal performance.
- (256×256)**:** The 256×256 tile size could not be executed due to limitations in the implementation: *"The minimum region width and height is 4"*.

Based on the analysis, the 4×4 tile size emerged as the optimal choice, offering a good balance of parallelism and efficiency for the virtual machine setup. The observed errors with larger tiles underscore the importance of carefully selecting region sizes compatible with the platform and application's requirements.