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

Yes, 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, consider 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.

Example:

```
std::mutex mtx;
void critical_section() {
    std::lock_guard<std::mutex> lock(mtx);
    // Critical code here
}
```

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.

Key Benefits: 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

Proper implementation of the thread pool destructor is essential to ensure that all resources are cleaned up correctly and that no tasks are left incomplete. The thread pool must ensure that:

- Threads are signaled to stop execution gracefully.
- Tasks in the queue are processed before shutdown.
- All threads are joined, ensuring their complete termination.

Steps for a Graceful Destructor: 1. Signal threads to stop using a flag (e.g., `done`). 2. Notify all waiting threads using `std::condition_variable`. 3. Join all threads to ensure they have completed execution.

Code Implementation:

```
~thread_pool() {
    done = true;
    condition_variable.notify_all();
```

```
    for (auto& t : threads) {
        if (t.joinable()) t.join();
    }
}
```

Why This Matters: Without proper synchronization, threads might access resources that have already been deallocated, leading to undefined behavior. The destructor ensures that tasks are either completed or safely discarded, maintaining consistency.

Advanced Considerations: - Implementing a task queue that supports prioritized shutdowns. - Logging unfinished tasks for post-mortem analysis.

Question 3: Performance Analysis

The `smallpt_thread_pool` binary was evaluated using different parallelization strategies. The strategies included partitioning work into rows, columns, and tiles of various sizes, such as 128×128, 256×256, 16×16, and 4×4. The goal was to assess the impact of task size on execution time and overall performance.

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.

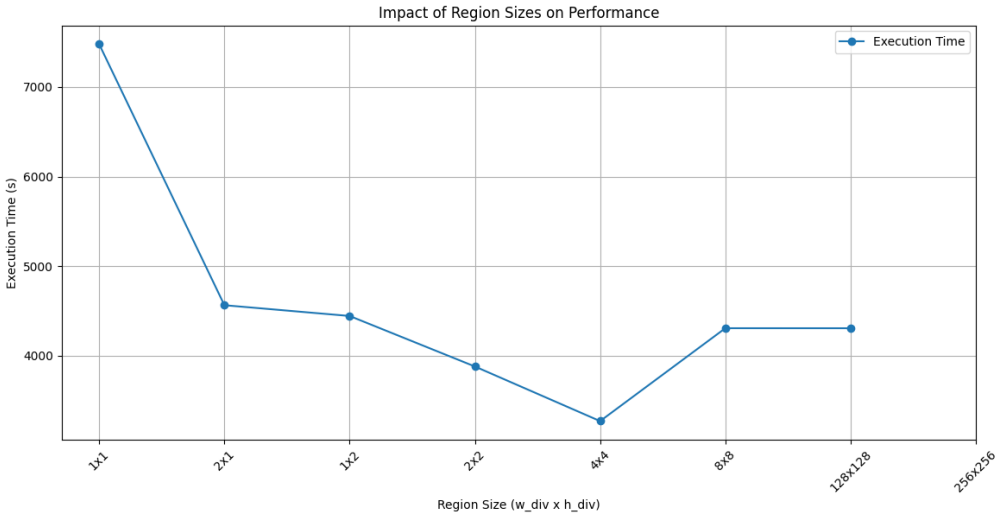

Figure 1: Execution times for different parallelization strategies and region sizes.

Discussion of Results

- **Small Tiles (16×16):** While small tiles offer higher parallelism by creating numerous independent tasks, the overhead of task management outweighs the benefits. This results in suboptimal performance.
- **Large Tiles (256×256):** Larger tiles reduce the number of tasks, lowering overhead. However, this strategy limits parallelism, as fewer tasks are available to distribute among threads.
- **Medium Tiles (128×128):** This configuration achieved the best performance by striking a balance between task granularity and management overhead. The tile size was well-suited to the workload, dividing it into manageable chunks without overwhelming the task queue.

Conclusion

Based on the analysis, the 128×128 tile size emerged as the optimal choice, offering a good balance of parallelism and efficiency. The observed errors with larger tiles underscore the importance of carefully selecting region sizes compatible with the application's requirements. Future work could explore dynamic adjustments to tile sizes based on runtime performance metrics.