Jacob Stein, Marek Pinto
CS3211 - Assignment 3

Outline and Implementation

Our initial approach was to simply use Tokio to make a capped pool of lightweight threads and run the tasks on threads based on availability. We realized that this was still not concurrent, because the while loop would wait for the Future to fulfill itself before executing the XOR and enqueue operations. This meant that a new Task would not run on another thread until the previous had finished, essentially serializing the program.

In order to circumvent this issue, we decided to split the work done with the result of the Task into two separate parts: the execute() function, and then XOR and enqueue. We knew we needed a system that would initiate multiple execute() calls on separate threads, and would only do work in the main thread for XOR and enqueue when execute() had finished running on a given thread. We also knew that XOR and enqueue could not be performed outside the main thread because we would not be able to safely pass around the taskq data structure concurrently in Rust.

We started by making a multi-input, single-read channel to control the flow of Task results from execute() running on multiple threads to the main thread. For each iteration of the while loop, a nested loop first reads all results off the channel to check for completed execute() tasks. For any Tasks it finds, it then performs the XOR operation with the output and adds the new task to the queue. When the channel is empty, we then use tokio::spawn to run execute() on the current Task in any available threads. This way, while execute() is running, the outer while loop can continue iterating without having to wait to perform XOR and enqueue.

With this implementation, we identified the edge case in which all tasks have been read from the queue, but not all execute() calls have finished running in the background. This means the outer while loop would stop running, and not all child tasks would be properly executed. To circumvent this, we created our own barrier. Every time a task is read, we increment a counter that starts at 0. Whenever we perform XOR and enqueue, we decrement that counter. The counter essentially keeps track of how many tasks are executing in the background. We added an adjunct condition to the outer while loop to keep running so long as there are background tasks executing. This ensured that all tasks would be properly handled.

Our implementation ended up being able to perform its function with a depth and children limit of 5 in between 5 to 8 seconds, almost 10 times faster than the serialized version. With a depth and children limit of 6, our program completed in around the same time the serialized version took to complete with limits of 5. We also added a check for storage usage to gauge our tradeoff between concurrency and space. Because we are using lightweight threads, our program uses

around 0.3 GB when it runs. This uses very little space, so the program does end up taking a very long time to complete for large values. Nonetheless, we were able to minimize the memory usage, which is the goal parameter of this assignment.

Concurrency Paradigm

Our primary concurrency primitive is a Tokio mpsc (multi-producer, single-consumer), which enables communication between the main thread and the spawned worker thread. Worker threads are created and used with the Tokio spawn macro, which can be used to perform the execute() method asynchronously. When the method is complete, it will communicate along the mpsc, and the main thread will then complete XOR and enqueue() for that task on its next iteration.

For the spawn macro itself, the move keyword transfers ownership of the ‘next’ variable to the spawned task closure. We also have the await keyword to ensure that the task is successfully completed and then transferred along the mpsc.

In summary we use Tokio’s spawn macro to do execute() asynchronously, then use an mpsc to read communications from the worker threads so that work that must be done in the main thread can then follow in the correct order.

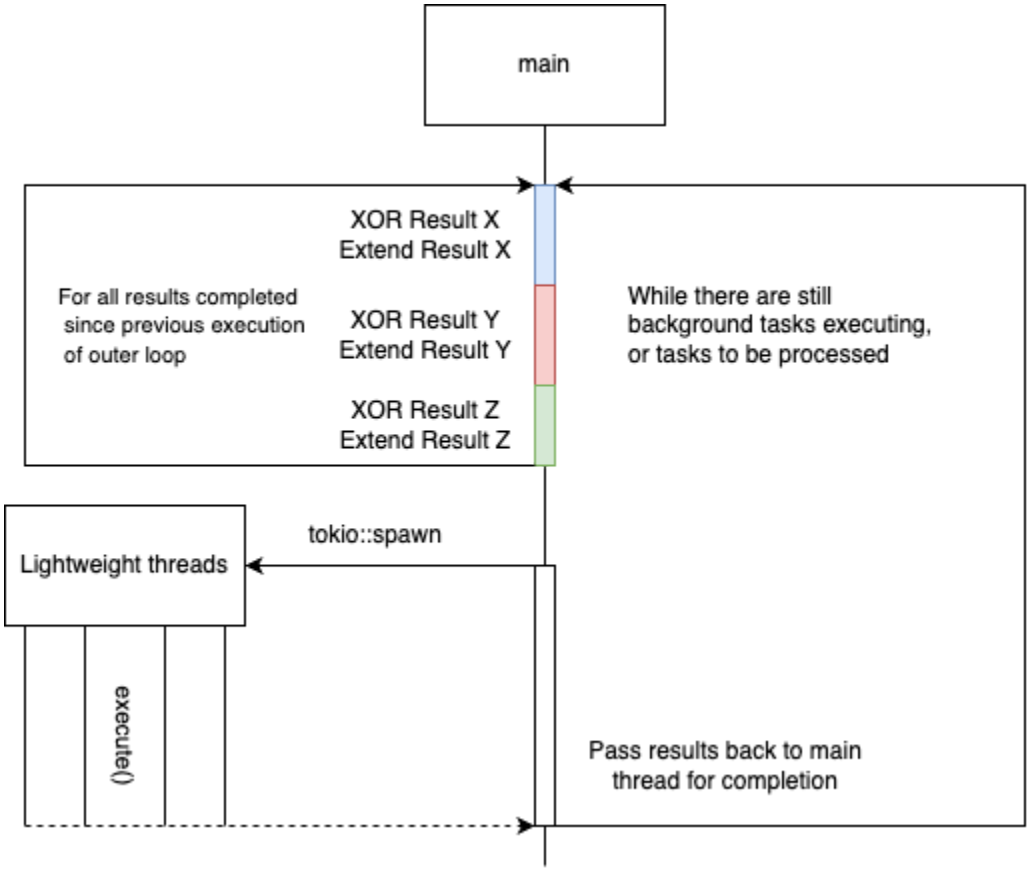
Task Scheduling

Each task is broken into two phases: execute() and XOR/enqueue. XOR/enqueue rely on shared data, so we can not safely make it run in parallel. The data enqueued is also reliant on the result of execute(), so the two phases must be serialized, but only one of them can run asynchronously.

On each iteration, we first complete the second phase for any tasks that have finished their first phase asynchronously and sent their latest TaskResult via the channel. Once this is complete, the program then starts the first phase on a worker thread for the pulled task. The loop will keep running, but at some point execute() for the pulled task will finish on the worker thread.

When the first phase finishes, it will send its TaskResult via the channel. On the next iteration of the loop, the second phase will execute on the main thread. Our program effectively maintains serialization for each task, but allows for the concurrent execution of multiple task executions at the same time.

Parallelism



Above is a diagram showing the execution of our program. Keep in mind, however, that the lightweight threads DO NOT need to complete execute() before the next iteration of the outer loop is entered. This is where the concurrency in our program comes into play. The outer loop can continuously process the results while the lightweight threads run execute() in the background. Whenever execute() finishes, the result will be passed via the channel and processed in main, yet other threads can continue running execute() on other tasks.

As mentioned earlier, phase one corresponds to execute() and phase two corresponds to XOR/Enqueue processing. For any given single task, phase one and two are serialized. For multiple tasks, their phase twos are serialized, but a number of phase ones can run in parallel in the background while a phase two is being completed on main. This is also an example of Amdahl's Law. The speedup of our program is limited by the fraction of it that is serialized, which is the execution of phase two for all tasks. This is represented by the color-coded bars in the diagram.

Alternative Implementations

First Iteration

Our original plan was to create a task pool using Tokio, and then simply assign execute() to a thread in the pool to execute asynchronously. We thought the for loop would continue running and start new execute() calls, but then realized that the loop would wait for execute() to run in the background in order to call XOR and enqueue before continuing to the next iteration of the loop.

These circumstances essentially meant that our program was serialized, with execute() running on a different thread while the main thread was waiting, which is actually worse than full serialization because it creates more overhead. We then realized that we should split the work done per task into two phases, and attempt to perform at least one of the phases concurrently.

Second Iteration

After realizing our mistake on the first iteration, we tried to move both phases into the lightweight threads. This quickly created issues because of the taskq VecDeque structure. Passing the structure between different lightweight threads would have opened the door for data synchronization issues, and would have required much more overhead. We quickly wrote off this idea.

Third Iteration

After realizing that the XOR/enqueue operations could not be executed outside the main thread, we decided to split into two phases and communicate using a channel. However, we realized that there was a possibility of our loop stopping when there were no tasks left in the taskq, even though tasks were executing in the background that would send additional subtasks via the channel. Therefore, we added a num_background variable that is incremented whenever a worker thread is created and decremented whenever results are received via the channel. Then, we added a condition to our while loop such that it wouldn't leave the loop until num_background was equal to 0 (meaning no tasks running in the background). This is what led to our final working product.