CS3211 Assignment 3

- Albert Sutiono (A0219773U)
- Daffa Fathani Adila (A0219822A)

Task Runner Implementation Description

Our implementation of the task runner is quite similar to the sequential implementation: maintain a hashmap to count tasks finished and the output/result. Our main logic is implemented as such:

1. Keep a counter of how many tasks are executed. This counter is initialized to 0.
2. We also maintain a channel (tx, rx) to store the result of each task execution (which may also contain the children task)
3. Generate initial tasks, send it to be executed by the **thread pool,** and increment the counter. The result of this execution will be sent to channel **tx**.
4. While there are tasks processing (counter > 0), we want to poll the result from the channel **rx** while also decrement the counter, signaling that a task has finished executing. We then iterate through the generated task from this result, and it will be sent to be executed by the thread pool. Also, increment the counter by 1 every time we send a task to be executed.
5. Keep repeating until all tasks have been processed (counter reaches 0), then print the result to the console.

Explain the main paradigm (concurrency primitive) used in your concurrent implementation:

The main paradigm that we use is the master worker. The main thread acts as the master thread that will spawn threads to execute the tasks and poll the result from the channel. Hence, we do not need any synchronization primitives to keep track of the result as the counter hashmap and output are only modified by the master thread.

Furthermore, the main concurrency primitive that we use is Thread Pool. The number of tasks could reach up to 1 million tasks, so Thread Pool will help to limit the resource/thread spawned which otherwise could crash the program if we designate each task to a single thread instead.

Explain how the tasks are scheduled and how new tasks are spawned:

The main thread polls from the result channel **rx** which also contains the next tasks to be scheduled. The main thread then schedules/sends all the next tasks to the Thread Pool to be executed.

New tasks are generated from the Task::execute function. The result of this execution will be sent to the Channel and later be processed by the master thread.

Will your implementation run in parallel? If yes, explain why?

Yes, as we set the number of threads in the Thread Pool to be greater than 1 (in our case it's 8 or 16), there should be multiple tasks that are executed by the thread in Thread Pool. The only task that is done sequentially is the update of the count_map and output, and spawning new threads to execute tasks. The bulk of the computation, which lies inside Task::execute() is executed in parallel within the Thread Pool.

In case you have tried multiple implementations, explain the differences and your evolution to the current submission.

1. Initially, we intended to use the Asynchronous model of concurrency which uses the Tokio library. We came up with an approach to batch the processing of the tasks and execute it asynchronously. However, this approach does not provide the maximum concurrency because of the tasks batching approach. Hence we decided not to use this implementation.
2. We decided to use Thread Pool for the reason mentioned above. Initially, we implemented it such that each thread/job in the task pool can issue a task to be executed in the task pool. The update of output and count_map is also done within each thread instead of the master thread. However, this approach becomes very messy as now we need to introduce a synchronization mechanism for the HashMap, Output, and Pool Thread itself by implementing Send and Sync Traits. We also faced a problem in joining the thread pool after all the tasks have been processed. Hence we decided to use the current implementation.