Lee Ee Jian - A0183105N
Leow Jit Yong - A0182998A

Assignment 3: Task Runner in Rust

1. Outline and brief explanation

In summary, our implementation makes use of worker threads with no designated master thread. Each thread primarily executes tasks from its local task queue, and can perform work stealing from the global queue or from other threads' task queue. This is facilitated through work-stealing data structures from the crossbeam library.

Initial tasks will be pushed into the global task queue by the main thread which the worker threads will steal from. Each worker thread will execute and generate children tasks, which will be added to their local task queue.

After spawning the worker threads, the main thread blocks until all worker threads have completed their join and exited.

2. Main implementation

Upon program execution, the main thread retrieves the number of CPUs that the machine it is running on has access to. This is the number of threads that will be spawned later in the program. Next, the main thread initialises the variables that will be shared amongst the spawned threads. The main shared variables, as well as their usages are summarised as follows:

Variables	Data structure	Usage
stealers	RwLock<Vec<Stealer<Task>>>>	Vector of stealer handle of every worker's queue protected by a RwLock.
injector	Injector<Task>	Global FIFO task queue where tasks are pushed and stolen from opposite ends.
queued_count	AtomicUsize	Total number of tasks that are waiting to be executed by all threads
num_idle	AtomicUsize	Number of idle worker threads
combined_output	AtomicU64	Final output required of program
num_hash, num_derive, num_rand	AtomicU64	Final count of each task type executed in program

Table 2.1

The injector is then populated with the initial tasks generated from the user-inputted seed. Next, each thread that is spawned acquires it's own shared pointer to the aforementioned shared variables and initialises its own local variables, whose usages are summarised as follows:

Variables	Data structure	Usage
local_queue	Worker<Task>	Local FIFO task queue which is a concurrent work-stealing deque

local_output	AtomicU64	Local output generated from execution of tasks in thread's local taskqueue
local_num_hash, loca_num_derive, local_num_rand	AtomicU64	Local count of each task type executed in thread's local taskqueue
is_idle	AtomicBool	True when thread has tasks to be executed, False otherwise

Table 2.2

The local_queue provides a method stealer() which creates a Stealer that may be shared among threads to steal tasks from its own task queue. Each thread proceeds to add its own Stealer to the list of shared stealers, which is synchronised by the RWLock. Next, the thread enters into an infinite loop to execute tasks.

To obtain a task to execute, the thread first looks into its local task queue, and then into the injector and lastly stealers. If a task is obtained, the worker executes the task, adds the resultant output into local_output, and updates the relevant local count variable. If a task cannot be obtained, there is no work to be done currently and so the threads change from active to idle state, in which is_idle becomes true and num_idle is incremented. The state is changed back to active (is_idle becomes false) whenever the thread is able to obtain a task in subsequent iterations of the infinite loop, in which case num_idle will be decremented.

Whenever a task cannot be obtained, the thread also performs a crucial check to see if queued_count is 0 and num_idle equals to the number of worker threads. This would mean that there are no tasks to be executed and all threads are idling. When this condition is met, it means that all tasks have finished executing and no new tasks will be further generated. If so, the thread breaks out of the infinite loop to exit. Before exiting, the threads updates the combined_output and shared count variables with its own local_output and local count variables.

When all threads have exited and completed their join() called in the main thread, the main thread unblocks and proceeds to print the tallied combined_output and count variables.

3. How we achieve concurrency

We achieve concurrency by spawning worker threads that can execute tasks and make progress on their tasks at the same time. If the worker threads share a cpu thread, their operations can still interleave with one another to achieve concurrency. To achieve this, we have adopted synchronisation primitives through the use of atomics and mutexes to prevent data races and problematic race conditions.

There are a number of variables that have to be shared between worker threads (refer to Table 2.1). We made use of atomics heavily for variables that perform operations such as addition and XOR, which are readily supported by Rust. The use of atomics helps prevent data races and is also more efficient than mutexes.

Use of Atomics and Memory Ordering

Recall that the condition to determine that all tasks executions are complete is when num_idle == n_workers and queued_count == 0.

The memory order for num_idle and queued_count is Release/Acquire because it is important to synchronise these two variables between any two worker threads. This is to prevent any data race conditions that may result in both the above condition to be true when in reality there are tasks to be run and/or worker threads that are still running.

Within a worker thread, we ensure the following happens-before constraint for each task execution: (1) Decrement num_idle → (2) Decrement queued_count → (3) Increment queued_count → (4) Increment num_idle.

With the use of Release/Acquire memory ordering, we can ensure that at any time when a thread observes an (4) incremented num_idle count (task goes to idle state after finishing task execution), the queued_count observed takes into account the (3) incrementing of queued_count (adding of children tasks) from that same thread that incremented num_idle.

Likewise, observing a (2) decremented queued_count (task execution) means that the observed (1) num_idle has also been decremented (change worker to non-idle because it's executing a task) by that same thread. This helps to prevent problematic race conditions which makes num_idle == n_workers and queued_count == 0 when in fact there are still tasks to be executed or worker threads running.

However, in our actual implementation as described in section 2, we do not perform the decrement and increment of num_idle before and after every task execution. This is to reduce the number of atomic operations performed to improve concurrency. As we still perform the update to num_idle in the same sequence as explained above, our correctness is still ensured.

We also make use of atomics for shared variables like num_hash, num_derive, num_rand and combined_output which the worker threads will update only once after all tasks are completed. As we perform a join() on all worker threads in the main thread, we ensure that all operations performed by the worker threads happen-before the operations after the join() in the main thread. Hence, we are able to use relaxed memory ordering for the mentioned atomic variables and ensure that the results printed in the main thread after the joins are the updated values.

Use of ReadWrite Locks

We also made use of a ReadWrite lock for the list of stealers shared between worker threads. We use RWlock because each worker thread only writes to this list once at the beginning, and all subsequent accesses are read operations. Since multiple threads can obtain the read lock at the same time, we improve concurrency for the read operations.

Use of Crossbeam::Deque

Crossbeam deque provides us with an optimised implementation (also uses atomics) to store our tasks locally in the worker queues, perform concurrent work stealing (from worker & injector queues) and perform concurrent access to Injector (global queue) between threads.

4. Running in Parallel

Apart from the operations that require synchronisation such as the use of atomics and mutexes as mentioned in the previous section, the bulk of the task execution (ie. the computation) are independent of each other and hence can be run in parallel. We spawn a worker thread for each available cpu thread and each worker thread will be able to run in parallel.

If we spawn more worker threads than available cpu threads, worker threads sharing the same cpu thread will run concurrently while worker threads using different cpu threads will run in parallel.

In view of running in parallel using multiple CPU threads, we made use of work-stealing functionality provided by Crossbeam::Deque to distribute tasks more evenly between worker threads. This aims to optimise the use of resources by reducing cases where worker threads are idle (ie. not working on any tasks) due to uneven distribution of workload (ie. other worker threads have many tasks to execute).

5. Differences and your evolution to the current submission

One difference between previous and current implementation is in the use of concurrent working stealing deques. In the previous iteration, the threads shared access to a common task queue, which was protected by a mutex. In order to obtain a task, each thread would have to obtain a lock to the task queue prior. In contrast, our current implementation allows for each thread to have their own local task queue. The current implementation is more concurrent as it does not require a critical section which is frequently accessed by all threads. In addition, in the event there are no tasks in the local queue, the thread is able to perform work-stealing, which maximises the number of active threads at any point of time and ensures a fair distribution of work in each thread.

Another difference between previous and current implementation is in the use of the local output and count variables. In the previous iteration, the threads shared access to a count_map and combined_output.These variables were accessed frequently by each thread because the variables required to be updated for every task executed. In contrast, our current implementation allows each thread to store and update local output and count variables which are only accessed by the thread itself. Each thread only updates the shared count variables exactly once before they exit. This drastically reduces the access to shared variables by threads, and thus increases concurrency.

Another difference is in determining the number of threads to spawn. In the previous implementation, the number of threads was simply hard-coded. In contrast, our current implementation determines the number of threads based on the number of available cpu threads. This allows for better parallelism since we can increase the number of threads when there are more cpu threads for work to be done in parallel. Conversely, we can reduce the number of threads when there are less cpu threads, which would have otherwise introduced context switching overheads between user threads for a limited number of cpu threads.