Task Runner Implementation

Our main submission task runner is a breadth-first-search (BFS) iterative implementation.

- 1 crossbeam unbounded channel is initialised in main and shared between the threads in the threadpool. There's only as many threads as there are cpu cores.
- Each thread takes a task from the channel whenever they're ready, processes the task, and atomically increments the global DERIVE, HASH or RANDOM counts by 1 depending on the task type.
- Result of task is atomically XOR with global OUTPUT
- INPUT_CNT and OUTPUT_CNT are atomically incremented by length of resultant tasks and respectively to keep track of completion status.
- Each newly generated task is sent into the channel for other available threads to read and process.
- Meanwhile in the main thread, it waits on a conditional variable. When all tasks are completed (INPUT_CNT == OUTPUT_CNT), then the conditional variable will be signalled and set to true, waking the main thread up to check the status of task completion. Main thread sees that all tasks are complete and prints the time taken and various counts.

Assumptions:

1. Number of generated tasks will always exceed completed tasks at any point in time unless there are no more pending tasks left
2. Processing of tasks are independent of each other.
3. Spawning of tasks are dependent only on their parent tasks of a higher height.
4. Tasks themselves are thread-safe

Main paradigm

It uses threads and a multi producer-multi consumer FIFO queue (crossbeam channel), globally accessible primitives like integers implemented in atomics. Threads are synchronised with the main thread when tasks are all complete using a conditional variable.

Scheduling and Spawning

Tasks are scheduled on a first come first serve basis in the FIFO queue. If a task was processed earlier but took a longer time than other tasks, spawned tasks will be added to the queue later.

Parallelism

Yes it can run in parallel. The crossbeam channel is MPMC, thus threads can read from the channel in parallel. As highlighted by assumptions 2,3 annd 4, these tasks are largely independent from each other and can be run in parallel.

Implementations explored

Our submitted implementation is the first solution we came up with as it's one of the easiest to implement and easily hits the max concurrency requirements (in terms of no idling threads/cpu cores).

The following implementations were explored in an attempt to hit the bonus requirement

BFS one height at a time

We identified that our earlier solution breaches the memory limit too quickly because tasks from lower heights are added to the queue while higher height tasks are still processing. Thus we tried to constrain the number of items in the queue to the number of generated tasks at a single height.

Github branch: thread_spawn

Concurrency: Not as optimal as main submission as threads will need to wait for all other threads to finish their tasks before proceeding to the next level.

Parallelism: Aside from waiting for thread completion at each level, tasks can still be processed in parallel.

Memory: Still exceeds 4GB as the number of tasks generated at each subsequent level exponentially increases. Furthermore, 2 task crossbeam channels (1 for current height, another for next height) are maintained at any time.

DFS Recursive

The previous solution breaches the memory limit too quickly because tasks from the same level are ALL expanded before having a chance to be processed in the next height. Thus we tried DFS such that one task expansion is followed by the completion of all children tasks under it. Worst case number of tasks in memory is cpu_count *(10000 ^ 10000), so actually it's worse than the previous implementation.
We also tried using **move** instead of cloning of tasks in previous implementations, which did see a visibly slower climb in memory usage. However, it exceeded 4GB nonetheless.

Github branch: dfs_recurse

Yet to try: Changing task.rs to provide an iterator instead of collecting all results into a vector