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

Github branch: threadpool+bonus

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 and 4, these tasks are largely independent from each other and can be run in parallel.

<u>Implementations explored</u>

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.

Asynchronous with Tokio

We also explored using Tokio's runtime and task scheduler to run the tasks as asynchronous functions. The main difference with the crossbeam implementation is that we did not need to implement a thread pool, and simply needed to wrap task execution in an asynchronous function, and call them. However, the implementation used BFS, one level at a time, which restricted the level of concurrency. The main challenge was dealing with the recursive nature of task creation, as asynchronous functions do not allow recursion. The benefit was that the amount of code was significantly less, showcasing the power of asynchronous programming in reducing mental overhead in coding tasks to run in parallel. However, considering that the main benefit of asynchronous programming is to avoid threads blocking on IO resources, resulting in context switching, it is probably not the best solution to the task scheduling problem, which is CPU-bound.

Github branch: tokio

Concurrency: same as BFS one height at a time

Parallel: Tokio runs the tasks in parallel for us

Memory: Exceeds 4GB. The likely cause of high memory usage is similar to other BFS solutions, as the number of tasks that we will await will still grow exponentially.

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. This is also because DFS has more potential to use less memory than BFS in theory given finite state space of 10000 max height and 10000 max children. However, as in task.rs children tasks are collected all at once, max number of tasks in memory can reach cpu_count *(10000 ^ 10000), so actually it's worse than the previous implementation in practice.
We also tried using **move** instead of cloning of tasks in earlier attempts, which did see a visibly slower climb in memory usage. However, it exceeded 4GB nonetheless.

Implementation:

- Spawn cpu_count number of threads, each thread is assigned an index from 0..cpu_count
- Each thread traverses the initial task queue cpu_count tasks at a time, starting from their assigned index, and processes the task at that index. E.g. thread 0 processes tasks at index 0, 24, 48 if cpu_count is 24.
- Child tasks are then recursively processed in the task_handler function, which outputs a final XOR value for all processed children tasks. Final xor value for parent is then atomically XOR with OUTPUT.

Concurrency Primitives: Threads, atomic counters and RWLock on the initial task queue.

DFS Recursive + modified task.rs

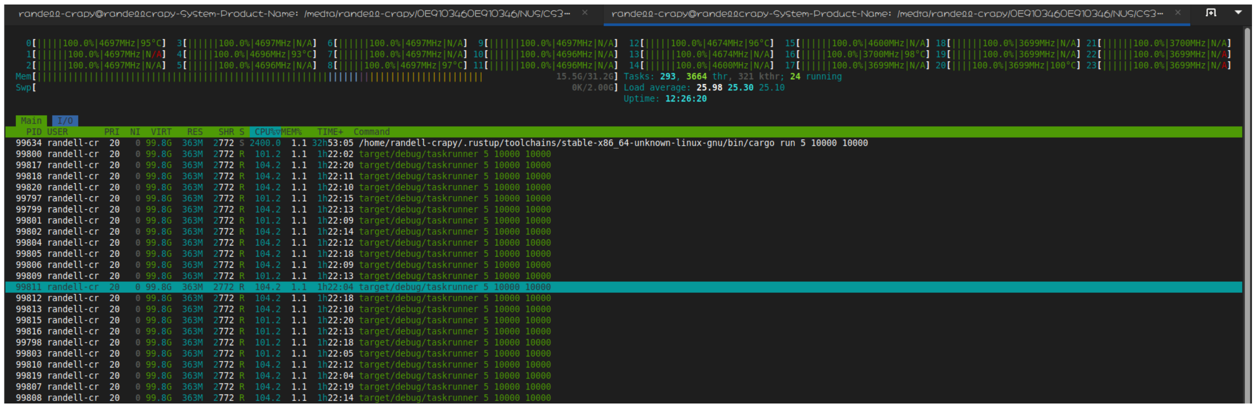
The previous solution breaches the memory limit too quickly because each level of recursion involves the expansion of the child tasks created from parent tasks. So ultimately I've resigned to the fact that we must prevent this immediate expansion from occurring before it is far from being processed.
Thus we modified task.rs, in particular the generate_set function to output an Iterator instead of a collected Vector. This lazily generates new tasks from parent tasks as opposed to all child tasks at one shot. In the task_handler function in main.rs, we dropped the "next" task after results were generated and dropped "new_task". We can drop new_task after it's processed because all child tasks would've been processed by then. Aside from these, implementation is basically the same as **DFS Recursive**

Github branch: dfs_recurse

Concurrency: Just like in **DFS Recursive**, it's not as optimal as main submission as some threads may be lucky and handle initial tasks that spawn significantly less tasks than others or tasks that in total take a lot less time to process. However, assuming this task handler is a daemon that doesn't really run to completion in systems that run forever like general and real-time operating systems, it's practically just as concurrent.

Parallelism: Aside from lucky threads waiting for unlucky threads with a lot more/ more time consuming tasks to handle, tasks are still processed in parallel.

Memory: In theory, this implementation maximum has 10000 * cpu_count of tasks. This is exactly the constant space complexity we're after! After running for 1.5 hours on my 24-core computer, the physical memory has seemingly capped to 363MB. Thus the memory optimisations have seemed to work. The image below is the result from htop command



```
randell-crapy@randellcrapy-System-Product-Name: /media/randell-crapy/0E9103460E910346/NUS/CS3⋯  ×   randell-crapy@randellcrapy-System-Product-Name: /media/randell-crapy/0E9103460E910346/NUS/CS3⋯  ×

    0[|||||100.0%|4697MHz|95°C]  3[|||||100.0%|4697MHz|N/A]  6[|||||100.0%|4697MHz|N/A]  9[|||||100.0%|4697MHz|N/A]  12[|||||100.0%|4674MHz|96°C]  15[|||||100.0%|4600MHz|N/A] 18[|||||100.0%|3699MHz|N/A] 21[|||||100.0%|3700MHz|N/A]
    1[|||||100.0%|4697MHz|N/A]  4[||||100.0%|4696MHz|93°C]  7[|||||100.0%|4697MHz|N/A] 10[|||||100.0%|4696MHz|N/A]  13[|||||100.0%|4674MHz|N/A]  16[||||100.0%|3700MHz|98°C] 19[|||||100.0%|3699MHz|N/A] 22[|||||100.0%|3699MHz|N/A]
    2[|||||100.0%|4697MHz|N/A]  5[|||||100.0%|4696MHz|N/A]  8[||||100.0%|4697MHz|97°C] 11[|||||100.0%|4696MHz|N/A]  14[|||||100.0%|4600MHz|N/A]  17[|||||100.0%|3699MHz|N/A] 20[||||100.0%|3699MHz|100°C] 23[|||||100.0%|3699MHz|N/A]
  Mem[||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||          15.5G/31.2G] Tasks: 293, 3664 thr, 321 kthr; 24 running
  Swp[                                                                                                 0K/2.00G] Load average: 25.98 25.30 25.10
                                                                                                                 Uptime: 12:26:20

  Main  I/O
    PID USER        PRI  NI  VIRT   RES   SHR S  CPU%▽MEM%   TIME+  Command
  99634 randell-cr   20   0 99.8G  363M  2772 S 2400.0  1.1 32h53:05 /home/randell-crapy/.rustup/toolchains/stable-x86_64-unknown-linux-gnu/bin/cargo run 5 10000 10000
  99800 randell-cr   20   0 99.8G  363M  2772 R  101.2  1.1  1h22:02 target/debug/taskrunner 5 10000 10000
  99817 randell-cr   20   0 99.8G  363M  2772 R  104.2  1.1  1h22:20 target/debug/taskrunner 5 10000 10000
  99818 randell-cr   20   0 99.8G  363M  2772 R  104.2  1.1  1h22:11 target/debug/taskrunner 5 10000 10000
  99820 randell-cr   20   0 99.8G  363M  2772 R  104.2  1.1  1h22:10 target/debug/taskrunner 5 10000 10000
  99797 randell-cr   20   0 99.8G  363M  2772 R  101.2  1.1  1h22:15 target/debug/taskrunner 5 10000 10000
  99799 randell-cr   20   0 99.8G  363M  2772 R  104.2  1.1  1h22:13 target/debug/taskrunner 5 10000 10000
  99801 randell-cr   20   0 99.8G  363M  2772 R  101.2  1.1  1h22:09 target/debug/taskrunner 5 10000 10000
  99802 randell-cr   20   0 99.8G  363M  2772 R  104.2  1.1  1h22:14 target/debug/taskrunner 5 10000 10000
  99804 randell-cr   20   0 99.8G  363M  2772 R  104.2  1.1  1h22:12 target/debug/taskrunner 5 10000 10000
  99805 randell-cr   20   0 99.8G  363M  2772 R  104.2  1.1  1h22:18 target/debug/taskrunner 5 10000 10000
  99806 randell-cr   20   0 99.8G  363M  2772 R  104.2  1.1  1h22:09 target/debug/taskrunner 5 10000 10000
  99809 randell-cr   20   0 99.8G  363M  2772 R  101.2  1.1  1h22:13 target/debug/taskrunner 5 10000 10000
  99811 randell-cr   20   0 99.8G  363M  2772 R  104.2  1.1  1h22:04 target/debug/taskrunner 5 10000 10000
  99812 randell-cr   20   0 99.8G  363M  2772 R  104.2  1.1  1h22:18 target/debug/taskrunner 5 10000 10000
  99813 randell-cr   20   0 99.8G  363M  2772 R  104.2  1.1  1h22:10 target/debug/taskrunner 5 10000 10000
  99815 randell-cr   20   0 99.8G  363M  2772 R  101.2  1.1  1h22:20 target/debug/taskrunner 5 10000 10000
  99816 randell-cr   20   0 99.8G  363M  2772 R  101.2  1.1  1h22:13 target/debug/taskrunner 5 10000 10000
  99798 randell-cr   20   0 99.8G  363M  2772 R  101.2  1.1  1h22:18 target/debug/taskrunner 5 10000 10000
  99803 randell-cr   20   0 99.8G  363M  2772 R  101.2  1.1  1h22:05 target/debug/taskrunner 5 10000 10000
  99810 randell-cr   20   0 99.8G  363M  2772 R  104.2  1.1  1h22:12 target/debug/taskrunner 5 10000 10000
  99819 randell-cr   20   0 99.8G  363M  2772 R  104.2  1.1  1h22:04 target/debug/taskrunner 5 10000 10000
  99807 randell-cr   20   0 99.8G  363M  2772 R  104.2  1.1  1h22:19 target/debug/taskrunner 5 10000 10000
  99808 randell-cr   20   0 99.8G  363M  2772 R  104.2  1.1  1h22:14 target/debug/taskrunner 5 10000 10000
```

Do note that virtual memory shown here is 100GB in total because each thread stack size has been set to 4GB. So that's 4 * 24 = 96GB statically allocated to the program minimally. It could definitely be less but the default thread stack size of 2MB is insufficient, so better overallocate for initial testing.