CS3211 Assignment 3 Report

Hubertus Adhy Pratama Setiawan - A0200816R
Simon Julian Lauw - A0196678A

Overview

The task runner is implemented using a threadpool + asynchronous channel. The implementation assumes that despite the large number of tasks being spawned, the output, results, and the task counters all can fit in 64 bits unsigned integer. Moreover, each task is independent of one another.

Concurrency Paradigm

The main concurrency paradigm being used in threadpool. A thread pool is a collection of pre-created threads that can be used to execute multiple tasks concurrently. In this case, we are using the implementation from threadpool crate and initialise the number of threads to be equal to the number of cpu(s) in the system. Such information can be found using the function from num_cpus crate.

In our implementation, we use asynchronous channels given by sync::mpsc module in the standard library crate to synchronise the main thread with the worker threads. The main thread is responsible to distribute the tasks to the worker threads through the channel, and each worker thread would be responsible to execute the task and send the result back to the main thread through the channel.

Implementation Details

The following algorithm is used:

1. Initialise a vector deque containing all the initial tasks
2. Initialise a threadpool, which internally contains task queues, with num_workers = num_thread available in the machine
3. Initialise the output = 0
4. Initialise number of active task = 0
5. For each task in the initial tasks:
    a. Submit the task to the threadpool, let each worker thread to execute the task which will return the output of current task + the children of the current task to the main thread using the channel
    b. Increment the number of active task by 1
6. While there are active tasks:
    a. Do blocking receive and wait for the worker thread to finish
    b. XOR the result with the output variable
    c. for each child of the children of the task being executed:
        i. submit each child to to the threadpool
        ii. Increment the number of active task by 1
    d. Decrement the number of active task by 1

Based on this algorithm, every time there is a task, submit the task to the task queue of the threadpool and let the worker threads grab the task from the task queue. Upon executing the task, the new tasks are being generated using the default generate_set method in task.rs, where each new task will get submitted to the task queue in the threadpool again until there are no more tasks available.

Parallelism

The implementation can run in parallel, as each worker thread in the threadpool executes tasks independently in the background. As long as there are tasks available in the task queue, the worker thread will just grab the task and execute it. Moreover, as each task is independent from one another, once the task is available in the queue, the worker thread can grab any task in arbitrary order.

Assuming the task queue in the thread pool is a FIFO queue, then our implementation is actually similar to running BFS where each node may get executed concurrently.

Running the task runner on different machines proves this claim:

1. Running the task runner with num_threads = 1 (on soctf-pdc-012 machine) takes ~50 seconds.
2. Running the task runner with num_threads = num_cpus = 8 (on soctf-pdc-012 machine) takes ~13 seconds.
3. Running the task runner with num_threads = num_cpus = 20 (on soctf-pdc-001 machine) takes ~8 seconds.

When run with more CPUs, the task throughput increases and we achieve higher speed-up. This implies the tasks are run in parallel even though we recognize that the actual speed-up is not linear because there are various overheads like managing the threads, sending and receiving the tasks from main thread to worker thread, etc.

Other Implementations

The following implementations were also tried:

1. Threadpool + async channel with non-blocking recv()
   Similar to the submitted code, but the recv() call is replaced with its non-blocking version (try_recv()). In addition, the main thread waits for all threads to finish before scheduling the newly generated tasks instead of sending them to the threadpool right away.
2. Threadpool + mutex
   Each thread will update the output variable and task queue directly instead of sending the result back to the master and let the master update the output and push the new tasks to the queue. This is done by passing an Arc (atomically reference counter) object containing mutex of the task queue to the worker thread. After all the threads exit (join()-ed by the main thread), the main thread is responsible to distribute the task to the worker threads again.
3. Asynchronous programming using tokio
   Similar to the submitted code, but we spawn a new tokio (green) thread for each task and then we await until all the handles are done. This method is arguably less efficient because Tokio is designed for I/O concurrency and there is no guarantee that the Tokio runtime could utilise all the cores within the machines.

We profiled all implementations and visualised the result using flamegraph. We found that the performance of threadpool + non-blocking receive is significantly slower than the other implementations. Looking at the flamegraph, we can see that the main thread sometimes polls an empty channel, wasting CPU cycles. Meanwhile, the performance of the other implementations are quite similar, even when compared against our main implementation (threadpool + channel). We find this quite interesting, as we expected these implementations to be slower because the tasks are sent in batches instead of scheduled right away when available. The flame graphs can be found in our repository, at /flamegraph.

Bonus Concept

For the bonus, we try to modify the task.rs and to support running the task runner with height = 10000 and max_children = 10000 under 4 GB memory constraint. To do so, we are trying to redefine the notion of task.

Let each task object be a finite stream of execution units. Therefore, executing each task actually means getting the current data in the stream, and it would return a new task stream with 1 less execution unit, together with a new task stream which is the child task if any. Pictorially, it looks like the following figure:

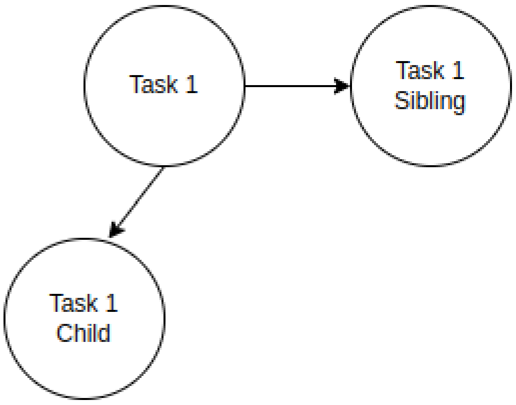

Figure 1. New Task Stream Execution

This notion of task is actually inspired by lazy evaluation of task generation. Moreover, redefining the task to be a stream theoretically allows the task runner to perform with higher speed-up since the scope of work to be done per task unit is less than the original task, making the task size more homogeneous and hopefully no thread will be wasted idling.

Moreover, to use the memory more efficiently, we try as much as possible for the task to be scheduled in DFS manner. Pictorially, it looks like the following figure:

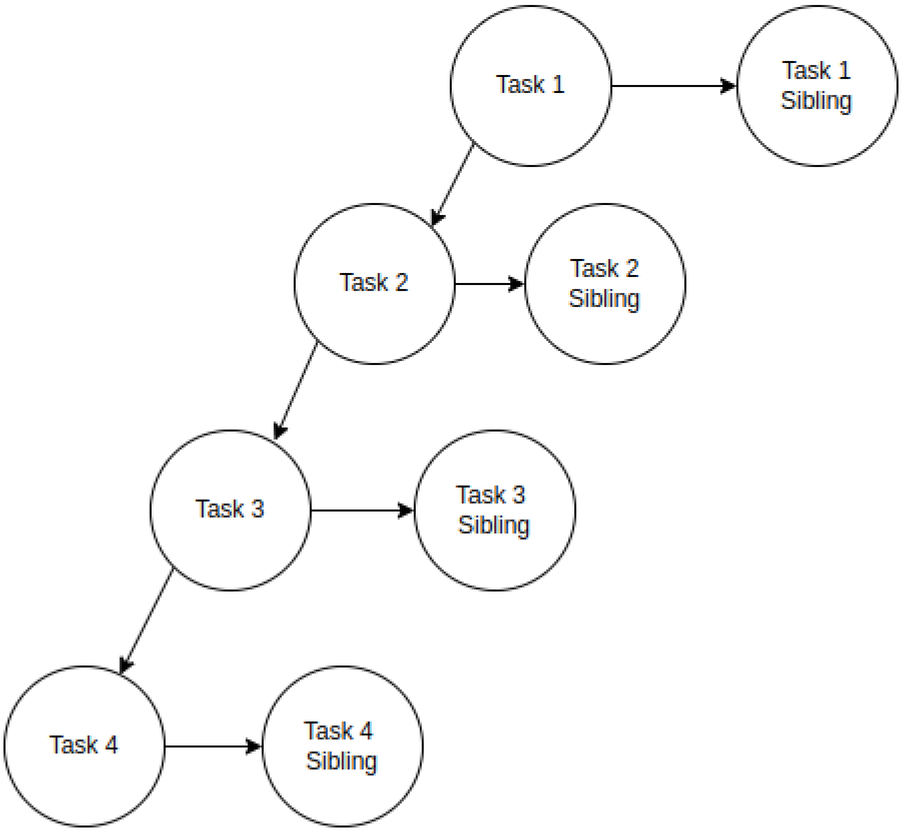

Figure 2. Task Scheduling Tree

By doing it in a manner similar to DFS, we recursively explore a subtree depth-first, and therefore creating around ~2 * 10000 tasks for height = 10000 and max_children = 10000. In total, we will have 20000 active tasks, with each task being 7 * 8 + 32 = 88 bytes in size. Therefore, in the serial execution, the total memory required is only ~ 20000 * 88 = 1.678 MB. However in practice, of course, the memory usage is much higher given that we are using threadpool to allow the task runner to perform at a higher level of concurrency, and the actual task execution tree doesn't follow DFS strictly since multiple nodes are explored at once.

Bonus Implementation

Similar to our main submission, we decided to use ThreadPool with asynchronous channel to execute the tasks. The initial implementation would just let the main thread distribute the task to the worker threads and collect the child and sibling tasks. Upon collecting the task, we put the sibling task onto the stack, and then the child task. Therefore, whenever there are idle threads, it would try to execute the child as much as possible before the sibling. This implementation can be found in /other_implementations/bonus/threadpool_dfs.rs.

However in the end, based on the flamegraphs, we realize that using try_recv is not performant and we decide to employ similar methods to the initial submission. Whenever the main thread gets the child and sibling tasks, it would immediately puts the child and then the sibling task into the threadpool task queue. Hopefully, with this method, the child will get executed first and the algorithm doesn't create the tasks exponentially.

Empirically, with height = 10000 and max_children = 10000, the memory usage plateaus at around 800 MB after ~2 hours of running it. Even though we believe that it would take forever for the task runner to finish :)