Sam Horradarn (SUNet: sirah)

Write up

Part 1

For both part A and B, all of the task system implementation uses thread pool to manage threads, except in `TaskSystemSerial` and `TaskSystemParallelSpawn` to reduce the overhead from thread creations at execution time. All of the implementation also uses dynamic assignment where each worker thread takes the first available task from a shared task queue, removes the task from queue so no other worker thread repeats the same task and begins executing the task. Once the execution is done, there is a separate queue to report the finished task execution back to main thread, so that the main thread knows when all tasks in the work queue have been finished and can return to the caller.

In part A, the bookkeeping is straightforward as there is only one `IRunnable*` task to execute at a given time. Instead of using sophisticated work queue, every thread shares a variable `current_task` that keeps track of the latest available task to execute, and another variable `task_done` to keep track of number of tasks that have finished. Each thread holds two separate mutexes on the shared variables: one for `current_task` and the other for `task_done`. The thread only locks the mutexes when it needs to increment the variables and unlocks them once the increment is done. This is so that the actual task execution can be done concurrently between threads.

In part B, the bookkeeping gets more complicated as the task system needs to track task dependencies. The execution steps are as follows:

1. The dependencies are tracked in a map, called `tasks_dep` with `TaskID` as key and `set<TaskID>` as value if the key `TaskID` has dependencies on other `TaskID` in the sets. This map is constructed during `runAsyncWithDeps()` as tasks are added to the system.
2. Upon execution of `sync()`, we can identify tasks that are ready to execute by scanning the map and adding it to a separate queue of ready tasks (called `ready_tasks`). `TaskID` that are ready to execute has an empty `set<TaskID>` associated with its key according to our design of the map. This `TaskID` key is also removed from the map.
3. Once the tasks from `ready_tasks` finish executing, they are added to another queue (called `done_tasks`) that is shared with the main thread.
4. The main thread will iterate through `tasks_dep` entries and remove `TaskID` that are present in `ready_tasks`, which is conceptually the same as removing the task dependencies from another task once we know that this `TaskID` is finished.
5. Repeat 2-4 until `tasks_dep` is empty and there are no more tasks waiting in `done_tasks`.

At the end of `run()` (for part A) and `sync()` (for part B), the thread pool is shut down by flipping the flag `done_work` that signals thread worker to stop spinning in the loop and waiting for more tasks. Then the threads are joined and finishes the teardown.

Part 2

The sequential task system performs best when there are small number of tasks and each task can be completed very quickly. This is because the overhead from thread creation and synchronization outweighs the benefits gained from concurrent execution. For example, in `super_super_light` test where there are relatively smaller tasks to execute compared to other tests like `ping_pong_*`, serial implementation finishes in 9.051ms, close to 8.805ms for the sleeping thread pool implementation and faster than 34.069ms of spawn-every-launch implementation.

The spawn-every-launch implementation will perform as well as the more advanced implementations when each task takes longer to compute or when there are large number of tasks. As each task gets more expensive to compute or the task increases in number, the overhead from thread creation will appear smaller when compared to the total execution time. For example, in `ping_pong_equal` where there are ~500k tasks and each task loops for 32 iterations, spawn-every-launch implementation takes 293.946ms compared to 271.48ms for the sleeping thread pool implementation. On the other hand, spawn-every-launch implementation will perform worse when there are small number of tasks or when each task is cheap to compute. This is visible in `super_super_light` test where spawn-every-launch implementation takes 37.286ms to complete compared to 9.322ms for the most advanced implementation.