Written assignment 2

Xavier Gonzalez: xavier18

JS Paul: jspaul

Question 1

The TaskSystem is a way to use the parallel resources to schedule and complete bulk task launches, including those launched asynchronously and with dependencies. In fact, in the implementation in part B for the synchronous run is simply the asynchronous call with no dependencies.

Each bulk task launch consists of some task of type `IRunnable`; a total number of task; and a set of dependencies (which may be empty). In order to run, we need to check whether its dependencies have been satisfied or not. We do this under a lock because we will be adding the bulk task launch to one of two (global) queues depending on whether its dependencies are satisfied or not (see discussion about tracking dependencies).

Once a task is ready to be run, it is then worked on my a thread in the thread pool, which is created once during construction of the TaskSystem (the size of the thread pool is the only required input). The threads are sleeping to avoid overhead but are woken up via condition variables when new work arrives, and they do the work. Finally, when work is done, they notify the main thread via a condition variable that all work is done.

Thread management

As suggested by part A, step 2, the implementation uses a persistent thread pool created once during construction. Worker threads are spawned in the constructor and continuously execute a `spinning()` function (other than for part 1 of part a), sleeping when no work is available and waking via condition variables when new tasks arrive (for part a part 3 and for part b).

We also need a mechanism do know when all the work is done.

In part A, After the thread have done their assigned work (see next section for discussion of task assignment), we increment a shared counter named `jobs_complete` using an atomic (to avoid race conditions). In this way, each worker thread can accurately tell if all the tasks have been completed, notifying the main thread via a condition variable once all tasks are done.

In part B, the condition variable is triggered when the queues holding the ready and waiting tasks are empty (see discussion of tracking dependencies), meaning all tasks are done

When threads are meant to be destroyed, a kill flag is triggered which wakes any sleeping threads and causes them to exit their spinning loops and terminate.

Dynamic Assignment

We use semi/dynamic assignment of jobs to threads, which lets us be flexible in balancing the load in all parts but part a part 1 (where jobs are taken 1 at a time by each thread).

However for the other parts, we use a batch size, as similar to the ideas presented in lecture. The gist is that it's not worth the overhead of waking up a thread just to do a single tiny task. On the other hand, if we had a thread to a large batch of tasks, the risk is that a pulse of long tasks could cause serious load imbalance. So, we pick a batch size as a compromise.

We use a batch size of std::max(1, num_total_tasks/num_threads/3) in part A. This batch size aims to split up the work among threads with plenty of batches to pull from to help mitigate load imbalance, while still keeping the overhead of waking up threads low.

In part B, we use a splicing system. We have a ready queue of batches of work for each task. When a task is moved to the ready queue, it is split up in num_threads batches, where each batch specifies a starting index of a job and the thread which processes it will process each job with the index `start_index + k * num_threads` for integer k until the job index surpasses the number of jobs in that task. This splicing system allows for load balancing as threads can pick new batches of work from the ready queue, which are generally equivalent in size for the same task. This helps mitigate load imbalance that could occur if some threads pick up long tasks while others pick up short tasks.

Tracking dependencies

To track dependencies in part B, there are a number of data structures set up.

One is a vector of ints called `num_jobs_left`. This vector is set up so that if we index into `num_jobs_left` with the appropriate `task_id`, we can see how many jobs are left before the task is complete. This value is decremented when jobs are finished, and serves as an indicator of when tasks that depend on this task can be started (i.e. when `num_jobs_left[task_id]` reaches 0). This is ok to do because we assign task_ids sequentially ourselves, using an atomic to ensure no repeats.

This vector is useful for deciding what to do when we run a bulk task with `runAsyncWithDeps`.

- If the `deps` of this bulk task launch are not met (as indicated by looking at the appropriate entries of `num_jobs_left`), then we add an appropriate instance of the `WaitingTask` struct to the `waiting_tasks` queue. This `WaitingTask` tracks the dependencies of this bulk task launch, and when the dependencies are met, we can launch the bulk task.
- If the `deps` of this bulk task launch are met, then we add multiple appropriate instances of the `TaskBatch` struct to the `ready_queue`. This `TaskBatch` represents part of a task that is ready to be executed, already batched and splicing specified, with all task dependencies met.

When a thread is awake, it first checks the `ready_queue` for any `TaskBatch`s or whether the `waiting_queue` is empty or if there is an active kill flag. If none of these conditions are met, the thread sleeps. A kill flag results in the termination of the thread spinning. If a ready task is not available, the thread simply spins with the anticipation that soon tasks will be complete to enable processing of waiting tasks. If there are tasks in the `ready_queue`, the thread begins processing some of the jobs in this task, in the splicing manner described previously.

Question 2

There are four different implementations in part A:

- serial
- parallel and always spawn

- parallel and use a thread pool and spin
- parallel and thread pool and sleep.

There are certain test cases where serial performs beter than any parallel implementation. This is typical in light tasks with minimal work required because there is an overhead to spawning threads.

For example, in `super super lightweight`, on myth, serials runs in around 9 ms, while the best (most sophisticated) parallel implementaion is around 10 ms.

In contrast, on myth, `fibonacci` takes around 1 second for serial, and only around 300-500 ms for the parallel implementations.

Let's compare and contrast these two tasks as way of understanding the benefits of serial implementation vs spawning threads.

`fibonacci` is implemented recursively and involves computing the 25th element of the fibonacci sequence. As a result, there is a lot of arithmetic to do in either implementation. Moreover, if we spawn threads and do them in parallel, each thread still has a lot of arithmetic to do, making the overhead of spawning threads worth it. That's why on this test, parallel is better than sequential.

`super super lightweight`, on the other hand, is a task that does no arithmetic at all. It only involves copying data from one array to another. Each operation is so lightweight that the overhead of spawning a thread is often not worth it. However, some amount of tuning matters. For example, on the myth machine, if we spawn with 3 threads, we can actualy get parallel+sleeping to be slightly faster than serial. So there is some balance: doing all of the operation in parallel is less costly than spawning one thread. But, already by the time we are spawning 4 threads, the overhead of spawning is too much and serial is faster.

`always spawn` is roughly comparable on `ping_pong` (equal and unequal), `recursive_fibonacci`, `math_operations_in_tight_for_loop_reduction_tree`, and `mandelbrot_chunked`. On the other tasks, thread pooling does better.

A great case study is look at `super_light` vs `ping_pong_equal`. Both of these tasks are identical, except that `ping_pong_equal` has 2^19 elements in the arrays, while `super_light` has 2^15. In `super_light`, the thread pool implementations are much better than always spawn. But in `ping_pong_equal`, they are roughly comparable. This is because in `super_light`, each run of the thread is not so heavy, so the overhead of spawning is more significant. In contrast, in `ping_pong_equal`, each thread has a lot more work to do, so the overhead of spawning is less significant. We see this theme of very heavy compute threads not showing so much of a benefit from thread pooling in the other tests as well, as `recursive_fibonacci`,`math_operations_in_tight_for_loop_reduction_tree`, and `mandelbrot_chunked` all involve heavy computation for each task.

Question 3

part a

A test we implemented for part a was `matMulTest`, which multiplied 5 2x2 matrices of the form {{2,2}, {0,2}}.

The test is meant to check if our implementations can still be correct in the context of matrix multplication, which involves mixing multiple linput variables. Matrix multiplication is also a very important primitive in

machine learning.

I was able to verify correctnes of the solution by checking the output of the test against a hand-calculated result. The result of the test did not cause me to change the implementation, but it did give me more confidence that the implementation was correct.

It is interesting that the parallel implementations were slower than serial on this test. This is likely because the number and size of the matrices multiplied together were small.

part b

For part b, to check correctness, we implemented `simpleAccumulateTest`. It was inspired by the structure of the sleep test, but the sleep test just printed out numbers for a visual inspection. I wanted to see if we could update a pointer asynchronously and get the correct result.

To do so, I used three tasks (A -> B -> C), where each task raised the current value of a pointer to a power (i.e., this operator is non-associative, unlike addition or multiplication, and so would raise an error if done out of order). Thus, the async implementation needs to be done correctly to give the right result. This test confirms the correctness of the async implementation.