STANFORD CS149- ASSIGNMENT 2

PART A)

Step 1: Move to a Parallel Task System
Solution:

In tasksys.h file we introduced a new variable num_threads in the `TaskSystemParallelSpawn` class ,to keep count of the threads available.

```
//ADDING A NEW VARIABLE
private:
    int num_threads;
```

Then we implemented the `TaskSystemParallelSpawn::run` function as per the requirement. This function first decides the number of threads to be used. If the number of tasks is less than the number of threads available, then we use threads equal to the number of tasks, otherwise we utilize all the available threads. Furthermore, we implemented thread synchronization using join() making sure that all tasks finish before returning.

- **How will you assign tasks to your worker threads? Should you consider static or dynamic assignment of tasks to threads?**
  For this, we determined the number of threads used dynamically based on the total number of tasks. Each thread executes tasks in a round-robin manner, ensuring even workload distribution among threads. This round robin technique improves the performance as compared to the simple linear thread allocation.

  Thread i will execute tasks : i, i + num_threads_to_use, i + 2*num_threads_to_use and so on
  For example, if num_total_tasks = 10 and num_threads = 3, the tasks will be distributed as:
  1. Thread 0 -> Task 0, Task 3, Task 6, Task 9
  2. Thread 1 -> Task 1, Task 4, Task 7
  3. Thread 2 -> Task 2, Task 5, Task 8

  We are using static task assignment as it reduces task scheduling overhead (compared to dynamic assignment). Each thread knows exactly which tasks it needs to execute which means low overhead and no need for locks and mutex.

- **Are there shared variables that need protection from simultaneous access?**
  In our implementation, each thread works independently, so there are no shared variables that require synchronization. Each thread gets its own set of tasks and executes them without modifying any shared state. Because we do not use a shared task queue, there's no need for synchronization.

```
void TaskSystemParallelSpawn::run(IRunnable* runnable, int num_total_tasks)
{
    //
    // TODO: CS149 students will modify the implementation of this
    // method in Part A.  The implementation provided below runs all
    // tasks sequentially on the calling thread.
    //

    int num_threads_to_use = std::min(num_threads, num_total_tasks);    //how many threads to use

    std::vector<std::thread> workers;    //worker threads

    for (int i = 0; i < num_threads_to_use; ++i)
    {
        workers.emplace_back([=]()
        {
            for (int task_id = i; task_id < num_total_tasks; task_id += num_threads_to_use)
            {
                runnable->runTask(task_id, num_total_tasks);
            }
        });
    }

    for (auto& worker : workers)
    {
        worker.join();
    }

}
```

This was the output after implementing this function:

```
laiba@DESKTOP-A2G0RVA:~/asst2/part_a$ ./runtasks -n 4 simple_test_sync
================================================================================
Test name: simple_test_sync
================================================================================
[Serial]:                [0.000] ms
[Parallel + Always Spawn]:               [0.462] ms
[Parallel + Thread Pool + Spin]:                 [0.000] ms
[Parallel + Thread Pool + Sleep]:                [0.000] ms
================================================================================
laiba@DESKTOP-A2G0RVA:~/asst2/part_a$ ./runtasks -n 4 mandelbrot_chunked
================================================================================
Test name: mandelbrot_chunked
================================================================================
[Serial]:                [366.623] ms
[Parallel + Always Spawn]:               [101.667] ms
[Parallel + Thread Pool + Spin]:                 [363.947] ms
[Parallel + Thread Pool + Sleep]:                [363.921] ms
```

Step 2: Avoid Frequent Thread Creation Using a Thread Pool

Solution:

We added some new variables in `TaskSystemParallelThreadPoolSpinning` class:

```
private:
    int num_threads;
    std::vector<std::thread> workers;
    std::queue<int> task_queue;
    std::mutex queue_mutex;
    std::atomic<int> tasks_remaining;
    IRunnable* runnable;
    int num_total_tasks;
    std::atomic<bool> stop;
```

In this step, we optimized task execution by implementing a thread pool in `TaskSystemParallelThreadPoolSpinning` constructor. Unlike `TaskSystemParallelSpawn`, which creates and destroys threads for each run() call, this implementation creates worker threads once and keeps them running (spinning) in a loop, continuously checking for new tasks. Tasks are added to the queue in the function: run(), and worker threads execute them.

Each worker thread checks the shared task queue for available tasks and executes a task if one is available. Then it decrements a counter (tasks_remaining) to track progress and continues the loop until stopped.

- **How might a worker thread determine there is work to do?**

Each worker thread spins in a while loop, repeatedly checking the task queue. It uses a mutex (queue_mutex) to safely access the queue.

1. If the queue is not empty, it removes a task and executes it.
2. If the queue is empty, it keeps looping (spinning) until a task arrives.

- **It is now non-trivial to ensure that run() implements the required synchronous behavior. How do you need to change the implementation of run() to determine that all tasks in the bulk task launch have completed?**
  The run() function ensures all tasks finish by:
  1. Pushing all tasks into the queue so workers can pick them up when needed.
  2. Tracking progress using tasks_remaining.
  3. Busy-waiting (while (tasks_remaining > 0);) until all tasks are completed.

```cpp
TaskSystemParallelThreadPoolSpinning::TaskSystemParallelThreadPoolSpinning(int num_threads)
    : ITaskSystem(num_threads), num_threads(num_threads), tasks_remaining(0), stop(false)
{
    // TODO: CS149 student implementations may decide to perform setup
    // operations (such as thread pool construction) here.

    for (int i = 0; i < num_threads; ++i)
    {
        workers.emplace_back([this]()
        {
            while (!stop)
            {
                int task_id = -1;
                // Fetch a task if available
                {
                    std::lock_guard<std::mutex> lock(queue_mutex);
                    if (!task_queue.empty())
                    {
                        task_id = task_queue.front();
                        task_queue.pop();
                    }
                }
                // Execute the task
                if (task_id != -1)
                {
                    runnable->runTask(task_id, num_total_tasks);
                    tasks_remaining--;
                }
            }
        });
    }
}
```

This was the output after implementing this function:

```
laiba@DESKTOP-A2G0RVA:~/asst2/part_a$ ./runtasks -n 4 simple_test_sync
================================================================================
Test name: simple_test_sync
================================================================================
[Serial]:                [0.000] ms
[Parallel + Always Spawn]:               [0.502] ms
[Parallel + Thread Pool + Spin]:                 [0.003] ms
[Parallel + Thread Pool + Sleep]:                [0.000] ms
================================================================================
laiba@DESKTOP-A2G0RVA:~/asst2/part_a$ ./runtasks -n 4 mandelbrot_chunked
================================================================================
Test name: mandelbrot_chunked
================================================================================
[Serial]:                [598.194] ms
[Parallel + Always Spawn]:               [200.260] ms
[Parallel + Thread Pool + Spin]:                 [224.917] ms
[Parallel + Thread Pool + Sleep]:                [609.404] ms
================================================================================
```

Step 3: Supporting Execution of Task Graphs

Solution:

For this part we added the following variables in `TaskSystemParallelThreadPoolSpinning` Class

```
private:
    int num_threads;
    std::vector<std::thread> workers;
    std::queue<std::function<void()>> task_queue;
    std::mutex queue_mutex;
    std::condition_variable condition;
    std::condition_variable main_thread_condition;
    std::atomic<int> tasks_remaining;
    std::atomic<bool> stop;
    IRunnable* runnable;
    int num_total_tasks;
```

We improved the spinning approach by making the worker threads sleep when no tasks are to be done, preventing CPU wastage. We used a condition variable (condition.wait()) that puts these threads to sleep. Main Thread Sleeps while waiting Instead of busy-waiting (which occupies CPU cycles).

The main thread waits on main_thread_condition and only wakes up when tasks_remaining == 0. Condition variables wake up threads using condition.notify_all() only when new tasks arrive. Similarly, main_thread_condition.notify_one() wakes up the main thread only when all tasks are done.

```
TaskSystemParallelThreadPoolSleeping::TaskSystemParallelThreadPoolSleeping(int num_th
: ITaskSystem(num_threads), num_threads(num_threads), stop(false), tasks_remaining(0)
{
    // TODO: CS149 student implementations may decide to perform setup
    // operations (such as thread pool construction) here.

    for (int i = 0; i < num_threads; ++i)
    {
        workers.emplace_back([this]()
        {
            while (true)
            {
                std::function<void()> task;
                {
                    std::unique_lock<std::mutex> lock(queue_mutex);
                    condition.wait(lock, [this]() { return stop || !task_queue.empty(

                    if (stop && task_queue.empty()) return;

                    task = std::move(task_queue.front());
                    task_queue.pop();
                }
                task();
                // Notify main thread if all tasks are done
                {
                    std::lock_guard<std::mutex> lock(queue_mutex);
                    tasks_remaining--;
                    if (tasks_remaining == 0)
                    {
                        main_thread_condition.notify_one();
                    }
                }
            }
```

After the implementation of all the above steps, i ran the ping pong test and got this output which clearly shows great improvement in speedup with each subsequent method

```
laiba@DESKTOP-A2G0RVA:~/asst2/part_a$ ./runtasks -n 16 ping_pong_equal
================================================================================
Test name: ping_pong_equal
================================================================================
[Serial]:                 [1146.531] ms
[Parallel + Always Spawn]:                [972.385] ms
[Parallel + Thread Pool + Spin]:                  [568.717] ms
[Parallel + Thread Pool + Sleep]:                 [548.478] ms
================================================================================
```

PART B)

For this part we added the following variables in `TaskSystemParallelThreadPoolSpinning` Class

```
//ADDING NEW VARIABLES HERE
private:

    void workerThread();

    int num_threads;
    std::vector<std::thread> workers;
    std::queue<std::function<void()>> taskQueue;     //tasks that are ready to be executed
    std::mutex queueMutex;
    std::condition_variable taskAvailable;           //to notify tasks availability
    std::atomic<bool> stop{false};                   //for stopping the execution

    // for dependencies
    std::unordered_map<int, int> taskDependencies;  //no of dependencies for each task
    std::unordered_map<int, std::vector<std::function<void()>>> dependentTasks;
    int nextTaskID = 0;
    std::mutex depMutex;
    std::condition_variable allTasksCompleted;
    int activeTasks = 0;
```

In this part we had to extend the `TaskSystemParallelThreadPoolSleeping` implementation done in part a to include asynchronous task execution involving task dependencies.

We created a pool of worker threads that continuously look for any task to be done.
If a task has no dependencies, it is immediately enqueued for execution and if dependencies exist, then the task is stored in a dependency tracker until all required tasks are completed.
Once the worker thread finishes a task it notifies the dependent task of its completion so that the dependent task can start with its execution.

We have implemented the sync function also to make sure that all tasks (including those with dependencies) finish execution before returning. We use condition variables for synchronization.

```
void TaskSystemParallelThreadPoolSleeping::run(IRunnable* runnable, int
num_total_tasks)
{

    runAsyncWithDeps(runnable, num_total_tasks, {});
    sync();

}
```

The is the output we got after this implementation which shows a significant speedup with this new approach

```
laiba@DESKTOP-A2G0RVA:~/asst2/part_b$ ./runtasks -n 4 ping_pong_equal
================================================================================
Test name: ping_pong_equal
================================================================================
[Serial]:                [1157.345] ms
[Parallel + Always Spawn]:               [1186.068] ms
[Parallel + Thread Pool + Spin]:                 [1130.919] ms
[Parallel + Thread Pool + Sleep]:                [698.028] ms
================================================================================
```