asst2 WriteUp

Part A

1 TaskSystemParallelSpawn

将系统简单的展开为并行的方式，使用多线程解决。

头文件中添加如下变量：

```C++
            std::thread* threads_pool;
            int num_threads;
            void threadRun(IRunnable* runnable, int num_total_tasks, std::mutex
    * m, int* curr_task);
```

利用mutex和共享变量来保证互斥访问

C++ | 复制代码

```cpp
TaskSystemParallelSpawn::TaskSystemParallelSpawn(int num_threads): ITaskSystem(num_threads) {
    this->num_threads = num_threads;
    this->threads_pool = new std::thread[num_threads];
}

TaskSystemParallelSpawn::~TaskSystemParallelSpawn() {
    delete [] threads_pool;
}

void TaskSystemParallelSpawn::threadRun(IRunnable* runnable, int num_total_tasks, std::mutex* m, int* curr_task){
    int curr_run_task = -1;
    while(curr_run_task < num_total_tasks){
        m->lock();
        curr_run_task = *curr_task;
        *curr_task = *curr_task + 1;
        m->unlock();
        if(curr_run_task >= num_total_tasks){
            break;
        }
        runnable->runTask(curr_run_task, num_total_tasks);
    }
}

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

    std::mutex* m = new std::mutex;
    int* curr_task = new int;
    *curr_task = 0;

    for(int i = 0; i < num_threads; i++){
        threads_pool[i]= std::thread(&TaskSystemParallelSpawn::threadRun, this, runnable, num_total_tasks, m, curr_task);
    }
    for(int i = 0; i < num_threads; i++){
        threads_pool[i].join();
    }
    delete m;
    delete curr_task;
}
```

2 TaskSystemParallelThreadPoolSpinning

前面那种构造方式在每次调用run的时候会产生创建线程的开销，当计算任务很低时这种开销非常明显，因此可以使用线程池的方法，在构建的时候就创建线程池。

工作线程不停的spinning来确定是否有更多任务来完成。

头文件中添加如下变量：

C++ | 复制代码

```
        std::vector<std::thread> threads_pool_;
        std::atomic<int> task_remained_;
        std::queue<int> task_queue;
        std::mutex mutex_;

        IRunnable* my_runnable_;

        int num_total_tasks_;
        bool killed_;
        void spinningThread();
```

threads_pool_中的线程需要从task_queue中获取对应的工作任务并执行。

C++ | 复制代码

```cpp
TaskSystemParallelThreadPoolSpinning::TaskSystemParallelThreadPoolSpinning(int num_threads): ITaskSystem(num_threads) {
    killed_ = false;
    my_runnable_ = nullptr;
    num_total_tasks_ = 0;
    for(int i=0;i<num_threads;++i)
        threads_pool_.emplace_back(&TaskSystemParallelThreadPoolSpinning::spinningThread, this);

}

TaskSystemParallelThreadPoolSpinning::~TaskSystemParallelThreadPoolSpinning() {
    killed_ = true;
    for(auto& t: threads_pool_)
        t.join();
}

void TaskSystemParallelThreadPoolSpinning::spinningThread(){
    int task_id;
    while(!killed_){
        task_id = -1;
        mutex_.lock();
        if(!task_queue.empty()){
            task_id = task_queue.front();
            task_queue.pop();
        }
        mutex_.unlock();

        if(task_id != -1){
            my_runnable_->runTask(task_id, num_total_tasks_);
            task_remained_--;
        }


    }
}

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

    task_remained_ = num_total_tasks;
    my_runnable_ = runnable;
    num_total_tasks_ = num_total_tasks;

```

```
42      for(int i=0;i<num_total_tasks;++i){
43          mutex_.lock();
44          task_queue.push(i);
45          mutex_.unlock();
46      }
47      while(task_remained_);
48  }
```

3 TaskSystemParallelThreadPoolSleeping

2中的缺点是没有task时thread会一直spinning等待，消耗cpu的资源，需要实现一种方法让worker thread没有task时处于sleep状态，有task被唤醒执行task。同时主线程也可能一直spinning等待worker thread完成工作在返回，同时需要解决主线程spinning的问题。

头文件中添加如下变量：

C++ | 复制代码

```
1           std::vector<std::thread> threads_pool_;
2           std::queue<int> task_queue;
3
4           std::mutex queue_mutex_;
5           std::mutex count_mutex_;
6
7           std::condition_variable queue_cond_;
8           std::condition_variable count_cond_;
9
10          IRunnable* my_runnable_;
11          int num_total_tasks_;
12          bool killed_;
13          int task_remained_;
14
15          void sleepingThread();
```

queue_mutex_用于对队列操作时进行互斥，count_mutex_用于对剩余任务做互斥操作。

queue_cond_用于队列操作后发送通知给等待的sleep线程，count_cond_用于给sleep的主线程发送通知说明任务已经全部完成。

```cpp
TaskSystemParallelThreadPoolSleeping::TaskSystemParallelThreadPoolSleeping(int num_threads): ITaskSystem(num_threads) {
    killed_ = false;
    my_runnable_ = nullptr;
    num_total_tasks_ = 0;
    task_remained_ = -1;

    for(int i=0;i<num_threads;++i)
        threads_pool_.emplace_back(&TaskSystemParallelThreadPoolSleeping::sleepingThread, this);
}
```

```cpp
TaskSystemParallelThreadPoolSleeping::~TaskSystemParallelThreadPoolSleeping() {
    killed_ = true;
    my_runnable_ = nullptr;
    queue_cond_.notify_all();

    for(auto& t: threads_pool_)
        t.join();
}
```

这里的queue_cond_.notify_all()用来唤醒睡眠的工作线程，使其遇到if(killed_)退出。

worker thread

```cpp
void TaskSystemParallelThreadPoolSleeping::sleepingThread(){
    int id;
    while(true){
        while(true){
            std::unique_lock<std::mutex> lock2(queue_mutex_);
            queue_cond_.wait(lock2,[&]{return !task_queue.empty()||killed_
;;});

            if(killed_)
                return;
            if(task_queue.empty())
                continue;

            id = task_queue.front();
            task_queue.pop();
            break;
        }
        my_runnable_->runTask(id, num_total_tasks_);
        std::unique_lock<std::mutex> lock4(count_mutex_);
        task_remained_--;
        if(task_remained_ == 0){
            lock4.unlock();
            count_cond_.notify_one();
        }
    }
}
```

unique_lock作用域为他最近的{}，出了作用域自动释放，在此处锁住队列，查看其中是否有task，没有的话进行睡眠并释放锁，如果有任务就退出循环执行任务，当剩余任务为0利用count_cond_通知主线程结束，进入析构函数，从析构函数中再推出其他的工作线程。

此处wait后面的lambda表达式返回如果为true表示跳过wait语句，继续执行下一行，这里的意思是如果队列不为空或者已经被killed了，就继续向下，因为如果不为空就正常执行task，如果killed的话在下一条语句退出工作线程。

main thread

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

    my_runnable_ = runnable;
    num_total_tasks_ = num_total_tasks;
    killed_ = false;
    task_remained_ = num_total_tasks;

    for(int i=0;i<num_total_tasks_;++i){
        std::unique_lock<std::mutex> lock2(queue_mutex_);
        task_queue.push(i);
    }
    queue_cond_.notify_all();

    while(true){
        std::unique_lock<std::mutex> lock3(count_mutex_);
        count_cond_.wait(lock3,[&](){return task_remained_ == 0;});
        if(!task_remained_) return;
    }

}
```

主线程通过for循环将所有任务加入task_queue，然后利用queue_cond_通知全部的工作线程可以执行了，最后主线程进入sleep等待全部任务执行完成后进行唤醒。

count_cond_后的lambda表达式表示如果剩余为0的话直接进入下一个语句进行返回，因为没有剩余任务了主线程也没有必要继续等待，直接退出即可。

Part B

在part b中需要支持异步启动任务，任务之间会存在依赖关系，执行的顺序必须严格遵守这样的约束关系。

在头文件中添加两个class：

C++ | 复制代码

```cpp
class TaskGroup{
    public:
        TaskID id;
        IRunnable* runnable;
        int num_total_tasks;
        TaskGroup(TaskID id, IRunnable* runnable, int num_total_tasks){
            this->id = id;
            this->runnable = runnable;
            this->num_total_tasks = num_total_tasks;
        }
        TaskGroup(const TaskGroup &task){
            this->id = task.id;
            this->runnable = task.runnable;
            this->num_total_tasks = task.num_total_tasks;
        }
};

class RunnableTask{
    public:
        TaskID id;
        IRunnable* runnable;
        int current_task;
        int num_total_tasks;

        RunnableTask(TaskID id, int current_task, IRunnable* runnable,  int num_total_tasks){
            this->id = id;
            this->runnable = runnable;
            this->current_task = current_task;
            this->num_total_tasks = num_total_tasks;
        }

        RunnableTask(const RunnableTask &task){
            this->id = task.id;
            this->runnable = task.runnable;
            this->current_task = task.current_task;
            this->num_total_tasks = task.num_total_tasks;
        }

};
```

TaskGroup用来保存当前一组task，runAsyncWithDeps时创建。

RunnableTask用来执行单个任务（相当于某一组task中的一个task），RunnableTask放在后面的任务队列中。

TaskSystemParallelThreadPoolSleeping中添加如下变量和成员函数：

C++ | 复制代码

```cpp
        bool killed_;
        int num_threads_;
        int current_taskid;

        std::vector<std::thread> threads;

        std::map<TaskID, std::set<TaskID>> task_dep_map_;
        std::map<TaskID, TaskGroup*> task_group_map_;
        std::map<TaskID, int> task_remained_map_;

        std::deque<RunnableTask*> task_queue_;
        std::deque<TaskID> finished_task_queue_;

        std::mutex finished_task_mutex_;
        std::condition_variable finished_task_cond_;

        std::mutex task_queue_mutex_;
        std::condition_variable task_queue_cond_;

        void worker_thread();
        void scanForReadyTasks();
        void removeTaskIDFromDependency(TaskID id);
```

创建线程和销毁线程与前面类似不赘述。

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


    //
    // TODO: CS149 students will modify the implementation of this
    // method in Parts A and B.  The implementation provided below runs all
    // tasks sequentially on the calling thread.
    //
    runAsyncWithDeps(runnable, num_total_tasks, {});
    sync();
    return;
}
```

run可以看作没有dep的runAsyncWithDeps，然后立刻执行sync。

```cpp
TaskID TaskSystemParallelThreadPoolSleeping::runAsyncWithDeps(IRunnable* runnable, int num_total_tasks,
                                                              const std::vector<TaskID>& deps) {


    //
    // TODO: CS149 students will implement this method in Part B.
    //
    if(deps.size()==0)
        task_dep_map_[current_taskid];

    for(auto dep:deps){
        task_dep_map_[current_taskid].insert(dep);
    }
    task_group_map_[current_taskid] = new TaskGroup(current_taskid, runnable, num_total_tasks);
    return current_taskid++;
}
```

如果当前deps为空，声明一下该键值，如果不为空，则插入到current_taskid对应的dep映射中，并新建一个TaskGroup加入到task_group_map_队列，表示每次调用runAsyncWithDeps都会创建一个新的任务组，这里的current_taskid也可以理解为GroupID。

```cpp
void TaskSystemParallelThreadPoolSleeping::scanForReadyTasks(){
    for(auto it = task_dep_map_.begin();it!=task_dep_map_.end();){
        auto tasks = it->second;
        if(tasks.empty()){
            TaskGroup* t = task_group_map_[it->first];
            task_remained_map_[t->id] = t->num_total_tasks;
            task_queue_mutex_.lock();
            for(int i=0;i<t->num_total_tasks;++i){
                task_queue_.push_back(new RunnableTask(t->id, i, t->runnable, t->num_total_tasks));
            }
            task_queue_mutex_.unlock();
            task_queue_cond_.notify_all();
            it = task_dep_map_.erase(it);
        }
        else{
            ++it;
        }
    }
}
```

扫描每一个TaskID对应的依赖deps，如果没有依赖的task，则将该TaskGroup中全部的task加入到task_queue_中并notify等待的线程，然后调用erase抹除掉该依赖项。

```cpp
void TaskSystemParallelThreadPoolSleeping::removeTaskIDFromDependency(TaskID finished_task){
    for(auto it = task_dep_map_.begin();it!=task_dep_map_.end();++it){
        it->second.erase(finished_task);
    }
}
```

finished_task表示已经完成的一组TaskGroup，可以遍历task_dep_map_找到所有依赖于该TaskGroup的项并消除。

C++ | 复制代码

```cpp
void TaskSystemParallelThreadPoolSleeping::sync() {

    //
    // TODO: CS149 students will modify the implementation of this method in Part B.
    //
    scanForReadyTasks();
    bool done=false;
    while(!done){
        std::unique_lock<std::mutex> finished_lock(finished_task_mutex_);
        finished_task_cond_.wait(finished_lock,[&]{return !finished_task_queue_.empty();});

        bool has_more_finished_task = true;
        while(has_more_finished_task){
            auto task_done_id = finished_task_queue_.front();
            finished_task_queue_.pop_front();
            task_remained_map_.erase(task_remained_map_.find(task_done_id));
            finished_lock.unlock();

            removeTaskIDFromDependency(task_done_id);

            finished_lock.lock();
            has_more_finished_task = !finished_task_queue_.empty();
        }

        finished_lock.unlock();

        scanForReadyTasks();

        finished_task_mutex_.lock();
        done = task_dep_map_.empty() && task_remained_map_.empty();
        finished_task_mutex_.unlock();

    }
}
```

同步线程，初始调用scanForReadyTasks将依赖为空的group中的task加入task_queue，然后进入循环，每当工作线程完成一个TaskGroup就进行唤醒，erase掉已经完成的线程同时remove其对应的依赖项，继续scanForReadyTasks，直到task_dep_map_和task_remained_map_均为空为止。

```cpp
void TaskSystemParallelThreadPoolSleeping::worker_thread(){
    while(!killed_){

        while(true){
            std::unique_lock<std::mutex> task_queue_lock(task_queue_mutex_);
            task_queue_cond_.wait(task_queue_lock, [&]{return killed_ || !task_queue_.empty();});

            if(task_queue_.empty()){
                task_queue_lock.unlock();
                break;
            }

            auto task = task_queue_.front();
            task_queue_.pop_front();
            task_queue_lock.unlock();

            task->runnable->runTask(task->current_task, task->num_total_tasks);

            finished_task_mutex_.lock();
            task_remained_map_[task->id]--;
            if(task_remained_map_[task->id]<=0){
                finished_task_queue_.push_back(task->id);
                finished_task_mutex_.unlock();

                finished_task_cond_.notify_all();
            }
            else{
                finished_task_mutex_.unlock();
            }
        }
    }

}
```

工作线程，等待scanForReadyTasks或者析构函数发来的notify，然后取出对应的task进行执行，如果该taskid对应的剩余任务为0，说明该group已经执行完成，通知同步线程进行相应的处理。