Homework 9

In this homework we'll convert a sequential program to a program that executes items in a work queue in parallel.

What I've given you:

In the HW9CPPAssigned directory, in addition to this .pdf you will find a working C++ program. The program has a working quicksort (QuickSort.h and QuickSort.cpp), a main.cpp file, and Command.h file, which will not be used now but will be used in the parallel version.

In main.cpp, NUMSORTS sorts are created, one in each iteration of the loop. The sort is then performed, and the time to execute the sort is printed. There is also a worker( ) function that is commented out but will be used in the parallel version.

In QuickSort, there are two constructors. One creates and initializes an array of the desired number of elements and the other, the zero arg constructor, creates an array of zero elements. The zero arg constructor will be used in the parallel version. The sort( ) function sorts the array, and is basically the entry point to the quick sort routine from outside of the class. quickSort and partition do the sort and partitioning of the data for the sort.

What you need to do:

1. Have the QuickSort function inherit from the Command abstract class, and implement the pure virtual functions declared in Command.
2. Write a DotProduct class that also inherits from the Command abstract class and implements its pure virtual functions. DotProduct will have two constructors, a zero arg and a constructor that takes an int that is the length of the *a* and *b* arrays whose dot product will be taken. DotProduct also needs to implement operator<<(…). I'll include sample output that will make it clear what the output should be.
3. A WorkerQueue class that implements a work queue, where work items are shared_ptr<Command> objects. The queue should be implemented using a C++ std::vector.

   In addition to any needed constructors (I used a single zero-arg constructor) you will need to implement a *get()* function that returns a shared_ptr<Command> of some entry in the queue, and removes that entry from the queue. Removing the entry from the queue should be properly synchronized using a std::mutex and a std::lock_guard.

   You will also need to implement a *put()* function, that takes a std::shared_ptr<Command> as an argument and returns a void, that adds the std::shared_ptr<Command> to the work queue. The add of the shared pointer to the queue should be properly synchronized.

4. In the file main.cpp, the worker function should be uncommented and the write that is done with identify( ) should be called. You should hold a lock different than the lock used for WorkQueue when calling identify so that there is not a race on std::ostream.

   In the main function, we will time the time it takes to

   a. add eight std::shared_ptr<Command> objects to the work queue. Four will point to dot products and four will point to quick sorts.

    b. Start four threads that run the worker function, passing your work queue as an argument.
    c. Wait until all four threads have finished.
5. Print out the execution time, the number of threads, etc., as shown in the sample output.

When compiling, use the -pthread option at the end of your list of files or you will get linker errors. You will need the following include files: <vector> for std:vector, <chrono> for timing (you will also need to put "using namespace std:chrono" after the includes and any defines in your main.cpp file), <memory> for std::share_ptr, <mutex> for locks, plus the normal stuff.

What to turn in:

A directory <userid> containing your code. If "g++ *.cpp -pthread" will compile your program, or "g++ -std=c11 *.cpp -pthread" will compile your program, no make file is needed. Otherwise include a make file which will build and run your program when "make" is executed.

Points:

Compiles, runs and gives reasonable output. The value of your numbers may differ from my solution. 2 points

Properly synchronized put() using a std::lock_guard: 2 points

Properly synchronized get() using a std::lock_guard: 2 points

Properly synchronized worker(WorkQueue) function using a std::lock_guard: 2 points

Properly starting threads: 1 point

Properly joining threads: 1 point