CPSC 351 Project 2 (200 points)

Multi-threaded Merge Sort

Write a C++ program to run a multi-threaded merge sort algorithm using pthreads.

For the input data, you should generate random numbers within a limit. Let us say you generated n random numbers. Split your input of n numbers into p equal-sized (almost) segments. All but one segment will contain floor(n/p) numbers.

Let us say you generated n = 25 random numbers within a limit of 0 - 100:

77 89 18 47 43 60 32 28 42 52 1 52 83 65 55 6 39 65 62 12 33 40 19 53 46

Let p = 5.

The data will be split as:

Split # 1: 77 89 18 47 43
Split # 2: 60 32 28 42 52
Split # 3: 1 52 83 65 55
Split # 4: 6 39 65 62 12
Split # 5: 33 40 19 53 46

For the first part of the merge sort, your program should create p concurrent threads. Each of the p threads should sort their respective segment concurrently (in parallel). You don't have to write your own sorting algorithm here; in fact, you can (should) use the C++ function sort().

After the threads have sorted their respective segments, the data will look like:

Split # 1: 18 43 47 77 89
Split # 2: 28 32 42 52 60
Split # 3: 1 52 55 65 83
Split # 4: 6 12 39 62 65
Split # 5: 19 33 40 46 53

For the next part, the merge sort will take adjoining pairs of splits and merge them together, in parallel. This will be done in rounds, where in each round a number of merges take place in parallel until there is only a single, completely sorted list remaining.

In our example, the first round consists of two merges in parallel: (Split #1 and Split #2) and (Split #3 and Split #4).

Split # (1 + 2): 18 28 32 42 43 47 52 60 77 89
Split # (3 + 4): 1 6 12 39 52 55 62 65 65 83
Split # 5: 19 33 40 46 53

The second round consists of only one merge:

Split # (1 + 2 + 3 + 4): 1 6 12 18 28 32 39 42 43 47 52 52 55 60 62 65 65 77 83 89
Split # 5: 19 33 40 46 53

And the final round also only consists of one merge:

Split # (1 + 2 + 3 + 4 + 5): 1 6 12 18 19 28 32 33 39 40 42 43 46 47 52 52 53 55 60 62 65 65 77 83 89

Compile: clang++ -pthread multi_threaded_merge_sort.cpp -o test

Example Run: ./test 25 100 5

Enable multiple processors in the virtual machine, for example: 4.

Goal 1: From the command line, the program should read in the count of numbers to be sorted, n, upper limit on the range of numbers, u and the number of segments they will be divided in, p.

Goal 2: The main should generate n random numbers within a limit of 1 - u (inclusive). On different runs, a different set of random numbers should be generated.

Hint: Use functions rand and srand for this.

These random numbers should be stored in a global data structure: a dynamic array or an std::vector, called arr. This original array should be printed from main.

Goal 3: The input should be divided into p segments. **Do not make new arrays for these segments.** All but one segment will contain floor(n/p) numbers. **The main should create p concurrent threads, one thread per segment. Each thread should call the same thread function, with its respective argument/s. In this function, the thread should sort its corresponding segment, using C++ sort() function. You should have the threads operate on the same original array, arr, although each thread operates on its respective segment.** After each thread has sorted its respective segment, it should print from within its thread function “Sorted floor(n/p) (example) numbers”. And then in the next line, it should print the sorted segment of numbers.

You will encounter race conditions in this task, possibly at several places.

To solve it:

You should and are allowed to use mutex locks in thread function, such as pthread_mutex_lock and pthread_mutex_unlock. But most importantly you are allowed to use the mutex locks ONLY in the thread function. Care should be taken that using these mutex locks DOES NOT BREAK the condition of parallel sorting. In other words, the concurrent threads should be sorting their respective segments in parallel.

If you encounter race conditions in main, you are NOT allowed to use ANY pthread library synchronization primitives including but not limited to: semaphores, mutexes, mutex locks such as pthread_mutex_lock and pthread_mutex_unlock and conditional variables such as pthread_cond_wait and pthread_cond_signal etc. You should solve that race condition (if any) in main, using another way, such as C++ language constructs.

Goal 4: The main should then create several rounds of threads to merge the segments. The segments are merged from left to right. That is segment 1 should be merged with segment 2. For any odd numbered segment, i , it is merged with segment i+1. **Note that merges are done in parallel.**

You don't have to write your own merge function here; in fact, you can (should) use the C++ function merge().

The main should create concurrent threads, one thread per 2 segments. Each thread should call the same thread function, with its respective argument/s. In this function, each thread should merge its corresponding 2 segments, using C++ merge() function. You can (should) use temporary array/s or std::vector/s to store the merged result of merging 2 segments. After each thread has merged its respective 2 segments, it should print from within its thread function “Merged floor(n/p) and floor(n/p) (example) numbers”. And then in the next line, it should print the merged segment of numbers.

You will encounter race conditions in this task, possibly at several places.

To solve it:

You should and are allowed to use mutex locks in thread function, such as pthread_mutex_lock and pthread_mutex_unlock. But most importantly you are allowed to use the mutex locks ONLY in the thread function. Care should be taken that using these mutex locks DOES NOT BREAK the condition of parallel merging. In other words, the concurrent threads should be merging their respective 2 segments in parallel.

If you encounter race conditions in main, you are NOT allowed to use ANY pthread library synchronization primitives including but not limited to: semaphores, mutexes, mutex locks such as pthread_mutex_lock and pthread_mutex_unlock and conditional variables such as pthread_cond_wait and pthread_cond_signal etc. You should solve that race condition (if any) in main, using another way, such as C++ language constructs.

Goal 5: Once you have a single sorted list, print it out from the main.

IMPORTANT GUIDELINES

On canvas, I have posted many example programs for threads. Take insights from these. The Concurrent threads example can serve as something to start with.

Please read in entirety. All specifications need to be adhered to. At some places, you are also given options, so there you can choose. But at places, where it says for example: which goals need to be completed and how and with what restrictions, then that must be followed.

Anything not specified in the description, there you are free to choose. Of course with everything said, the solution should be your own work, and collaboration is only allowed "WITHIN" the group.

FILES TO BE SUBMITTED

1) multi_threaded_merge_sort.cpp
2) .txt file will name/s and email id/s

GROUP WORK

This project can be done individually or in a group of maximum 3 people. For a group of 2 or 3, each of the group members needs to submit. If one group member fails to submit, that person gets 0. Indicate in an additional .txt file, the names and email ids of members in the group. If working individually, indicate in the .txt, your name and email id.

Blurb for your resume

Use your GitHub account as a ready-to-show portfolio of your programming projects to potential employers.