CSCE 735 Fall 2023

HW 3: Parallel Merge Sort Using OpenMP

This assignment builds on the last one where you developed a thread-based parallel merge-sort from starter serial code. The following description is replicated from the previous assignment, however, the problems after that are new.

You are provided with a program `sort_list_openmp.c` that sorts a list of integers in ascending order using an iterative variant of the merge sort algorithm. A recursive approach to merge sort consists of splitting the list into two sublists, sorting the two sublists recursively, and then merging them. The merged list is constructed one element at a time by comparing pairs of elements – one from each sublist – and placing the smaller of the two in the merged list. The pointer that keeps track of the sublist elements to be compared advances to the next when its element is placed in the merged list.

The merging process is inherently serial. To develop a parallel merge sort, it is worthwhile to consider a bottom-up approach which results in an iterative algorithm. Starting from an unsorted list, pairs of adjacent elements are "merged" into sorted sublists of size 2. This is followed by merging adjacent lists into a larger list that is twice the size. The algorithm iterates through $k = \log_2(n)$ steps, or levels, where each level consists of merging adjacent sublists into a list twice the size. The last level merges two halves of the original list into a fully sorted list, and is equivalent to the first step in the recursive version described earlier.

To develop a parallel merge sort with $p = 2^q$ threads, we begin at level (k – q) of the iterative algorithm described in the preceding paragraph. At this level, we have p sorted sublists, each consisting of n/p elements (for simplicity assume both n and p are powers of 2). Let us denote the sublist for thread i by L_i. The remaining q steps of the algorithm result in merging pairs of sublists for the last q levels as described in the iterative algorithm.

In order to employ all the threads in the merger process, thread i is given the responsibility of determining the location of every element of its own sublist L_i in the merged list, and placing it directly in that location. For an element v, it is sufficient to figure out how many elements in the paired list L_{i-1} (or L_{i+1}) are smaller than v, which can be determined by either a linear search or binary search for v in of L_{i-1}. Note that at higher levels where the sublists are larger than L_i, thread i remains responsible for n/p elements originally assigned to it; however, now it has to consider larger sublists when computing the location of an element v. This is a bit involved but should be evident when you examine the code provided to you. See figure at the end for additional clarification.

While the code is serial, it is written in a way that identifies the work to be done by each thread, which should allow straightforward parallelization using threads. The only difference from the iterative algorithm is that sublists L_i at level (k – q) are sorted by quick-sort instead of merge-sort.

To compile and execute the code, use the commands:

```
module load intel
icc -qopenmp -o sort_list_openmp.exe sort_list_openmp.c
./sort_list_openmp.exe <k> <q>
```

where <k> and <q> are integer arguments that specify the number of elements in the list n = 2^k, and the number of threads p = 2^q. The output of a sample run is shown below.

```
./sort_list_openmp.exe 20 4
List Size = 1048576, Threads = 16, error = 0, time (sec) = 0.1414, qsort_time = 0.1260
```

Note that in its present form, the code is not multi-threaded. Thus `Threads` is merely reporting the value 2^q for the input q.

1. (70 points) Revise the code to implement parallel merge sort via OpenMP. The code should compile successfully and should report `error=0` for the following instances:
   ```
   ./sort_list_openmp.exe 4 1
   ./sort_list_openmp.exe 4 2
   ./sort_list_openmp.exe 4 3
   ./sort_list_openmp.exe 20 4
   ./sort_list_openmp.exe 24 8
   ```
2. (20 points) Plot speedup and efficiency for all combinations of k and q chosen from the following sets: k = 12, 20, 28 ; q = 0, 1, 2, 4, 6, 8, 10. Comment on how the results of your experiments align with or diverge from your understanding of the expected behavior of the parallelized code.
3. (10 points) For the instance with k = 28 and q = 5 experiment with different choices for OMP_PLACES and OMP_PROC_BIND to see how the parallel performance of the code is impacted. Explain your observations.

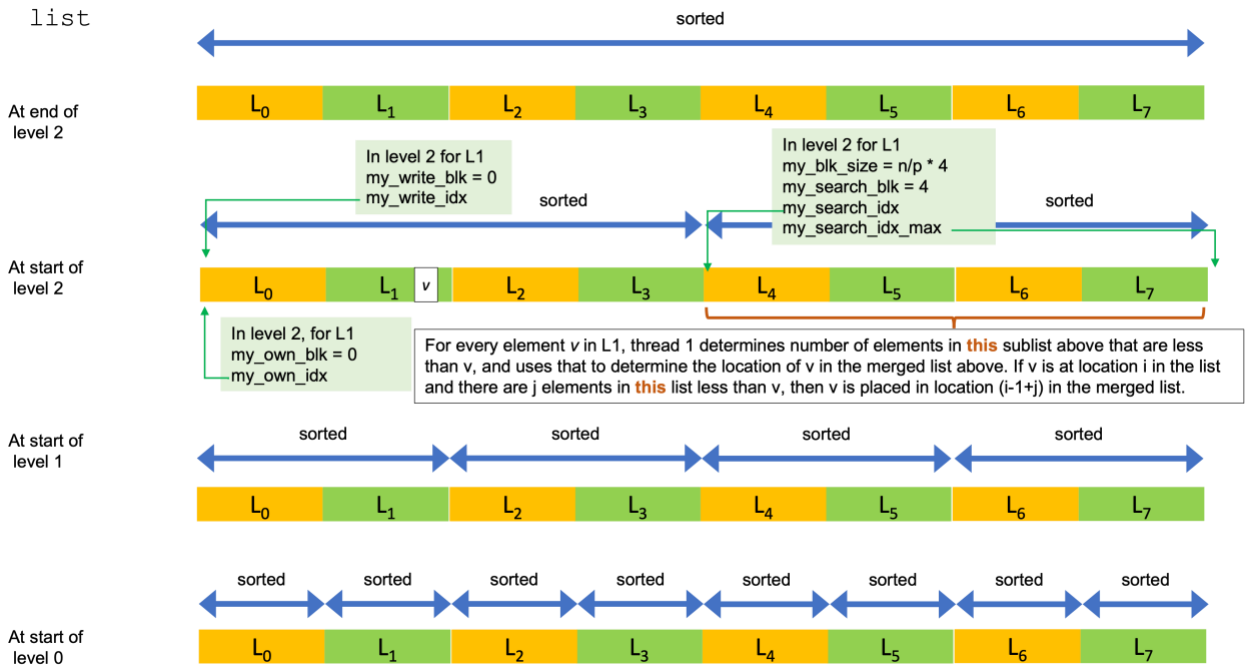
Submission

Upload two files to Canvas:

1. Submit the file `sort_list_omp.c` that implements your OpenMP-based parallel merge sort for Problem 1.
2. Submit a single PDF or MSWord document with your responses for Problems 2 and 3.

Helpful Information

1. You may use Grace for this assignment.
2. Load the compiler module prior to compiling your program. Use:
   `module load intel`
3. Compile C programs with OpenMP pragmas using `icc` with the switch `-qopenmp`. For example, to compile `code.c` to create the executable `code.exe`, use
   `icc -qopenmp –o code.exe code.c`
4. The run time of a code should be measured when it is executed in dedicated mode. Create a batch file and submit your code for execution via the batch system on the system.


list
sorted
At end of level 2
L0 L1 L2 L3 L4 L5 L6 L7
In level 2 for L1
my_write_blk = 0
my_write_idx
sorted
In level 2 for L1
my_blk_size = n/p * 4
my_search_blk = 4
my_search_idx
my_search_idx_max
sorted
At start of level 2
L0 L1 v L2 L3 L4 L5 L6 L7
In level 2, for L1
my_own_blk = 0
my_own_idx
For every element v in L1, thread 1 determines number of elements in this sublist above that are less than v, and uses that to determine the location of v in the merged list above. If v is at location i in the list and there are j elements in this list less than v, then v is placed in location (i-1+j) in the merged list.
At start of level 1
sorted sorted sorted sorted
L0 L1 L2 L3 L4 L5 L6 L7
sorted sorted sorted sorted sorted sorted sorted sorted
At start of level 0
L0 L1 L2 L3 L4 L5 L6 L7