CSCI 176 (Parallel Processing) Spring 2021 Prog. Assignment 4 (20 pts) Due: April 05 (M)

Parallel merge sort with OpenMP.

Consider the merge sort algorithm that you studied in data structures/algorithm course(s).
For a given problem size n, each thread is assigned a local list of n/p elements.
In this version of parallel merge sort, each thread performs 'qsort' (GNU library quicksort) to sort the local list instead of implementing the recursion-based method.
Merging step is a tree reduction, e.g., thread_0 merges locally sorted lists from thread_0 and thread_1, thread_2 merges locally sorted lists from thread_2 and thread_3, and so on.
Eventually, thread_0 produces the final globally sorted list.
A more detailed guide is shown below.

Programming guide ----------

1. Access command line arguments for n (total number of elements in the list) and p (total number of threads); create a list (dynamic array) of n integers; initialize the list with random numbers – each random number should be less than or equal to n, and it is suggested to use 'parallel for' loop.
2. Time checking starts at this point; please use 'omp_get_wtime()' that OpenMP supports.
3. Launch multiple threads; in each thread, create a local list of size n/p, copy list[my_start ~ my_end] to the local list, and perform qsort (GNU library quicksort); each thread is responsible for updating the global list with the sorted local list. It is suggested to use 'parallel for' loop for local list copy steps.
4. Merging steps need a while loop, which you practiced in HW1; pseudo-code is shown below:

```
divisor = 2; core-difference = 1;
while (divisor <= thread_count)
{ //use barrier synchronization here;
  //determine whether this thread is sender or receiver;
  //if receiver, perform merging operation by calling merge(..) function;
  //there is nothing to do for a sender;
  divisor *= 2; core_difference *= 2;
 }
```

   Merge(..) function should be defined separately; one suggested idea is that the function updates the global list with the merged result.
5. Time checking ends at this point; please use 'omp_get_wtime()' that OpenMP supports.
6. Finally, please make a function to check whether the final list is completely sorted or not.
   This function is easy to define, i.e., using a for loop (i=0 ~ n-1), check whether (list[i] > list[i+1]).

Submission:

Before compilation, include good documentation (global doc, each func head doc, etc.) in the source code; and submit source code and screenshot(s) of execution output.

For the output, 1. Show the trace result with exec time for n=100 and p=4;
2. Without trace, show the time and correctness for n=100000000, p=1,2,4,8.
Please see the sample outputs shown in the next page.

Sample output with trace (n=40, p=4 case):

```
Original list:
24 7 36 34 16 27 13 10 28 15 24 7 18 34 16 12 21 15 11 33 14 33 27 28
28 17 36 3 3 26 40 10 16 39 17 30 31 26 15 22

Thread_2, local_list: 14 33 27 28 28 17 36 3 3 26
Thread_0, local_list: 24 7 36 34 16 27 13 10 28 15
Thread_3, local_list: 40 10 16 39 17 30 31 26 15 22
Thread_1, local_list: 24 7 18 34 16 12 21 15 11 33

Thread_2, sorted local_list: 3 3 14 17 26 27 28 28 33 36
Thread_3, sorted local_list: 10 15 16 17 22 26 30 31 39 40
Thread_0, sorted local_list: 7 10 13 15 16 24 27 28 34 36
Thread_1, sorted local_list: 7 11 12 15 16 18 21 24 33 34

sorted list:
3 3 7 7 10 10 11 12 13 14 15 15 15 16 16 16 17 17 18 21 22 24 24 26 26
27 27 28 28 28 30 31 33 33 34 34 36 36 39 40
**verified that list1 is sorted.

Using P=4, n=40, Time taken: 0.000296116 sec
```

Sample output without trace (n=1000000000, p=1,2,4,8):

```
**verified that list is sorted.
Using P=1, n=1000000000, Time taken: 407.193 sec

**verified that list is sorted.
Using P=2, n=1000000000, Time taken: 262.296 sec

**verified that list is sorted.
Using P=4, n=1000000000, Time taken: 199.321 sec

**verified that list is sorted.
Using P=8, n=1000000000, Time taken: 197.879 sec
```