COL380 Assignment 1: Sorting with OpenMP Tasks

Read the following sorting algorithm and implement it using OpenMP framework. **You must use OpenMP tasks to implement the algorithm**. Try to parallelize your implementation as much as possible. Implementation is expected to scale with number of CPUs.

Algorithm ParallelSort(A, n, p):
Input:
A – an array of unsigned integers
n – number of elements in *A*
p – number of buckets

Procedure:

1. Divide *A* into A_0, A_1, …, A_{p-1}, *p* buckets of size $n/p \pm 1$ each as follows. Each A_i contains contiguous elements of A.

2. From each bucket A_i, select first *p* elements as pseudo-splitters. Let R = [r_0, r_1, …, r_{p*p-1}] be the sorted list of p^2 pseudo-splitters. This sorting may use ParallelSort or SequentialSort.

3. Select *p-1* equally spaced splitters from *R* as follows. Let *S* = [s_0, s_1, …, s_{p-2}] be the selected splitters such that $s_j = R[(j+1)*p]$ for *j* in *0* to *p-2*.

4. (Using tasks) Split *A* into *p* partitions B_0, B_1, …, B_{p-1} such that for any element *a* in partition B_i, $s_{i-1} < a \leq s_i$. Assume $s_{-1} = -\infty$ and $s_p = \infty$.

5. Let n_i denote the number of elements in partition B_i. Sort each partition B_i in a separate task which uses *SequentialSort*(B_i, n_i) if n_i < *Threshold*, and *ParallelSort*(B_i, n_i, *p*) *otherwise*. *SequentialSort* is sequential sorting of your choice implemented in a task.

6. Return concatenation of sorted partitions B_i.

Note that Threshold is minimum number of elements to switch to sequential sorting. Use *Threshold* = 2*n/p*, twice the expected size of the partitions. Grading will be in correctness as well as efficiency, scaling up to 24 cores.

Submission Instructions

Submit a single zip file named [Your Entry Number].zip on Moodle with the following:

1. An outline of your OpenMP task-based implementation and design choices made. Explain the degree of your parallelism and scalability (with the help of a graph like below).
2. Graphs of number of CPUs vs. execution time for (at least up to) 24 CPUs and with array of size at least up to 2^{32}.

3. Sources implementing the function signatures provided in the header “psort.h”. Do not include any data files.
4. A makefile that builds a library named “psort” (libsort.a or libsort.so) with the implementation of the functions provided in “psort.h”.

Note

1. You are expected to implement the sorting method ParallelSort().
2. ParallelSort() works in-place. It need not be stable.
3. You are free to define new functions/variables/classes as per your requirement outside specifications of psort.h. Make sure you include those in your submission and in the makefile. Do not change psort.h.
4. You are also given a driver.cpp file which reads the input data from a data file, calls the sorting code, and prints the sorting time. You can use it to run your implementation. You need not include this file in final submission.
5. A sample input data file is provided which has the following format. The first line contains two unsigned integers *n* and *p,* respectively followed by *n* lines containing *n* unsigned integers.