**CSC 338 Parallel and Distributed Computing**

**Exercise No. 16, December 4, 2017**

**A parallel sort**

**Goal**

Understand odd-even transposition sort in OpenMP

**Background**

Odd-even transposition sort is carried out in two phases. In the even phases each odd-subscripted element (a[1], a[3], etc) is compared with the element to its “left” (a[0], a[2], etc). If the elements are out of order they're swapped. During the odd phases each element is compared with the one on its right; again, the elements are swapped if they're out of order. The elements are guaranteed to be in order after n phases, where n is the number of elements. For example, consider the array [9, 7, 8, 6]. In phase 0, 7 is compared with 9 and the elements are swapped; 6 is compared with 8 and the elements are swapped, leading to [7, 9, 6, 8]. In phase 1, 9 and 6 are swapped. Consider which elements are compared in phases 2 and 3 and convince yourself that phase 2 and 3 complete the process (no changes are made in phase 3).

Here is the algorithm:

for (phase = 0; phase < n; phase ++)

if (phase % 2 == 0)

for (i = 1; i < n; i += 2)

if (a[i-1] > a[i]) swap(&a[i-1], &a[i])

else

for (i = 1; i < n-1; i++)

if (a[i] > a[i+1]) swap(&a[i], &a[i+1])

You may recall that loop-carried dependencies cause problems in OpenMP (a loop-carried dependency is when a value calculated in one iteration is used in calculating a value in the next iteration). The algorithm posted above has a loop-carried dependency in the outer loop. Consider that, in the example, phase 0 should swap 9 and 7, as well as 8 and 6. But if the odd and even phases are carried out simultaneously, the comparison in the odd phase might be between 7 and 8, rather than 9 and 6.

**Procedure**

Look at the code for omp\_odd\_even1.c, in the exercise folder. Most of the code is support – getting command line functions, generating numbers, and so forth. It's instructive to understand how all those things work, though, so make sure you understand what each part of the code is doing. Also study the comments at the top of the program and make sure you understand how to use it. Note, too, how the code is timed.

Now turn your attention to the function that actually carries out the sort. Note the use of pragmas to parallelize the two phase-loops. In OpenMP, the parallel for directive must be followed by one block of code, which will be carried out in parallel. Fortunately, there is an implied barrier at the end of the block – no thread will proceed until all threads have finished the block. So all the hand-wringing above turns out to be hot air – in this case, we don't have to worry about the two phases going on simultaneously.

However, there is another potential problem. The parallel for directive might fork, then join, a new team of threads every time the block executes (some implementations might be smart enough to join the threads only after all the computation is finished). But we can, in fact, control that. Look at omp\_odd\_even2.c. How does it fork a team of threads once, then use those thread to carry out the phases of the algorithm?

Compile and execute both programs with 1, 2, 3, and 4 threads and with a large enough list of numbers to see a difference in execution time if it exists (I used n = 5000). Compare the execution times (run each comparison several times so you can get a reasonably stable idea of how much time each program takes). Is the second program faster than the first? Can you conclude that our implementation of the first program forks a new team of threads on each iteration of the loops?