Connie Shi
Professor Zahran - Lab 2

Lab 2: Modified Traveling Salesman

Note: I included 300 cities because the run times did not differ that much with such small input files (from 2 to 32 cities, the speed up was unnoticeable).

Cities	# Threads
2	2
4	2
8	2
16	2
32	4
64	4
128	8
300	16

I used #pragma omp parallel for num_thread(thread_count) in my program to parallelize the for-loop that does the computation for the greedy traversal of the graph. The number of threads was chosen based on performance after multiple testing. I couldn't come up with a "formula" to find the number of cities that will have the best performance, so I tested number of threads for each power of 2 up to the number of cities and found that there is one optimal number that runs in the best time. From cities 2 to 16, I would have had more or less the same performance if I just used 1 thread, but refrained because this lab is supposed to be OpenMP parallel. I used 2 to be on the safe side. This actually doesn't make much sense, because the

contention of 2 threads on such a small input size could actually create a worse performance than just one thread by itself.

Conclusion:

There is a general trend of an increase in speed up (time of serial divide by time of parallel) as we increase the number of cities. This is expected because as the number of cities increases, the sequential one-thread version will have to do more work by itself. However, when we parallelize with OpenMP, there are more threads that are sharing the work in parallel so it runs faster. In fact, with 300 cities, there is about 5x speedup so this number would greatly increase with more than 300 cities (though that may cause an overflow in the data).

The 5 causes for poor performance using OpenMP are sequential code, communication, load imbalance, synchronization and compiler non-optimization.

There is only one section that is declared critical in my program. This section is where the update of the best_path pointer takes place and is necessary. Two or more threads can try to change this pointer to their private, newly found, shorter path at the same time which would result in non-determinism in this update of a global variable. As this section is quite small, I have optimized it as much as possible. There is minimized communication.

I used #pragma omp parallel for with a dynamic schedule. While each thread should do about the same amount of work for a full path (traversing all n cities), using the branch and bound algorithm, my program returns when a path exceeds the distance of the shortest path already found. This means that at some iteration, a thread will travel until the end after n cities. Other iterations, however, can terminate early. Using a static scheduler means that threads might idle wait for others to finish. So to best minimize load imbalance, dynamic scheduler will give idle threads work to do.

Since there is one for loop that is executed in parallel, synchronization is minimized with just the one implicit barrier at the end of the for loop. This barrier is necessary (this is why I did not use nowait) because we do not want threads that have finished “early” to try and print the best path before all paths have been checked.

I unfortunately can’t do anything about compiler non-optimization (yet) so there isn’t anything to say.

This was a really fun lab to play around with. ☺