Assignment 1: Traveling Salesman

CMSC 502: Parallel Algorithms

Steven Hernandez
Department of Computer Science
Virginia Commonwealth University
Richmond, VA USA
hernandezsm@vcu.edu

Introduction

For assignment 1, we looked to find optimal paths for the traveling salesman problem (TSP) through a sequential program, a threaded program and finally an MPI solution.

Implementation

Each subsequent solution builds upon the previous program, such that the dynamic programming logic used to create the optimal TSP path for the sequential program (Fig. 1) was used to determine the optimal sub-paths per thread and process for the threaded and MPI programs (Fig. 2).

Sequential Implementation

The sequential program simply follows along with the dynamic programming algorithm given, thus there is little to note on the implementation.

Threaded/MPI Implementation

To have a chance at increasing the number of cities that the system can calculate paths for, we implement both a threaded version and an MPI version. Both implementations take the exact same steps aside from the way the programs communicate between processes or threads. First, each program loads all data as expected, then splits this data into a grid with number of blocks *b* which is divisible by *4*. Logistical sizes for *b* are both *16* and *4*. For each of these blocks, the data contained in the region of the block is either sent to an individual thread or MPI process. Each of these blocks contain a sub-graph of points from the global graph with which, the sequential TSP implementation is employed to find the optimal TSP solution for this subset of points. As soon as this is completed, the thread or process transmits the steps taken for the optimal path as an array of city-identifiers (a based-1 array). The results of these gridded sub-solutions can be seen in Fig. 2.

Stitching Sub-paths

Because each thread in the threaded program and each process in the MPI program are calculated independently of one another, these sub-path solutions are obviously not connected in any way which would allow for a single path. Instead, using a nearest neighbor approach, we take the following general approach:

```
1. Pick a random starting grid block g
2. While there are grid blocks left to connect
3.     For each allowed point p in g:
4.         For each grid block g' which has not yet been connected:
5.             For each point p' in g':
6.                 If distance(p, p') < distance(p, nearest-neighbor):
7.                     Set nearest-neighbor := p'
8.     Connect to nearest-neighbor
9.     g = g'
```

In the above, me must consider that once we determine the nearest neighbor from some subsequent grid block, we are limited in the number of nodes with which we can exit the block. Given an ordered list of cities c = [1 2 3 4 5 6] where the order of the cities corresponds to the order in which to travel to each city, if we enter the grid point at city *3* then we can have only two choices. Upon entering at city *3*, we can either travel the rest of the cities in an increasing direction such that our next cities to travel to are *4,5,6* then considering the order list as a circular list, *1* before finally reaching city *2* where we are then forced to exit to some other grid block (or complete the path by returning to the first

grid point). Alternatively, we can travel in a decreasing direction from 3 to 2,1,6,5,4 where again, we must exit to some other grid block. Because of this, we have decreased the possible number of neighbors we must test.

We can see the result of this method in Fig. 3. Obviously, we can assume with all of the line intersections found in this path, we are not achieving quite the best path. Handling these inversions can be seen in Fig. 4, which is a much better solution even from a simple visual inspection. [1] Explains in detail an algorithm for determining whether two lines are intersecting. To determine this across the entire path, we must loop through the points in the path continuously until finally no intersections are found.

Analysis

As expected, the sequential version of the code is not as fast or able to handle the same number of cities, however, it does provide us with optimal solutions to the Traveling Salesman Problem. We can see in Fig. 5; the sequential version tops out at 25 cities. In fact, around the 20 city mark, we start seeing a sharp increase in time required for computation. On the other hand, we can see that both the MPI and threaded versions easily handle this number of cities in their computation. We can see that with 4 processes, the MPI version reaches this point at the 65 city level. The Threaded version with 16 threads seems to easily handle even 200 cities before requiring a large computation time increase.

Surprisingly, even though both MPI and threaded versions complete their computation in much less time than the sequential time, distance determined by these methods are very close (Fig. 6) to the optimal solution determined by the Sequential program. Figure 7 shows error is less than 10% for city counts between 15 and 25 in the given tests. Do note that both Fig. 6 and Fig. 7 present the same line for both the threaded version and the MPI version. This is because both methods use the same algorithm for calculating sub-paths, stitching sub-paths and handling inversions.

Issues

Because of the complexity of the program, while I was able to get all parts build (stitching, handling of inversions, both MPI and threaded versions), there are certain edge cases that have not been worked out as of yet for the project. For example, there are certain cases when the MPI version reaches a segmentation fault as a result of requiring too large memory blocks to be allocated. As mentioned in class, we are expected to handle these sorts of problems in the following project however.

Conclusion

Both threaded and MPI versions of this algorithm increases the possible problem size while keeping the required computation time down to a much lower amount than a sequential alternative. While an increase in complexity in software develop is required, the performance improvement allows for much more complex and thus useful problems to be solved.

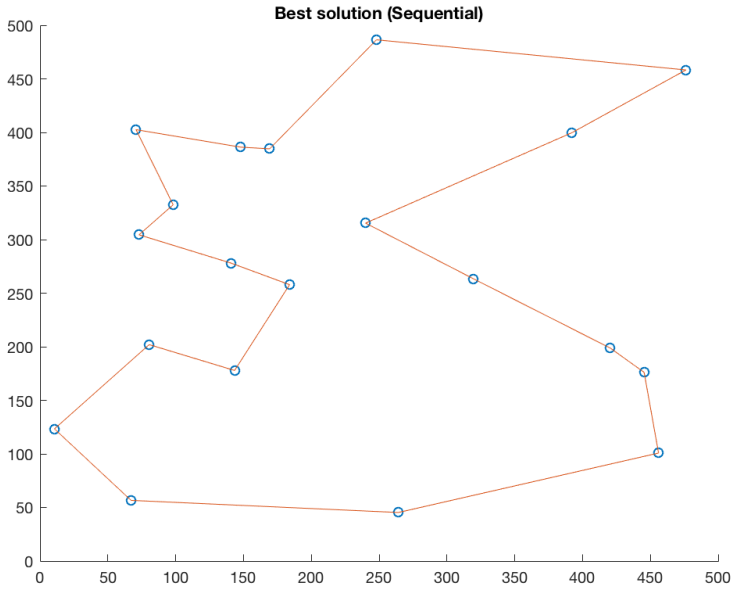

Figure 1. Optimal Solution from Sequential Program

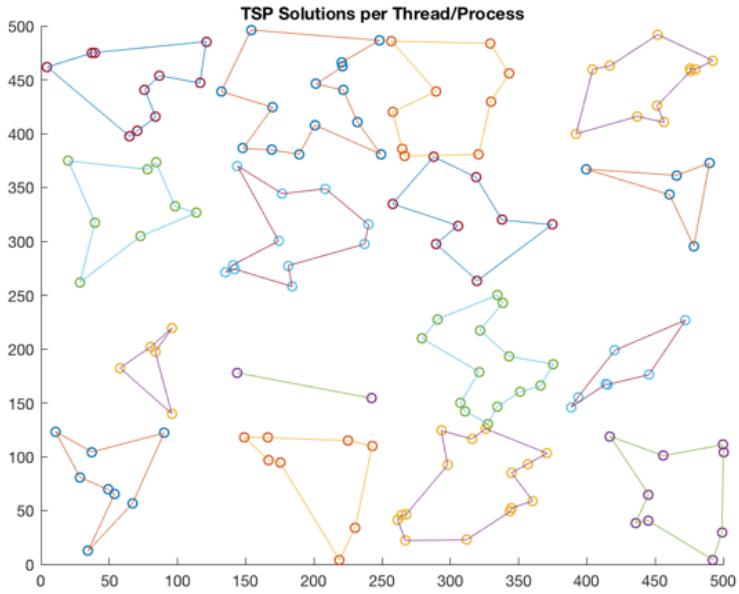

Figure 2. Optimal TSP Solutions for Sub-Graphs (Threaded/MPI)

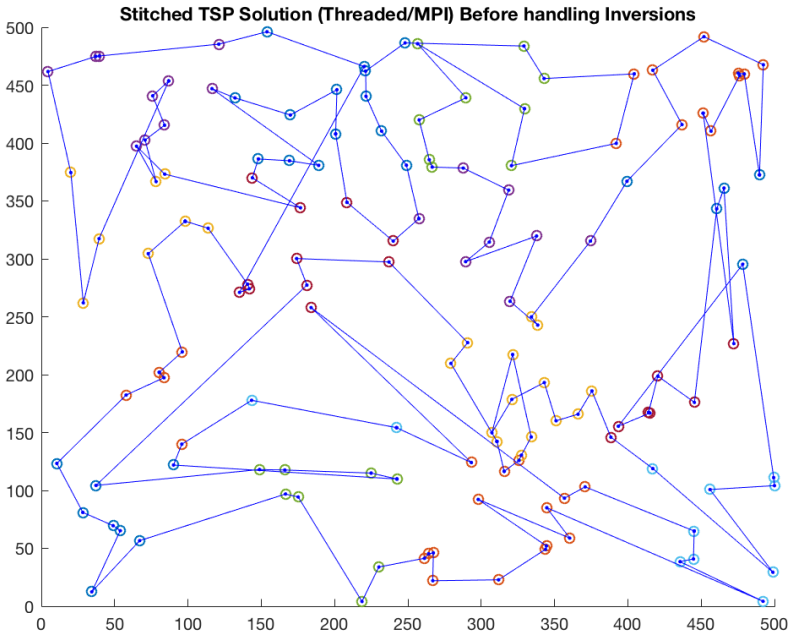

Figure 3. Stitched Solution Before Handling Inversions

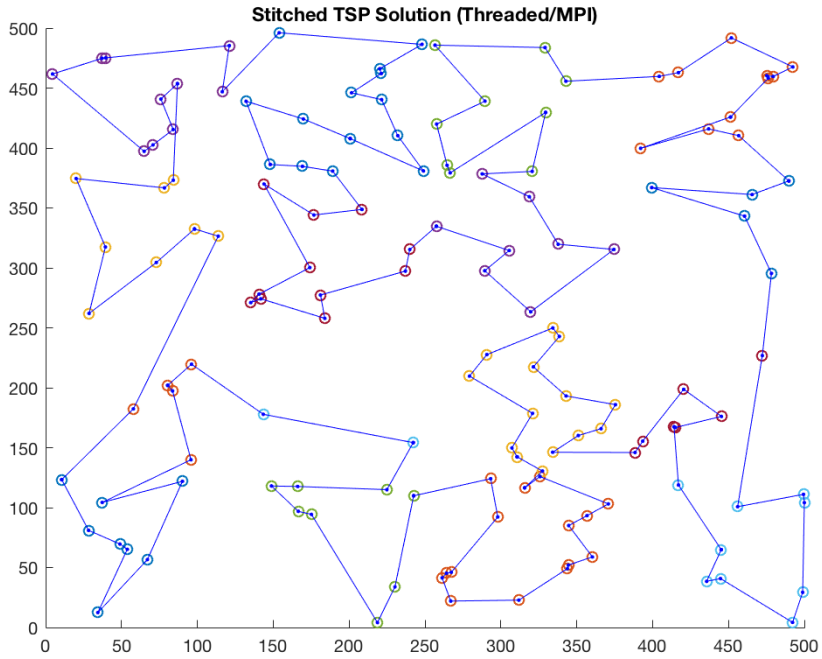

Figure 4. Stitched Optimal TSP Solution (Threaded/MPI)

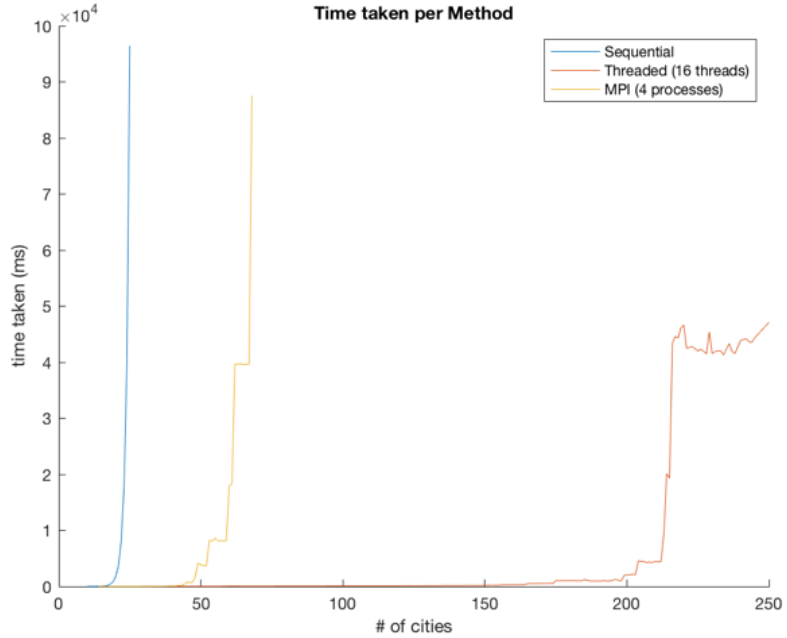

Figure 5. Computation Time for each Method

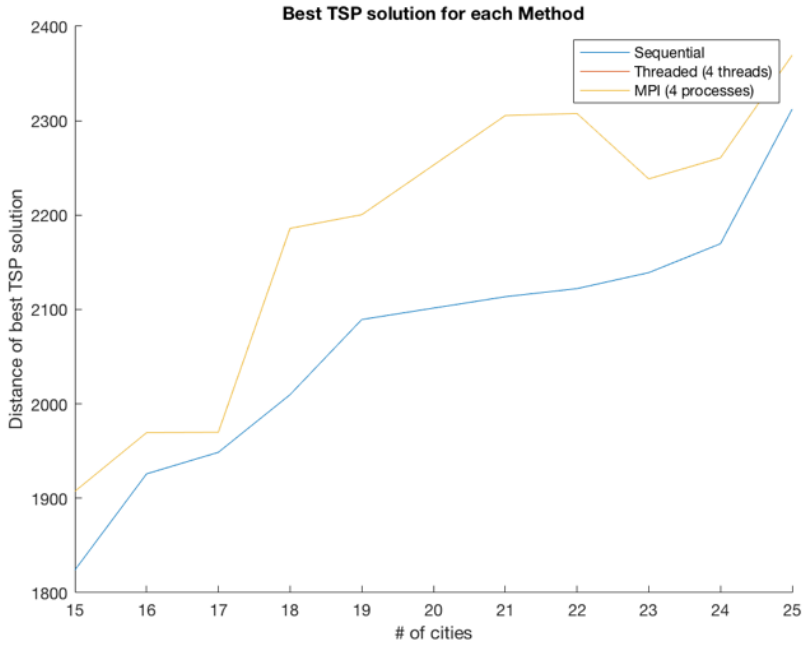

Figure 6. Optimal Distance Calculated Per Method

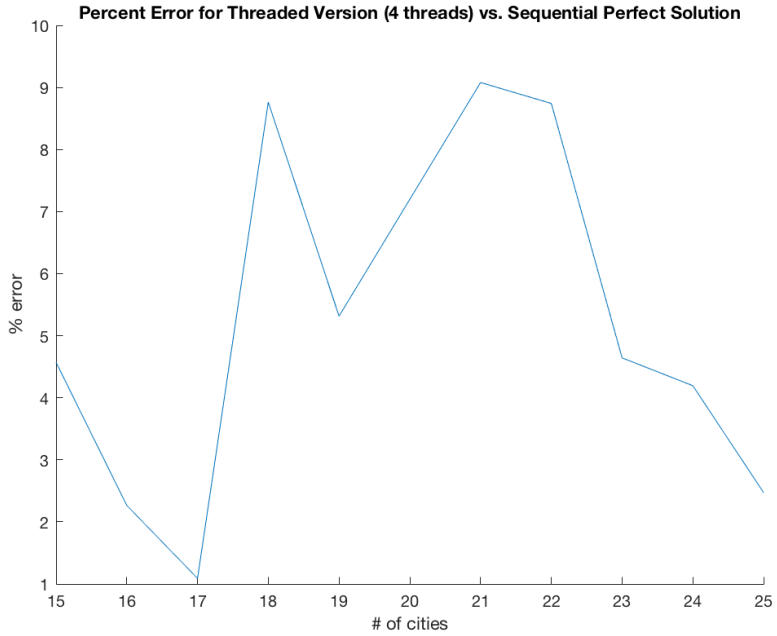

Figure 7. Percent Error for Threaded Version (4 threads) vs. Sequential

References

[1] “How to check if two given line segments intersect?,” GeeksforGeeks, 29-May-2018. [Online]. Available: https://www.geeksforgeeks.org/check-if-two-given-line-segments-intersect/. [Accessed: 10-Oct-2018].