Assignment 2: Parallel Traveling Salesman

CMSC 502: Parallel Algorithms

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

Introduction

In this project, we looked to implement a fully parallel MPI version of the Traveling Salesman Problem (TSP). In the previous iteration, we parallelized calculating sub-TSP solutions per process, but did not handle stitching in parallel; resulting in inefficient computation times. In this project, we looked to increase the parallelism and thus decrease computation times by stitching sub-TSP solutions together in parallel, Additionally, the method used for stitching provides much fewer inversions (although it may be assumed that the solution should result in no inversions).

In addition to this main goal of this project, we also look to improve the general abilities of the MPI implementation from the previous implementation. We are successful in this case. The previous method caused errors in certain cases when reversing the dynamic programming solution when for example floating point math would change values by small decimal points thus resulting in incorrect equality evaluations. Additionally, issues were solved in the use of infinity on the server when compared to running on a local machine.

Finally, the implementation is more generalized from the previous implementation in that the implementation can work on any square number of processors. Given the max number of processes possible on the server is *40*, previously we were only able to run the implementation on number of processes $p = 2^{2n}$ where n is a positive integer which in our case only allowed the implementation to work when $p \in \{1, 4, 16\}$. This new implementation allows for any square number of processors $p = n^2$ which means we can achieve much greater variety of processors; again, in our case $p \in \{1, 4, 9, 16, 25, 36\}$. This gives us more flexibility in that we are using a higher percentage of available resources available to us.

Process

We begin by looking at the process taken by the MPI implementation. Each process first selects a number of random cities within a 100x100 range which can be seen in Figure 1 where all 36 processes select 23 random city locations. This occurs without communication, followed directly by calculating the sub-TSP solution for the random city locations selected, Figure 2. At this point, each process begins making first communications with one another, first stitching complete rows by column as per the specification. For each process, the process will either send to a destination or received from some source. Because of this, we should in essence be able to avoid any barriers given adequate tagging of messages, however this was not implemented here. After complete rows are stitched, these rows are stitched in the same algorithm. Note, when we have numbers of processors that are not $p = 2^{2n}$ we can't guarantee as many processors are doing work as expected. For example, Figure 88 shows the case when $p = 2^{2n}$ or more simply, when dimensionality of each row and thus each column is $d = 2^n$. We see in this figure; a simple pattern appears in each time t_i where each process is either sending or receiving a message (or the process has completed all work and thus can be closed). However, given this implementation allows for any square number of processes $p = n^2$ we cannot guarantee these cases. Figure 99 shows the case when in the first-time instances, P_7 would normally require a message from P_8, however P_8 does not exist thus causing a deadlock. Figure 1010 as well shows a case where deadlock is possible, however in this case, this isn't in the first-time instance t_1, but instead later on in time instance t_2. We can see that each of the three cases in these figures, three communication periods are required even though less work and less communication is actually occurring as the number of processors drops from eight down to seven until 6 in Figure 1010 (do note that this would also be the case for p=5 but not p=4 which itself would require only 2 communication steps). Luckily, this won't affect the computation too much, but it should be considered. For example, if we had a higher number of processors on the server, we would be able to have p=64 which would might be a better case. Luckily, the logic involved in determining whether or not a process should send, receive or wait until the next time instances only results in constant overhead *O(1)*.

Analysis
The algorithm was run on each combination of processors $p \epsilon \{1, 4, 16\}$ with number of cities per processor $c \epsilon \{x | x < 20\}$. Figure 4 shows Time taken for calculating TSP solution for each combination. As we reach a threshold between 20-25 cities per processor, the times taken quickly increase. Before this point, the time taken is rather constant implying the communication overhead of communicating between each process takes the highest amount of time. We can see very obviously that while a higher number of processes takes more time with lower number of cities per block (Figure 5), we see these higher number of processes shine when there are a high number of cities per block, beating any attempt with fewer numbers of processes. Notice, I found that the algorithm for stitching did not result in no inversions, so the code did require handling inversions. The same analysis from above can be seen when handling inversion in Figure 6 and Figure 7 respectively.

Because each block creates the same number of random cities, we aren't able to evaluate the accuracy of a single process to 36 processes (i.e. 1 process cannot handle 36 cities). Figure 11 shows the TSP solution as calculated for different numbers of processors for given number of total cities. We can see in this that with more processors, we get significantly high total distance for the TSP solution implying that there are better solutions, if only we could compute more nodes per individual process. Additionally, Figure 12 shows the percent error when using 4,9 and 16 processors compared to the more perfect solution calculated with a single process. We can see that with 4 processes, we end up with 100% error meaning that the solution is twice as long as the optimal solution. Even worse, with 9 processes, error reach the 300% mark both of which make sense when looking at the distances plotted in Figure 11. As a result, as more processors are added, we get worse results.

Implementation Time Complexity
The implementation takes the following general algorithm

1.	Generate K random cities for all *n* blocks (where n=b^2)	O(K)
2.	Calculate sub-TSP	O(K^2*2^K)
3.	Merge all Columns per Row (*b* blocks per row)	O($(Kn)^2$*logb)
4.	Merge Rows into Full TSP	O($(Kn)^2$*logb)
5.	Compute full path length	O(Kn)

Notice, when merging the rows and columns, each iteration we not only merge the contents from two blocks in parallel, but also handle any inversions as they show up. This allows us to handle inversions on smaller sets of cities, but also handle inversions in parallel which is faster than handling them all at the end at a single node.

Also, because we are limited to a certain number of cities, we can consider any value related to *K* as a constant; thus, our algorithm can be evaluated simply with regards to b and n (which are a function of each other).

Sequential Time Complexity
The key complexity comes in merging all blocks, each of which must be added sequentially to an increasing total value.

So, given K cities per block, the first instance merges comparing K cities in block 1 to K cities in block 2. The second merge then compares 2K cities (those merged in the last instance to K cities in block 3 and so on up to the final block.

$$\sum_{i=1}^{n} K * (K * i) = \frac{n(n+1)}{2} * K^2$$

Which again, since we don't consider K is:

$$O\left(\frac{n(n+1)}{2}\right)$$

Parallel Time Complexity
Parallel on the other hand handles many of these operations in parallel thus, assuming communication iterations start and complete at the same time, the longest running process (process rank=0) must merge $\sqrt{n}$ times.

$$\sum_{i=1}^{\sqrt{n}} K * K * i^2 = 2^{\sqrt{n}} * K^2$$

Simplified without *K* leaves us with

$$O\left(2^{\sqrt{n}}\right)$$

Thus, we can calculate speed up as:

$$S = \frac{\frac{n(n+1)}{2}}{2^{\sqrt{n}}} = \frac{n(n+1)}{2 * 2^{\sqrt{n}}}$$

Thus, efficiency would be

$$E = \frac{\frac{n(n+1)}{2 * 2^{\sqrt{n}}}}{n} = \frac{n+1}{2 * 2^{\sqrt{n}}}$$

From here, we can look at cost

$$pT_p = \frac{n^2(n+1)}{2 * 2^{\sqrt{n}}} \ (because\ p = n)$$

There is overhead in the parallel algorithm such that

$$T_o = n^2 \log(b) = n^2 \log(p) = p^2 logp$$

Thus, we can see that the isoefficiency W can be computed as

$$W = \mathrm{K} * p^2 logp$$

Where K in this case is a constant of effeciency.
Based on this, the number of processors required for a problem set with size of N is

$$p = \sqrt{N}$$

The biggest issue with this method is in the total Amount of memory required for this algorithm which is lower bounded by the amount of space required to calculate the sub TSP per process. In total this comes to:

$$M = p(K^2 * 2^K)$$

Where K is the number of cities within each individual process. We do *free()* any no longer needed memory as soon as possible in the current implementation so that we don't hold onto useless space per process.

Conclusion

In this project, improvements were made not only by stitching sub-TSP blocks in parallel, but also in fixing bugs from the previous implementation as well as in allowing the implementation to utilize the higher number of processors available from the server.

Figures

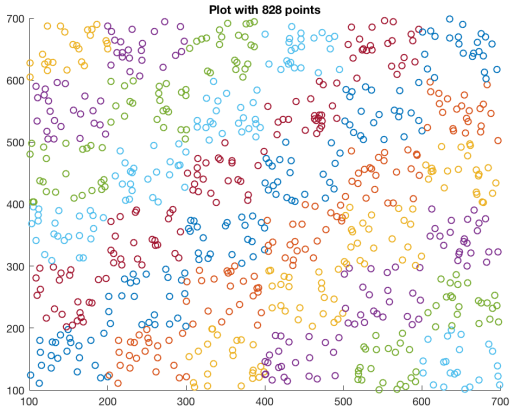

Figure 1 Randomly selected points for each 100x100 processor block.

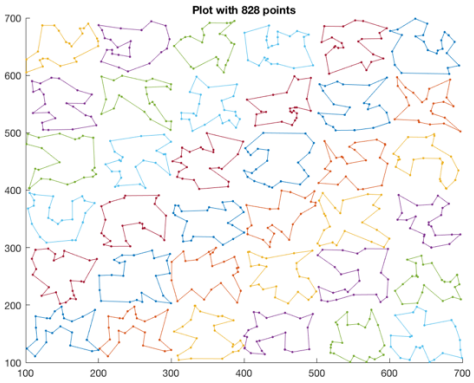

Figure 2 Sub-TSP solutions computed per processor block.

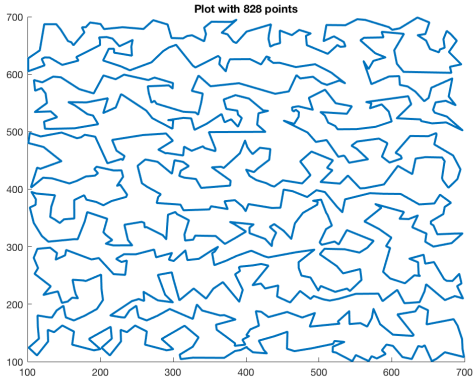

Figure 3 Final TSP across all 828 points showing no inversions.

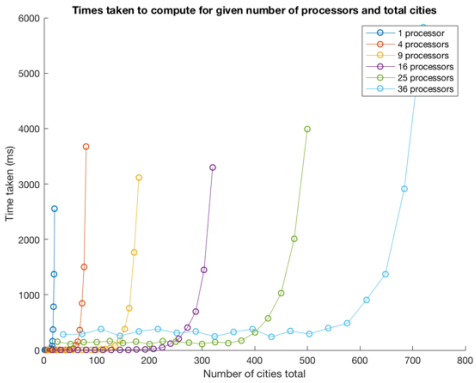

Figure 4 Time taken for calculating TSP solution for different numbers of processors. As we reach a threshold between 16-20 cities per processor, the times taken quickly increase after this point.

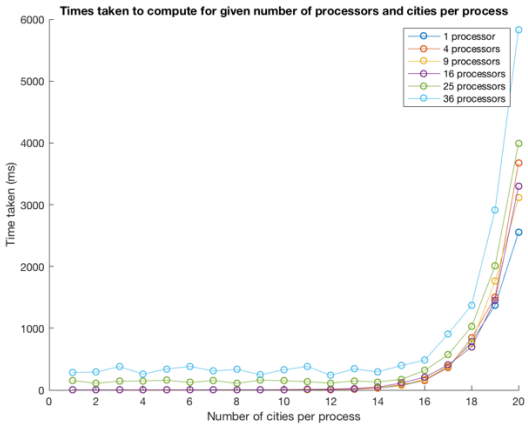

Figure 5 Time taken comparing number of cities per process shows an obvious higher time when higher numbers of processors are involved, as would be expected given higher amounts of communication required

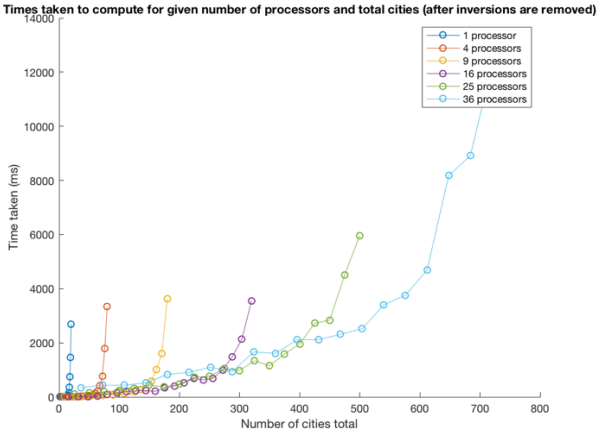

Figure 6 Time taken for each processor after handling inversions.

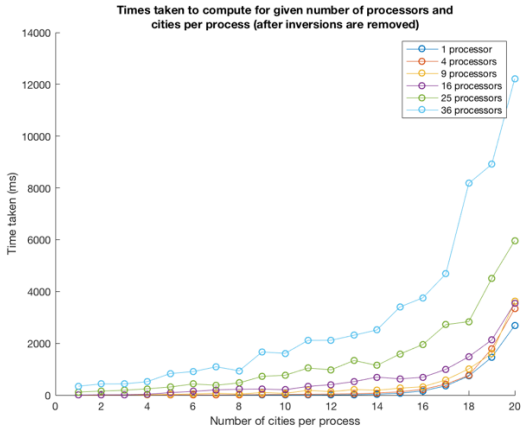

Figure 7 Time taken comparing number of cities per process shows an obvious higher time when higher numbers of processors are involved, as would be expected given higher amounts of communication required

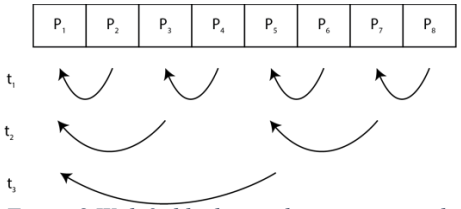

Figure 8 With 2^n blocks per dimension, our algorithm simply allows for all log(x) processes to complete the expected stitching work at each time frame.

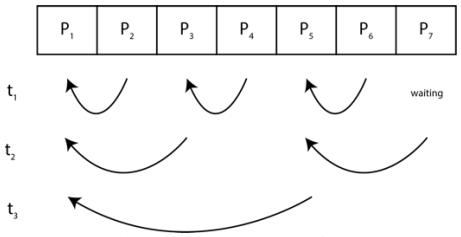

Figure 9 When dimensionality of a row is not 2^n; in this example p=7, process P_7 must know not to expect a message from P8 at t_1 otherwise a deadlock will occur.

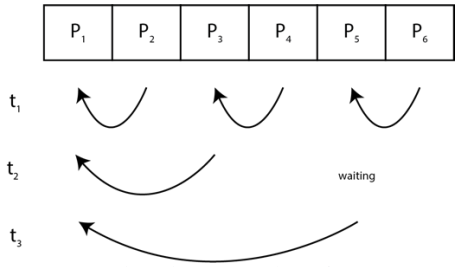

Figure 10 When dimensionality of a row is not 2^n; in this example p=6, P_5 must know not to expect a message from P_7 at t_2 otherwise a deadlock will occur.

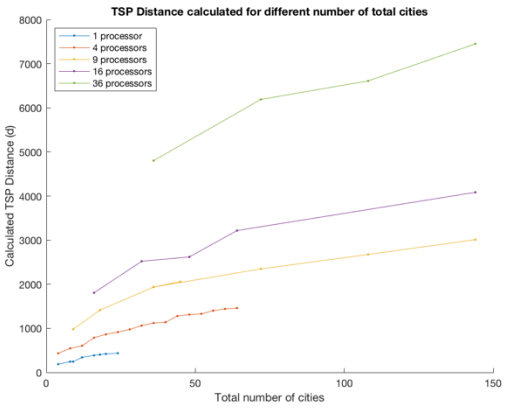

Figure 11 TSP Distances calculated for different total number of cities as calculated by different number of processors.

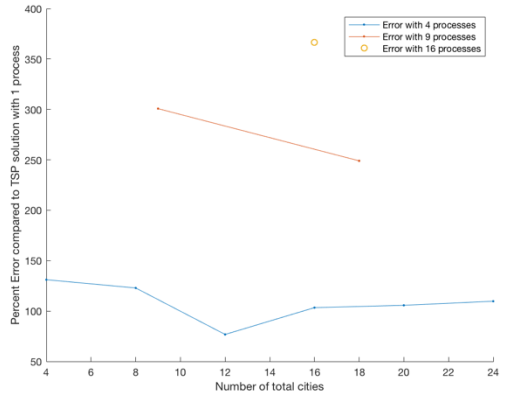

Figure 12 Calculated Percent Error for TSP solutions with different numbers of processors showing with more processors, there is more higher error.