Solving Traveling Salesman Problem Using Parallel Genetic Algorithm and Simulated Annealing

Fan Yang

May 18, 2010

Abstract

The traveling salesman problem (TSP) is to find a tour of a given number of cities (visiting each city exactly once and returning to the starting city) where the length of this tour is minimized. In this project, we implemented solutions to the traveling salesman problem using parallel genetic algorithm and simulated annealing, and compared the parallel performance and results of these two implementations. In performance analysis, we used the results from running serial implementation of the two algorithms as baselines for comparison. Furthermore, we studied the effect of changing the rate of convergence on the parallel performance of these two algorithms. The two algorithms are implemented in C and parallelized using the MPI library. The study indicates that both the genetic algorithm and simulated annealing implementations achieve good parallel speedup. We found that increasing the rate of convergence improves the optimization result (prior to reaching a plateau) but also significantly increases the running time.
1 Introduction

Optimization problems have numerous applications such as in computer wiring, planning, and logistics. Numerous approaches have been created to solve these optimization problems efficiently. In this paper, we examine simulated annealing algorithm and the genetic algorithm in the context of solving the traveling salesman problem, a well-known combinatorial optimization problem in operations research and theoretical computer science. The idea of the traveling salesman problem is to find a tour of a given number of cities (visiting each city exactly once and returning to the starting city) where the length of this tour is minimized. We investigate the performance of the parallel implementation of these two algorithms in solving the traveling salesman problem.

2 Simulated Annealing Algorithm for TSP

In simulated annealing algorithm, the problem of finding an optimal solution is analogous to a process of reducing the overall “energy” of a system via state transitions. In the traveling salesman problem, a state is defined as a route, or a permutation of the cities to be visited. The energy of a state is the total distance of a route. In every iteration of the algorithm, a candidate state transition, also known as a neighbor of the current state, is generated by exchanging a pair of cities. The neighbor route is then evaluated by an acceptance function that makes the state transition if the neighbor route has a lower energy or chosen with a probability that depends on the difference between the energies and on a global parameter $T$ (called the temperature).

2.1 Serial Version:

1. Initialization: Generate a random candidate route and calculate fitness value for the route.

2. Repeat following steps NO_OF_ITERATIONS times:

   1. Generate a random candidate route.
   2. Calculate fitness value for the candidate route.
   3. If the candidate route has a lower energy or chosen with a probability that depends on the difference between the energies, accept the candidate route.
   4. Repeat steps 1-3 NO_OF_ITERATIONS times.

   5. Output the best route found.

   6. End the algorithm.

   7. Print the best route found.

   8. End the algorithm.

   9. End the algorithm.
3. Return the best result

2.2 Parallelized version:

1. On each thread:

(a) Initialization: Generate a random candidate route and calculate fitness value for the route.

(b) Repeat following steps NO_OF_ITERATIONS times:

   if ITERATION_COUNT != CONVERGENT_COUNT

   i. Neighbor function: Generate a neighbor route by exchanging a pair of cities.

   ii. Acceptance function: Evaluate the neighbor route for acceptance - if accepted, replace current route with neighbor route.

   else

   i. MPI_Barrier: share the best result among threads by sending all the results to root and have root broadcast the best result

2. MPI_Barrier: return the best result among all the threads.

3 Genetic Algorithm for TSP

Genetic algorithms are typically implemented as a computer simulation in which a population of abstract representations (called chromosomes) of candidate solutions (called individuals) to an
optimization problem evolves toward better solutions. The evolution starts from a population of completely random individuals and happens in generations. In each generation, the fitness of the whole population is evaluated, multiple individuals are stochastically selected from the current population (based on their fitness), modified (mutated or recombined) to form a new population, which becomes current in the next iteration of the algorithm. In the context of the traveling salesman problem, the random population is a population of randomly generated candidate routes. In each iteration of the genetic algorithm, two new routes are created by crossover and mutation using the two best routes of the current generation. These two new routes then replace the worst two routes to form a population for a new generation. Below is the step-by-step description of the genetic algorithm used for traveling salesman problem.

### 3.1 Serial Version:

1. Initialization: Generate N random candidate routes and calculate fitness value for each route.

2. Repeat following steps NO\_OF\_ITERATIONS times:
   
   (a) Selection: Select two best candidate routes.
   
   (b) Reproduction: Reproduce two routes from the best routes.
   
   (c) Generate new population: Replace the two worst routes with the new routes.

3. Return the best result

### 3.2 Parallelize Version:

1. On each thread:
   
   (a) Initialization: Generate N random candidate routes and calculate fitness value for each route.
(b) Repeat following steps NO_OF_ITERATIONS times:

if ITERATION_COUNT != CONVERGENT_COUNT

i. Selection: Select two best candidate routes.

ii. Reproduction: Reproduce two routes from the best routes.

iii. Generate new population: Replace the two worst routes with the new routes.

else

i. MPI_Barrier share the best result among threads by sending all the results to root and have root broadcast the best result

2. MPI_Barrier return the best result among all the threads

4 Methods

A data set of 15 cities with its coordinates is used for this project. This data set is from a published paper - “Enhanced Traveling Salesman Problem Solving by Genetic Algorithm Technique”.

<table>
<thead>
<tr>
<th>Cities (15 Total)</th>
<th>Coordinates</th>
</tr>
</thead>
<tbody>
<tr>
<td>a, b, c, d, e, f, g, h, i, j, k, l, m, n</td>
<td>(110,54), (236,110), (153,151), (227,49), (307,176), (220,211), (341,90), (149,91), (335,40), (371,150), (218,161), (334,239), (148,227), (49,128), (183,39)</td>
</tr>
</tbody>
</table>

Figure 1: Cities in the traveling salesman problem

In comparing the performance of the parallel implementations and the serial implementations, we set some of the algorithm-specific parameters to constant values. For simulated annealing algorithm, the default cooling constant is set to 0.9999 and the start temperature is set at 10.0. For genetic algorithm, the default population size is set to 1000. The simulated annealing algorithm
is run for a total of $10^6$ iterations on each trial while the genetic algorithm is run for a total of $10^5$ iterations per trial. The discrepancy in the number of trials for these two algorithms is due to the fact that one iteration of genetic algorithm takes approximately 10 times longer than one iteration of simulated annealing. For each distinct condition, we conduct a total of 10 trials and analyze the results using mean and standard deviations. The distance between cities are computed as Euclidean distance.

This program is written in C with MPI. We ran the program on the Beowulf cluster with 1 to 8 compute nodes.

5 Results

5.1 Running Time

![Simulated annealing algorithm running time](image)

Figure 2: Simulated annealing parallel running time

Figure 2 shows that while running the same total number of iterations of simulated annealing algorithm, the running time decreases as the number of processors increases. The figure also indicates that the standard deviation of the running time results in each case is negligible, which
means that the running time values for multiple trials in each of the study are consistent.

![Genetic algorithm running time](image)

**Figure 3:** Genetic algorithm parallel running time

Similarly, Figure 3 shows that the running time of genetic algorithm also decreases as the number of processors increases. The reason that the results for simulated annealing and genetic algorithm are not displayed in the same graph is that the two algorithms have different setup, such as the number of iterations used and other algorithm specific parameters. The standard deviation of the running time results is noticeably bigger than the one from simulated annealing. This difference could be due to the fact that there is more computation per iteration, which could lead to a higher chance for compute nodes to reach the MPI Barrier’s at different times, resulting in a slightly more varied set of running time results.

### 5.2 Parallel Speedup

Parallel speedup is measured by the following formula:

$$S(N) = \frac{\text{Execution time on a single processor}}{\text{Execution time on } N \text{ processors}} = \frac{t_1}{t_N}$$
In this section, we graph the parallel speedup curves for simulated annealing and genetic algorithm.

![Simulated annealing algorithm parallel speedup](image1)

**Figure 4: Simulated annealing parallel speedup**

Figure 4 shows a near-linear parallel speedup for the simulated annealing algorithm. The curve approximates the linear line at all points, which means that the implementation has little to no deterioration in scalability.

![Genetic algorithm parallel speedup](image2)

**Figure 5: Genetic algorithm parallel speedup**

Figure 5 shows a near-linear parallel speedup for the genetic algorithm. Unlike the figure 4 for simulated annealing, the speedup curve for the genetic algorithm starts to drift away a bit when
there are 7 or 8 processors. This deviation from the linear line could be caused by the longer per-iteration running time in the genetic algorithm, which could lead more delays due to processors waiting for other processors to reach the MPI Barrier’s.

5.3 Route Optimization Performance

This section discusses performance of each algorithm in finding the shortest route. The results shown in figure 6 and figure 7 is collected by taking the distance of the shortest route, among all the compute nodes, at the end of each trial.

![Simulated annealing algorithm distance results](image)

Figure 6: Simulated annealing distance results

Figure 6 shows a gradual decrease in the distance of the shortest route as the number of processors increases in the simulated annealing algorithm. While there’s slight anomaly between the result for 4 processors and the result for 5 processors, the large standard deviation for the result from 5 processors leads us to believe that there could be an outlier that swung the result away from the linear trend.

The results indicate that having more processors generally achieves better optimization results. This result is intuitive as the processors share the best result among them at a specified con-
vergence frequency. Having more processors increases the likelihood that the best result among the processors, each time the processors converge, outperforms the best result from the previous convergence.

![Genetic algorithm distance results](image)

Figure 7: Genetic algorithm distance results

Similarly, figure 7 shows a gradual decrease in the distance of the shortest route as the number of processors increases in the genetic algorithm. Comparing the graphs of optimization results of these two algorithms, it appears that the genetic algorithm achieves slightly better results than simulated annealing at each data point and improves the result at a faster rate as the number of processors increases. One possible reason for such results is that there are more random routes generated, a whole population of routes instead of just one, and more mutations in genetic algorithm each iteration, both inversion and crossover rather than one neighbor route from exchanging a pair of cities.

5.4 Rate of Convergence

In this section we examine the effect of changing the rate of convergence on the running time and the optimization results. The rate of convergence is a measure of how frequently the processors
share the best result among themselves. At each convergence, all the processors send the shortest route to the root processor and wait at a barrier before the root processor broadcast out the best result to all the processors. The idea is that the higher rate of convergence can improve optimization while requiring more work and communication among the processors, and we aim to determine the exact effects of the rate of convergence, which could provide us with insights on optimal values for the rate of convergence.

In order to test the rate of convergence, we set the number of processors to 4 and varied the rate of convergence from 0% to 50%. Since we are studying the rates of convergence within a range of 50%, we picked 6 rates of convergence that would give us a sense of the trend or key points to interpolate other points of interest.

5.4.1 Effect on Running time

![Simulated annealing algorithm convergence running time](image)

Figure 8: Effect of rate of convergence on simulated annealing parallel running time

Figure 8 shows that for simulated annealing, there is a gradual increase in the running time as the rate of convergence increases. The serial implementation of the simulated annealing algorithm has a running time of 21.74 seconds. Using interpolation from the 6 data points, we observe that the
running time goes above the serial implementation at rate of convergence at 35%.

![Graph showing genetic algorithm convergence running time](image)

Figure 9: Effect of rate of convergence on genetic algorithm parallel running time

Similarly, for genetic algorithm, figure 9 indicates a gradual increase in the running time as the rate of convergence increases. The serial implementation of the genetic algorithm has a running time of 12.51 seconds. Using interpolation from the 6 data points, we observe that the running time exceeds the serial implementation at convergence rate of 15%.

Even though the increase in the running time for both the simulated annealing and the genetic algorithm is sub-linear, it quickly goes above the running time for a serial implementation of the simulated annealing and genetic algorithm. This result confirms our earlier hypothesis and indicates that the frequent communication and use of MPI Barriers can significantly increase the running time.
5.5 Effect on Route Optimization

![Simulated annealing convergence distance results](image)

Figure 10: Effect of rate of convergence on simulated annealing route optimization

Figure 10 shows that for simulated annealing, the distance of the shortest route decreases as we increase the rate of convergence. However, the shortest distance quickly plateaus starting at 10% rate of convergence, and stays near the 1220 level for shortest distance. The standard deviations also decreases significantly as we increase the rate of convergence. This shows that the results are consistent and stable starting from 10% rate of convergence.

![Genetic algorithm convergence distance results](image)

Figure 11: Effect of rate of convergence on genetic algorithm route optimization
Similarly, figure 11 shows that for the genetic algorithm, the distance of the shortest route decreases as we increase the rate of convergence. However, the shortest distance quickly plateaus starting at 5% rate of convergence, and stays near the 1220 level for shortest distance. Similar to the graph for simulated annealing, the standard deviations also decreases significantly as we increase the rate of convergence. This shows that the results are consistent and stable starting from 5% rate of convergence.

In terms of optimization, the results quickly plateau after a certain rate of convergence is reached: 10% for simulated annealing and 5% for the genetic algorithm. According to the results from the previous section, we know that parallel simulated annealing underperforms its serial counterpart at 35% and parallel genetic algorithm underperforms its serial counterpart at 15%. It is possible to run the parallel simulated annealing and genetic algorithms at 10% and 5%, respectively, and still outperform the serial implementation. Therefore, if our goal were to optimize the shortest route results for each configuration (number of processors), the rate of convergence should be set near 10% for simulated annealing and 5% for genetic algorithm.

6 Conclusion

The study indicates that both the genetic algorithm and simulated annealing implementations achieve a near-linear parallel speedup. We found that increasing the rate of convergence improves the optimization result (prior to reaching a plateau) but also significantly increases the running time. If the goal were to optimize for finding the shortest route, we would increase the rate of convergence to a point where the optimization performance plateaus. The optimal rate of convergence was determined by running the algorithm at various rates of convergence with the other parameters fixed. We found that the optimal values for running on four processors for simulated annealing and genetic algorithm to be 10% and 5%, respectively. These values of the rate of convergence
finds the shortest path while still outperforming the serial implementation. This result not only confirms our understanding of the tradeoffs between running time and distance optimization, but also motivates further research to determine the exact rate of convergence for all possible setups.
References
