THE PARALLEL IMPLEMENTATION ON GENETIC ALGORITHM FOR SOLVING TRAVELLING SALESMAN PROBLEM

Hao Chen, Mutian Xu

Department of Electrical Engineering and Computer Science
University of California, Irvine

ABSTRACT

Traveling salesman problem (TSP) is quite known in the field of combinatorial optimization. There had been many flexible attempts to address this problem using genetic algorithms (GA). With the significantly increasing of the dataset in real world, the sequential method constrains the performance of algorithms. This paper proposes three parallel methods based on using sequential GA to solve TSP problem, aiming at improving the efficiency of sequential solutions. In this paper, we parallelized the initial population and the evaluation parts through dividing them into several sub parts and used different cores to process different sub parts. Besides, we parallelized another greedy algorithm when initializing the population to set an upper bound, and we also provided non-migration and migration methods, which will be introduced in the introduction section. Using these parallel methods, we ultimately got a better path with shorter distance than that of sequential algorithm and reached four times speedup.

1. INTRODUCTION

The Travelling Salesman Problem (TSP) is one of the best-known NP-hard problems, and many algorithms have been developed to solve this problem. One of them is to use genetic algorithm (GA) to solve TSP because of its robustness and flexibility, these will be explained in detail in Background section.

For solving this problem, many researchers proposed different methods. In 2005, Lawrence V. Snyder presents a heuristic to solve the generalized traveling salesman problem. The procedure incorporates a local tour improvement heuristic into a random-key genetic algorithm. The algorithm performed quite well when tested on a set of 41 standard problems with known optimal objective values [1]. In 2001, E. Cantu-Paz. presents theoretical developments that improve our understanding of the effect of the algorithm's parameters on its search for quality and efficiency [2]. In 2005, Milena Karova introduces the solution, which includes a genetic algorithm implementation in order to give a maximal approximation of the problem, modifying a generated solution with genetic operators [3]. In 2006, Plamenka-Borovska investigates the efficiency of the parallel computation of the travelling salesman problem using the genetic approach on a slack multicomputer cluster. For the parallel algorithm design functional decomposition of the parallel application has been made and the manager/workers paradigm is applied [4]. In 2009, Abhishek Verma presents how genetic algorithms can be modeled into the MapReduce model. They describe the algorithm design and implementation of simple and compact GAs on Hadoop, an open source implementation of MapReduce [5]. In 2010, Fan Yang implements solutions to the traveling salesman problem using parallel genetic algorithm and compared the parallel performance and results of these two implementations [6]. In 2013, Harun Rait Er shows a parallel genetic algorithm implementation on MapReduce framework in order to solve Traveling Salesman Problem [7].

Previously, using GA to solve TSP is becoming a main topic because GA is time efficient and a good approximation for TSP. However, this method sometimes can stuck to local optima or takes considerable time when the number of cities increases. Genetic algorithm is a kind of probability search algorithm whose performance is affected by control parameters such as population size, hybridization and mutation probability, and sometimes it will converge to the local optimal solution [7]. Since the algorithm requires a large population size and the population evolves from generation to generation, the fitness function calculation needs to be carried out continuously, and the calculation amount is quite large. Therefore, parallelizing the genetic algorithm to improve the reliability and efficiency of the algorithm is the main content of the current research.

In this paper, we divided the population into several sub-populations and use different cores to process these sub-populations simultaneously. Besides, inspired by existing methods, we also tried to parallelize the evaluation part through using multiple processes to calculate fitness value of each individual solution simultaneously. Besides, compared to other works, we parallelized another algorithm greedy algorithm, which will be illustrated in Background, when initializing the initial population. This could increase the

converging speed of the genetic algorithm In addition, we also have non-migration and migration methods when generating the next generation.

The migration method would transfer optimal individuals of each sub-population to other sub-populations to generate the next generation. In this way, every sub population could get good genes of other sub populations. However, the non-migration method would not do this. Consequently, user could choose which method to use according to their aims. Non-migration method costs less time than the migration method since it avoids the inter-process communication. However, the migration method could give better results. Finally, different with traditional parallelization on genetic algorithm using openMP or CUDA, in our experiment, we directly use the multiprocessing library of python 3 to use multiple cores to process these calculations, which is easier than openMP and CUDA, and it is very easy for users to learn it.

2. BACKGROUND

In this section, Travelling Salesman Problem (TSP), Genetic algorithm (GA) and parallel computation will be illustrated respectively.

Travelling Salesman Problem. Travelling Salesman Problem (TSP) is not only a combinatorial optimization problem, but also a NP-hard problem, which means it could not be solved in polynomial time. It could be described as that a salesman will start from a city in one set of cities, then visit all other cities and return to the start city. The problem is calculating the path with shortest distance. Some researchers have researched a series of algorithms to solve this problem, such as Brute Force Algorithm, Dynamic Programming Algorithm [8], Nearest Neighbor Algorithm and Genetic Algorithm [9]. This paper will focus on Genetic Algorithm.

Genetic Algorithm. Genetic Algorithm is a computational intelligence method to solve combinatorial optimization problem. It is based on natural evolution, keeping the population with high fitness and filter out the population with low fitness. It could be divided into five parts, Population initialization, Evaluation, Crossover, Mutation and Next Generation [9]. In the Population initialization part, a number of random individual solutions will be generated, these individual solutions are called population. In TSP, initial population are random paths that including all the cities. In the Evaluation part, we firstly create a fitness function, the higher fitness value of the individual solution, the better the solution is. After calculating the fitness value of each individual solution, a selection function will make a difference. In the selection function, there is a threshold, only the individual solutions with higher fitness value than the threshold could be kept, and others will be filtered out. The kept individual solutions will be used to generate the next generation. In TSP, we use the reciprocal of the distance to represent fitness value, which means the shorter distance of the path, the bigger the fitness value is. As for the Crossover part, two individual solutions will be selected, and we called them parents. We select part of solution of parent one and part of solution of parent two, then combine these two parts to generate a new solution, which is child. In the Mutation part, we randomly swap the order of the cities in a solution to generate a new solution. Finally, in the Next Generation part, the new solutions generated in Crossover part and Mutation part will be regarded as new populations and repeat the whole process again. When starting the algorithm, a condition will be set, the condition can be iteration times or a fixed path distance, once the condition is satisfied, the algorithm will be terminated.

Parallel computation. In terms of parallel computation, it could be briefly described as using multiple threads to operate a series of instructions simultaneously. The attempt of parallel computation is reducing the running time of the algorithm, improve the efficiency of the algorithm and improve resources utilization. In this paper, we will illustrate how to parallelize Genetic Algorithm to improve the efficiency and reduce the running time.

Greedy Algorithm. Greedy Algorithm is a kind of algorithm used to solve TSP. In this algorithm, the salesman will start from the start location and visit a city that has the shortest distance with the start location. Then the salesman will choose an uvisited city which has the shortest distance with the current city. If all the cities are visited, the salesman will return to the start location. The time upper bound of this algorithm is O(n^2). This algorithm is very quick and could give a local optimal result.

Multiprocessing library of python. Multiprocessing is a library of python and it could execute multiple processes simultaneously. The basic idea of multiprocessing is using different cores to independently execute different processes at the same time. Each process will run on the corresponding core and use its own resources. Nowadays, most of the computers have 2, 4 or even 8 cores. However, many programs only use one core to execute their processes, which is a waste of resources. Multiprocessing could help to improve the CPU utilization and increase the computing speed.

3. METHOD

In this section, the sequential method will be introduced at the beginning, and three kinds of parallel methods will be illustrated and we will show how we optimized these methods. Since we used multiprocessing library of python 3 to parallelize the algorithm, we will also explain how to implement multiprocessing in our algorithm.

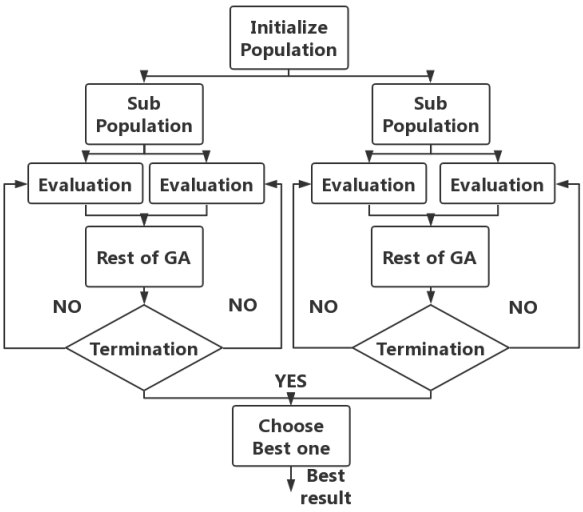

Fig. 1. Parallel method one.

Sequential method. We firstly used the sequential method to implement genetic algorithm. We generated 800 different paths in the population initialization part. Then these paths will be evaluated in the evaluation part, and we regarded the reciprocal of the total distance of a path as the fitness value of this path. Then we will sort these paths according to their fitness value. In the selection function, only the top 400 paths would be kept and others would be filter out. After that, the kept good paths would be used to create new paths through crossover and mutation functions. In our algorithm, we set the iteration time to 1500, so when the algorithm iterates 1500 times, it would terminate and output the best path. Based on this sequential method, we came up with three parallel methods.

Parallel method one.. Inspired by the existing methods, we firstly divided the initial population into two sub populations. Since each sub population could independently execute the genetic algorithm, we used two independent threads to process these two sub populations. Besides, since the evaluation function is the most complex function of genetic algorithm, and it takes the majority of the total time, we also used another two threads to parallelize this function. Similar with separating initial population into two sub populations, we separated the input data of the evaluation function into two sub inputs, and used two independent threads to process these two sub inputs simultaneously. Figure 1 shows the flow chart of this method. The following code snippet shows how to use multiprocessing library of python to parallelize the genetic algorithm.

```
import multiprocessing as mp
def initial_population():
    '''Randomly generate 800 different path,
       store these paths in list A'''
    return A
def GA(lives,q):
    '''Execute the genetic algorithm
    store the best path in list result'''
    q.put(result)
if __name__=='__main__':
    A = initial_population()
    A_1, A_2 = A[:400],A[400:]
    q1 = mp.Queue()
    q2 = mp.Queue()
    p1 = mp.Process(target=func,
        args=(A_1,q1,))
    p2 = mp.Process(target=func,
        args=(A_2,q2,))
    p1.start()
    p2.start()
    p1.join()
    p2.join()
    res1 = q1.get()
    res2 = q2.get()
    '''Compare res1 and res2 and
    output the better one'''
```

However, we found this method does not work well,the total time used to run the algorithm only decreased slightly compared with sequential method. This is because we used the multiprocessing library of python to implement parallelization, so we had to use extra cores to parallelize the evaluation function. However, if we have extra cores, we can use these cores to parallelize all the functions instead of only parallelizing the evaluation function. This would be more efficient. In addition, we also found this method could not always generate a good path, the total distance of the output path is not very short. This is because there is no upper bound for the genetic algorithm, it could only randomly generate paths, which cannot ensure we can always get a good result. Based on these drawbacks, we made some improvements of this method.

Parallel method two. Inspired by the drawbacks of the previous method, we made some improvements. Firstly, we parallelized greedy algorithm when initializing the population. Although greedy algorithm could only give a local optimal result, the result is better than the result of 'method one'. Besides, we divided the initial population into four sub populations and used four processes to independently process the four sub populations. Each process would execute the whole genetic algorithm on the corresponding sub population. In addition, after dividing the sub populations, we also sent the result of greedy algorithm to each sub population to set an upper bound. In this way, each sub population could output a result which has shorter distance than that of greedy algorithm. After each sub population terminates its iteration, it would give the best result. Then

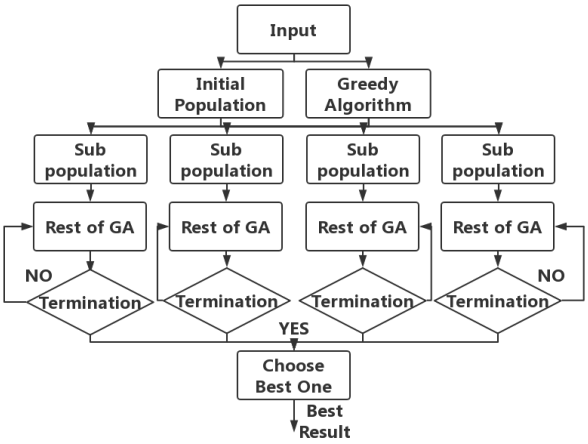

Fig. 2. Parallel method two.

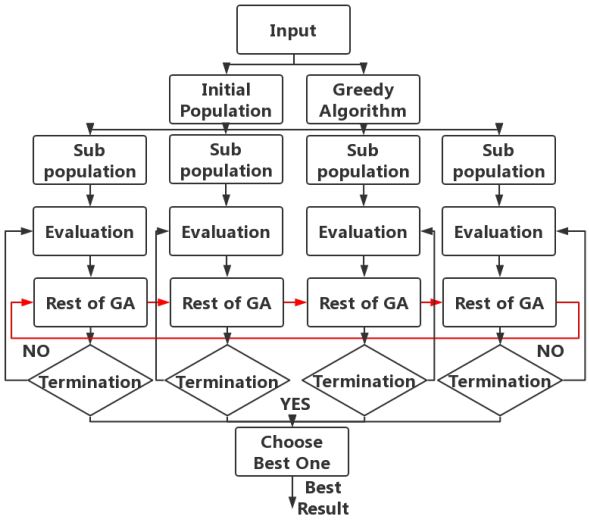

Fig. 3. parallel method three.

we would choose a best result among four results of four sub populations and output the best result. Figure 2 reveals the flow chart of this method. In this method, we realized task parallelism through parallelizing 'greedy algorithm' and 'population initialization' function. Besides, we also realized data parallelism through parallelizing four sub populations. However, since there is no communication between different sub populations, every sub population could only use its own good genes to generate new paths. Good genes are the good results of this sub population. In this way, we could only get the local optimal result. Consequently, we added 'migration' between different sub populations to solve this problem.

Parallel Method three. This method added 'migration' among different sub populations based on parallel method two. To add migration among different sub populations, we created a shared global list. After evaluation function and selection function, we would pick the top 25 best results of each sub population, and put these results into the shared global list. Then the shared global list should include 100 good results, then we would send this list to every sub population. In this way, each sub population could use these 100 good results to generate new paths instead of solely using own good results. In this way, we increased the number of good genes of each sub population, which could help each sub population output a better result. Figure 3 shows the flow chart of this method. The following code snippet shows how to use the 'Manager' library to create a shared global list, and how to use the 'Rlock' library to avoid race condition. Only the process acquired the lock could access to the shared global list and change it.

```
import multiprocessing as mp
def initial_population():
    '''Randomly generate 800 different path,
       store these paths in list A'''
    return A
def GA(lives,array,lock,q):
    '''Execute the genetic algorithm
    store the best path in list result'''
    lock.acquire()
    array.append(paths)
    lock.release()
    q.put(result)
if __name__ == '__main__':
    A = initial_population()
    A_1, A_2 = A[:400], A[400:]
    a = mp.Manager()
    array = a.list()
    lock = mp.RLock()
    q1 = mp.Queue()
    q2 = mp.Queue()
    p1 = mp.Process(target=func,
        args=(A_1,array,lock,q1,))
    p2 = mp.Process(target=func,
        args=(A_2,array,lock,q2,))
    p1.start()
    p2.start()
    p1.join()
    p2.join()
    res1 = q1.get()
    res2 = q2.get()
    '''Compare res1 and res2 and
    output the better one'''
```

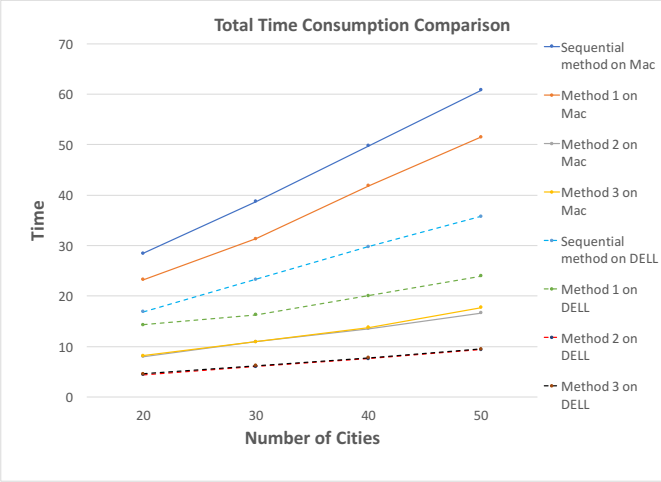

Fig. 4. Total time consumption comparison.

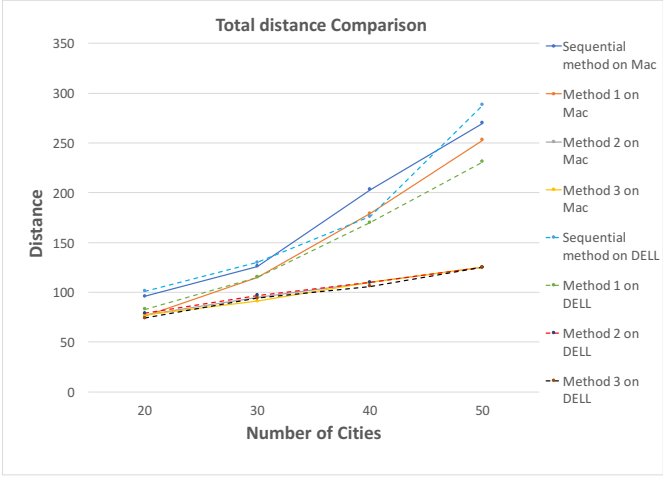

Fig. 5. Total distance comparison.

4. EXPERIMENTAL RESULTS

In this section, we will explain our experiments in detail. We will introduce the software and hardware we used and show you our inputs and outputs of the sequential method and three parallel methods. Finally, we will compare these methods based on their output results and time consumption.

Experimental setup. In our experiment, we used two different hardware equipment to test our algorithms for avoiding coincidence. The first hardware equipment we used is the DELL computer. The graphics is NVIDIA GeForce GTX 1080. The processor is Intel Core i7-8700, the processor speed is 3.2GHz x 12, the memory is 32GB and there are 6 cores and 12 threads. Another hardware equipment we used is MacBook pro. The processor of this hardware is Intel Core i7, the processor speed is 2.7GHz, the memory is 16GB and there are 4 cores and 8 threads.

As for the software system, we used python 3 to implement our sequential and parallel algorithm. Multiprocessing library of python was imported to parallelize the algorithm, and time module is imported to calculate each algorithms time consumption. In terms of the inputs, we would input the list of cities need to be visited, start location, and the distance adjacent matrix among different cities. We made a dataset which stored 50 cities coordinates, and the start locations coordinate is set to (0,0). After executing the algorithm, the algorithm would output the best path, the total distance of the path and the total time consumption of the algorithm. In this project, we did eight sets of experiments. We tested the four algorithms using four test cases where there are 20 cities, 30 cities, 40 cities and 50 cities. All the test cases were run in both hardware equipment.

Results. In our experiments, we used the result and time consumption of the sequential method to set the baseline. All three parallel methods would be used to compare with the sequential method. Figure 4 shows the total time consumption comparison among four methods and figure 5 shows the total distance comparison among these four methods. In these two figures, solid lines represent data got from MacBook Pro and dotted lines represent data got from DELL computer. Although data got from two computers are different, the ratios are similar. Consequently, tendency of solid lines and that of dotted lines are similar too. Based on the experimental results, we could found that method one does not work well. Compared with sequential method, it only improves a little on both total distance and time consumption. Fortunately, method two and method three improve a lot compared with sequential method and method one. Method two and method three could output a path with much shorter distance than that of sequential method. Besides, these two methods only consumed 1/4 time of sequential method. We used four cores to parallelize the algorithm and got four times speedup, which represents we nearly get a linear speed up. Comparing method two and method three, we could found method three could give a better path than method two. This proves that the 'migration' among different sub populations is important. Besides, we could also found that method three consume a little more time than method two. This is because it takes some to finish the interprocess communication.

5. CONCLUSIONS

In this paper, we discussed how to use genetic algorithm to sequentially solve travelling salesman problem, and we also introduced three different methods to parallelize genetic algorithm to improve the algorithms performance. In conclusion, our work is successful and important in three aspects.

- We realized both task parallelism and data parallelism in method two and method three.
- Our parallel methods work well. We used four cores to parallelize the genetic algorithm in method two and method three, and we approximately got four times speedup.
- Different with traditional parallel method using CUDA and OpenMP, we provided a new method to parallelize genetic algorithm using multiprocessing library of python.

6. FURTHER COMMENTS

In the future, we will improve the crossover and mutation functions of our genetic algorithm. In this project, we used the most basic method of crossover and mutation. In the crossover, we just randomly choose and combine two paths, and in the mutation, we also randomly choose two cities in a path and swap their position. Consequently, in the future, we will try some more complex crossover algorithm and mutation algorithm, and we could also parallelize two crossover algorithms in the crossover function. In addition, we would also try to use OpenMP and CUDA to implement our parallel method, and compare the results with this paper.

7. REFERENCES

[1] Mark S. Daskin Lawrence V. Snyder a, *, "A random-key genetic algorithm for the generalized traveling salesman problem," *European Journal of Operational Research*, , no. 174, pp. 38–53, 2005.

[2] Erick Cant-Paz, "Efficient and accurate parallel genetic algorithms," 2001.

[3] StoyanPenev Milena Karova, VassilSmarkov, "Genetic operators crossover and mutation in solving the tsp problem," *International Conference on Computer Systems and Technologies CompSysTech*, 2005.

[4] Plamenka Borovska, "Solving the travelling salesman problem in parallel by genetic algorithm on multicomputer cluster," *International Conference on Computer Systems and Technologies CompSysTech*, 2006.

[5] David E. Goldberg Abhishek Verma, Xavier Llor'a and Roy H. Campbell, "Scaling simple and compact genetic algorithms using mapreduce," *International Conference on Intelligent Systems Design and Applications*, 2009.

[6] Fan Yang, "Solving traveling salesman problem using parallel genetic algorithm and simulated annealing," 2010.

[7] Nadia Erdoan Harun Rait Er, "Parallel genetic algorithm to solve traveling salesman problem on mapreduce framework using hadoop cluster," *The International Journal of Soft Computing and Software Engineering*, 2013.

[8] Kshitij Pathak Chetan Chauhan, Ravindra Gupta, "Survey of methods of solving tsp along with its implementation using dynamic programming approach," *International Journal of Advance Research*, vol. 1, March 2013.

[9] Rohit Tanwar Khushboo Arora, Samiksha Agarwal, "Solving tsp using genetic algorithm and nearest neighbour algorithm and their comparison," *International Journal of Scientific Engineering Research*, January 2016.