Implementation of a Parallel Genetic Algorithm for Solving the Traveling Salesman Problem

Ali Abbas

Abstract—Traveling Salesman Problem (TSP) is of great interest in many fields of study due to its wide array of applications. Finding an exact solution to TSP is impractical, therefore we use a Parallel Genetic Algorithm (PGA) to find good solutions. We used an Island model of PGA, where each island is a thread that runs the basic GA on a sub-population, and the results are combined after all threads are finished. We used a Synchronous Dual Queue to handle the data exchange between threads that is needed for the migration between islands. The results of evaluating the implementation on 3 problems in TSPLIB showed that the implementation either found the optimal solution or at least found a good solution that was at most 3% worse than the optimal solution. We also showed that the speedup of multi-threading over sequential implementation was linear up to the number of physical CPU cores.

Index Terms—Parallel Genetic Algorithm, Island Model, Traveling Salesman Problem, Synchronous Dual Queue.

1 Introduction

Traveling Salesman Problem (TSP) is a well known NP-hard problem about finding the shortest tour between cities, such that the traveling salesman visits each city once and returns back to the starting city. It has many real world applications such as optimizing the distribution of logistics between multiple locations, navigation, robotic path finding, and optimizing integrated circuit design. Because TSP is an NP-hard problem, finding an exact solution for a large number of cities is not practical, so instead I will use genetic algorithms (GA) to find approximate solutions that can be good enough for practical applications.

Implementing GA sequentially wastes a lot of computational resources, since most modern day computers have multi-core processors that can potentially yield better performances if the implementation can be executed concurrently on multiple threads. I will use an island model of parallel GA to implement the concurrent version. In this model best individuals migrate to other islands. This has been studied extensively in the literature, for example Wang et al., (2020) implemented a parallel GA based on island model on GPU CUDA to solve the TSP [1].

Migration presents a challenge to a naive parallel implementation, since each thread needs to receive and send some data to other threads, which will require some form of synchronisation. I will implement a synchronous dual queue, which is a concurrent data structure that can be used for the migration.

The remainder of the paper is structured as follows. In section 2, the genetic algorithm is explained. Section 3 discusses the application of GA to TSP. In section 4 the island model of parallel GA, and Synchronous Dual Queue is presented. Section 5 presents the experimental results. Finally, section 6 concludes the paper.

2 Genetic Algorithm

Genetic algorithm (GA) is biologically inspired from the process of evolution in nature. In GA, a population consists of individuals where each individual is represented by its chromosome, which encodes all the relevant information about the individual in genes. Each gene can be viewed as encoding a single attribute of the individual, for example eye color, or height. Each chromosome can be represented by an array of integers, where each element of the array, represents a gene.

To simulate the process of evolution, we first start by creating a population of randomly generated individuals (chromosomes), where each individual is evaluated on how it is fitted to survive in the environment by applying a fitness function which assigns a fitness value to it. To simulate the process of mating, we use two genetic operators: crossover and mutation. To apply these operations first we need to select the individuals that will undergo the operation by applying a selection algorithm. There are different selection algorithms available, but here we will use a well known selection algorithm called tournament selection. In tournament selection, first we randomly select k (tournament size) individuals from the population and then with a probability p (tournament probability), the individual with highest fitness value is selected. To apply crossover, we select two individuals which are called parents, and exchange their genes according to a certain crossover operator, and produce two new individuals called children. To apply mutation, each gene of a child will be modified to a new value according to a certain mutation operator with probability determined by mutation rate. Children represent the next generation of the population, so we replace parents with children and repeat the whole process until a certain number of generations have passed. Figure 1 shows the flowchart of the basic genetic algorithm.

3 Applying Genetic Algorithm to the Traveling Salesman Problem

In order to find a solution to the TSP, we first need to represent a solution of the TSP as an individual (chromosome) in GA. We can assign a number to each city, and list the cities in the order that they are visited. Since

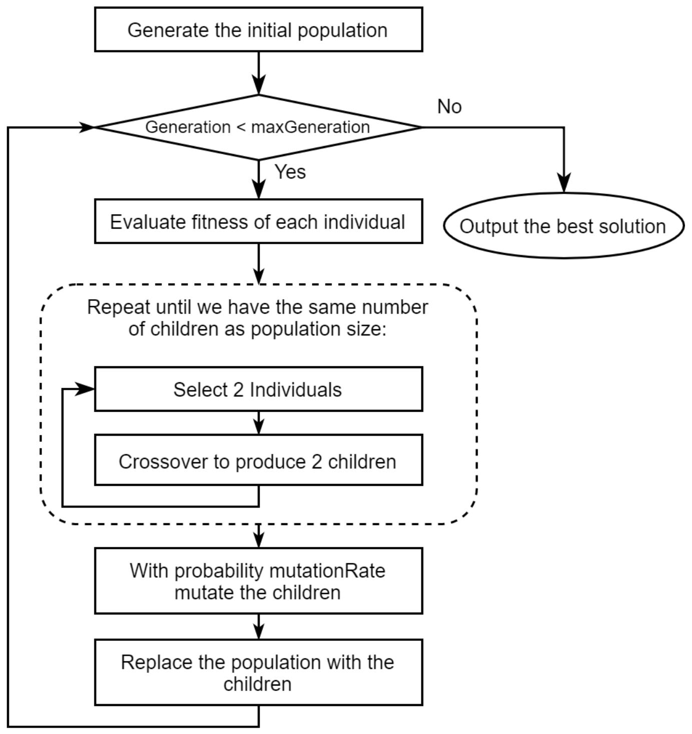
Generate the initial population
Generation < maxGeneration
No
Yes
Evaluate fitness of each individual
Output the best solution
Repeat until we have the same number of children as population size:
Select 2 Individuals
Crossover to produce 2 children
With probability mutationRate mutate the children
Replace the population with the children

Fig. 1. Genetic Algorithm Flowchart

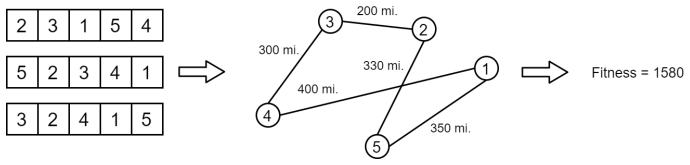

Fig. 2. Example of TSP encoded population and sample tour representation

the last visited city is also the first visited city, we only represent it once at the beginning of the list. For example the [5, 2, 3, 4, 1], represents this tour: 5 → 2 → 3 → 4 → 1 → 5. The fitness is calculated by summing all the distances along the tour. Figure 2 shows a sample population and tour.

For crossover, we use heuristic greedy crossover (HGreX) [2]. To perform HGreX on parent1 and parent2, we first select a random city c and add it to child 1. Let d1 be the distance between c and the next city cp1 in parent 1 (this means that cp1 is in the next location to c in parent 1). Likewise, let d2 be the distance between c and the next city cp2 in parent 2. Then we consider these four possibilities:

1) Both cp1 and cp2 has already been added to child 1: Select one unvisited city from each parent, and calculate the distance from these cities to the last added city in child 1. Add the closest city to the child 1.
2) cp1 has already been added to child 1: Add cp2 to child 1.
3) cp2 has already been added to child 1: Add cp1 to child 1.
4) Neither cp1, nor cp2 has been added to child 1: Calculate the distance from cp1 and cp2 to the last added city in child 1. Add the closest city to the child 1.

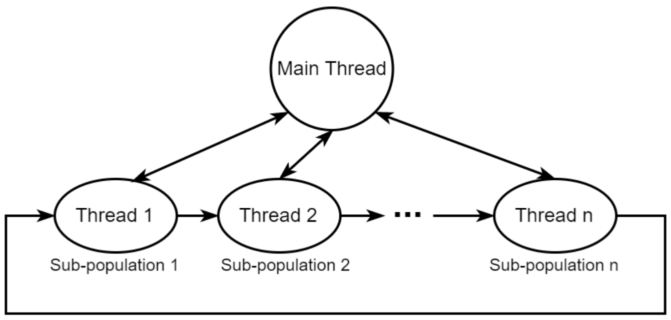

Fig. 3. Island Model of Parallel GA

For mutation, we use a heuristic mutation (HM) that was proposed by Vahdati et al., (2009) for solving TSP. It works by first selecting a random city c in the parent, then we find the closest city cp to the selected city and reverse the order of cities between those c and cp such that cp becomes the next city after c, and the city before cp becomes the second next city and so on (in this approach when we reach the end of an array we loop back to the beginning) [3].
To allow for more diversity in later generations where most individuals have converged and become very similar, we also use Exchange Mutation (EM) [4], which simply selects two cities in the list and swaps them. For example if cities 8 and 1 are selected from [5, 6, 8, 2, 7, 1, 3, 4], then the resulting chromosome after mutation will be [5, 6, 1, 2, 7, 8, 3, 4].
We always use HM in every generation, and after applying HM, we apply EM with probability determined by a variable mutation rate which is calculated in each generation as the maximum mutation rate * (generation / number of generations). This allows for more diversity in later generations where most individuals have converged and become very similar.

4 Parallel Genetic Algorithm

Genetic Algorithm can naturally be extended to run in parallel, since a population can easily be divided into sub-populations that can be distributed among different threads. One popular method of parallelizing GA is the island model approach, where each sub-population can be thought of living on an island and are evolving independently of other islands but every few generations, best individuals from an island will migrate to another island. The migration mechanism allows for the best solutions found to propagate through the whole population, while the isolation of islands helps in keeping the diversity of the solutions such that we are not trapped in local optima.
Figure 3. Shows the island model of Parallel GA (PGA), where each thread is running the basic GA (shown in Figure 1) with a sub-population that sends its two best individuals to the next thread every m generations, where m is the migration interval that determines how frequently the migration occurs. The migration occurs with a probability mr, where mr is the migration rate that determines the probability of a successful migration (there's a chance that

the migration fails and the best individuals are not sent to the next thread). The receiving thread replaces its worst two individuals with the received individual with probability mr, every m generations.

4.1 Synchronous Dual Queue

The migration of individuals between islands can be thought of as exchange of data between threads, which poses a serious challenge to the parallel implementation. For example thread A can be in its 5th generation when it needs the data from thread B which is in its 3rd generation. So thread A must wait for two generations until thread B can provide the data that it needs. A solution to this problem can be found using a variation of a synchronous data structure. One such data structure is Synchronous Dual Queue described in detail in [5].

Synchronous Dual Queue is a dual data structure that splits enqueue and dequeue methods into two steps: reservation and fulfillment. If a dequeuer tries to remove an item from an empty queue, instead of throwing an exception, it inserts a reservation object into the queue, then spins on the item in the reservation object until it's non-null, which means an enqueuer fulfilled the reservation by providing an item. Similarly, if an enqueuer does not find any reservations in the queue, it adds a new node, but instead of immediately returning, it spins on the item in the node until it becomes null, which means it was taken by a dequeuer. In this way, enqueuers and dequeuers can meet and sync with each other while exchanging data [5].

Synchronous Dual Queue as described in [5] is not suitable for implementing the migration, because first we have all the islands enqueue their data and wait for a dequeuer, but there are no dequeuers, since all threads are waiting in the enqueue stage. Therefore I modified the implementation to allow for the enqueue method to not spin waiting for a dequeuer, instead it returns immediately after adding a node or fulfilling a reservation. In this way, the enqueuer can run a head adding items and not worry about being in perfect sync with the dequeuer, because eventually the dequeuer will get all the items in order. So the only blocking method will be dequeue, because a thread still can't advance if it does not receive the data it needs, so it must wait for an enqueuer to provide them.

5 Experimental Results and Evaluation

To evaluate the implementation, I used the symmetric traveling salesman problem (TSP) samples from TSPLIB, which is a library of TSP problem samples with given optimal solutions for a subset of the problems [6]. I selected berlin52.tsp (52 locations in Berlin (Groetschel)), pr76.tsp (76-city problem (Padberg/Rinaldi)), and rd100.tsp (100-city random TSP (Reinelt)), which represent different levels of difficulty as number of cities grow from 52 to 76, and 100. Each file contains the coordinates of the cities that we used to calculate the distance matrix, that returns the distance between any two cities. I also calculated the optimal tour length from the given optimal tours in TSPLIB, which for berlin52 is 7544.365902, for pr76 is 108159.438274, and for rd100 is 7910.396210.

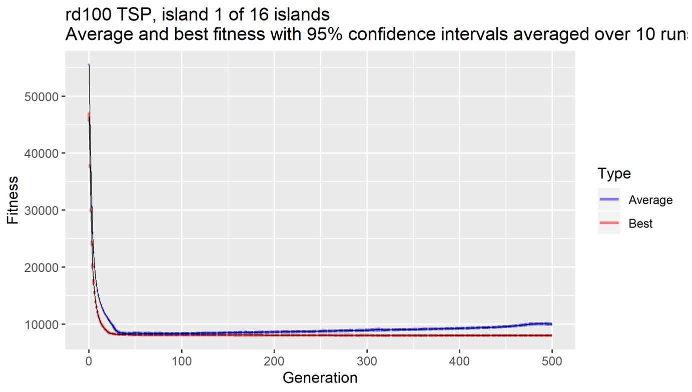

Fig. 4. Fitness vs Generation of thread 1 for rd100 problem with 16 threads

I ran the experiments on a Windows 10 Laptop with Intel Core i7-4720HQ CPU @ 2.6 GHz with 4 physical cores and hyperthreading, 16 GB RAM, and Nvidia GeForce GTX 970M Graphic Card. Java 8 was used to code the implementation.

To evaluate the effect of the number of threads on execution times, I varied the number of threads (number of islands) with these values: 1, 2, 4, 8, 16, and 32. Each problem with a given number of threads was executed 10 times such that I can obtain 95% confidence intervals on execution times to account for inevitable variation that occurs in concurrent programming, since we don't know how the threads will be scheduled. The following parameters was shared between all problems:

- Number of Runs: 10
- Tournament Size: 4
- Tournament Probability: 1
- Crossover Rate: 1
- Mutation Rate: 0.7
- Migration Interval: 50
- Migration Rate: 0.8
- Random Seed: 8512376

Random seed is used for the random number generator to guarantee that executing the program with the same parameters, always yields the same results. Number of generations was 500 for berlin52 and rd100, and was 200 for pr76. Population size was 100008 for pr76 and rd100, and was 10008 for berlin52.

For berlin52, 96.6% of all runs found the optimal solution. For pr76, only 5% of all runs found the optimal solution, but even the worst solutions were only 3% longer than the optimal, which means it's adequate for most practical purposes. For rd100, only 8.3% of all runs found the optimal solution, but even the worst solutions were only 2% longer than the optimal, which means it's adequate for most practical purposes. Figure 4 shows the fitness vs generations of thread 1 for rd100 problem with 16 threads. Results are averaged over 10 runs and 95% confidence intervals are added as well. This graph is very typical of all such graphs, where the population rapidly converges to a very good solution, but takes a long time to find the optimal solution, since the good solution is a local optima that traps the GA and makes it harder to find the global optima. As we can see the

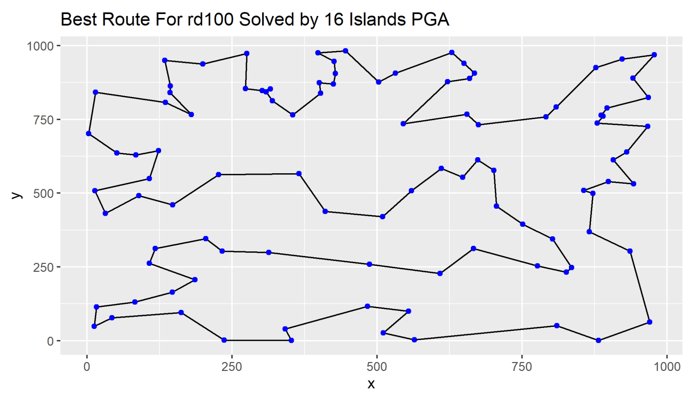

Fig. 5. Optimal solution for the rd100 problem, using 16 threads

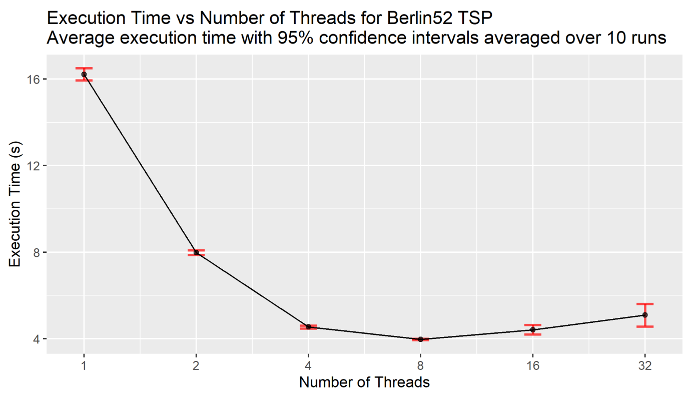

Fig. 6. Execution time vs number of threads for berlin52 problem

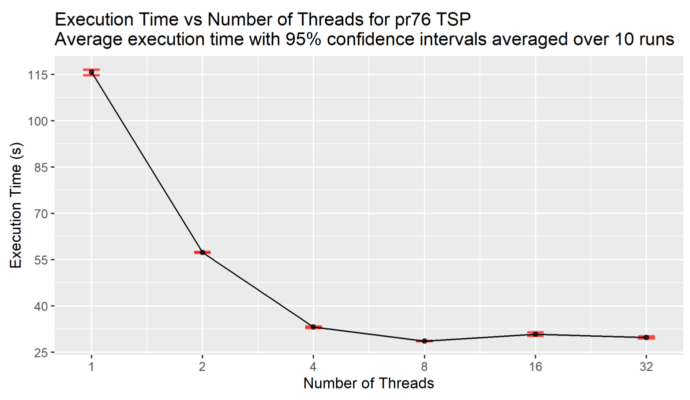

Fig. 7. Execution time vs number of threads for pr76 problem

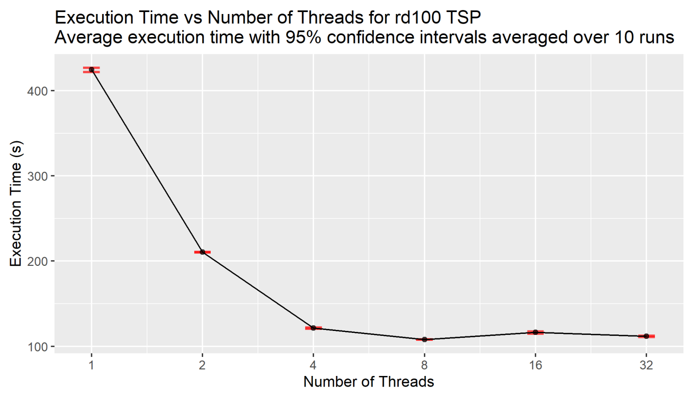

Fig. 8. Execution time vs number of threads for rd100 problem

average fitness becomes slightly worse, since we are using a variable mutation rate that introduces more variation as the number of generations increases.
Figure 5 Shows the optimal solution for the rd100 problem, using 16 threads.

Figure 6 Shows the execution time vs number of threads for the berlin52 problem. As we can see, up to the number of 4 physical CPU cores, the speedup is linear with the number of threads, ie. as the number of threads doubles, the execution time halves. Since we have hyperthreading in our CPU, we see a slight benefit from increasing the threads to 8, but beyond that, the execution time gets worse as the overhead of using more threads eliminates any benefits of having more threads. The same trend can be seen in Figure 7 and Figure 8 which show the execution time vs number of threads for the pr76 and the rd100 problems respectively.
Table 1 Shows the speedup over sequential run values for all the problems.

TABLE 1
Speedup of multithreading over sequential run

Number of Threads	berlin52	pr76	rd100
2	2.034101	2.017519	2.016330
4	3.574414	3.495511	3.496269
8	4.092739	4.043584	3.927391
16	3.678433	3.750353	3.650471
32	3.189116	3.889517	3.796263

6 Conclusion

The PGA with the island model was effective in solving berlin52, pr76, and rd100 problems of the TSPLIB. In all cases, either the optimal solution was found or the found solution was within the 3% margin of error of the optimal solution. In regards to the relation between execution time and the number of threads, we observed a linear relationship when number of threads were smaller than or equal to the number of physical CPU cores, after that up to the number of threads supported by hyperthreading, which was 8 for my CPU, there was a slight improvement, but beyond that no improvement was observed. Also the implementation of Synchronous Dual Queue worked very well and was able to synchronize the threads without any issues.

References

[1] B. Wang, H. Zhang, J. Nie, J. Wang, X. Ye, T. Ergesh, M. Zhang, J. Li, and W. Wang, "Multipopulation genetic algorithm based on gpu for solving tsp problem," *Mathematical Problems in Engineering*, vol. 2020, 2020.
[2] K. Puljić and R. Manger, "Comparison of eight evolutionary crossover operators for the vehicle routing problem," *Mathematical Communications*, vol. 18, no. 2, pp. 359–375, 2013.
[3] G. Vahdati, M. Yaghoubi, M. Poostchi *et al.*, "A new approach to solve traveling salesman problem using genetic algorithm based on heuristic crossover and mutation operator," in *2009 International Conference of Soft Computing and Pattern Recognition*. IEEE, 2009, pp. 112–116.

[4] P. Larranaga, C. M. H. Kuijpers, R. H. Murga, I. Inza, and S. Dizdarevic, "Genetic algorithms for the travelling salesman problem: A review of representations and operators," *Artificial Intelligence Review*, vol. 13, no. 2, pp. 129–170, 1999.

[5] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming, revised first edition*. Morgan Kaufmann, 2012.

[6] *TSPLIB*, Universität Heidelberg, 2013 (accessed October 14, 2020). [Online]. Available: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp/