

Parallelizing GA Based Heuristic approach for TSP

PROJECT REPORT

Submitted by

Tejas G Naik – 17BCE2093

Rohan Gupta – 17BCE2150

Kakshak Porwal – 17BCE0699

Course Code: CSE4001

Course Title: Parallel and Distributed Computing

Under the guidance of

Dr. S. Anto

Associate Professor, SCOPE,

VIT , Vellore.



VIT[®]
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

NOVEMBER, 2019

INDEX

1. Introduction	5
1.1. Genetic Algorithm	5
1.2. Travelling Salesman Problem	5
1.3. Genetic Algorithm approach to TSP	6
2. Literature Survey	7
3. Overview of the Work	9
3.1. Problem description	9
3.1.1 Various methods & algorithm of TSP	10
3.2. Software Requirements	12
3.3. Hardware Requirements	12
4. System Design	12
4.1. The genetic approach for solving TSP problem	12
4.2. Parallel genetic algorithm for TSP	13
4.3. Master-slave model	15
5. Implementation	17
5.1. Description of Modules/Programs	17
5.1.1 Initialisation	18
5.1.2 Selection	18
5.1.3 Crossover	19
5.1.4 Mutation	19
5.1.5 Modalities	19
5.2. Pseudocode	20
5.2.1 Serial Algorithm	20
5.2.2 Complexity and problem with serial approach	21
5.2.3 Parallelization Strategies	21
5.3. Source Code	22

5.3.1 Main TSPGEN.C	22
5.3.2 Essential functions implemented in project	25
5.4. Test Cases	30
5.4.1 Sample Test Case 1	30
5.4.2 Sample Test Case 2	33
5.5. Execution Snapshots	35
6. Conclusion	41
7. Future Scope	41
8. References	42

ABSTRACT

Travelling Salesperson Problem being a classic combinatorial optimization problem is an interesting but a challenging problem to be solved. It falls under the class of NP hard problem and becomes non-solvable for large data set by traditional methods like integer linear programming and branch and bound method, being the earlier popular approaches. Given the list of cities and distances between them, the problem is to find the shortest tour possible which visits all the cities in list exactly once and ends in the city where it starts. Despite the Traveling Salesman Problem is NP-Hard, a lot of methods and solutions are proposed to the problem. One of them is Genetic Algorithm (GA). GA is a simple but an efficient heuristic method that can be used to solve Traveling Salesman Problem. Genetic algorithm is powerful technique to discover optimal for traveling salesman problem (TSP). Genetic Algorithm based solutions emerged as the most popular tool to solve this which is a heuristic mechanism to find the closest approximate solution to the problem. In this project, we approach Travelling Salesman Problem using GA approach, a natural selection technique which repeatedly modifies a population of individual solutions, with the added power of modern computing systems. Here we come up with a parallel version of GA for both multicore and many core architectures in order to make some of the challenging problems like Vehicle Routing Problem for google maps, DNA sequencing and many more optimization problems involving a good amount of complex computation and data handling. The project presents a comparative analysis of the results for various degree and structure of graphs depicting close approximation to the accuracy in terms of most optimal path for traversal with highly reduced execution time.

Key words: Parallel Computing, TSP, Genetic Algorithms, Parallelism Profiling, Performance Estimation, MPI.

1 Introduction

1.1 Genetic Algorithm

A wide spectrum of application areas demands time-consuming computation for solving various optimization problems i.e. finding local extrema, more often global extrema or near-optimal solutions. One of the most efficient approaches for global optimization is to utilize genetic algorithms [1, 2]. Genetic algorithms are powerful and promising techniques for search problems to find out approximate solution to optimization. Genetic algorithms, being a particular class of evolutionary algorithms, imply the concepts of evolutionary biology (recombination, inheritance, natural selection, mutation). Genetic Algorithm is a biological method which helps evolution. It is used to solve constrained and unconstrained problems of optimization. They belong to the class of Evolutionary Algorithms. The concepts of natural evolution like selection, mutation and crossover is used for problem solving. Optimal solution is selected after successive generations. Therefore, it is highly suited for problems involving optimization and search-based solution to find the approximate solution. The most popular utilizations of GA are for computer simulation where candidates for an optimal solutions (individuals), presented abstractly by chromosomes, undergo evolution by performing recombination and mutation and the newly created populations tend to produce the optimal or near-optimal solution. The main goal of genetic algorithms is to search a solution space imitating the biological metaphor and, therefore, they are inherently parallel. Computer clusters [3, 4] are widely used nowadays for solving time-consuming problems. The applications of GA are computer simulation where subjects undergo evolution by repeated recombination and mutation, thus generating optimal or near optimal solutions.

1.2 Travelling Salesman Problem

The Traveling Salesman Problem (often called TSP) is a classic algorithmic problem in the field of computer science and operations research[1]. It is focused on optimization. In this context, better solution often means a solution that is cheaper, shorter, or faster. TSP is a mathematical problem. It is most easily expressed as a graph describing the locations of a set of nodes. The traveling salesman problem was defined in the 1800s by the Irish mathematician W. R. Hamilton and by the British mathematician Thomas Kirkman. Hamilton's Icosian Game was a recreational puzzle based on finding a Hamiltonian cycle.[2] The general form of the TSP appears to have been first studied by mathematicians during the 1930s in Vienna and at Harvard, notably by Karl Menger.

The travelling salesman problem (TSP) is a well-known combinatorial problem [5, 6, 7]. The idea of the TSP is to find the shortest tour of a group of cities without visiting any town twice, but, practically, it implies the construction of a Hamiltonian cycle within a weighted fully connected undirected graph. Therefore, this is a problem of combinatorial graph search. The TSP is maybe one of the most explored problems in computer science. The applications of the TSP problem are numerous – in computer wiring, job scheduling, minimizing fuel consumption in aircraft, vehicle routing problem, robot learning, etc. The purpose of this project is to investigate the performance of the parallel implementation of the TSP problem using genetic algorithms.

1.3 GENETIC ALGORITHM APPROACH TO TSP

Standard TSP involves a Hamiltonian Cycle of least weightage inside a fully connected undirected graph $G = (V, E)$. The possible number of paths in the naive algorithm is $n!$ and as the n increases, it becomes increasingly difficult to find the optimal solution in terms of time and space complexity. Therefore, by using Genetic Algorithm, we compare the number of cities with the number of chromosomes. Input of the Euclidean distances between any 2 cities is calculated. As Genetic Algorithm is a never-ending biological process, it does not have any fixed number of repetitions. Therefore, the number of iterations has been taken as input.

Random sample paths are calculated with the help of the cities.[5] They may or may not have optimal solution. A criteria/fitness function is applied to all the paths. Criteria = (Path with the max. distance + Path with min. distance)/2. Only the paths less than the criteria move on to the next iteration. This process continues for the specified number of iterations. Each iteration is followed by a mutation at the end which uses one-point crossover method to introduce new paths. This one-point crossover takes the first half of a path and joins it with another path to create a new second half of the original path.

2 Literature Survey

Genetic Algorithm is comprised of fitness evaluation, selection, crossover and mutation phases. The most time-consuming stage in the algorithm is fitness evaluation. Especially if the population size is very large, fitness evaluation turns into a big problem to be solved in sequential GA. Another problem encountered in a sequential genetic algorithm is that it sometimes gets stuck in a local search space. To overcome the problems described above, Parallel Genetic Algorithm (PGA) is used. Fitness evaluation problem can be solved using multi-core or super computers. If getting stuck in sub-optimal search space is the problem, PGAs provide an appropriate solution for the problem. The calculation of fitness evaluation, the use of single or multiple population, in case of multiple populations, the way individual exchange is carried out are some of the criteria according which PGAs are classified. PGAs are therefore classified into Master-slave, multiple populations with migration and multiple populations without migration. [1]

Bernd Reisleben and Peter Merz developed method for approximation optimal for symmetric and asymmetric TSP. This method combines the local search and genetic algorithm; local search find local optimal in search space and genetic algorithm find the local optimal space to discover global optimal. This method produce optimal solution in considerable time. [2]

The paper discusses various techniques for solving the interesting NP hard combinatorial problem TSP. The paper discusses in brief the exact and traditional algorithms to solve the problem. But due to computational limitations, heuristics to the approach were deduced. In this paper, we have discussed genetic algorithm as the heuristic approach to TSP solving, being the most popular among others. But solving these heuristics again have computational limitations and processing time is significantly high when the algorithm is executed over a single-core machine. So, the paper comes with the execution of the proposed parallel model over OpenMP for CPU based optimization and CUDA to achieve high performance efficiency over graphics processor. The experimental results show that there is a significant cut-down in the execution time of the algorithm over the two parallel algorithm execution models. CUDA being the API developed to handle heavy compute intensive tasks show better performance for the algorithm than OpenMP. [3]

Heuristic methods and genetic approaches are the most appropriate ways to solve the travelling salesman problem. Multiple optimal solutions can be obtained for this problem by using various combinations of selection, crossover and mutation techniques. We have surveyed many approaches and many combinations of genetic operators for this problem and the used combination of genetic operators in this paper is the optimized one among them. [4]

This paper proposed a new parallel approach in solving the TSP. It introduced a simple and effective solution that is clearly a valuable choice in comparison to classical approaches. By utilizing the power of parallel processing, the solution for the problem could be gotten faster while their quality seems to be better. According to the paper, many more mathematical problems could benefit from today's advances of technology, especially in computer science. There are still more rooms to develop our algorithm and we hope to contribute more in the future.[5]

The paper 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. Performance estimation and parallelism profiling have been made on the basis of MPI-based parallel program implementation. [6]

This paper addressed traveling salesman problem using the randomized biased genetic algorithm to discover optimal. RBGA uses elitist technique and bias selection of parent for mating to generate individual. RBGA is compared to GA w.r.t 30 benchmark traveling salesman problems over two criterion i.e average mean error and average mean execution time. As complete simulation results shown, RBGA perform better against GA for TSP. [7]

Four conceptually different PGA approaches and one hybrid of two of these approaches were compared using the TSP as an example application problem. For fair comparisons, all PGAs were developed based on the same baseline SGA, implemented on the same 16 thin PES of an IBM SP2 parallel machine, and tested using the same set of initial populations on the same set of TSP instances. Two techniques were identified as the most effective ones for improving the PGA performance: (1) inter-PE information exchange during the evolution progress (e.g., migration of the locally best chromosomes) and (2) tour segmentation and recombination. [8]

Parallel Genetic Algorithms have been successfully applied to solve some complex combinatorial optimization problems like Traveling Salesman Problem to reduce the required time to find a feasible solution. The use of Graphics Processing Units (GPUs) has become increasingly popular in last years, and they are not only used for graphical computations but also used for general purpose parallel computation. Therefore, they are increasingly used in high performance computing applications with parallel implementation. In this paper, it is aimed to develop a GPU based parallel genetic algorithm on TSP for decreasing solution time. Performance of implemented system is measured with the different size of population due to its relevance with the number of parallel executions on GPU. [9]

In this paper, an approach for approximately solving symmetric and asymmetric traveling salesman problems (TSP) has been presented. The approach is based on the combination of local search heuristics and genetic algorithms; local search techniques were used to efficiently find local optima in the TSP search space, and genetic algorithms were used to search the space of local optima to find the global optimum. The performance results presented for several symmetric and asymmetric TSP instances have shown that the approach is able to produce high quality solutions in reasonable time.[10]

3 Overview of work

3.1 Problem description

In nature, there exist many processes which seek a stable state. These processes can be seen as natural optimization processes. Over the last 30 years, several attempts have been made to develop global optimization algorithms which simulate these natural optimization processes. There are mainly three reasons why TSP has been attracted the attention of many researchers and remains an active research area. First, a large number of real-world problems can be modelled by TSP. Second, it was proved to be NP-Complete problem. Third, NP-Complete problems are intractable in the sense that no one has found any really efficient way of solving them for large problem size.[8] Also, NP complete problems are known to be more or less equivalent to each other; if one knew how to solve one of them one could solve the rest. The TSP finds application in a variety of situations such as automatic drilling of printed circuit boards and threading of scan cells in a testable VLSI circuit, X-ray crystallography, etc. The methods that provide the exact optimal solution to the problem are called Exact Methods. An implicit way of solving the TSP is simply to list all the feasible solutions, evaluate their

objective function values and pick out the best. However, it is obvious that this “exhaustive search” is grossly inefficient and impracticable because of vast number of possible solutions to the TSP even for problem of moderate size.[6] Since practical applications require solving larger problems, hence emphasis has shifted from the aim of finding exactly optimal solutions to TSP, to the aim of getting heuristically, ‘good solutions’ in reasonable time and ‘establishing the degree of goodness’. Genetic algorithm (GA) is one of the best algorithms that have been used widely to solve the TSP instances.

3.1.1 The various methods and algorithm for T.S.P

These are some of the various methods for solving travelling salesman problem developed by mathematicians and computer scientist.

- ✓ **The shortest path algorithm:** This approximately finds the shortest path all pairs of node or vertices on a weighted graph.
- ✓ **The simple Insertion Algorithms:** This is based on the branch- and -bound algorithm. It derives an n-city tour from an (n-1) city tour. This process is applied recursively to build up to modify and produce a complete n-city solution.
- ✓ **Genetic Algorithm:** The idea of genetic algorithm is to stimulate the way nature uses evolution to solve T.S.P
- ✓ **The Elastic Net Methods:** This is a kind of artificial neural network, which is used primarily for optimization problem.
- ✓ **The Simulated Annealing Algorithm:** This involves the simulation of a physical process of annealing which is then used to develop a software program to solve T.S.P
- ✓ **Ant Algorithm:** This tries to use real ant abilities to solve various optimization problems.

We have developed a solution to the Traveling Salesman Problem (TSP) using a Genetic Algorithm (GA). In the Traveling Salesman Problem, the goal is to find the shortest distance between N different cities. The path that the salesman takes is called a tour. Testing every possibility for an N city tour would be $N!$ math additions. 52 cities tour would have to measure the total distance of be 3.041×10^{64} different tours. Adding one more city would cause the time to increase by a factor of 51.[7] Obviously, this is an impossible solution. A genetic algorithm can be used to find a solution in less time. Although it might not find the best solution, it can find a near perfect solution for a 52-city tour in less than a minute. There are a couple of basic steps to solving the traveling salesman problem using a GA.[5]

- ✓ First, create a group of many random tours in what is called a population. This algorithm uses a greedy initial population that gives preference to linking cities that are close to each other.
- ✓ Second, pick 2 of the better (shorter) tours parents in the population and combine them to make 2 new child tours. Hopefully, these children tour will be better than either parent.

A small percentage of the time the child tours are mutated. This is done to prevent all tours in the population from looking identical. The new child tours are inserted into the population replacing two of the longer tours. The size of the population remains the same. New children tours are repeatedly created until the desired goal is reached. As the name implies, Genetic Algorithms mimic nature and evolution using the principles of **Survival of the Fittest**. [4]

These parameters are to control the operation of the Genetic Algorithm:

- ✓ **Population Size** - The population size is the initial number of random tours that are created when the algorithm starts. A large population takes longer to find a result. A smaller population increases the chance that every tour in the population will eventually look the same. This increases the chance that the best solution will not be found.
- ✓ **Neighbourhood / Group Size** - In each generation, this number of tours are randomly chosen from the population. The best 2 tours are the parents. The worst 2 tours get replaced by the children. For group size, a high number will increase the likelihood that the really good tours will be selected as parents, but it will also cause many tours to never be used as parents. A large group size will cause the algorithm to run faster, but it might not find the best solution.
- ✓ **Mutation %** - The percentage that each child after crossover will undergo mutation when a tour is mutated; one of the cities is randomly moved from one point in the tour to another.
- ✓ **Nearby Cities** - As part of a greedy initial population, the GA will prefer to link cities that are close to each other to make the initial tours. When creating the initial population this is the number of cities that are considered to be closed.
- ✓ **Nearby City Odds %** - This is the percent chance that any one link in a random tour in the initial population will prefer to use a nearby city instead of a completely random city. If the GA chooses to use a nearby city, then there is an equally random chance that it will be any one of the cities from the previous parameter.

- ✓ **Maximum Generations** - How many crossovers are run before the algorithm is terminated.
- ✓ **Random Seed** - This is the seed for the random number generator. By having a fixed instead of a random seed, you can duplicate previous results as long as all other parameters are the same. This is very helpful when looking for errors in the algorithm.
- ✓ **City List** - The downloadable version allows you to import city lists from XML files. Again, when debugging problems, it is useful to be able to run the algorithm with the same exact parameters.

3.2 Software Requirements

- ✓ Linux operating system (Ubuntu)
- ✓ MPI (Messaging Passing Interface)

3.3 Hardware Requirements

- ✓ CPU for fast computation
- ✓ 8 GB RAM
- ✓ i7 or i5 intel processor

4 System Design

4.1 The genetic approach for solving the TSP problem

The standard TSP may be presented mathematically as finding the Hamiltonian cycle of minimal weight within a weighted fully connected undirected graph $G = (V, E)$ where the vertices present the cities, the edges denote the intercity paths, and the weights of the edges represent the intercity distances. The deterministic method to solve the TSP problem involves traversing all possible routes, evaluating corresponding tour distances and finding out the tour of minimal distance. The total number of possible routes traversing n cities is $n!$ and, therefore, in cases of large values of n it becomes impossible to find the cost of all tours in polynomial time.[9] Parallel processing helps reducing the computation time of the TSP problem. Furthermore, our intention is to apply the genetic approach for finding the near-optimal solution of the TSP problem and to investigate the impact of the programming model on the performance of parallel system for that particular application.[7] The biological metaphor involves evolving populations where the size of the population is 40 and each individual is denoted by 40 chromosomes. Chromosomes present the order of cities in which the salesman will visit them. In our case permutation encoding is used where every chromosome is a string

of numbers that represent a position in a sequence. For the chromosome's permutation coding is used which is the best method for coding ordering problems. The fitness represents the length of the tour. The selection is performed following the rules of the roulette wheel method – the individuals of the highest fitness are selected for parents. The method of recombination is that of two crossover points – two parts of the first parent are copied, and the rest between these two parts is taken in the same order as in the second parent. The mutation applied is of the normal random type and involves changing of the city order.[5] The parallel genetic computation of the TSP on the cluster platform has been performed for 40 generations, the population size being 40, the genome has been represented by 10 bits, the probability of crossover happening used is 0.75, and the probability of each bit being mutated has been 0.1. After a new city is added or deleted it is necessary to create and evolve new populations presented by the corresponding chromosomes and the genetic algorithm is restarted.[3]

4.2 Parallel genetic algorithm for TSP

The algorithm, which is illustrated in Figure 3, starts with initializing m populations called pop_1 , pop_2 , ... and pop_m . These populations are fetched to separate computing threads. Afterwards, population evaluation process is carried out to estimate each candidate's fitness in the populations mentioned above. After this procedure, if the global stop condition is met, the algorithm will end. Otherwise, the next steps are taken. It is necessary to mention that the global stop condition is fulfilled when the fitness threshold is met or when the number of exchanging among slaves in the cluster exceeds a predefined value. The usage of populations is clearer from now on. We implement a computer cluster including m workstations. Each one will serve a thread and do the same job as each other.[2] The previously generated populations are distributed into m computers: pop_1 into computer 1, pop_2 into computer 2, ... and pop_m into computer m , respectively. In the next stage, selection, crossover and mutation phases are processed for a limited number of runs. This number of iterations is predetermined and is called local stop condition. The migration condition is simply comparing the iteration of each loop with this value. If a thread finishes sooner than the others, the migration procedure will wait for the rests to finish before proceeding. The purpose of this procedure is to select the best solution from m threads for broadcasting back to the population evaluation phase.[3]

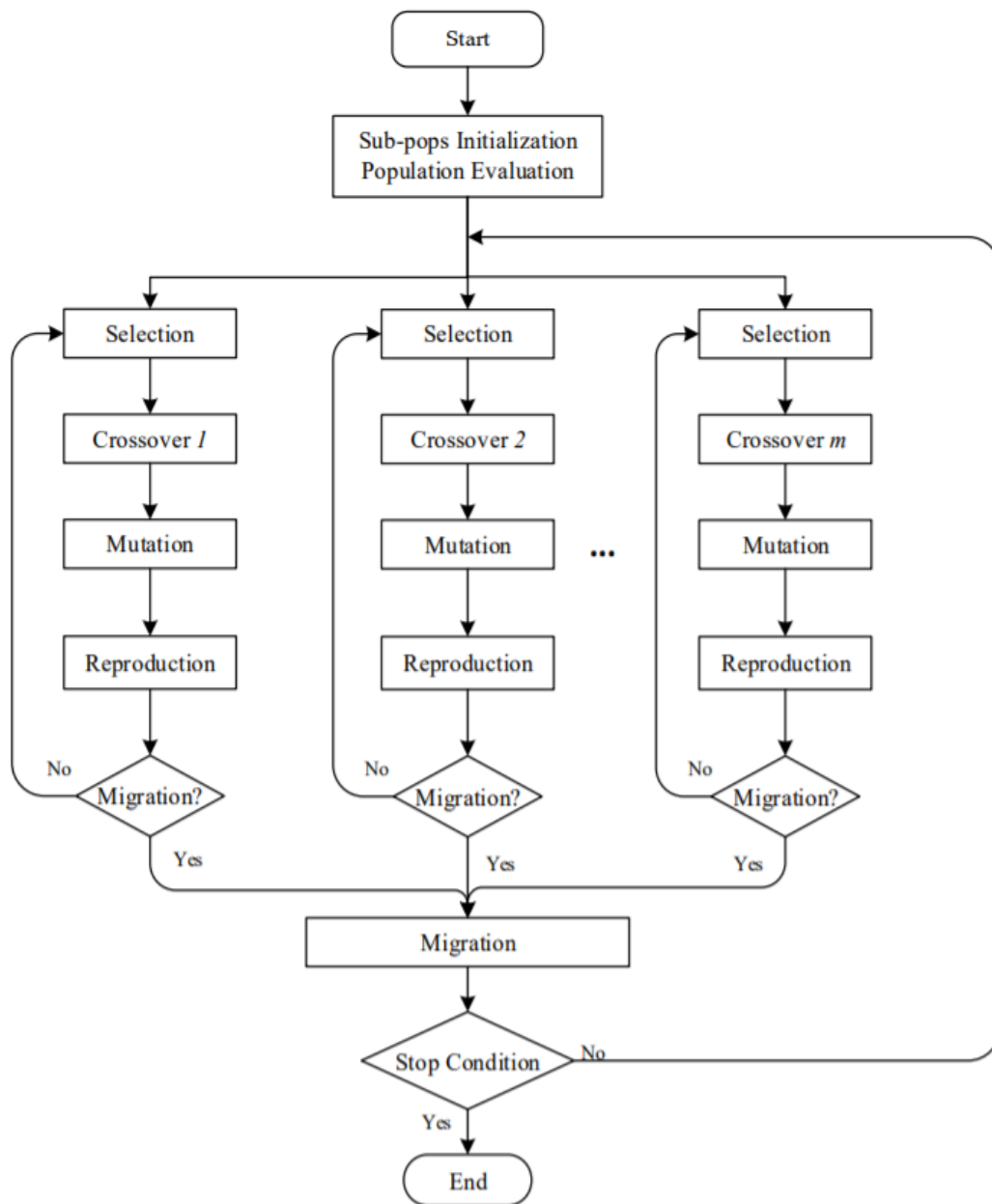


Figure 1. Flow of Parallel Genetic Algorithm

4.3 Master-Slave Model

Master-Slave Model (MSM) is essentially an architecture to increase speed, scale and computing power. For GAs, distribution of crossover and mutation operations, and in some cases fitness calculation, can be done to different processors [9]. This approach is especially appropriate for today's context where multicore, multiprocessor and distributed systems are available nearly everywhere. What still needs to be done is utilizing the computing power of those facilities to solve the problem faster, more exactly and more efficiently.

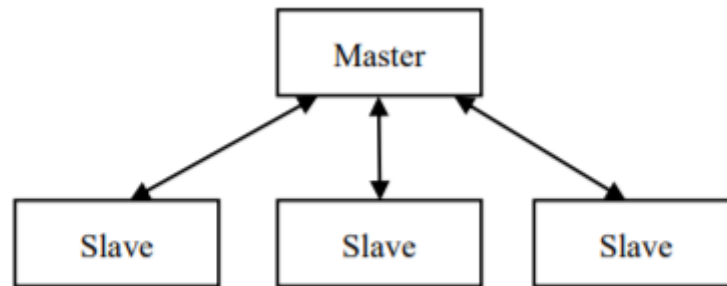


Figure 2. Master-Slave Model

Firstly, MSM assigns a fraction of the population to the available processors/computer systems for evolutionary operation. From this time, it can work in two methods. The first one is synchronous mode. Once the population is assigned to the processors, synchronous MSM waits for all the processors to complete their operations and return the result before evaluating the new population [9]. Meanwhile, the second one is asynchronous, which does not wait for slow processors to return their results.

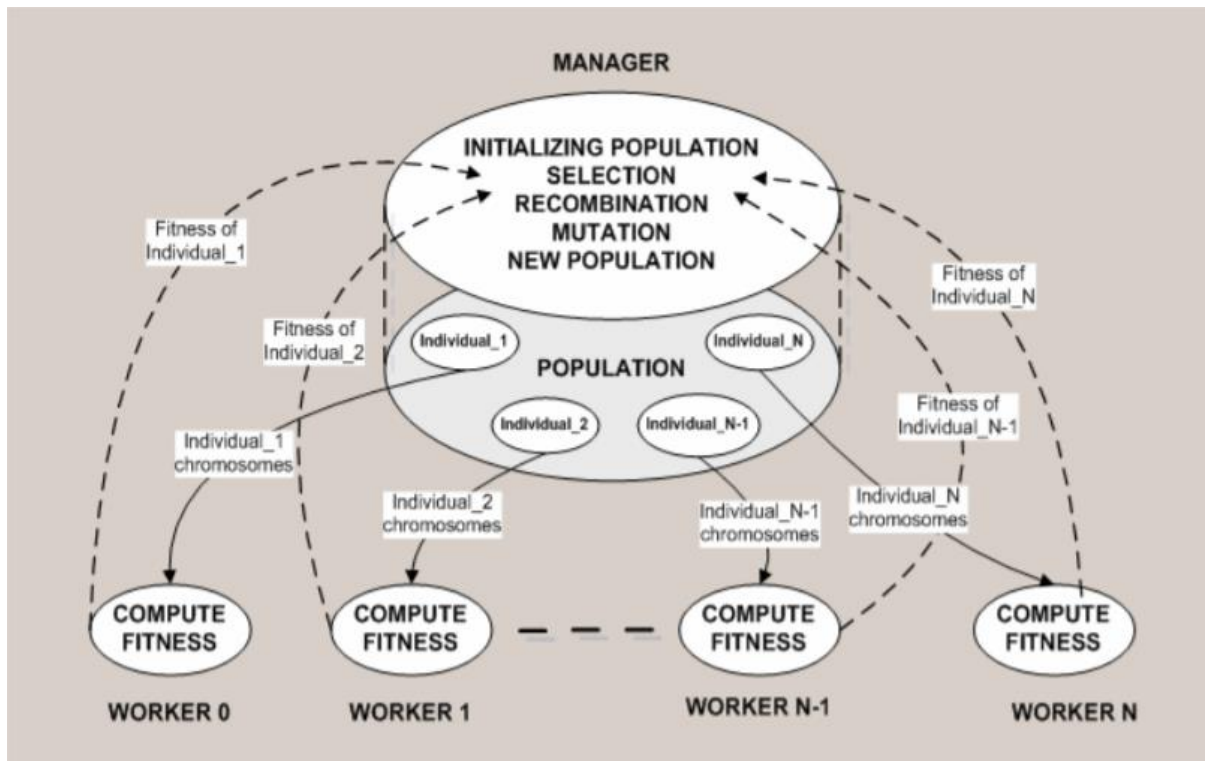


Figure 3 Master slave model of computing the TSP by genetic algorithm

The master process (rank 0) performs all genetic operations and distributes the computational load among the worker processes. It performs the following activities:

1. Initializes population
2. Sends a genome to be evaluated; Distributes the individuals among the workers by sending the chromosomes of the individuals (MPI_Send) to the workers to evaluate their fitness.
3. Receives (MPI_Irecv) the evaluated fitness from workers
4. Evaluates the fitness of the entire population
5. Performs roulette wheel selection
6. Performs recombination (selects 2 points for crossover)
7. Performs normal random mutation (two cities are chosen and exchanged)
8. Generates new population
9. In case the number of the current generation is less than the maximum number - to 2
10. In case the number of the current generation equals the maximum number sends a termination message to all workers
11. Prints the computed shortest path.

The operations of a worker process are as follows:

1. Signals the master that it is ready for work
2. Waits for genomes to be sent by the manager until a termination message is received
3. Evaluates the fitness of each chromosome of the individual
4. Sends the computed fitness of the individual to the manager
5. Receives termination message from the manager

The master sends the chromosomes to the workers by means of synchronous send transaction (MPI_Send) because of the insignificant delay. Once the master has sent the genomes to all the worker processes it waits for them to finish by executing the MPI function MPI_Waitall. The master gets the computed fitness values from the workers through a nonblocking communication transaction (MPI_IRecv) using request for keeping track of the completion of the communication transaction.[4] Performing the communication transactions requires sending the size of the array being transferred, the type of the elements, the source of the message, the specific tag denoting the message, and the communicator specifying the group of processes taking part in the particular communication transaction (MPI_COMM_WORLD). The first thing to do by the workers is to inform the master that they are ready for work. Throughout the computational process they are waiting for genomes from the master and get them by means of synchronous receive operation (MPI_Recv). Worker processes die upon receiving a terminating message from the master.[9]

5 Implementation

5.1 Description of modules/programs

The exact application involved finding the shortest distance to fly between eight cities without visiting a city more than once. The table below shows the distances between each city in kilometres. The search space for this problem involves every possible permutation of routes that visit each city once. As there are eight cities to be visited, and because once a city has been visited that city is not eligible to be travelled to again, the size of the search space stands as $8!$ (40320) possible permutations.[6]

5.1.1 Initialisation

Upon initialisation, each individual creates a permutation featuring an integer representation of a route between the eight cities with no repetition featured. A corresponding array with the string equivalent of these indexes is created to output when a solution is found. A fitness function calculates the total distance between each city in the chromosome's permutation. For example, in the ordering above, the distance between the cities represented by '0' and '4' is added to an overall sum, then the distance between the cities represented by '4' and '1' is added, and so on. The chromosome's fitness is set to the overall sum of all distances within the permutation. The smaller the overall distance, the higher the fitness of the individual.[9]

5.1.2 Selection

There are two popular methods to apply when performing selection in genetic algorithms, roulette wheel selection and tournament selection. In our first implementation, we used the roulette wheel method to select two parents to produce two offspring within each generation. Each chromosome calculates a selection probability with a higher fitness correlating to a higher probability. An overall sum is created of the fitnesses of each chromosome within the population. Each chromosome's probability is then calculated by dividing it's individual fitness by the total probability sum which results in a float value in the range of 0 to 1. An array is created to store the individual probabilities and act as the roulette wheel in which to determine the parents.[7]

To select the parents, a random number is generated between 0 and 1. The number is then tested against the array values to test which two indexes it lies between and therefore which chromosome to select as a parent. This process is iterated twice to select two parents and a check is in place to guarantee that the two parents are different. Below is a representation of the probabilities generated by 5 different chromosomes based upon their individual fitness. On the right is the representation of the array in which to test the randomly generated number against. The first value always serves as 0, the next as $0 + \text{chromosome1.probability}$, the next as $\text{chromosome1.probability} + \text{chromosome2.probability}$, and so on for all entries.

Tournament selection also involves this process of calculating probability bias however features a stronger stochastic element in choosing parents. The tournament consists of considering only a select few chromosomes as parents rather than the whole population. The size of the tournament is assigned at the beginning of the program, and the chromosomes are

chosen from a random starting point to create a tournament of the set size. The same process as the roulette wheel is used to determine the two parents for the crossover phase.

5.1.3 Crossover

As the solution requires that no city to be visited more than once, using a classical crossover operator may often lead to generating weaker offspring. A conventional crossover operator may select one half to copy from the first section and the remaining half to copy from the second. This does not prevent copying duplicate cities into the offspring chromosome and will result in a penalty for visiting the same city more than once. However, using the order 1 approach helps to preserve the non-repeating feature of the parent chromosomes[1]. To create the first child, copy part of the first parent's chromosome to the child's chromosome. Then choose valid non-repeating numbers from the second parent in the order that they appear to the empty values in the child's chromosome. To create the chromosome for the second child, repeat this process inversely by copying part of the second parent's chromosome and using valid values from the first parent for the remaining values.

5.1.4 Mutation

For this specific problem, the standard mutation action is modified to avoid creating repetition of cities visited. A conventional mutation method of replacing a value with a random city would duplicate cities and not fulfil the problem's criteria. Instead, two random indexes are selected in the array holding the city indexes and these two values are swapped to maintain.

5.1.5 Modalities

Genetic algorithms have two modalities, steady-state and generational. Steady-state utilises an elitist selection process in which the best n chromosomes are of the population are carried over to the next generation. By keeping the best-ranked chromosome, this implementation does not risk losing its best solution so far. Generational does not utilise this approach and instead only carries over any offspring produced in the crossover phase.[8]

The four variations are noted in the results as:

- Steady-State Approach with Roulette Wheel Selection (SS-RW)
- Steady-State Approach with Tournament Selection (SS-T)
- Generational Approach with Roulette Wheel Selection (G-RW)
- Generational Approach with Tournament Selection (G-T)

Each variation was tested 50 times and the minimum, maximum, and average number of generations needed to reach the solution was recorded. The same mutation rate of 0.15 and population size of 25 was used for each implementation. The stochastic aspect of the mutation threshold and selection process removes consistency from the performance of GA. Another fifty tests may yield entirely different results for each variation that may be more expected or entirely contradict what would be logically assumed. However, if these results do accurately describe the behaviour of the variations, the steady-state approach and tournament selection method may benefit in more creative applications, where exploration and a slow convergence may demonstrate an auditory or artistic process. Overall, considering the total size of the search space mentioned in the introduction, the genetic algorithm serves well in finding a solution in a relatively small number of generations.

The Travelling Salesman Problem (often called, TSP) is a classic algorithmic problem in the field of computer science. It is focused on optimization. In this context, better solution often means a solution that is cheaper. TSP is a mathematical problem. It is most easily expressed as a graph describing the locations of a set of nodes. TSP can be modelled as an undirected weighted graph, such that cities are the graph's vertices, paths are the graph's edges, and a path's distance is the edge's length. It is a minimization problem starting and finishing at a specified vertex.

5.2 Pseudocode

5.2.1 Serial Algorithm:

```

push(stack, tour)
while stack is non empty
    tour ← pop(stack)
    if citycount (tour) = n
        if besttour(tour)
            updatebesttour(tour)
    else for b = n – 1 downto 1
        if feasible (tour,b)
            add(city , tour)
            push(stack, tour)
            removelast(tour, city )

```

5.2.2 Complexity and problem with serial approach

The problem is a famous NP hard problem. There is no polynomial time know solution for this problem.

The complexity of the algorithm is $O(n!)$ where n is the total number of cites or number of vertices in an undirected graph.

- 1) Consider city 1 as the starting and ending point.
- 2) Generate all $(n-1)!$ Permutations of cities.
- 3) Calculate cost of every permutation and keep track of minimum cost permutation.
- 4) Return the permutation with minimum cost.

When we increase the problem size by increasing the number of nodes in a graph the complexity increases factorially as the number of tours is equal to $n!$ And the cost for overall computation becomes expensive.

5.2.3 Parallelization Strategies

There are primarily two strategies for parallelizing the algorithm. Both the strategies will parallelize the Tree Search. One would use a common stack and each process will do computations for the tour on the top of the stack until the stack is empty. The second one will parallelize the Tree search by initially splitting the stack into parts depending on the number of threads and each process will work on its own stack.

The first strategy is to parallelize the computations on different tours while using the same stack.

```
#pragma omp parallel private(tour) reduction(min:minCost)
    while(!isEmpty(stack))
        #pragma omp critical
            tour ← pop(s)
            if citycount (tour) = n
                if besttour(tour)
                    updatebesttour(tour)
            else for b = n - 1 downto 1
                if feasible (tour,b)
                    add(city , tour)
                    #pragma omp critical
                    push(stack, tour)
                    removelast(tour, city )
```

The second strategy splits the main stack and distributes it equally among the given number of threads. So, each process has a local stack which is part of the main stack and it does the computations only on this stack. No sharing of the main stack.

```
#pragma omp parallel private(tour) reduction(min:minCost)
```

```
    stack s1=split_stack(main_stack,rank)
    while(!isEmpty(s1))
        tour←pop(s1)
        if citycount (tour) = n
            if besttour(tour)
                updatebesttour(tour)
        else for b = n – 1 downto 1
            if feasible (tour,b)
                add(city , tour)
                push(s1, tour)
                removelast(tour, city)
```

5.3 Source Code

5.3.1 Main TSPGEN.C

```
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <stdio.h>
#include "mpi.h"
#include "common.h"
#include "map.h"
#include "population.h"
int mpiId = 0;
int main(int argc, char **argv){
    tspsMap_t map;
    tspsConfig_t config;
    tspsPopulation_t population;
    unsigned long int numGenerations = 0;
    int mpiNumProcs = 0;
    double start, end;
```

```

//starting MPI directives
MPI_Init(NULL,NULL);
MPI_Comm_size(MPI_COMM_WORLD,&mpiNumProcs);
MPI_Comm_rank(MPI_COMM_WORLD,&mpiId);
logg("* Starting tspgen...\n");
// parse the command line args
if(readConfig(&config, argc, argv) != TSPS_RC_SUCCESS){
    return TSPS_RC_FAILURE;
}
// parse the map
logg("* Parsing map...\n");
if(parseMap(&map) != TSPS_RC_SUCCESS){
    logg("Error! Unable to read map 'maps/brazil58.tsp'!\n");
    return TSPS_RC_FAILURE;
}
// initialize random seed:
srand ( time(NULL)*mpiId );
logg("* Generating population...\n");
if(generatePopulation(&population, &config) != TSPS_RC_SUCCESS){
    logg("Error! Failed to create a new random population!");
    return TSPS_RC_FAILURE;
}
// start a timer (mpi_barrier + mpi_wtime)
MPI_Barrier(MPI_COMM_WORLD);
start = MPI_Wtime();
logg("* Initializing reproduction loop...\n");
while(1){
    numGenerations++;
    calculateFitnessPopulation(&population, &map);
    if(numGenerations % 500 == 0){
        logg("- Generation %d...\n", numGenerations);
        printIndividual(&population.individuals[0], "Current Top Individual");
    }
    sortPopulation(&population);
}

```

```

    if(config.numGenerations > 0 && numGenerations == config.numGenerations){
        logg("* Max number of generations [%d] reached!\n", config.numGenerations);
        break;
    }
    crossoverPopulation(&population, &config);
    mutatePopulation(&population, &config);
    // migrate population at every n generation
    if(numGenerations % config.migrationRate == 0){
        migrateIndividuals(&population, mpiId, mpiNumProcs, &config);
    }
}
// join all the populations
joinPopulations(&population, mpiId, mpiNumProcs);
sortPopulation(&population);
// get the best inidividual and print it
printIndividual(&population.individuals[0], "Top Individual");
printIndividual(&population.individuals[config.populationSize-1], "Worst (of the top ones)
Individual");
// stop the timer
MPI_Barrier(MPI_COMM_WORLD); /* IMPORTANT */
end = MPI_Wtime();
if(mpiId == 0) { /* use time on master node */
    printf("* Runtime = %f\n", end-start);
}
logg("* tspgen finished!\n");
free(population.individuals);
free(map.nodes);
MPI_Finalize();
return TSPS_RC_SUCCESS;
}

```


5.3.2 Essential functions implemented in project

//Each process generates its own sub-population using the pseudo random number generator

```
int generatePopulation(tspsPopulation_t *pop, tspsConfig_t *config)
{
    pop->numIndividuals = config->populationSize;
    pop->individuals = (tspsIndividual_t*)malloc(pop->numIndividuals *
sizeof(tspsIndividual_t));
    int i;
    for(i=0; i<pop->numIndividuals; i++){
        generateRandomChromosome(&pop->individuals[i], i);
    }
    return TSPS_RC_SUCCESS;
}
```

//Generates a new random chromosome

```
int generateRandomChromosome(tspsIndividual_t *ind, int index){
    int ii;
    for(ii=0; ii<NUM_NODES; ii++){
        ind->chrom[ii] = ii; // city index starts from zero
    }
    struct timeval tv;
    gettimeofday(&tv, NULL);
    int usec = tv.tv_usec+index;
    srand48(usec);
    size_t i;
    for (i = NUM_NODES - 1; i > 0; i--) {
        size_t j = (unsigned int) (drand48()*(i+1));
        int t = ind->chrom[j];
        ind->chrom[j] = ind->chrom[i];
        ind->chrom[i] = t;
    }
    return TSPS_RC_SUCCESS;
}
```

//This function calculates the fitness value of each chromosome the parsed map stored in the directory

```
double calculateFitnessChromosome(int *chromosome, tspsMap_t *map){
    int fitnessValue = 0.0;
    int i, firstCity, secondCity;
    int xd, yd;
    for (i=0; i<NUM_NODES-1; i++){
        firstCity = chromosome[i];
        secondCity = chromosome[i+1];
        xd = map->nodes[firstCity].x - map->nodes[secondCity].x;
        yd = map->nodes[firstCity].y - map->nodes[secondCity].y;
        fitnessValue = fitnessValue + rint(sqrt(xd*xd + yd*yd));
    }
    return fitnessValue;
}
```

```
void swap (int *a, int *b){
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

```
int compare (const void *a, const void *b)
{
    tspsIndividual_t * popA = (tspsIndividual_t *)a;
    tspsIndividual_t * popB = (tspsIndividual_t *)b;
    return ( popA->fitness - popB->fitness );
}
```

//Mutate the population

```
int mutatePopulation(tspsPopulation_t *pop, tspsConfig_t *config){
    tspsIndividual_t *ind = NULL;
    int alreadySwaped[NUM_NODES];
    int mutationRate = config->mutationRate * 100;
    int index1, index2;    //the index of the nodes to be swapped
    int aux;
    int i, j;
```

```

for(i=config->numElitism; i<pop->numIndividuals; i++){
    if(rand()%100 > mutationRate)
        continue;
    memset(alreadySwaped, 0, sizeof(alreadySwaped));
    ind = &pop->individuals[i];
    //mutate!
    //swap mutationSize nodes in the chromosome
    for(j=0; j<config->mutationSize; j++){
        index1 = rand() % NUM_NODES;
        //if already swaped, jump to the next of the list
        while(alreadySwaped[index1] !=0){
            if(index1 + 1 < NUM_NODES)
                index1++;
            else
                index1 = 0;
        }
        alreadySwaped[index1] = 1;
        index2 = rand() % NUM_NODES;
        //if already swaped, jump to the next of the list
        while(alreadySwaped[index2] !=0){
            if(index2 + 1 < NUM_NODES)
                index2++;
            else
                index2 = 0;
        }
        alreadySwaped[index2] = 1;
        //swap the nodes
        aux = ind->chrom[index1];
        ind->chrom[index1] = ind->chrom[index2];
        ind->chrom[index2] = aux;
    }
}

```

```

    return TSPS_RC_SUCCESS;
}
//sort the population by increasing fitness
int sortPopulation(tspsPopulation_t *pop){
    qsort(pop->individuals, pop->numIndividuals, sizeof(tspsIndividual_t), compare);
    return TSPS_RC_SUCCESS;
}
// selection of two parents based on sampling from the multinomial distribution
rndNumber_one = rand() / (double) RAND_MAX;
rndNumber_two = rand() / (double) RAND_MAX;
//count = count+2;
for (i = 0; i < config->populationSize; i++)
{
    offset_one += pop->individuals[i].probability;
    if (rndNumber_one < offset_one)
    {
        pick_one = i;
        break;
    }
}
for (i = 0; i < config->populationSize; i++)
{
    offset_two += pop->individuals[i].probability;
    if (rndNumber_two < offset_two)
    {
        pick_two = i;
        break;
    }
}
offset_one=0; offset_two=0;
//swapping the segment of chromosome between the two delimiters
for (i = cross_pone; i < cross_ptwo; i++) {
    int tem = child_1[i];
    child_1[i] = child_2[i]; //

```

```

        child_2[i] = tem; //
        vec_1[num] = child_1[i];
        vec_2[num] = child_2[i];
        num++;
    }
    //copy rest of the segment from parents to children
    for(i = 0; i<cross_pone; i++){
        child_1[i] = pop->individuals[pick_one].chrom[i];
        child_2[i] = pop->individuals[pick_two].chrom[i];
    }
    for(i = cross_ptwo; i<NUM_NODES; i++){
        child_1[i] = pop->individuals[pick_one].chrom[i];
        child_2[i] = pop->individuals[pick_two].chrom[i];
    }
    int buf, flag=0;
    //solving the conflicts that arise due to above swapping procedure.
    for (i = 0; i < num; i++ ){
        if(flag==1){
            i=i-1;
            flag=0;
        }
        for (j = 0; j < num; j++ ){
            if((vec_1[i]==vec_2[j])&&(vec_1[i]!=vec_2[i] )){
                buf = vec_1[i];
                vec_1[i] = vec_1[j];
                vec_1[j] = buf;
                flag=1;
                continue;
            }
        }
    }
    for (i=0; i<num; i++){
        for(j = 0; j<cross_pone; j++){
            if(vec_1[i]==child_1[j]){

```

```

        child_1[j] = vec_2[i];
    }
    if(vec_2[i]==child_2[j]){
        child_2[j] = vec_1[i];
    }
}

for(j = cross_ptwo; j<NUM_NODES; j++){
    if(vec_1[i]==child_1[j]){
        child_1[j] = vec_2[i];
    }
    if(vec_2[i]==child_2[j]){
        child_2[j] = vec_1[i];
    }
}
}

// replacing the worst candidates in the previous generations with the new children.
for (i=0; i<NUM_NODES; i++){
    pop->individuals[config->populationSize-count-1].chrom[i] = child_1[i];
    pop->individuals[config->populationSize-count-2].chrom[i] = child_2[i];
}

free(vec_1);
free(vec_2);

```

5.4 Test Cases

5.4.1 Sample Test Case 1

Input Berlin Map

NAME: berlin52
TYPE: TSP
COMMENT: 52 locations in Berlin (Groetschel)
DIMENSION: 52
EDGE_WEIGHT_TYPE: EUC_2D
NODE_COORD_SECTION
1 565.0 575.0
2 25.0 185.0
3 345.0 750.0
4 945.0 685.0
5 845.0 655.0
6 880.0 660.0

```
7 25.0 230.0
8 525.0 1000.0
9 580.0 1175.0
10 650.0 1130.0
11 1605.0 620.0
12 1220.0 580.0
13 1465.0 200.0
14 1530.0 5.0
15 845.0 680.0
16 725.0 370.0
17 145.0 665.0
18 415.0 635.0
19 510.0 875.0
20 560.0 365.0
21 300.0 465.0
22 520.0 585.0
23 480.0 415.0
24 835.0 625.0
25 975.0 580.0
26 1215.0 245.0
27 1320.0 315.0
28 1250.0 400.0
29 660.0 180.0
30 410.0 250.0
31 420.0 555.0
32 575.0 665.0
33 1150.0 1160.0
34 700.0 580.0
35 685.0 595.0
36 685.0 610.0
37 770.0 610.0
38 795.0 645.0
39 720.0 635.0
40 760.0 650.0
41 475.0 960.0
42 95.0 260.0
43 875.0 920.0
44 700.0 500.0
45 555.0 815.0
46 830.0 485.0
47 1170.0 65.0
48 830.0 610.0
49 605.0 625.0
50 595.0 360.0
51 1340.0 725.0
52 1740.0 245.0
EOF
```

Output

```
kakshak@kakshak-HP-Notebook:~/Desktop/tspgen-genetic-algorithm$ mpiexec -n 2 ./tspgen
* Starting tspgen...
* Config:
-- Population Size = [5000]
-- Mutation Rate = [0.100000]
-- Num of Generations = [10000]
-- Num of Elitism = [100]
-- Mutation Size = [2]
-- Migration Rate = [500]
```

```

-- Migration Share = [0.010000]
* Parsing map...
* Generating population...
* Initializing reproduction loop...
- Generation 500...
** Current Top Individual
-- fitness = [15716]
-- chromos = [28][11][3][15][49][29][38][9][8][22]
             [19][20][6][1][41][48][14][42][35][45]
             [46][25][27][51][13][12][5][50][10][26]
             [32][33][2][18][7][40][44][36][21][31]
             [24][23][47][0][17][30][37][43][4][39]
             [34][16]

-----
- Generation 1000...
** Current Top Individual
-- fitness = [13309]
-- chromos = [47][36][39][33][48][44][7][40][8][9]
             [17][2][16][1][41][6][49][18][30][22]
             [46][25][26][51][13][12][10][27][11][50]
             [32][4][5][38][24][37][45][19][0][43]
             [14][42][3][31][21][23][35][28][15][29]
             [20][34]

-----
- Generation 1500...
** Current Top Individual
-- fitness = [11854]
-- chromos = [32][9][3][5][4][15][39][33][48][31]
             [0][20][6][1][41][16][17][38][36][35]
             [46][25][26][12][13][51][10][50][27][11]
             [24][47][43][18][7][40][8][2][22][29]
             [49][28][19][30][21][23][45][34][14][37]
             [44][42]

-----
- Generation 2000...
** Current Top Individual
-- fitness = [11294]
-- chromos = [39][35][3][5][33][43][36][4][47][38]
             [31][16][6][1][41][20][17][0][15][24]
             [46][25][26][12][13][51][10][50][11][27]
             [32][42][9][8][7][40][44][2][22][29]
             [49][28][19][30][21][23][45][34][14][37]
             [18][48]

* Max number of generations [10000] reached!
** Top Individual
-- fitness = [8437]
-- chromos = [24][3][5][14][39][38][48][31][0][21]
             [30][20][41][1][6][16][2][17][29][28]
             [46][25][26][12][13][51][10][27][11][50]
             [32][42][9][8][7][40][18][44][22][19]
             [49][15][43][33][34][35][36][37][4][23]
             [47][45]

-----
** Worst (of the top ones) Individual
-- fitness = [14391]
-- chromos = [29][17][3][4][44][43][24][48][2][36]
             [23][47][45][37][11][10][50][32][35][16]

```


[20][41][6][1][28][46][13][51][12][26]
[25][27][42][9][8][7][40][18][15][31]
[30][5][33][21][0][38][39][49][34][14]
[19][22]

* Runtime = 73.813744

* tspgen finished!

kakshak@kakshak-HP-Notebook:~/Desktop/tspgen-genetic-algorithm\$

5.4.2 Sample Test Case – 2

Input of Brazil Map

Brazil

NAME: brazil58

TYPE: TSP

COMMENT: 52 cities in Brazil (Ferreira)

DIMENSION: 52

EDGE_WEIGHT_TYPE: EXPLICIT

EDGE_WEIGHT_FORMAT: UPPER_ROW

EDGE_WEIGHT_SECTION

2635 2713 2437 1600 2845 6002 1743 594 2182 2906 1658 464 3334 3987 2870 2601 330
3049 1302 3399 1946 1278 669 627 2878 1737 3124 2878 307 5217 799 3305 3716 2251
2878 3467 4316 2963 512 2515 4850 1937 367 3601 3936 2430 2691 2087 1861 2358 2263
1425 2266 2166 3870 1417 739
314 2636 666 1096 4645 693 2889 287 772 1135 2875 1424 2185 1193 846 2142 1127 3104
1484 490 990 1950 2855 975 926 1214 599 2535 3860 3027 1407 1811 359 1060 1557 2959
394 2740 98 3538 856 2026 1710 1733 508 194 532 2906 435 335 2470 137 234 2072 1196
1517
2730 706 791 4588 922 2991 217 760 1050 2915 1119 1776 1451 5410 2182 846 3333 578
721 1030 1990 2895 670 835 909 287 2575 3803 3067 1102 1505 158 439 1248 2902 287
2780 276 3436 771 2066 1395 1740 407 279 334 3135 337 235 2699 451 546 1882 2699
1557
2824 3457 6083 2120 2040 2844 2973 3937 1958 3710 4363 2941 3207 1991 3512 1571
3392 2013 2922 2368 2170 3336 3503 3500 2884 2109 5298 2108 3643 4092 2934 3093
3843 4391 2763 2083 2543 4925 3236 2107 3939 4274 3008 2522 3006 806 2936 3026 947
2464 2364 4208 3116 2390
1247 4746 716 2223 584 1442 694 2089 1577 2230 1609 1003 1260 1297 2535 1511 908
324 1116 2183 1126 260 1367 996 1863 3961 2355 1565 1959 660 1106 1710 3060 991
2068 523 3594 434 1354 1861 2196 658 750 563 2929 586 668 2493 494 557 2130 524 851
5126 1653 3545 708 1298 1342 3451 347 1126 1989 250 2736 138 3786 888 1452 1671
2336 3431 124 1165 189 640 3110 4341 3603 489 855 730 364 598 3440 841 3316 919 3974
1063 2620 940 1090 722 921 761 3866 652 754 3430 1096 1191 1232 1582 2098
4267 6553 4826 3830 5951 6417 5423 6089 3142 4876 5676 5181 6678 4720 4074 5071
5724 6675 5000 4993 5221 4494 6265 785 6776 5265 5756 4628 4747 5511 1686 4276 6472
4447 2230 5167 5784 5561 5891 4902 4353 4922 6480 4759 4771 6044 4299 4199 5830
5271 5404
2296 923 1175 1282 2162 1906 2559 1133 1352 1422 1651 2423 1588 209 1030 1461 2420
1538 965 1716 1112 2010 3482 2521 1839 2288 1000 1289 2039 2551 955 2217 715 3125
1145 1529 2131 2466 1087 718 1083 2225 1015 960 1789 537 436 2400 1238 1199
2869 3459 2366 147 3741 4399 3417 3163 924 3462 681 3734 2493 1934 1377 130 3402
2449 3531 3219 343 5766 228 3724 4128 2883 3277 3870 4835 3214 184 2746 4875 2492
925 4017 4362 2985 2973 2831 1544 2913 2891 1103 2717 2655 4504 1972 1447

1004 781 2793 1036 1694 1694 458 1844 757 3125 1096 827 908 1702 2773 587 844 826
455 2453 4041 2945 1019 1423 75 572 1165 3140 560 2658 227 3674 502 1944 1312 1650
170 383 169 3136 98 80 2700 411 508 1584 948 1435

Output

kakshak@kakshak-HP-Notebook:~/Desktop/tspgen-genetic-algorithm\$ mpiexec -n 2 ./tspgen

* Starting tspgen...

* Config:

-- Population Size = [5000]

-- Mutation Rate = [0.100000]

-- Num of Generations = [10000]

-- Num of Elitism = [100]

-- Mutation Size = [2]

-- Migration Rate = [500]

-- Migration Share = [0.010000]

* Parsing map...

* Generating population...

* Initializing reproduction loop...

- Generation 500...

** Current Top Individual

-- fitness = [17384]

-- chromos = [28][0][35][33][44][4][36][24][45][25]
[15][43][18][30][17][22][7][8][9][32]
[48][34][11][26][51][13][46][38][21][12]
[27][50][14][2][16][6][1][41][19][49]
[29][42][10][20][37][23][47][5][3][39]
[31][40]

* Max number of generations [10000] reached!

** Top Individual

-- fitness = [8548]

-- chromos = [43][33][34][35][38][48][31][0][21][17]
[30][22][19][49][28][15][45][24][3][50]
[11][27][26][10][51][13][12][25][46][29]
[1][6][41][20][16][2][40][7][8][9]
[18][44][39][36][37][47][23][4][5][14]
[42][32]

** Worst (of the top ones) Individual

-- fitness = [13227]

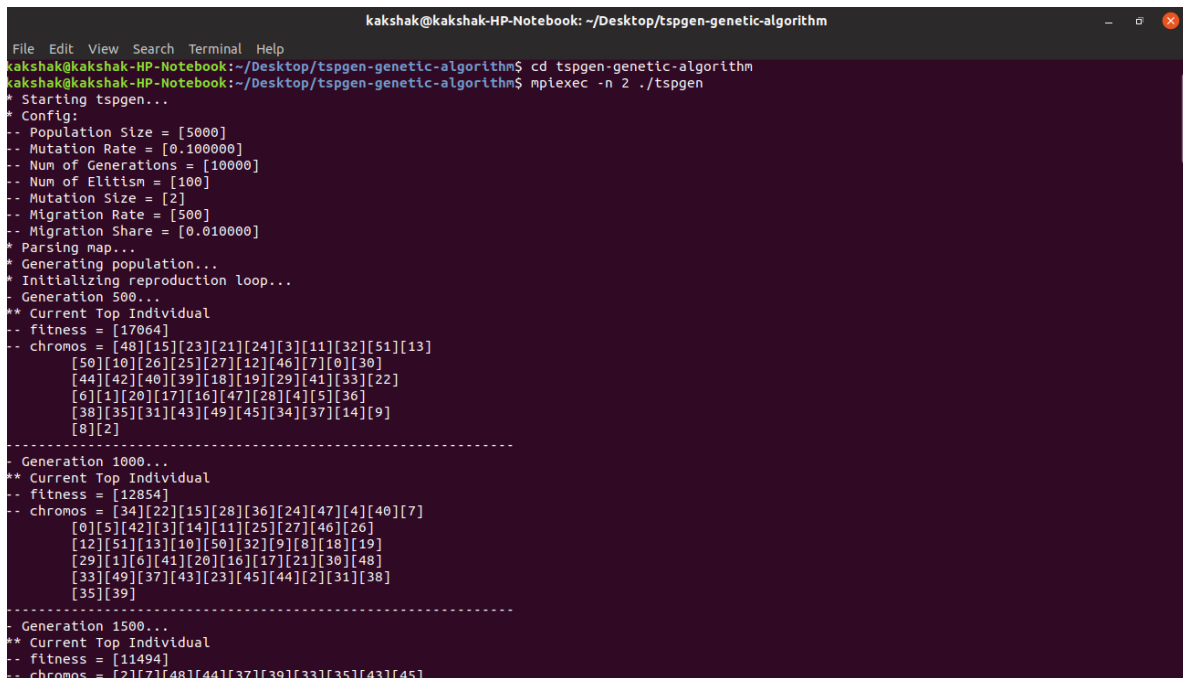
-- chromos = [34][5][33][38][44][48][35][31][32][17]
[30][22][19][49][28][15][45][26][24][50]
[11][27][46][10][51][13][12][25][3][29]
[1][6][41][16][20][2][40][7][8][9]
[18][21][37][36][39][47][23][4][0][14]
[42][43]

* Runtime = 71.847401

* tspgen finished!

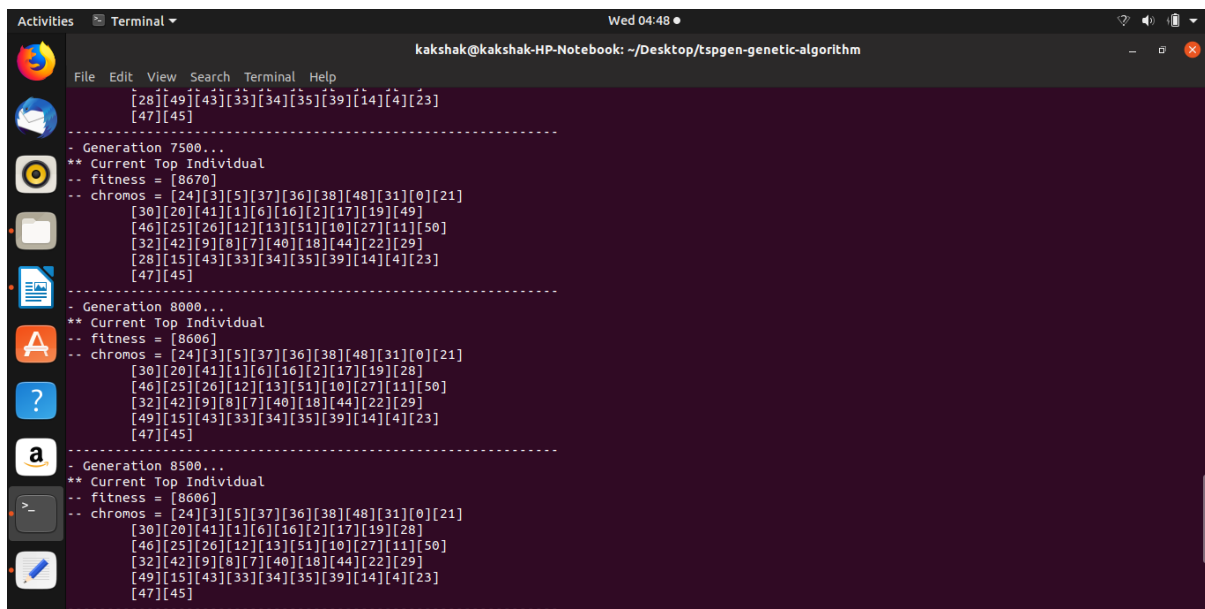
kakshak@kakshak-HP-Notebook:~/Desktop/tspgen-genetic-algorithm\$

5.5 Execution Snapshots



```
kakshak@kakshak-HP-Notebook: ~/Desktop/tspgen-genetic-algorithm
File Edit View Search Terminal Help
kakshak@kakshak-HP-Notebook:~/Desktop/tspgen-genetic-algorithm$ cd tspgen-genetic-algorithm
kakshak@kakshak-HP-Notebook:~/Desktop/tspgen-genetic-algorithm$ mpirun -n 2 ./tspgen
* Starting tspgen...
* Config:
-- Population Size = [5000]
-- Mutation Rate = [0.100000]
-- Num of Generations = [10000]
-- Num of Elitism = [100]
-- Mutation Size = [2]
-- Migration Rate = [500]
-- Migration Share = [0.010000]
* Parsing map...
* Generating population...
* Initializing reproduction loop...
- Generation 500...
** Current Top Individual
-- fitness = [17064]
-- chromos = [48][15][23][21][24][3][11][32][51][13]
[50][10][26][25][27][12][46][7][0][30]
[44][42][40][39][18][19][29][41][33][22]
[6][1][20][17][16][47][28][4][5][36]
[38][35][31][43][49][45][34][37][14][9]
[8][2]
-----
- Generation 1000...
** Current Top Individual
-- fitness = [12854]
-- chromos = [34][22][15][28][36][24][47][4][40][7]
[0][5][42][3][14][11][25][27][46][26]
[12][51][13][10][50][32][9][8][18][19]
[29][1][6][41][20][16][17][21][30][48]
[33][49][37][43][23][45][44][2][31][38]
[35][39]
-----
- Generation 1500...
** Current Top Individual
-- fitness = [11494]
-- chromos = [2][7][48][44][37][39][33][35][43][45]
```

Figure 4 Sample Output



```

[28][49][43][33][34][35][39][14][4][23]
[47][45]
-----
- Generation 7500...
** Current Top Individual
-- fitness = [8670]
-- chromos = [24][3][5][37][36][38][48][31][0][21]
[30][20][41][1][6][16][2][17][19][49]
[46][25][26][12][13][51][10][27][11][50]
[32][42][9][8][7][40][18][44][22][29]
[28][15][43][33][34][35][39][14][4][23]
[47][45]
-----
- Generation 8000...
** Current Top Individual
-- fitness = [8606]
-- chromos = [24][3][5][37][36][38][48][31][0][21]
[30][20][41][1][6][16][2][17][19][28]
[46][25][26][12][13][51][10][27][11][50]
[32][42][9][8][7][40][18][44][22][29]
[49][15][43][33][34][35][39][14][4][23]
[47][45]
-----
- Generation 8500...
** Current Top Individual
-- fitness = [8606]
-- chromos = [24][3][5][37][36][38][48][31][0][21]
[30][20][41][1][6][16][2][17][19][28]
[46][25][26][12][13][51][10][27][11][50]
[32][42][9][8][7][40][18][44][22][29]
[49][15][43][33][34][35][39][14][4][23]
[47][45]
-----
```

Figure 5 Sample Output

```
kakshak@kakshak-HP-Notebook: ~/Desktop/tspgen-genetic-algorithm
File Edit View Search Terminal Help
[34][43]
-----
- Generation 2500...
** Current Top Individual
-- fitness = [9922]
-- chromos = [7][40][44][31][48][36][35][38][45][49]
[28][15][39][4][24][11][27][46][25][26]
[12][13][51][10][50][32][9][8][18][22]
[29][1][6][41][20][16][2][17][30][0]
[21][19][34][23][5][3][42][37][47][14]
[33][43]
-----
- Generation 3000...
** Current Top Individual
-- fitness = [9656]
-- chromos = [7][40][44][31][48][0][35][38][39][49]
[28][15][45][4][3][11][27][26][25][46]
[12][13][51][10][50][32][9][8][18][19]
[29][1][6][41][20][16][2][17][30][22]
[21][34][36][23][24][5][42][37][14][47]
[33][43]
-----
- Generation 3500...
** Current Top Individual
-- fitness = [9433]
-- chromos = [40][18][44][31][48][38][35][34][43][49]
[28][15][39][4][24][11][27][26][25][46]
[12][13][51][10][50][32][9][8][7][22]
[29][1][6][41][20][16][2][17][30][0]
[21][19][33][23][5][3][42][37][47][14]
[36][45]
-----
- Generation 4000...
** Current Top Individual
-- fitness = [9244]
-- chromos = [7][18][44][31][48][38][35][34][43][49]
[28][15][45][4][24][11][27][26][25][46]
[12][13][51][10][50][32][9][8][40][19]
[29][1][6][41][20][16][2][17][30][22]
[21][19][33][23][5][3][42][37][47][14]
[36][45]
-----
```

Figure 6 Sample Output

```
kakshak@kakshak-HP-Notebook: ~/Desktop/tspgen-genetic-algorithm
File Edit View Search Terminal Help
-----
- Generation 6500...
** Current Top Individual
-- fitness = [8871]
-- chromos = [40][18][44][31][48][38][35][33][43][49]
[28][15][45][3][24][11][27][26][25][46]
[12][13][51][10][50][32][9][8][7][19]
[29][1][6][41][20][16][2][17][30][22]
[21][0][34][36][4][5][39][37][23][47]
[14][42]
-----
- Generation 7000...
** Current Top Individual
-- fitness = [8793]
-- chromos = [40][18][44][31][48][34][35][33][43][49]
[28][15][45][47][24][11][27][26][25][46]
[12][13][51][10][50][32][9][8][7][19]
[29][1][6][41][20][16][2][17][30][22]
[21][0][38][36][23][5][4][39][37][14]
[3][42]
-----
- Generation 7500...
** Current Top Individual
-- fitness = [8752]
-- chromos = [40][18][44][31][48][34][35][33][43][49]
[28][15][45][47][24][11][27][26][25][46]
[12][13][51][10][50][32][9][8][7][19]
[29][1][6][41][20][16][2][17][30][22]
[21][0][38][36][4][23][37][39][14][5]
[3][42]
-----
- Generation 8000...
** Current Top Individual
-- fitness = [8735]
-- chromos = [40][18][44][31][48][35][34][33][43][49]
[28][15][45][47][24][11][27][26][25][46]
[12][13][51][10][50][32][9][8][7][19]
[29][1][6][41][20][16][2][17][30][22]
[21][0][38][36][4][23][37][39][14][5]
[3][42]
-----
```

Figure 7 Sample Output

```

-----
* Max number of generations [10000] reached!
** Top Individual
-- fitness = [8619]
-- chromos = [40][18][44][31][48][35][34][33][43][49]
             [28][15][45][47][24][11][27][26][25][46]
             [12][13][51][10][50][32][9][8][7][19]
             [29][1][6][41][20][16][2][17][30][22]
             [21][0][38][39][36][37][23][4][14][5]
             [3][42]
-----
** Worst (of the top ones) Individual
-- fitness = [14202]
-- chromos = [35][19][22][7][47][38][33][15][42][9]
             [0][21][45][32][36][39][18][44][48][34]
             [37][16][24][11][50][10][51][13][12][26]
             [27][25][46][28][29][1][6][41][20][2]
             [17][30][5][14][4][23][49][0][40][31]
             [43][3]
-----
* Runtime = 73.393219
* tspgen finished!
kakshak@kakshak-HP-Notebook:~/Desktop/tspgen-genetic-algorithm$

```

Figure 7 Output of TSP of Berlin Map

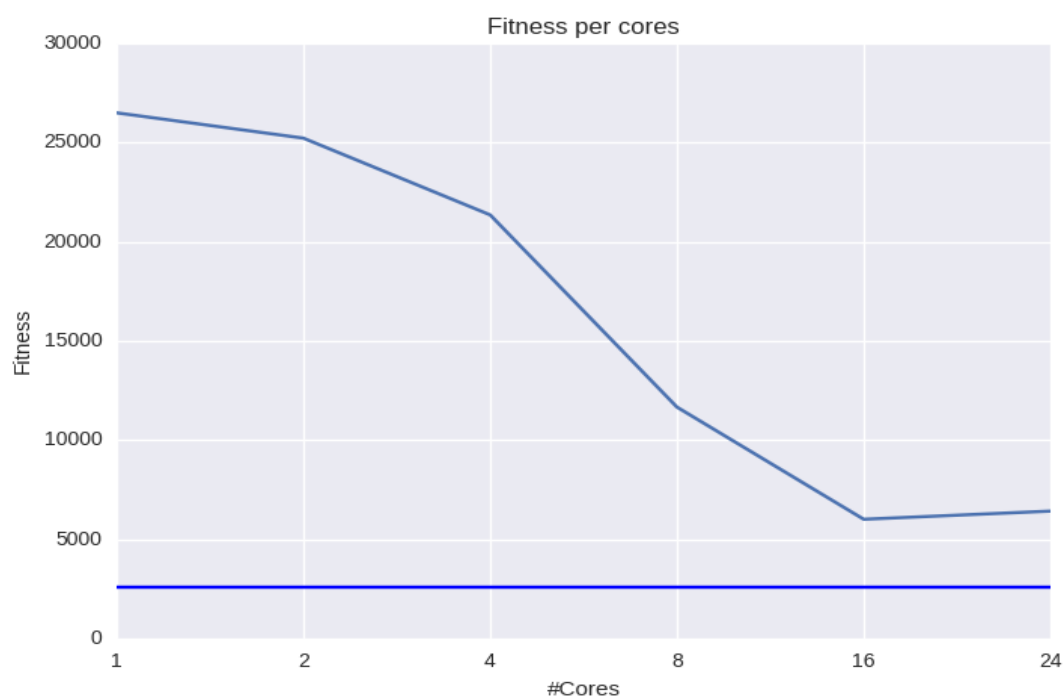


Figure 8 Fitness vs Cores

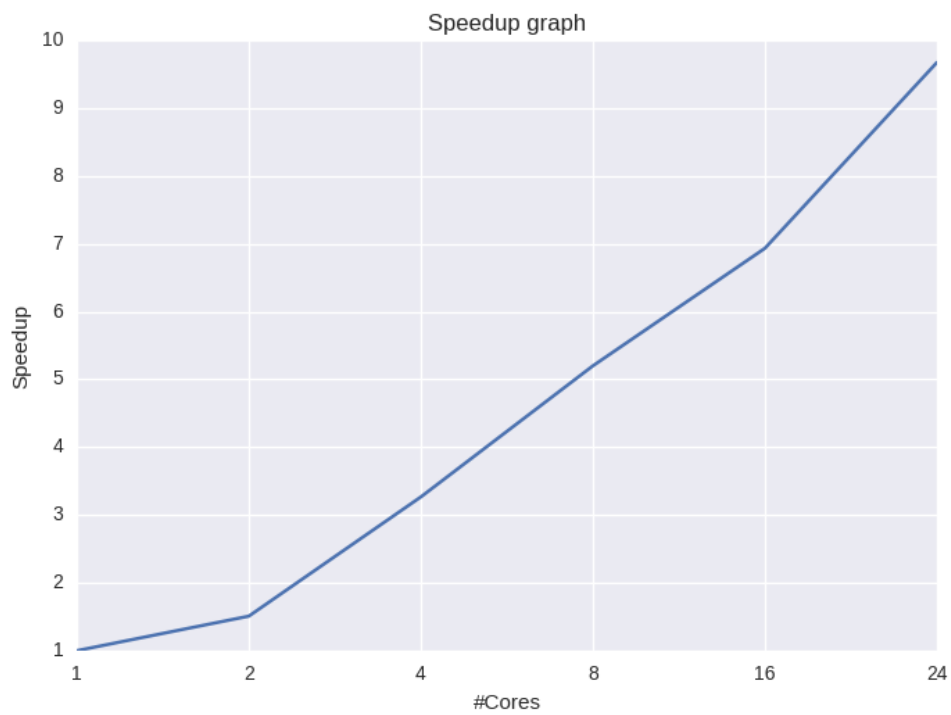


Figure 9 Speedup vs Cores

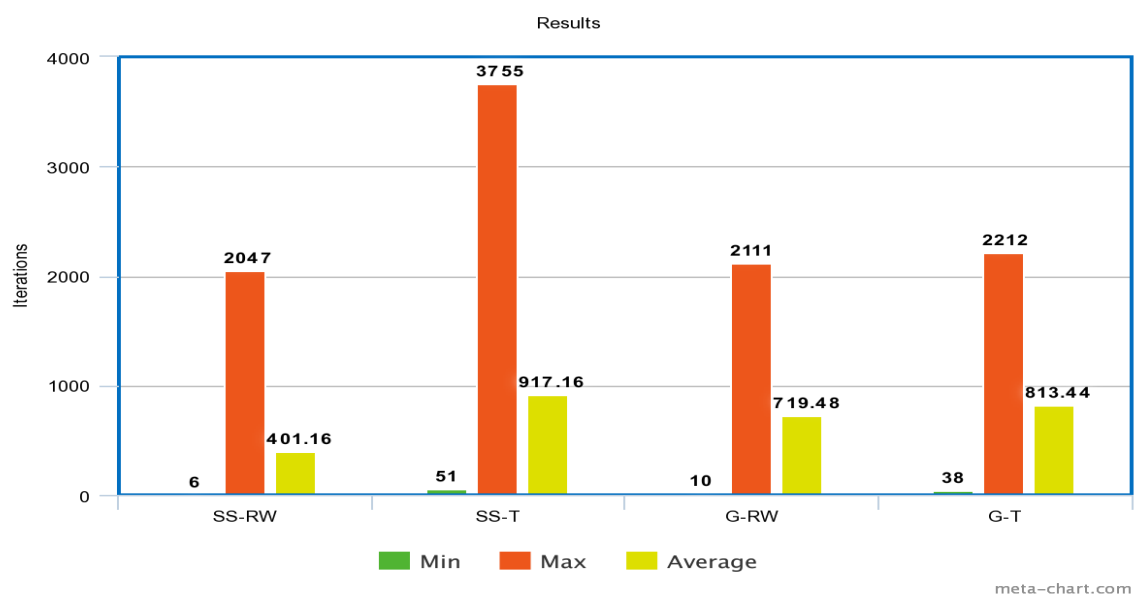
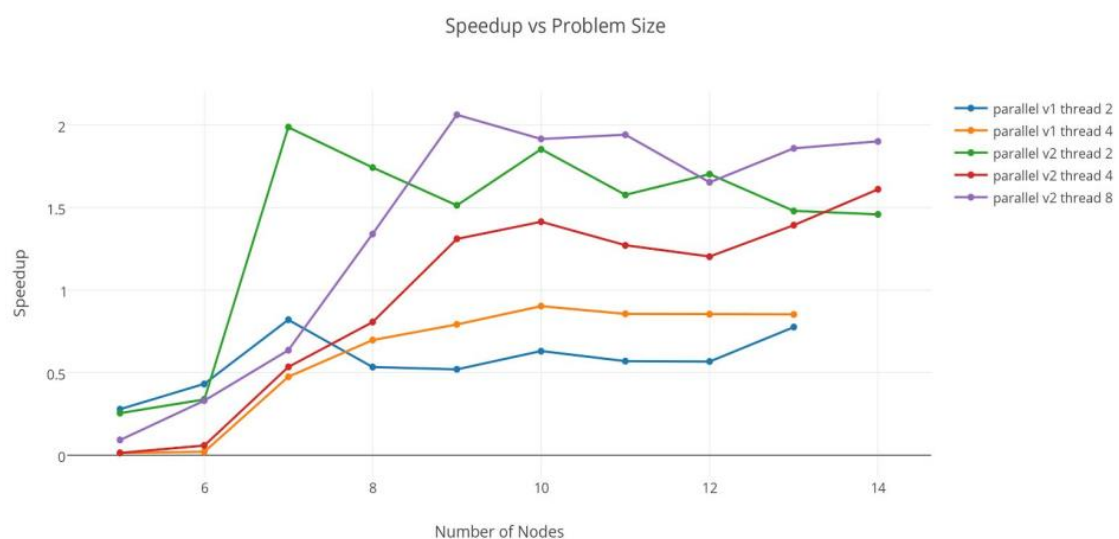
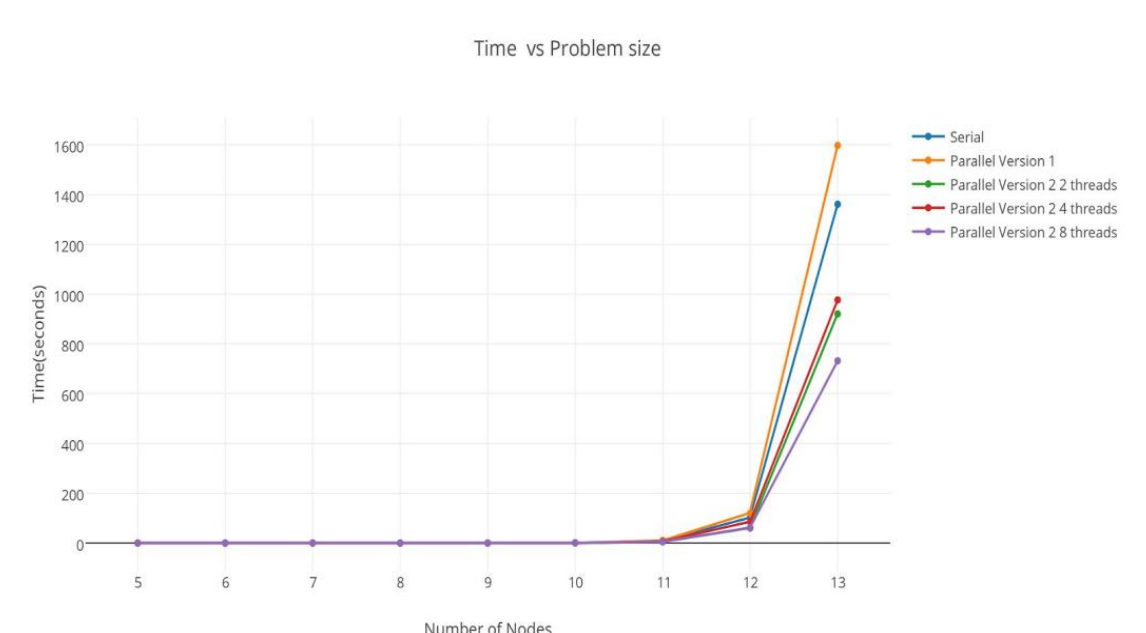


Figure 10 Results vs Iteration

As we can see from the graph the time taken by the serial code is much more than the parallel code. Also, as we can increase the number of threads for parallel version 2 the time taken by the code decreases.



X-axis: Number of nodes

Y-axis: Speedup

Figure 11,12 Speedup vs Problem Size

As we can see from the figure that as the number of processors increase for a fixed number of nodes(sufficiently large) the speedup also increases. For lesser number of nodes the speedup does not increase with increase in the processors. The parallel overhead generated in splitting the stack might be the reason for this.

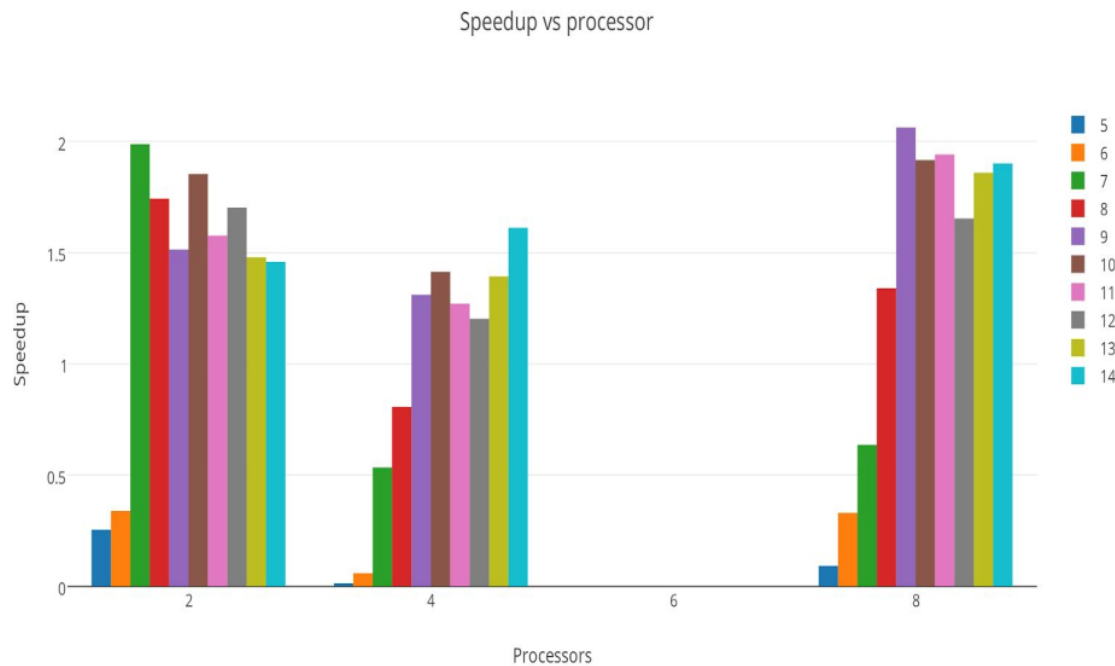


Figure 13 Speedup vs Processors

As we can see from the graph the first version is not that good and does not give any speedup. The parallel overhead generated due to critical section of pushing and popping into the stack might be the reason for this. For second version, we can see that when the problem size is small the speedup is more in case of a smaller number of threads whereas compared to a greater number of threads. The parallel overhead generated in case of more number of threads might be a reason for this. But as we increase the problem size the speedup obtained in case of 2 threads gradually decreases. Also, as the problem size increases the speedup also increases for a greater number of threads because the increase in parallel part is dominant over the serial part. We calculated the Karp Flatt metric for a fixed problem size (number of nodes=14) and we saw that the metric was slightly increasing when we increased the number of processors.

Number of Processors	Karp Flatt Metric
2	0.3708
4	0.4509
8	0.4586

6. Conclusion

As it can be seen from results mentioned the first parallelization strategy does not give any speedup. This is due to the parallel overhead generated due to the critical section for popping and pushing onto the same stack. On the other hand, the second parallelization strategy gives speedup as the problem size increases. When comparing for 2,4 and 8 threads the code gives more speedup when there are 2 threads and when the problem size is small. This is because the overhead generated in splitting the stack is more in 4 and 8 threads. But as the size increases the parallel overhead becomes negligible with respect to the total time and thus 4 and 8 threads give more speedup.

7. Future Scope

The second parallelization strategy splits the stack and distributes it among the processes. Each process computes the minimum tour in its allocated subtree. There is no communication between the processes. If at any point of time if the cost for reaching a node from source is more than the cost of global tour then there is no meaning in going further deep in the tree. So, it is possible that a process may complete its computation much before the other processes. In this case it will have to wait while the other processes work. When a process runs out of work, it can busy wait for more work or notification that program is terminating A process with work can send part of it to an idle process Alternatively, a process without work can request such form other process(es).

8. References

- [1] Harun Raşit Er, Prof. Dr. Nadia Erdoğan, Parallel Genetic Algorithm to Solve Traveling Salesman Problem on MapReduce Framework using Hadoop Cluster, ICJSE, March 2018
- [2] Freisleben, Bernd, Peter Merz, A genetic local search algorithm for solving symmetric and asymmetric traveling salesman problems, IEEE, June 2016
- [3] Rahul Saxena, Monika Jain, Parallelizing GA Based Heuristic approach for TSP over CUDA and OpenMP, IEEE, 2017
- [4] Sahib Singh Juneja, Pavi Saraswat, Travelling Salesman Problem Optimization Using Genetic Algorithm, IEEE, 2019
- [5] Tung Khac Truong, Binh Thanh Dang, An Improved Parallel Genetic Algorithm for The Traveling Salesman Problem, ISA, 2016
- [6] Plamenka Borovska, Solving the Travelling Salesman Problem in Parallel by Genetic Algorithm on Multicomputer Cluster, IEEE, 2016
- [7] Indresh Kumar Gupta, Abha Choubey, Randomized Bias Genetic Algorithm to Solve Traveling Salesman Problem, IEEE, 2017
- [8] Lee Wang, Anthony A. Maciejewski, A COMPARATIVE STUDY OF FIVE PARALLEL GENETIC ALGORITHMS USING THE TRAVELING SALESMAN PROBLEM, IEEE, 2015
- [9] Ugur Cekmez, Mustafa Ozsiginan, Adapting the GA Approach to Solve Traveling Salesman Problems on CUDA Architecture, IEEE 2015
- [10] Bernd Reisleben, Peter Merz ,A Genetic Local Search Algorithm for Solving - Symmetric and Asymmetric Traveling Salesman Problems, IEEE,2016
- [11] ByungIn Kim, Jaelk Shim, Min Zhang, Comparison of TSP Algorithms, IEEE,2015
- [12] Noraini Mohd Razali, John Geraghty, Genetic Algorithm Preformance with Different Selection Strategies, Springer Link, 2017
- [13] Kylie Bryant, Arthur Benjamin, Genetic Algorithms and The Travelling Salesman Problem, Springer Link, 2015
- [14] Larry Yaeger, Artificial Life as an approach to Artificial Intelligence, Intro to Genetic Algorithms, Elseiver, 2016
- [15] Milena Karova, Vassil Smarkov, Stoyan Penev, Genetic operators crossover and Mutation, IEEE,2015
- [16] Miguel Rocha, Jose Neves, Preventing Premature Convergence to Local Optima in Genetic Algorithms, Elseiver, 2017