Indian Institute of Technology Indore

Report for Lab Assignment 4 - GA

Parallel Computing Lab (CS 359)

Author
Aniket Sangwan (cse180001005@iiti.ac.in)

April 23, 2021

Contents

Chapter 1

Report

1.1 Problem Statement

Write a parallel program to solve Travelling Salesman Problem using Genetic Algorithm. TSP is given as:

Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?

- Each city needs to be visited exactly one time.
- We must return to the starting city, so our total distance needs to be calculated accordingly

1.2 Approach

TSP is a NP hard problem, i.e. no accurate solution can be found for this problem in polynomial time complexity. Genetic Algorithm (GA) is a search-based optimization technique based on the principles of Genetics and Natural Selection. It is frequently used to find optimal or near-optimal solutions to difficult problems which otherwise would take a lifetime to solve. This algorithm continues refining the search space to reach the maximum value for a given function along with mutation to prevent local maximum.

The Genetic Algorithm proceeds in the following steps:

- **Creating population:** We randomly shuffle the list of city IDs to create a population of pop_size and **parallelize** the loop generating individuals.
- **Determining fitness:** We calculate the distance of route taken and taken fitness to be the inverse of distance. Our aim is to maximise the fitness of solution.
- **Ranking routes:** We sort the population in the order of decreasing fitness and return it in a new vector. We can **parallelize the two for loops** in this function which are used to calculate the fitness and to store it in descending order.

```
        #pragma omp parallel for
        for(int i=0;i<fitnessResults.size();i++){

```

```
            sorted_population[i]=population[fitnessResults[i].S];
            fitness_scores[i]=(fitnessResults[i].F);

    }

```

- **Selecting the mating pool:** First,'elite_size' number of individuals with highest fitness are automatically carried to the next generation, before applying any selection algorithm. We then use fitness proportionate selection to select the parents that will be used to create the next generation. The two loops used for implementing elitism and calculating total fitness are **parallelized**. Then the loop implementing roulette's selection is also **parallelized** using OMP FOR.

```
        // Using fitness proportionate selection

        #pragma omp parallel for
        for(int i=0;i<fitness_scores.size()-elite_size;i++){

            double pick=(double)rand()/RAND_MAX;
            pick*=100;
            for(int j=0;j<fitness_scores.size();j++){

                if(j==fitness_scores.size()-1){
                    matingPool[i+elite_size]=(population[j]);
                    break;
                }
                if(j==fitness_scores.size()-1 || fitness_scores[j]<=
    pick && fitness_scores[j+1]>pick){
                    matingPool[i+elite_size]=(population[j]);
                    break;
                }

            }
        }

```

- **Breeding Population:** We use mating pool to breed children and use elitism to retain the best routes. These for loops are also **parallelized**.

```
        #pragma omp parallel for
        for(int i=0;i<length;i++){

            vector<City>child=breed(matingpool[i],matingpool[
    matingpool.size()-i-1]);
            children[i+elite_size]=(child);

    }

```

- **Mutating Population:** Two cities in every individual of the population are mutated using swap mutation with the probability of 'mutationRate'.

1.3 Input

The input is provided as variables in the program.

- The user can select the number of cities, population size, elite size, mutation rate and no of generations.
- If needed, the coordinates of cities can be taken as input.

1.4 Output

- The first line outputs the initial total distance.
- The second line outputs the final minimized total distance of the solution.
- The final line outputs the time taken (in sec) by the program.

The image below shows sample input and output.

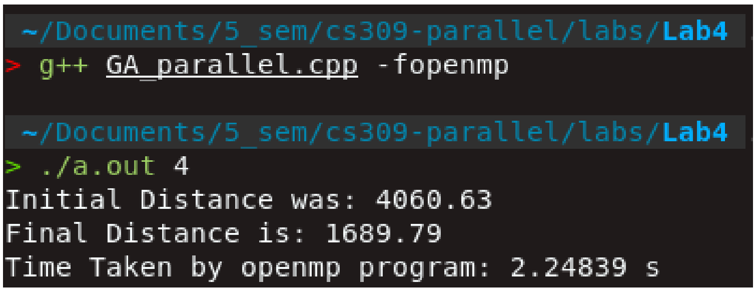

Figure 1.1: Sample input and output

1.5 Runtime and Convergence Analysis

To evaluate the performance for different cases, I used **three different values for population sizes** (200, 400 and 800) while keeping number of cities = 40, Number of elite routes = 19, mutation rate = 0.01 and number of generations = 1000. I compared the runtime of each test case for serial and parallel programs and generated the following graph to visualise the performance.

1.5.1 Test Case 1 (Population size=200)

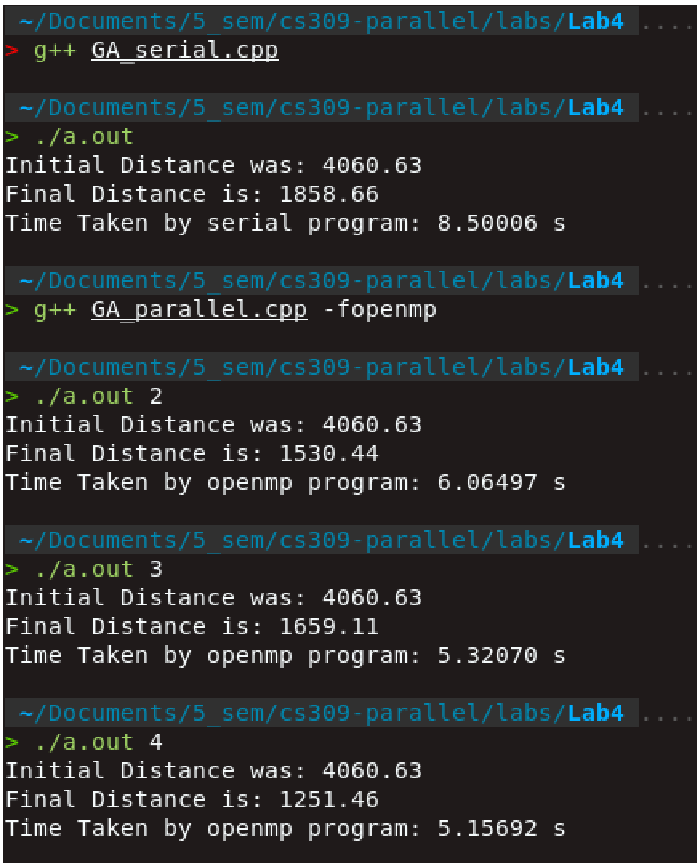

Figure 1.2: Output when population=200

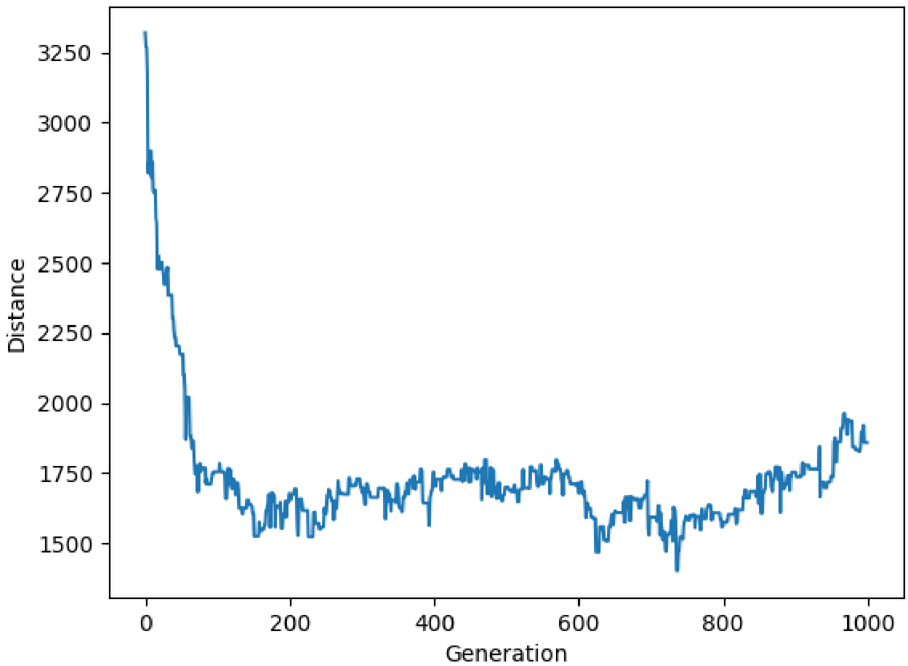

Figure 1.3: Convergence Graph when population=200

1.5.2 Test Case 2 (Population size=400)

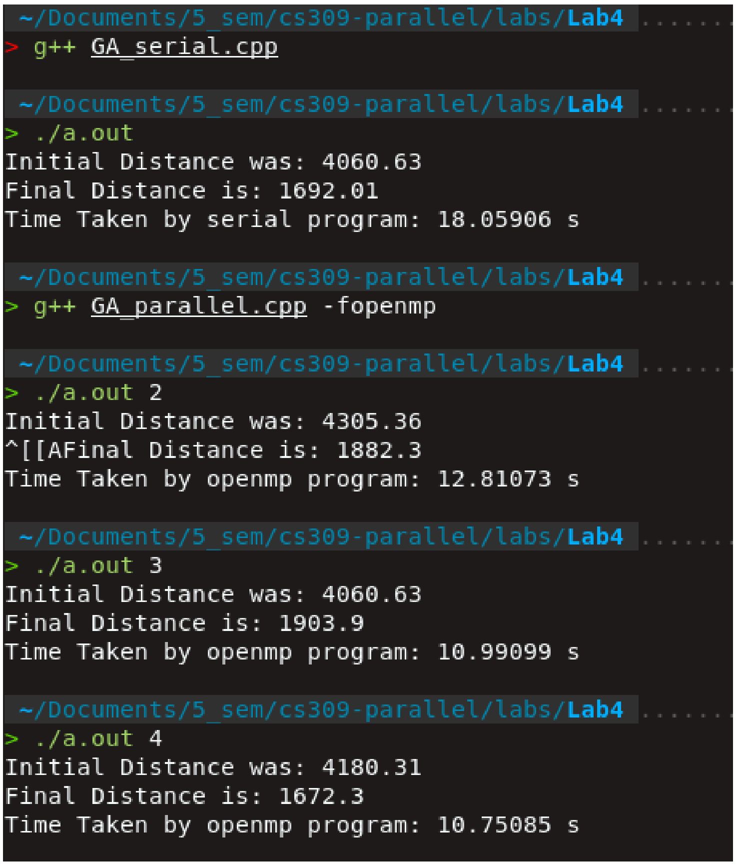

Figure 1.4: Output when population=400

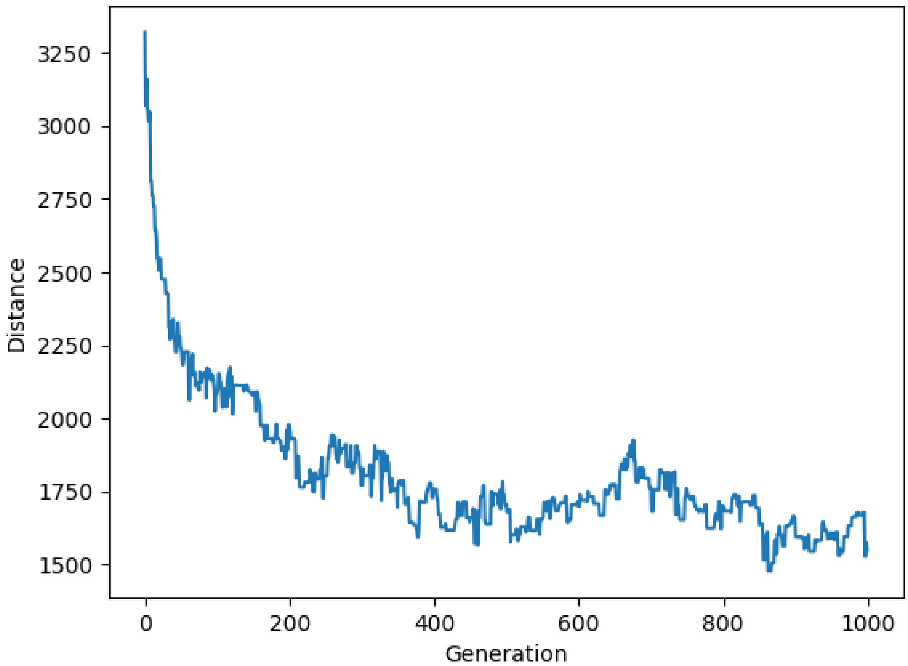

Figure 1.5: Convergence Graph when population=400

1.5.3 Test Case 3 (Population size=800)

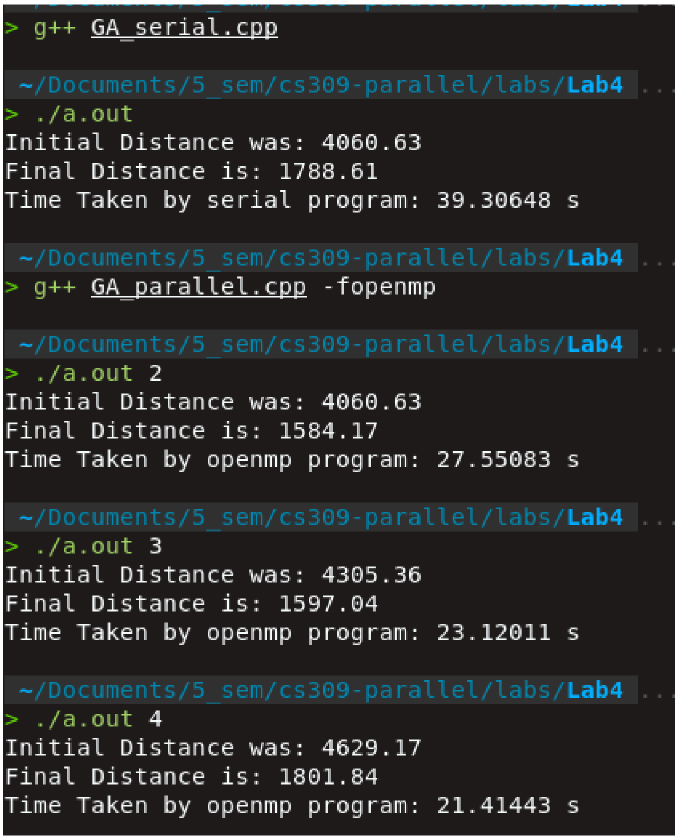

Figure 1.6: Output when population=800

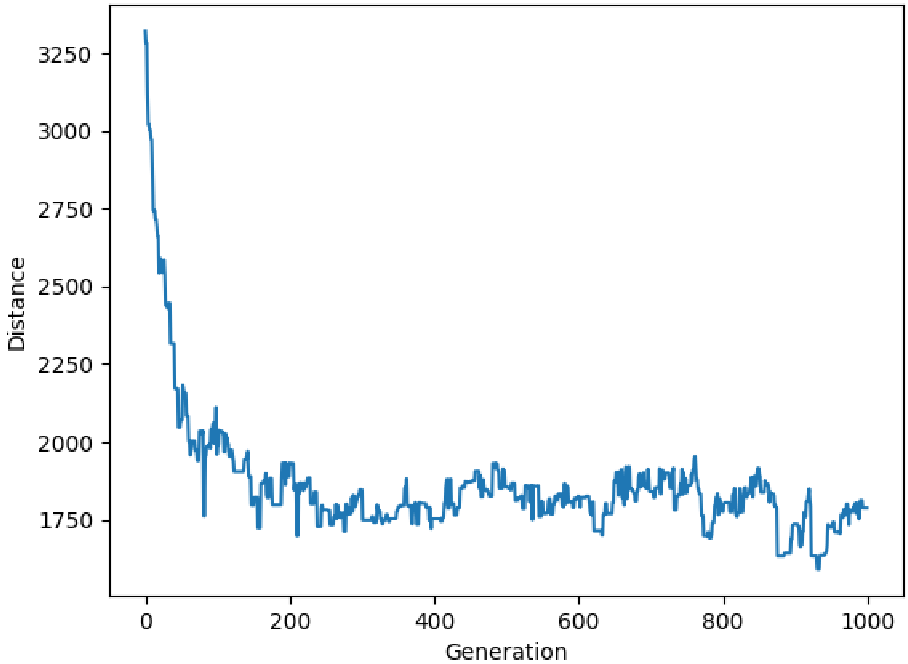

Figure 1.7: Convergence Graph when population=800

1.6 Runtime Comparsion

	Runtime (in sec)			
Population	1 Process	2 Processes	3 Processes	4 Processes
200	8.50006	6.06497	5.32070	5.15692
400	18.05906	12.81073	10.99099	10.75085
800	39.30648	27.55083	23.12011	21.41443

Table 1.1: Run-Times

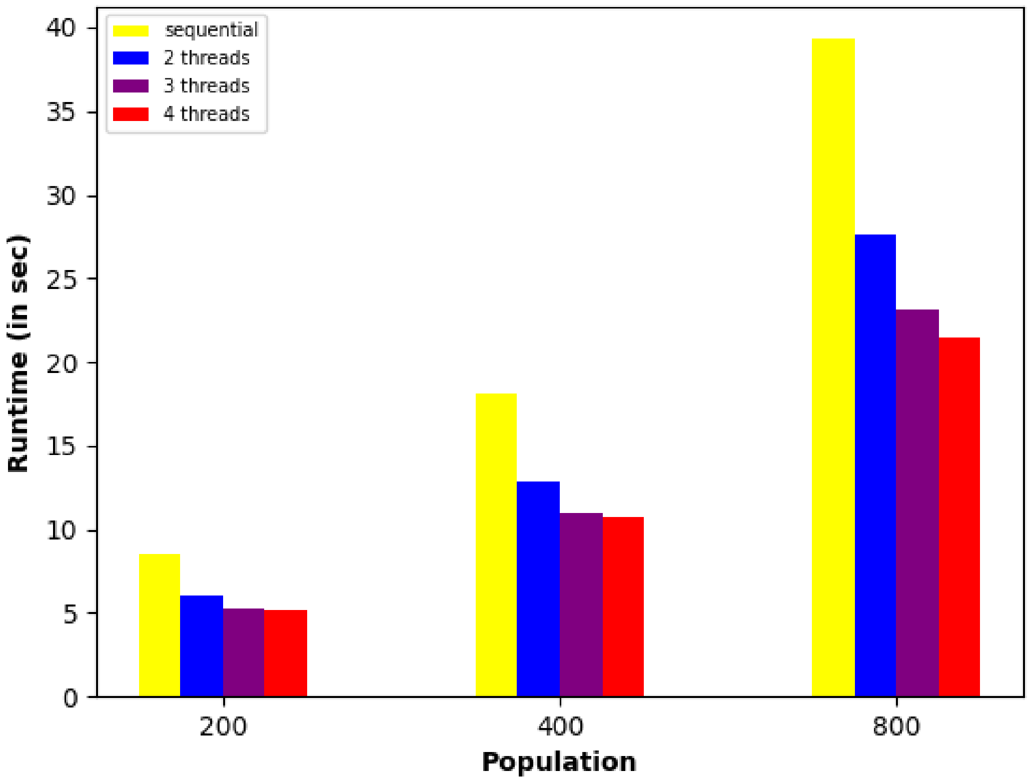

Figure 1.8: Change in runtime with change in population for serial and parallel programs

1.7 Conclusion

We conclude that the runtime reduces when the number of threads are increased while keeping population constant. The algorithm converges to a specific range in a few hundred iterations and the convergence is faster if the population size is large. We observe an average speedup of 2 for every population size when the number of threads are increased from 1 to 4. The Breeding function helps in the diversification process and Mutate function works for the intensification phase by helping to avoid local convergence.

1.8 Implementation

```

    #include <bits/stdc++.h>
#include <omp.h>
using namespace std;

#define double long double
#define pb push_back
#define F first
#define S second

int size; // No of threads

vector<double>fitness_scores; // Stores the fitness values of every individual
int w=1; // Current id of city

// Structure for City in TSP problem
struct City{

    int id; // Stores the id of the city (Unique for every city)
    int x,y; // Coordinates of the city

    // Constructor for cities
    City(int a,int b){
        id=w;
        x=a;
        y=b;
        w++;
    }

    // Calculates distance between two cities
    double distance(City c){

        double xdis=abs(x-c.x);
        double ydis=abs(y-c.y);
        double distance=sqrt(xdis*xdis+ydis*ydis);

        return distance;

    }

    // Prints the coordinates of the city
    void print_coord(){
        cout<<"("<<x<<", "<<y<<")\n";
    }

};

// Calculates the total distance of the route taken
double routeDistance(vector<City>route){

    double path_dist=0,fitness=0.0;

    // TSP starts and ends at the same place. So the initial city is inserted again
    route.pb(route[0]);

```

```
    // Calculates pairwise distance

    for(int i=0;i<route.size()-1;i++){

        City fromCity=route[i];
        City toCity=route[i+1];

        path_dist+=fromCity.distance(toCity);

    }

    return path_dist;

}

/* Returns fitness value of given route.
We aim to minimize the distance. So, Fitness is taken to be the inverse of
    distance,
because a larger fitness score is considered better
*/
double routeFitness(vector<City>route){

    double fitness=1.0/routeDistance(route);
    return fitness;

}

// Randomly creates route using the city list. Randomly selects the order
    in which we visit the cities
vector<City> createRoute(vector<City>CityList){

    shuffle(CityList.begin(),CityList.end(),std::default_random_engine(rand
    ()));

    return CityList;

}

// Looping through the create route function to generate full population
vector<vector<City>> initialPopulation(int pop_size, vector<City>CityList){

    vector< vector<City> >population;

    #pragma omp parallel
    {
        vector< vector<City> >population_private;

        #pragma omp for nowait
        for(int i=0;i<pop_size;i++)
            population_private.pb(createRoute(CityList));

        #pragma omp critical
        population.insert(population.end(),population_private.begin(),
    population_private.end());

    }
    return population;

```

```
    }

// Sorts the population according to their fitness score and returns in a new vector
vector<vector<City>> rankRoutes(vector<vector<City>>population){

    vector<pair<double,int>>fitnessResults; // Stores fitness value and inital route id
    fitnessResults.resize(population.size());

    #pragma omp parallel for
    for(int i=0;i<population.size();i++){
        fitnessResults[i].F=routeFitness(population[i]);
        fitnessResults[i].S=i;
    }

    // Sorts in descending order wrt fitness value
    sort(fitnessResults.begin(),fitnessResults.end(),greater <pair<double,int>> ());

    vector<vector<City>>sorted_population; // Will store population sorted in decreasing fitness value
    sorted_population.resize(population.size());

    #pragma omp parallel for
    for(int i=0;i<fitnessResults.size();i++){

        sorted_population[i]=population[fitnessResults[i].S];
        fitness_scores[i]=(fitnessResults[i].F);

    }

    return sorted_population;

}

/*
 Using fitness proportionate selection to select the parents that will be used
to create the next generation.

First,'elite_size' number of individuals with highest fitness are automatically carried to the next
generation, before applying any selection algorithm.

Then it completes the mating pool using fitness proportionate selection.

*/
vector<vector<City>> mating_Pool(vector<vector<City>>population, int elite_size){

    vector<vector<City>>matingPool; // Stores the mating pool
    matingPool.resize(fitness_scores.size());


    double tot_fitness=0; // Stores total fitness

    #pragma omp parallel
```

```
    {
      // Using elitism
    #pragma omp for
    for(int i=0;i<elite_size;i++)
        matingPool[i]=(population[i]);

    #pragma omp for reduction(+:tot_fitness)
    for(int i=0;i<fitness_scores.size();i++)
        tot_fitness+=fitness_scores[i];
    }
    // Assigning fitness weighed probability
    for(int i=0;i<fitness_scores.size();i++){

        fitness_scores[i]=100*(fitness_scores[i]/tot_fitness);
        if(i!=0) fitness_scores[i]+=fitness_scores[i-1];

    }

    // Using fitness proportionate selection

    #pragma omp parallel for
    for(int i=0;i<fitness_scores.size()-elite_size;i++){

        double pick=(double)rand()/RAND_MAX;
        pick*=100;
        for(int j=0;j<fitness_scores.size();j++){

            if(j==fitness_scores.size()-1){
                matingPool[i+elite_size]=(population[j]);
                break;
            }
            if(j==fitness_scores.size()-1 || fitness_scores[j]<=pick &&
    fitness_scores[j+1]>pick){
                matingPool[i+elite_size]=(population[j]);
                break;
            }

        }
    }

    return matingPool;

}

// Using ordered crossover to breed for next generation.
vector<City> breed(vector<City>parent1, vector<City>parent2){

    vector<City>child;

    // Randomly selecting a subset from first parent string
    int geneA=((double)rand()/RAND_MAX)*parent1.size();
    int geneB=((double)rand()/RAND_MAX)*parent1.size();

    int startGene=min(geneA, geneB);
    int endGene=max(geneA, geneB);

    int n=parent1.size();
    bool used[n+1]={0};
```

```
    // Filling the remainder of the route from second parent string in the order in which they appear.
    for(int i=startGene;i<endGene;i++){
        child.pb(parent1[i]);
        used[parent1[i].id]=1;
    }

    for(int i=0;i<parent2.size();i++){
        if(used[parent2[i].id]==0)
            child.pb(parent2[i]);
    }

    return child;

}

// Using mating pool to breed children. Using elitism to retain the best routes.
vector<vector<City>> breedPopulation(vector<vector<City>>matingpool, int elite_size){

    vector<vector<City>>children;
    children.resize(matingpool.size());
    int length=matingpool.size()-elite_size;

    for(int i=0;i<elite_size;i++)
        children[i]=(matingpool[i]);

    shuffle(matingpool.begin(),matingpool.end(),std::default_random_engine(rand()));

    // Filling the rest of the generation
    #pragma omp parallel for
    for(int i=0;i<length;i++){

        vector<City>child=breed(matingpool[i],matingpool[matingpool.size()-i-1]);
        children[i+elite_size]=(child);

    }

    return children;

}

/* Using swap mutation to mutate.
Helps in avoiding local convergence
mutationRate-> Probability that two cities will swap their position
*/
vector<City> mutate(vector<City>&individual, double mutationRate){

    for(int swapped=0;swapped<individual.size();swapped++){

        double prob=(double)rand()/RAND_MAX;

        if(prob<mutationRate){

```

```
            double swapWith=(double)rand()/RAND_MAX;
            swapWith*=individual.size();

            swap(individual[swapped], individual[swapWith]);

        }

    }

    return individual;

}

// Using mutate function to mutate the complete population
vector<vector<City>> mutatePopulation( vector<vector<City>>population,
    double mutationRate){

    vector<vector<City>>mutatedPop;
    mutatedPop.resize(population.size());

    for(int ind=0;ind<population.size();ind++){
        vector<City>mutatedInd = mutate(population[ind],mutationRate);
        mutatedPop[ind]=(mutatedInd);
    }

    return mutatedPop;

}

// Produces a new generation using all the functions above
vector<vector<City>> nextGeneration( vector<vector<City>>currentGen, int
    elite_size, double mutationRate){

    // Rank the routes in the current generation
    vector<vector<City>> popRanked=rankRoutes(currentGen);

    // cout<<routeDistance(popRanked[0])<<", ";

    // Determining potential parents and creating mating pool
    vector<vector<City>> matingpool=mating_Pool(popRanked, elite_size);

    // Creating new generation
    vector<vector<City>> children=breedPopulation(matingpool, elite_size);

    // Applying mutation
    vector<vector<City>> nextGen=mutatePopulation(children, mutationRate);

    return nextGen;

}

void Genetic_Algo( vector<City>population, int popSize, int elite_size,
    double mutationRate, int generations){

    // Creating initial population from city list
    vector<vector<City>>pop=initialPopulation(popSize, population);

    cout<<"Initial Distance was: "<<routeDistance(pop[0])<<endl;
```

```
    for(int i=0;i<generations;i++){

        fitness_scores.clear();
        fitness_scores.resize(popSize);
        pop=nextGeneration(pop, elite_size, mutationRate);

    }


    cout<<"Final Distance is: "<<routeDistance(pop[0])<<endl;

}

int main(int argc, char **argv){
  ios_base::sync_with_stdio(false);cin.tie(NULL);cout.tie(NULL);

    omp_set_num_threads(atoi(argv[1]));

    int no_of_cities=40; // No of cities
    int popSize=800;     // Population Size
    int eliteSize=19;    // Elite Size
    double mutationRate=0.01;    // Rate of mutation
    int generations=1000;    // No of generations

    vector<City>cityList; // Stores the initial list of cities

    // Assigning random coordinates to cities
    for(int i=0;i<no_of_cities;i++){
        double x=200*((double)rand()/RAND_MAX);
        double y=200*((double)rand()/RAND_MAX);
        cityList.pb(City(x,y));
    }

    double start_time=omp_get_wtime();

    Genetic_Algo(cityList, popSize, eliteSize, mutationRate, generations);

    double final_tot=omp_get_wtime()-start_time;
    // Prints time taken
    cout<<fixed<<setprecision(5) <<"Time Taken by openmp program: "<<
   final_tot <<" s "<<endl;

return 0;
}
```