

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

1	Report	1
1.1	Problem Statement	1
1.2	Approach	1
1.3	Input	3
1.4	Output	3
1.5	Runtime and Convergence Analysis	3
1.5.1	Test Case 1 (Population size=200)	4
1.5.2	Test Case 2 (Population size=400)	5
1.5.3	Test Case 3 (Population size=800)	6
1.6	Runtime Comparsion	7
1.7	Conclusion	7
1.8	Implementation	7

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.

```
1 | #pragma omp parallel for
2 |   for (int i=0;i<fitnessResults.size();i++){
3 |
```

```

4         sorted_population[i]=population[fitnessResults[i].S];
5         fitness_scores[i]=(fitnessResults[i].F);
6
7     }
8

```

- **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.

```

1         // Using fitness proportionate selection
2
3         #pragma omp parallel for
4         for(int i=0;i<fitness_scores.size()-elite_size;i++){
5
6             double pick=(double)rand()/RAND_MAX;
7             pick*=100;
8             for(int j=0;j<fitness_scores.size();j++){
9
10                if(j==fitness_scores.size()-1){
11                    matingPool[i+elite_size]=(population[j]);
12                    break;
13                }
14                if(j==fitness_scores.size()-1 || fitness_scores[j]<=
pick && fitness_scores[j+1]>pick){
15                    matingPool[i+elite_size]=(population[j]);
16                    break;
17                }
18            }
19        }
20    }
21

```

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

```

1         #pragma omp parallel for
2         for(int i=0;i<length;i++){
3
4             vector<City>child=breed(matingpool[i],matingpool[
matingpool.size()-i-1]);
5             children[i+elite_size]=(child);
6
7         }
8

```

- **Mutating Population:** Two cities in every individual of the population are mu-

tated using swap mutation with the probability of 'mutationRate'.

1.3 Input

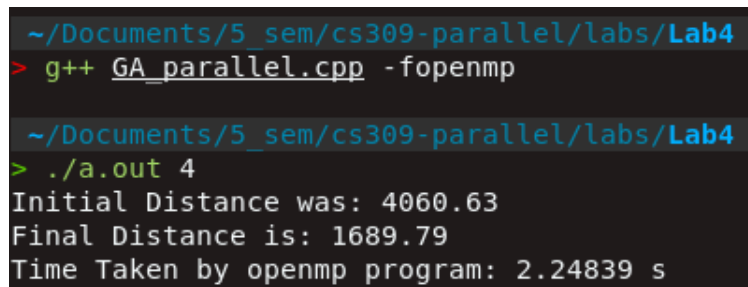
The input is provided as variables in the program.

- 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.



```
~/Documents/5_sem/cs309-parallel/labs/Lab4  
> g++ GA_parallel.cpp -fopenmp  
  
~/Documents/5_sem/cs309-parallel/labs/Lab4  
> ./a.out 4  
Initial Distance was: 4060.63  
Final Distance is: 1689.79  
Time Taken by openmp program: 2.24839 s
```

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)

```
~/Documents/5_sem/cs309-parallel/labs/Lab4 ...  
> g++ GA_serial.cpp  
  
~/Documents/5_sem/cs309-parallel/labs/Lab4 ...  
> ./a.out  
Initial Distance was: 4060.63  
Final Distance is: 1858.66  
Time Taken by serial program: 8.50006 s  
  
~/Documents/5_sem/cs309-parallel/labs/Lab4 ...  
> g++ GA_parallel.cpp -fopenmp  
  
~/Documents/5_sem/cs309-parallel/labs/Lab4 ...  
> ./a.out 2  
Initial Distance was: 4060.63  
Final Distance is: 1530.44  
Time Taken by openmp program: 6.06497 s  
  
~/Documents/5_sem/cs309-parallel/labs/Lab4 ...  
> ./a.out 3  
Initial Distance was: 4060.63  
Final Distance is: 1659.11  
Time Taken by openmp program: 5.32070 s  
  
~/Documents/5_sem/cs309-parallel/labs/Lab4 ...  
> ./a.out 4  
Initial Distance was: 4060.63  
Final Distance is: 1251.46  
Time Taken by openmp program: 5.15692 s
```

Figure 1.2: Output when population=200

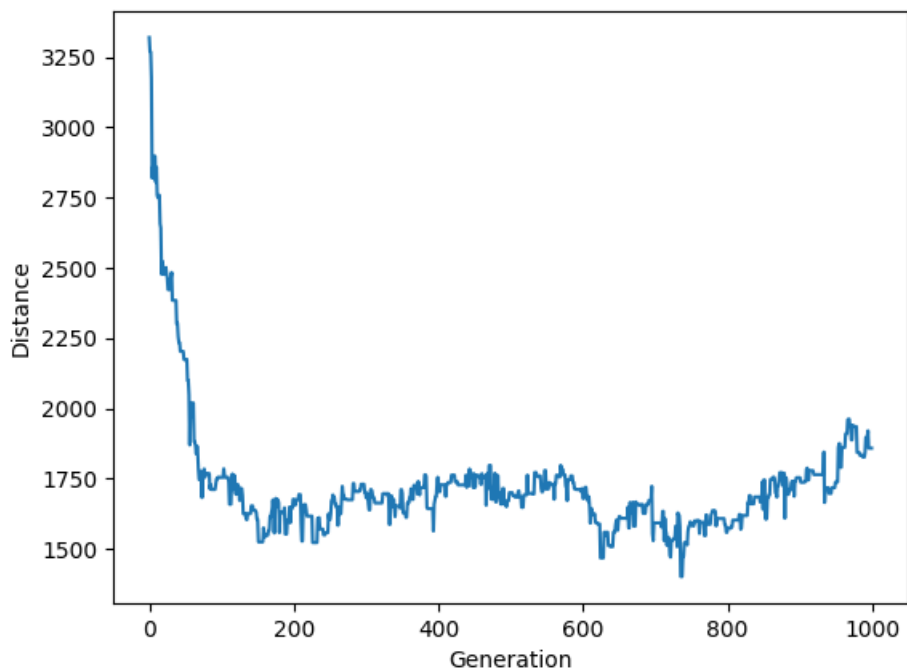


Figure 1.3: Convergence Graph when population=200

1.5.2 Test Case 2 (Population size=400)

```
~/Documents/5_sem/cs309-parallel/labs/Lab4 .....  
> g++ GA_serial.cpp  
  
~/Documents/5_sem/cs309-parallel/labs/Lab4 .....  
> ./a.out  
Initial Distance was: 4060.63  
Final Distance is: 1692.01  
Time Taken by serial program: 18.05906 s  
  
~/Documents/5_sem/cs309-parallel/labs/Lab4 .....  
> g++ GA_parallel.cpp -fopenmp  
  
~/Documents/5_sem/cs309-parallel/labs/Lab4 .....  
> ./a.out 2  
Initial Distance was: 4305.36  
^[[AFinal Distance is: 1882.3  
Time Taken by openmp program: 12.81073 s  
  
~/Documents/5_sem/cs309-parallel/labs/Lab4 .....  
> ./a.out 3  
Initial Distance was: 4060.63  
Final Distance is: 1903.9  
Time Taken by openmp program: 10.99099 s  
  
~/Documents/5_sem/cs309-parallel/labs/Lab4 .....  
> ./a.out 4  
Initial Distance was: 4180.31  
Final Distance is: 1672.3  
Time Taken by openmp program: 10.75085 s
```

Figure 1.4: Output when population=400

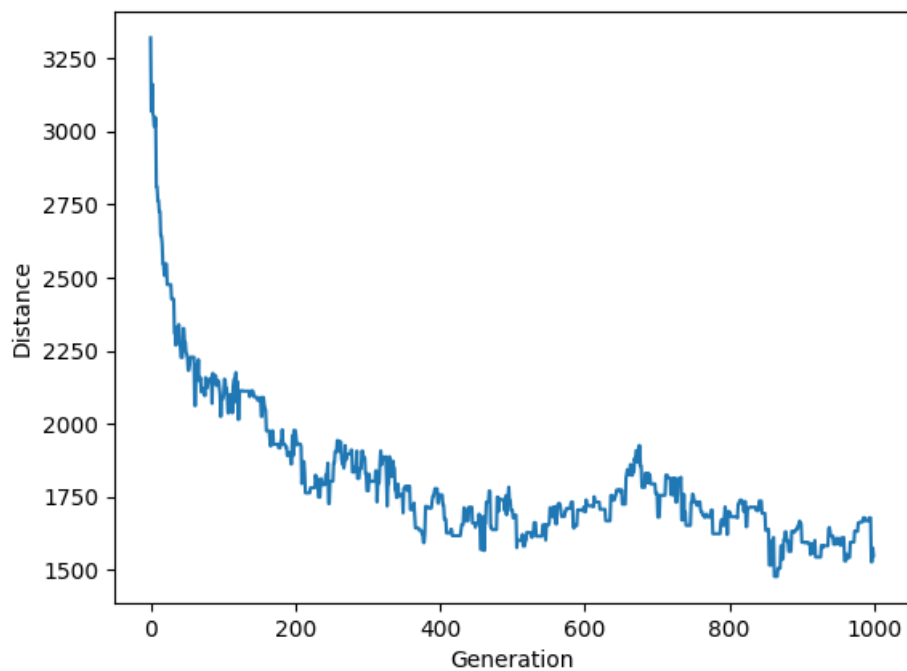


Figure 1.5: Convergence Graph when population=400

1.5.3 Test Case 3 (Population size=800)

```
> g++ GA_serial.cpp

~/Documents/5_sem/cs309-parallel/labs/Lab4 ..
> ./a.out
Initial Distance was: 4060.63
Final Distance is: 1788.61
Time Taken by serial program: 39.30648 s

~/Documents/5_sem/cs309-parallel/labs/Lab4 ..
> g++ GA_parallel.cpp -fopenmp

~/Documents/5_sem/cs309-parallel/labs/Lab4 ..
> ./a.out 2
Initial Distance was: 4060.63
Final Distance is: 1584.17
Time Taken by openmp program: 27.55083 s

~/Documents/5_sem/cs309-parallel/labs/Lab4 ..
> ./a.out 3
Initial Distance was: 4305.36
Final Distance is: 1597.04
Time Taken by openmp program: 23.12011 s

~/Documents/5_sem/cs309-parallel/labs/Lab4 ..
> ./a.out 4
Initial Distance was: 4629.17
Final Distance is: 1801.84
Time Taken by openmp program: 21.41443 s
```

Figure 1.6: Output when population=800

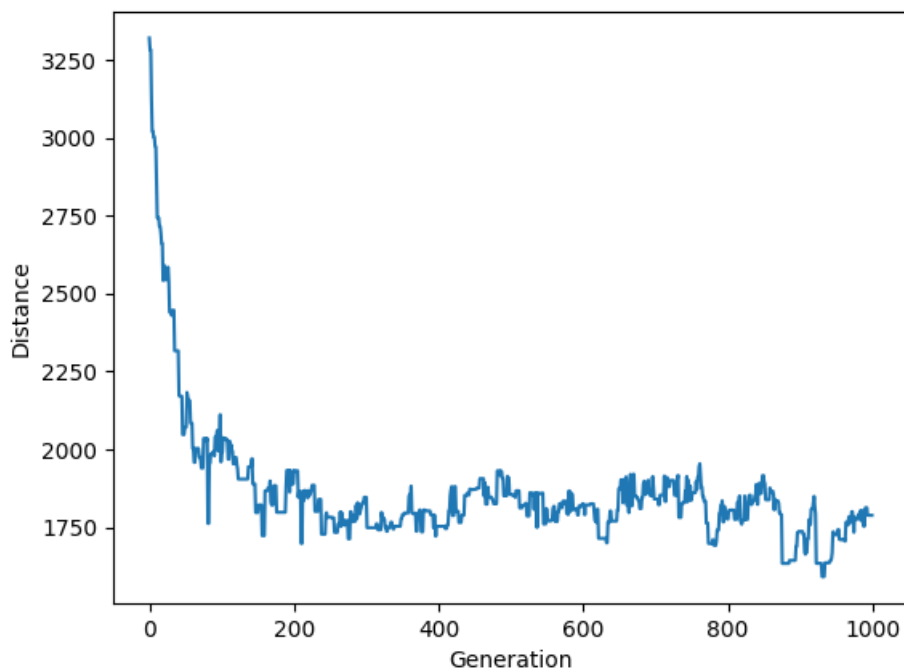


Figure 1.7: Convergence Graph when population=800

1.6 Runtime Comparision

Population	Runtime (in sec)			
	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

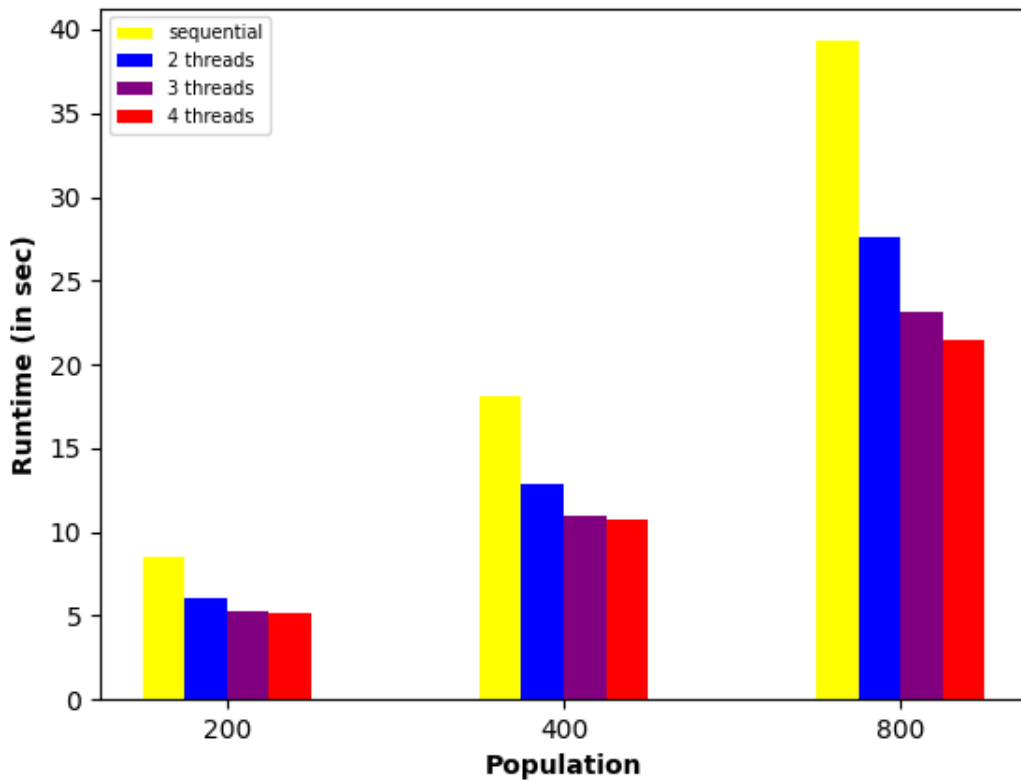


Figure 1.8: Change in runtime with change in population for serial and parallel programs

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

```

1      #include <bits/stdc++.h>
2
3      #include <omp.h>
4      using namespace std;
5
6      #define double long double
7      #define pb push_back
8      #define F first
9      #define S second
10
11     int size; // No of threads
12
13     vector<double>fitness_scores; // Stores the fitness values of every
        individual
14     int w=1; // Current id of city
15
16     // Structure for City in TSP problem
17     struct City{
18
19         int id; // Stores the id of the city (Unique for every city)
20         int x,y; // Coordinates of the city
21
22         // Constructor for cities
23         City(int a,int b){
24             id=w;
25             x=a;
26             y=b;
27             w++;
28         }
29
30         // Calculates distance between two cities
31         double distance(City c){
32
33             double xdis=abs(x-c.x);
34             double ydis=abs(y-c.y);
35             double distance=sqrt(xdis*xdis+ydis*ydis);
36
37             return distance;
38         }
39     }
40
41     // Prints the coordinates of the city
42     void print_coord(){
43         cout<<"("<<x<<" , "<<y<<"\n";
44     }
45
46 };
47
48 // Calculates the total distance of the route taken
49 double routeDistance(vector<City>route){
50
51     double path_dist=0,fitness=0.0;
52
53     // TSP starts and ends at the same place. So the initial city is
        inserted again
54     route.pb(route[0]);
55

```

```

56 // Calculates pairwise distance
57
58 for (int i=0;i<route.size()-1;i++){
59
60     City fromCity=route[i];
61     City toCity=route[i+1];
62
63     path_dist+=fromCity.distance(toCity);
64
65 }
66
67 return path_dist;
68
69 }
70
71 /* Returns fitness value of given route.
72 We aim to minimize the distance. So, Fitness is taken to be the inverse of
73 distance,
74 because a larger fitness score is considered better
75 */
76 double routeFitness(vector<City>route){
77
78     double fitness=1.0/routeDistance(route);
79     return fitness;
80 }
81
82 // Randomly creates route using the city list. Randomly selects the order
83 // in which we visit the cities
84 vector<City> createRoute(vector<City>CityList){
85
86     shuffle(CityList.begin(),CityList.end(),std::default_random_engine(rand()));
87
88     return CityList;
89 }
90
91 // Looping through the create route function to generate full population
92 vector<vector<City>> initialPopulation(int pop_size, vector<City>CityList){
93
94     vector< vector<City> >population;
95
96     #pragma omp parallel
97     {
98         vector< vector<City> >population_private;
99
100         #pragma omp for nowait
101         for (int i=0;i<pop_size;i++)
102             population_private.pb(createRoute(CityList));
103
104         #pragma omp critical
105         population.insert(population.end(),population_private.begin(),
106             population_private.end());
107     }
108     return population;
109 }

```

```

110     }
111
112 // Sorts the population according to their fitness score and returns in a
113 // new vector
114 vector<vector<City>> rankRoutes(vector<vector<City>>population){
115     vector<pair<double,int>>fitnessResults; // Stores fitness value and
116     // initial route id
117     fitnessResults.resize(population.size());
118
119 #pragma omp parallel for
120 for(int i=0;i<population.size();i++){
121     fitnessResults[i].F=routeFitness(population[i]);
122     fitnessResults[i].S=i;
123 }
124
125 // Sorts in descending order wrt fitness value
126 sort(fitnessResults.begin(),fitnessResults.end(),greater<pair<double,
127 int>>());
128
129 vector<vector<City>>sorted_population; // Will store population sorted
130 // in decreasing fitness value
131 sorted_population.resize(population.size());
132
133 #pragma omp parallel for
134 for(int i=0;i<fitnessResults.size();i++){
135     sorted_population[i]=population[fitnessResults[i].S];
136     fitness_scores[i]=(fitnessResults[i].F);
137 }
138
139 return sorted_population;
140 }
141
142 /*
143 Using fitness proportionate selection to select the parents that will be
144 used
145 to create the next generation.
146 First,'elite_size' number of individuals with highest fitness are
147 automatically carried to the next
148 generation, before applying any selection algorithm.
149 Then it completes the mating pool using fitness proportionate selection.
150
151 */
152 vector<vector<City>> mating_Pool(vector<vector<City>>population, int
153 elite_size){
154
155     vector<vector<City>>matingPool; // Stores the mating pool
156     matingPool.resize(fitness_scores.size());
157
158     double tot_fitness=0; // Stores total fitness
159
160 #pragma omp parallel

```

```

161 {
162     // Using elitism
163     #pragma omp for
164     for (int i=0;i<elite_size;i++)
165         matingPool[i]=(population[i]);
166
167     #pragma omp for reduction(+:tot_fitness)
168     for (int i=0;i<fitness_scores.size();i++)
169         tot_fitness+=fitness_scores[i];
170 }
171 // Assigning fitness weighed probability
172 for (int i=0;i<fitness_scores.size();i++){
173
174     fitness_scores[i]=100*(fitness_scores[i]/tot_fitness);
175     if(i!=0) fitness_scores[i]+=fitness_scores[i-1];
176
177 }
178
179 // Using fitness proportionate selection
180
181 #pragma omp parallel for
182 for (int i=0;i<fitness_scores.size()-elite_size;i++){
183
184     double pick=((double)rand()/RAND_MAX);
185     pick*=100;
186     for (int j=0;j<fitness_scores.size();j++){
187
188         if(j==fitness_scores.size()-1){
189             matingPool[i+elite_size]=(population[j]);
190             break;
191         }
192         if(j==fitness_scores.size()-1 || fitness_scores[j]<=pick &&
fitness_scores[j+1]>pick){
193             matingPool[i+elite_size]=(population[j]);
194             break;
195         }
196     }
197 }
198 }
199
200 return matingPool;
201 }
202 }
203
204 // Using ordered crossover to breed for next generation.
205 vector<City> breed(vector<City>parent1, vector<City>parent2){
206
207     vector<City>child;
208
209     // Randomly selecting a subset from first parent string
210     int geneA=((double)rand()/RAND_MAX)*parent1.size();
211     int geneB=((double)rand()/RAND_MAX)*parent1.size();
212
213     int startGene=min(geneA, geneB);
214     int endGene=max(geneA, geneB);
215
216     int n=parent1.size();
217     bool used[n+1]={0};

```

```

218
219 // Filling the remainder of the route from second parent string in the
order in which they appear.
220 for(int i=startGene;i<endGene;i++){
221     child.pb(parent1[i]);
222     used[parent1[i].id]=1;
223 }
224
225 for(int i=0;i<parent2.size();i++){
226     if(used[parent2[i].id]==0)
227         child.pb(parent2[i]);
228 }
229
230 return child;
231 }
232 }
233
234 // Using mating pool to breed children. Using elitism to retain the best
routes.
235 vector<vector<City>> breedPopulation(vector<vector<City>>matingpool, int
elite_size){
236
237     vector<vector<City>>children;
238     children.resize(matingpool.size());
239     int length=matingpool.size()-elite_size;
240
241     for(int i=0;i<elite_size;i++){
242         children[i]=(matingpool[i]);
243
244     shuffle(matingpool.begin(),matingpool.end(),std::default_random_engine(
rand()));
245
246     // Filling the rest of the generation
247     #pragma omp parallel for
248     for(int i=0;i<length;i++){
249
250         vector<City>child=breed(matingpool[i],matingpool[matingpool.size()-
i-1]);
251         children[i+elite_size]=(child);
252
253     }
254
255     return children;
256 }
257 }
258
259 /* Using swap mutation to mutate.
260 Helps in avoiding local convergence
261 mutationRate-> Probability that two cities will swap their position
262 */
263 vector<City> mutate(vector<City>&individual, double mutationRate){
264
265     for(int swapped=0;swapped<individual.size();swapped++){
266
267         double prob=(double)rand()/RAND_MAX;
268
269         if(prob<mutationRate){
270

```

```

271         double swapWith=(double)rand()/RAND_MAX;
272         swapWith*=individual.size();
273
274         swap(individual[swapped], individual[swapWith]);
275
276     }
277
278 }
279
280 return individual;
281
282 }
283
284 // Using mutate function to mutate the complete population
285 vector<vector<City>> mutatePopulation( vector<vector<City>>population ,
    double mutationRate){
286
287     vector<vector<City>>mutatedPop;
288     mutatedPop.resize(population.size());
289
290     for(int ind=0;ind<population.size();ind++){
291         vector<City>mutatedInd = mutate(population[ind],mutationRate);
292         mutatedPop[ind]=(mutatedInd);
293     }
294
295     return mutatedPop;
296
297 }
298
299 // Produces a new generation using all the functions above
300 vector<vector<City>> nextGeneration( vector<vector<City>>currentGen , int
    elite_size , double mutationRate){
301
302     // Rank the routes in the current generation
303     vector<vector<City>> popRanked=rankRoutes(currentGen);
304
305     // cout<<routeDistance(popRanked[0])<<" ";
306
307     // Determining potential parents and creating mating pool
308     vector<vector<City>> matingpool=mating_Pool(popRanked, elite_size);
309
310     // Creating new generation
311     vector<vector<City>> children=breedPopulation(matingpool, elite_size);
312
313     // Applying mutation
314     vector<vector<City>> nextGen=mutatePopulation(children, mutationRate);
315
316     return nextGen;
317
318 }
319
320 void Genetic_Algo( vector<City>population , int popSize , int elite_size ,
    double mutationRate , int generations){
321
322     // Creating initial population from city list
323     vector<vector<City>>pop=initialPopulation(popSize , population);
324
325     cout<<"Initial Distance was: "<<routeDistance(pop[0])<<endl;

```

```

326
327     for (int i=0;i<generations;i++){
328
329         fitness_scores.clear();
330         fitness_scores.resize(popSize);
331         pop=nextGeneration(pop, elite_size , mutationRate);
332
333     }
334
335
336     cout<<"Final Distance is: "<<routeDistance(pop[0])<<endl;
337
338 }
339
340 int main(int argc, char **argv){
341     ios_base::sync_with_stdio(false); cin.tie(NULL); cout.tie(NULL);
342
343     omp_set_num_threads(atoi(argv[1]));
344
345     int no_of_cities=40; // No of cities
346     int popSize=800;     // Population Size
347     int eliteSize=19;    // Elite Size
348     double mutationRate=0.01; // Rate of mutation
349     int generations=1000; // No of generations
350
351     vector<City>cityList; // Stores the initial list of cities
352
353     // Assigning random coordinates to cities
354     for (int i=0;i<no_of_cities;i++){
355         double x=200*((double)rand()/RAND_MAX);
356         double y=200*((double)rand()/RAND_MAX);
357         cityList.pb(City(x,y));
358     }
359
360     double start_time=omp_get_wtime();
361
362     Genetic_Algo(cityList , popSize , eliteSize , mutationRate , generations);
363
364     double final_tot=omp_get_wtime()-start_time;
365     // Prints time taken
366     cout<<fixed<<setprecision(5) <<"Time Taken by openmp program: "<<
final_tot <<" s "<<endl;
367
368     return 0;
369 }

```