Indian Institute of Technology Indore

# Report for Lab Assignment 4 - RDA

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 Red Deer 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. Red Deer algorithm is a population based meta heuristic algorithm which can be used to find near optimal solutions to difficult problems. It continues refining the search space to reach the minimum value while considering the exploration and exploitation phases satisfactorily.

The Red Deer algorithm for TSP proceeds in the following steps:

- **Creating population:** We select random numbers between 0 and 1 to proceed with our algorithm and generate 'popSize' number of individuals (Red Deer). Encoding scheme of meta-heuristic algorithms is used to generate city routes using these values. The loop generating RDs is **parallelized**.

```
        #pragma omp parallel for
        for(int i=0;i<pop_size;i++){
            RD_list[i].resize(no_of_cities);
            for(int j=0;j<no_of_cities;j++){
                RD_list[i][j]=get_rand();
            }
        }

```

- **Determining fitness:** We convert the current Red Deer to route using encoding scheme and calculate the distance of the route taken.
- **Ranking Red Deers:** We sort the population in the order of increasing 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 ascending order.

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

```

```
        #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);

    }

```

- **Separating Male from Hinds:** First,'no_of_males' number of individuals with minimum distance are selected as males and the rest of the RDs are considered to be hinds. Notably, male RDs are the good solutions in this algorithm. Here the male RDs maintain the intensification properties, while hind RDs consider the diversification phase of the algorithm.
- **Roaring male RDs:** Male RDs try to increase their grace by roaring. We find the neighbors of the male RD, and if the objective functions of the neighbors are better than the male RD, we replace them with the prior ones. The following equation is used to update the position of males:

$$male_{new} = male_{old} + a_1 * ((UB - LB) * a_2) + LB : if a_3 >= 0.5$$
$$male_{new} = male_{old} - a_1 * ((UB - LB) * a_2) + LB : if a_3 >= 0.5,$$

  where a1, a2 and a3 are generated randomly by a uniform distribution between zero and one. If the new position is better than the previous one, the position is updated.
- **Select commanders and stags:** Male RDs are divided into two types, namely commanders and stags. The number of commander RDs are given by:

$$N_{comm} = \gamma * (N_{male})$$

$$N_{stags} = N_{male} - N_{comm}$$

- **Fight between commanders and stags:** Each commander fights with every stag and two new solutions are generated. The best among the commander and generated solutions is assigned the new commander. The new solutions are generated using following equations:

$$New_1 = (Comm + Stag)/2 + b_1 * ((UB - LB) * b_2) + LB$$

$$New_2 = (Comm + Stag)/2 - b_1 * ((UB - LB) * b_2) + LB,$$

  where b1 and b2 are generated randomly by a uniform distribution between zero and one.

- **Forming harems:** Here, we form the harems. A harem is a group of hinds in which a male commander seized them. The number of hinds in harems depends on the power of male commanders

- **Mating process for commanders:** Commanders mate with $\alpha$ percent of hinds in his harem and $\beta$ percent of hinds in some randomly selected harem. The offspring is generated using the following formula:

$$offs = (Comm + Hind)/2 + (UB - LB) * c,$$

  where c is generated randomly by a uniform distribution between zero and one. This process is **parallelized** using the below technique:

```
    #pragma omp parallel
    {
        vector<vector<double>>offs_temp;
        for(int i=0;i<commanders.size();i++){
            for(int j=0;j<alpha*harems[i].size();j++){

                c=get_rand();
                vector<double>child(sz);
                for(int k=0;k<sz;k++){
                    child[k]=(commanders[i][k]+harems[i][j][k])/2+(UB-LB)*c;
                }
                offs_temp.pb(child);
            }
        }

        #pragma omp critical
        offs.insert(offs.end(),offs_temp.begin(),offs_temp.end());
    }
```

- **Mating of stags with their nearest hind:** Each stag mates with its closest hind. The offspring is generated using the same formula as above. This process is also **parallelized** using the same technique as above.

- **Selecting the next generation:** To select the next generation, two different strategies have been followed. In the first one, we keep all the male RD in the next generation. The second strategy refers to the remainder of the population in the next generation. We choose hinds out of all hinds and offspring generated by mating process regarding the fitness value by using the roulette wheel mechanism.

```
#pragma omp parallel for reduction(+:tot_fitness)
    for(int i=0;i<offs.size();i++){
        fitnessscores[i]=routeFitness(offs[i], index);
        tot_fitness+=fitnessscores[i];
    }
```

```
#pragma omp parallel
    {
        vector<vector<double>>nextGen_temp;

        #pragma omp for nowait
        for(int i=0;i<pop_size-males.size();i++){

            double pick=get_rand();
            pick*=100;
            for(int j=0;j<fitnessscores.size();j++){

                if(j==fitnessscores.size()-1){
                    nextGen_temp.pb(offs[j]);
                    break;
                }
                if(pick>100-fitnessscores[j]){
                    // cout<<j<<endl;
                    nextGen_temp.pb(offs[j]);
                    break;
                }

            }
        }

        #pragma omp critical
        nextGen.insert(nextGen.end(),nextGen_temp.begin(),nextGen_temp.end());

    }
```

## 1.3 Input

The input is provided as variables in the program.

- The user can select the number of cities, population size, number of males, alpha, beta, gamma and no of generations.

## 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.

## 1.5 Runtime and Convergence Analysis

To evaluate the performance for different cases, I used **three different values for population sizes** (100, 200 and 400) while keeping number of cities = 40, Number of males=30, alpha=0.9, beta=0.4, gamma=0.7 and number of generations=1000. I compared the runtime of each test case for serial and parallel programs and generated graphs to visualise the performance.

### 1.5.1 Test Case 1 (Population size=100)

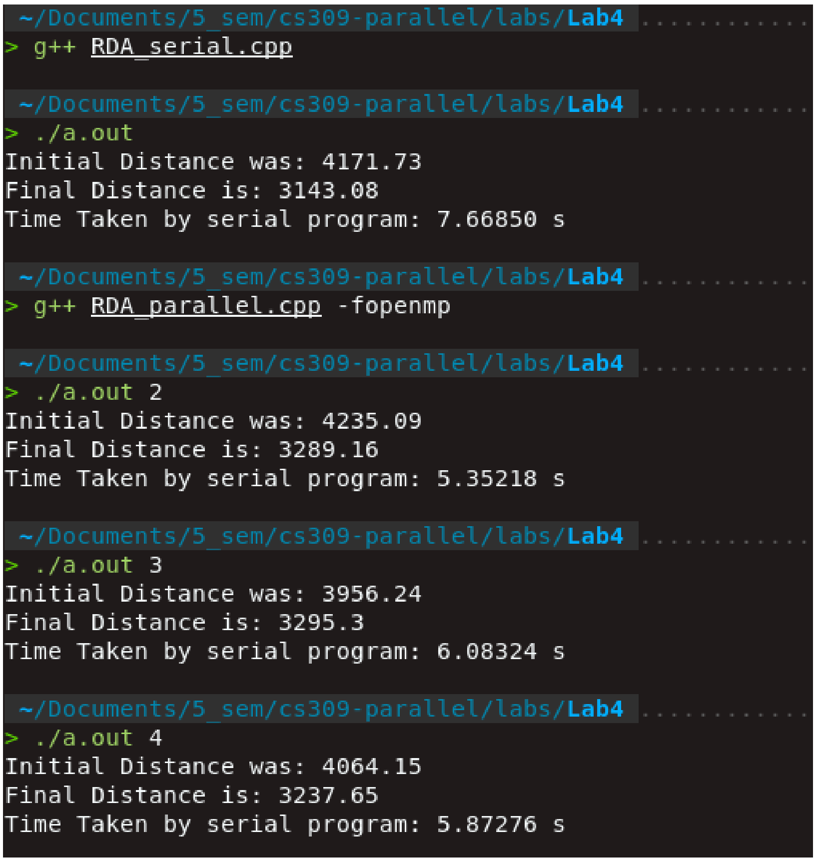


Figure 1.1: Output when population=100

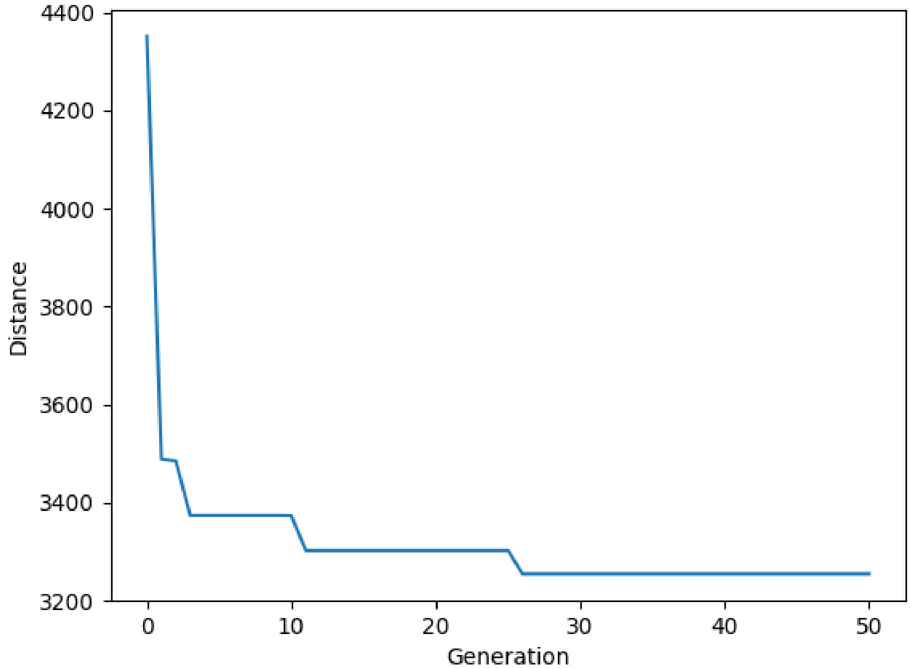


Figure 1.2: Convergence Graph when population=100

### 1.5.2 Test Case 2 (Population size=200)

```
~/Documents/5_sem/cs309-parallel/labs/Lab4
> g++ RDA_serial.cpp

~/Documents/5_sem/cs309-parallel/labs/Lab4
> ./a.out
Initial Distance was: 3428.08
Final Distance is: 3196.64
Time Taken by serial program: 13.31405 s

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

~/Documents/5_sem/cs309-parallel/labs/Lab4
> ./a.out 2
Initial Distance was: 3932.9
Final Distance is: 2987.16
Time Taken by serial program: 11.22058 s

~/Documents/5_sem/cs309-parallel/labs/Lab4
> ./a.out 3
Initial Distance was: 4167.99
Final Distance is: 3019.71
Time Taken by serial program: 10.69063 s

~/Documents/5_sem/cs309-parallel/labs/Lab4
> ./a.out 4
Initial Distance was: 4380.34
Final Distance is: 3229.53
Time Taken by serial program: 8.08117 s
```

Figure 1.3: Output when population=200

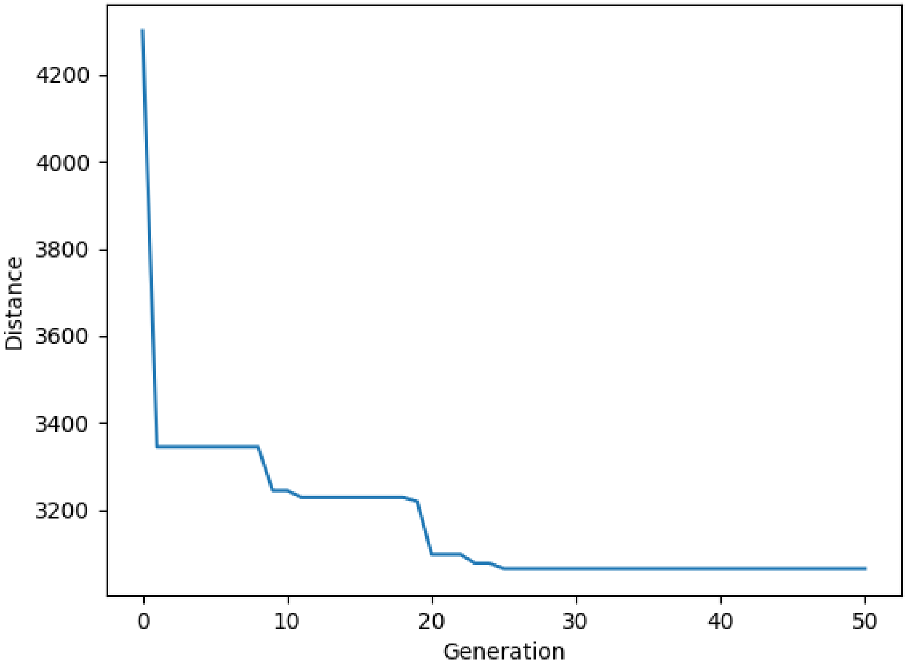


Figure 1.4: Convergence Graph when population=200

### 1.5.3 Test Case 3 (Population size=400)

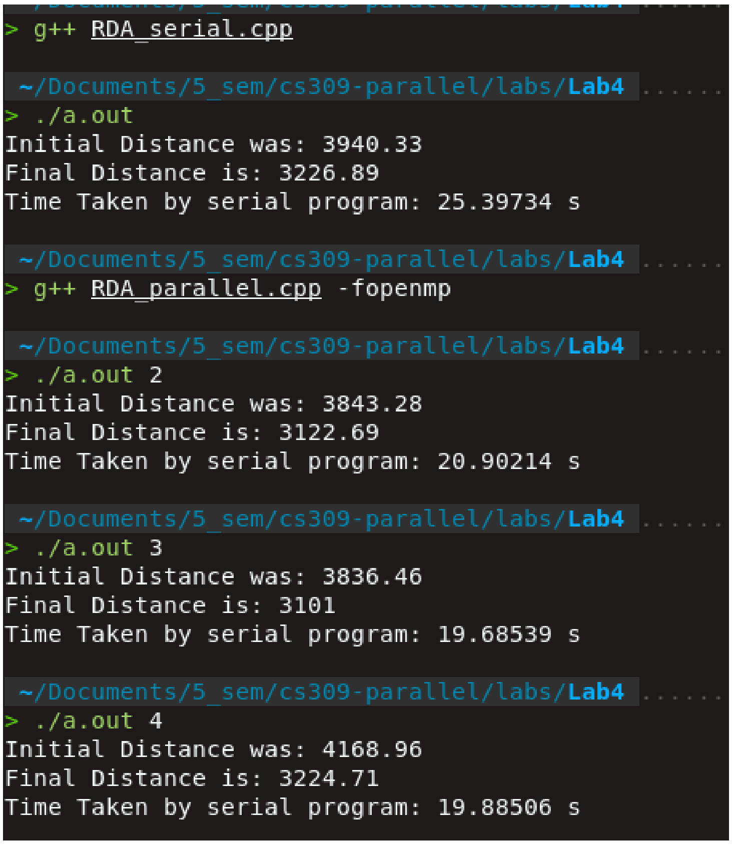


Figure 1.5: Output when population=400

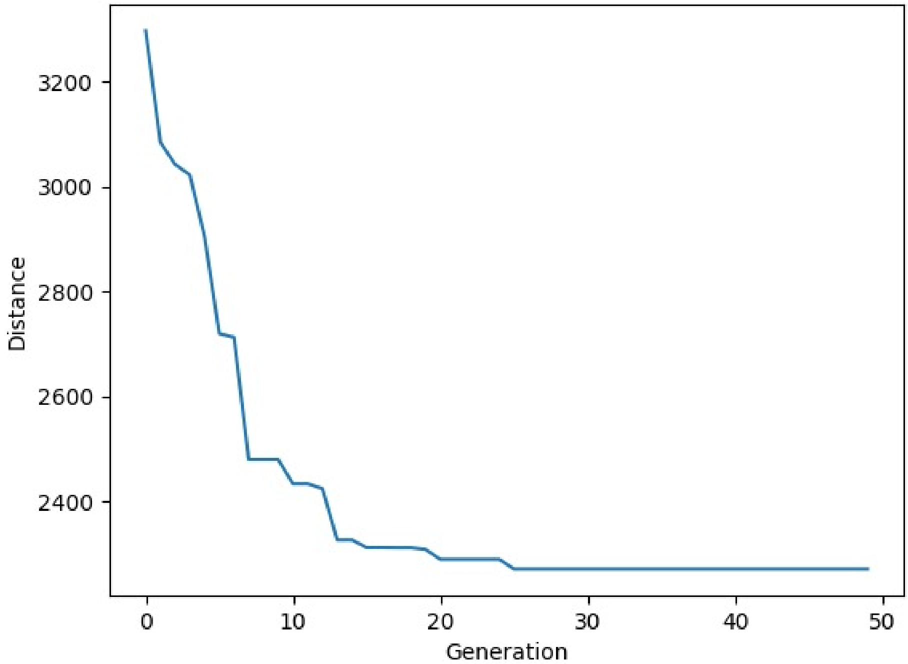


Figure 1.6: Convergence Graph when population=400

| | Runtime (in sec) | | | |
|---|---|---|---|---|
| Population | 1 Process | 2 Processes | 3 Processes | 4 Processes |
| 100 | 7.66850 | 5.35218 | 6.08324 | 5.87276 |
| 200 | 13.31405 | 11.22058 | 10.69063 | 8.08117 |
| 400 | 25.39734 | 20.90214 | 19.68539 | 19.88506 |

Table 1.1: Run-Times

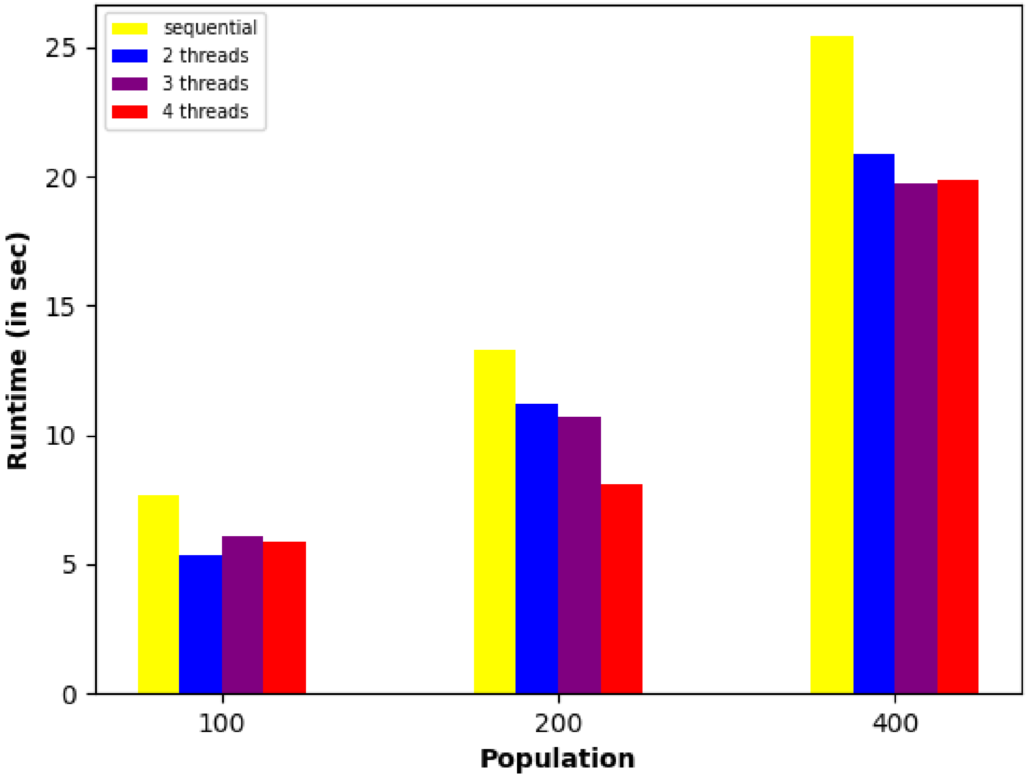


Figure 1.7: Change in runtime with change in population for serial and parallel programs

## 1.6 Conclusion

We conclude that the runtime usually reduces when the number of threads are increased while keeping population constant. The speedup is not as expected because of the pre-calculations and overhead involved in the algorithm. Along with this, there are certain steps in the algorithm that cannot be parallelized due to data dependency and critical regions. We also observe an increase in the speedup with increase in population size due to the dependency of parallelized loops on 'popSize'.

## 1.7 Implementation

```cpp
/*
Author: Aniket Sangwan (180001005)
*/

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

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

const int inf=1e6;

mt19937 rnd(chrono::system_clock::now().time_since_epoch().count());
const double RANDOM_MAX = 4294967295;

// Returns random value
double get_rand(){
    return (double)rnd() / RANDOM_MAX;
}


vector<double>fitness_scores; // Stores the fitness values of every Red Deer

double LB=-1, UB=1; // Lower and Upper Bound used in the algorithm

// 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

    // 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<double>route, City index[]){
```

```
    double path_dist=0,fitness=0.0;

    vector<double>sorted_RD=route;
    sort(sorted_RD.begin(), sorted_RD.end());
    map<double,int>RD_to_city;

    for(int i=0;i<sorted_RD.size();i++){
        RD_to_city[sorted_RD[i]]=i;
    }

    // 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= index[RD_to_city[route[i]]];
        City toCity= index[RD_to_city[route[i+1]]];

        path_dist+=fromCity.distance(toCity);

    }

    return path_dist;

}

/* Returns fitness value of given route.
We aim to minimize the distance. Here, I am trying to minimize the fitness
    too
*/
double routeFitness(vector<double>route, City index[]){

    double fitness=routeDistance(route, index);
    return fitness;

}


// Randomly creates initial population of Red deer using randomizer
vector<vector<double>> initialPopulation(int pop_size,int no_of_cities){

    vector<vector<double>>RD_list;

    RD_list.resize(pop_size);

    #pragma omp parallel for
    for(int i=0;i<pop_size;i++){
        RD_list[i].resize(no_of_cities);
        for(int j=0;j<no_of_cities;j++){
            RD_list[i][j]=get_rand();
        }
    }

    return RD_list;

```

```
}

// Sorts the population in increasing fitness score and returns in a new vector
vector<vector<double>> rankRDs(vector<vector<double>>population, City index[]){

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

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

    // Sorts in ascending order wrt fitness value
    sort(fitnessResults.begin(),fitnessResults.end());

    vector<vector<double>>sorted_population; // Will store population sorted in increasing 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;

}

// Separates males and hinds by separating 'no_of_males' Red deers with minimum fitness values
vector<vector<vector<double>>> Separate_male_and_hinds(vector<vector<double>>population, int no_of_males, int no_of_hinds, City index[]){

    vector<vector<vector<double>>>Separate;
    Separate.resize(2);

    Separate[0].resize(no_of_males);
    Separate[1].resize(no_of_hinds);

    for(int i=0;i<no_of_males;i++)
        Separate[0][i]=population[i]; // Store males


    for(int i=no_of_males;i<population.size();i++)
        Separate[1][i-no_of_males]=population[i]; // Stores Hinds

    return Separate;
}

```

```
/* Simulating the roaring process.
Finds the neighbours of male RD and if their OF is better, we replace them
    with the male_new
*/
vector<vector<double>> roar(vector<vector<double>>males, City index[]){

    double a1=get_rand(),a2=get_rand(),a3=get_rand();

    for(int i=0;i<males.size();i++){

        vector<double>male_new(males[i].size());

        // Updating the position of males
        for(int j=0;j<males[i].size();j++){
            if(a3>=0.5)
                male_new[j]=males[i][j]+a1*((UB-LB)*a2)+LB;
            else
                male_new[j]=males[i][j]-a1*((UB-LB)*a2)+LB;

        }
        if(routeFitness(male_new,index)<routeFitness(males[i],index))
            males[i]=male_new;
    }

    return males;

}

// Separates commanders and stags by separating gamma*(no_of_males) with
    minimum fitness into commanders
vector<vector<vector<double>>> Separate_commanders_and_stags(vector<vector<
    double>>males, double gamma){

    int no_of_commanders=gamma*males.size();
    vector<vector<vector<double>>>Separate;
    Separate.resize(2);

    Separate[0].resize(no_of_commanders);
    Separate[1].resize(males.size()-no_of_commanders);

    for(int i=0;i<no_of_commanders;i++)
        Separate[0][i]=males[i];


    for(int i=no_of_commanders;i<males.size();i++)
        Separate[1][i-no_of_commanders]=males[i];

    return Separate;
}

/*
Each commander fights with every stag and two new solutions are generated.
    The best among the
commander and generated solutions is assigned the new commander
*/
void Fight(vector<vector<double>>&commanders, vector<vector<double>>stags,
    City index[]){

```

```
    int no_of_stags=stags.size();
    int sz=commanders[0].size();

    for(int i=0;i<commanders.size();i++){

        for(int j=0;j<stags.size();j++){
        vector<double> New1(sz),New2(sz);    // Two new solutions
        double b1=get_rand(),b2=get_rand();

            for(int k=0;k<sz;k++){
                New1[k]=(commanders[i][k]+stags[j][k])/2.0+b1*((UB-LB)*b2)+LB;
                New2[k]=(commanders[i][k]+stags[j][k])/2.0-b1*((UB-LB)*b2)+LB;
            }

            // Calculating minimum fitness and
            double fitness1=routeFitness(commanders[i], index);
            double fitness2=routeFitness(New1, index);
            double fitness3=routeFitness(New2, index);
            double minfitness=min(fitness1,min(fitness2,fitness3));

            if(fitness2==minfitness)commanders[i]=New1;
            else if(fitness3==minfitness)commanders[i]=New2;
        }
    }

}

// Forming harems. Hinds are divided proportionally among male commanders according to their power
vector<vector<vector<double>>> form_harems(vector<vector<double>>& commanders, vector<vector<double>>hinds, City index[]){

    double tot_fitness=0;    // Stores total fitness
    shuffle(hinds.begin(),hinds.end(),std::default_random_engine(rand()));

    // Stores fitness values
    vector<double>fitness;
    fitness.resize(commanders.size());

    for(int i=0;i<commanders.size();i++){
        fitness[i]=routeFitness(commanders[i], index);
        tot_fitness+=fitness[i];
    }

    //
    vector<vector<vector<double>>>harems; // Stores the hinds in each harem
    harems.resize(commanders.size());

    int sz=commanders[0].size();
    int hinds_taken=0;  // No of hinds assigned to harems
    int no_of_hinds=hinds.size();    // Total no of hinds
    int harem_size=(fitness[0]/tot_fitness)*no_of_hinds;     // No of hinds in next harem

    // Proportionally assigning harems
    for(int i=0;i<commanders.size();i++){
```

```
        harems[i].resize(harem_size);
        for(int j=hinds_taken;j<hinds_taken+harem_size;j++){

            harems[i][j-hinds_taken]=(hinds[j]);

        }
        hinds_taken+=harem_size;
        if(i<commanders.size()-2)
            harem_size=(fitness[i+1]/tot_fitness)*no_of_hinds;
        else
            harem_size=no_of_hinds-hinds_taken;
    }

return harems;

}

/*
Mating commander of a harem with alpha percent of hinds in his harem
Mating commander of a harem with beta percent of hinds in another random
    harem
*/
vector<vector<double>> Mate_alpha_beta(vector<vector<double>>commanders,
    vector<vector<vector<double>>>harems, double alpha, double beta){

    int sz=commanders[0].size();
    int c=get_rand();
    vector<vector<double>>offs;

    #pragma omp parallel
    {
        vector<vector<double>>offs_temp;
        for(int i=0;i<commanders.size();i++){
            for(int j=0;j<alpha*harems[i].size();j++){

                c=get_rand();
                vector<double>child(sz);
                for(int k=0;k<sz;k++){
                    child[k]=(commanders[i][k]+harems[i][j][k])/2+(UB-LB)*c
    ;
                }
                offs_temp.pb(child);
            }
        }

        #pragma omp critical
        offs.insert(offs.end(),offs_temp.begin(),offs_temp.end());
    }

    #pragma omp parallel
    {
        vector<vector<double>>offs_temp;
        for(int i=0;i<commanders.size();i++){

            int w=rand()%harems.size(); // Randomly selected harem
            while(w==i) w=rand()%harems.size();

            for(int j=0;j<beta*harems[w].size();j++){
```

```
                    c=get_rand();
                    vector<double>child(sz);
                    for(int k=0;k<sz;k++){
                        child[k]=(commanders[i][k]+harems[w][j][k])/2+(UB-LB)*c
    ;
                    }
                    offs_temp.pb(child);
                }
            }

            #pragma omp critical
            offs.insert(offs.end(),offs_temp.begin(),offs_temp.end());
        }

return offs;
}

// Mating stag with the nearest hind
vector<vector<double>> Mate_stag_hind(vector<vector<double>>stags, vector<
    vector<double>>hinds, City index[]){

     vector<vector<double>>offs;

     // Finding hind with minimum distance from 'stags[i]' stag and mating
    with it
     #pragma omp parallel
     {
         vector<vector<double>>offs_temp;

         #pragma omp for nowait
         for(int i=0;i<stags.size();i++){
             int cindx=-1,cdist=inf;
             for(int j=0;j<hinds.size();j++){

                 vector<double>diff(stags[i].size());

                 for(int k=0;k<stags[i].size();k++)
                     diff[k]=abs(stags[i][k]-hinds[j][k]);

                 int diff_fitness=routeFitness(diff, index);
                 if(diff_fitness < cdist)
                     cdist=diff_fitness,cindx=j;

             }

             // Generating offspring with the selected hind
             int j=cindx;
             double c=get_rand();
             vector<double>child(stags[i].size());
             for(int k=0;k<stags[i].size();k++){
                 child[k]=(stags[i][k]+hinds[j][k])/2+(UB-LB)*c;
             }
             offs_temp.pb(child);

         }
         #pragma omp critical
         offs.insert(offs.end(), offs_temp.begin(), offs_temp.end());
     }
```

```
return offs;
}

// Selecting the next generation using elitism and roulette wheel mechanism
vector<vector<double>> NextGen(vector<vector<double>>offs, vector<vector<
    double>>males, int pop_size, City index[]){

     vector<vector<double>>nextGen;

     // All the males are selected in the next generation
     for(int i=0;i<males.size();i++){
          nextGen.pb(males[i]);
     }

     vector<double>fitnessscores;
     fitnessscores.resize(offs.size());

     double tot_fitness=0; // Stores total fitness

     #pragma omp parallel for reduction(+:tot_fitness)
     for(int i=0;i<offs.size();i++){
          fitnessscores[i]=routeFitness(offs[i], index);
          tot_fitness+=fitnessscores[i];
     }

     // Assigning fitness weighed probability
     for(int i=0;i<fitnessscores.size();i++){

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

     }

     // Using fitness proportionate selection (Roulette Wheel mechanism)
     #pragma omp parallel
     {
          vector<vector<double>>nextGen_temp;

          #pragma omp for nowait
          for(int i=0;i<pop_size-males.size();i++){

               double pick=get_rand();
               pick*=100;
               for(int j=0;j<fitnessscores.size();j++){

                    if(j==fitnessscores.size()-1){
                         nextGen_temp.pb(offs[j]);
                         break;
                    }
                    if(pick>100-fitnessscores[j]){
                         // cout<<j<<endl;
                         nextGen_temp.pb(offs[j]);
                         break;
                    }

               }
          }
```

```
        #pragma omp critical
        nextGen.insert(nextGen.end(),nextGen_temp.begin(),nextGen_temp.end
    ());

    }

return nextGen;
}

// Using all the above functions to get next generation
vector<vector<double>> nextGeneration(vector<vector<double>>population, int
     popSize, int no_of_males, double alpha, double beta, double gamma, City
     index[]){

    vector<vector<double>>popRanked=rankRDs(population, index);

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

    vector<vector<vector<double>>>separate = Separate_male_and_hinds(
    popRanked, no_of_males, popSize-no_of_males, index);
    vector<vector<double>>Males=separate[0];
    vector<vector<double>>Hinds=separate[1];

    Males=roar(Males, index);
    Males=rankRDs(Males, index);

    separate=Separate_commanders_and_stags(Males, gamma);
    vector<vector<double>>Commanders=separate[0];
    vector<vector<double>>Stags=separate[1];

    Fight(Commanders, Stags, index);

    vector<vector<vector<double>>>harems=form_harems(Commanders, Hinds,
    index);

    vector<vector<double>>offs=Mate_alpha_beta(Commanders, harems, alpha,
    beta);

    vector<vector<double>>offs1=Mate_stag_hind(Stags, Hinds, index);

    offs.insert(offs.end(),offs1.begin(),offs1.end());

    offs.insert(offs.end(),Commanders.begin(),Commanders.end());
    offs.insert(offs.end(),Stags.begin(),Stags.end());
    offs.insert(offs.end(),Hinds.begin(),Hinds.end());

    vector<vector<double>>nextGen=NextGen(offs, Males, popSize, index);

return nextGen;
}

void RD_Algo( vector<City>cityList, int popSize, int no_of_males, int
    generations,double alpha, double beta, double gamma, City index[]){

    // Creating initial population from city list
    vector<vector<double>>population;

```

```
    population=initialPopulation(popSize, cityList.size());

    cout<<"Initial Distance was: "<<routeDistance(population[0], index)<<endl;

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

        fitness_scores.clear();
        population=nextGeneration(population, popSize, no_of_males, alpha, beta, gamma, index);

    }
    cout<<"Final Distance is: "<<routeDistance(population[0], index)<<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=100;     // Population Size
    int no_of_males=30; // No of males
    int generations=100;    // No of generations

    double alpha=0.9;
    double beta=0.4;
    double gamma=0.7;

    int no_of_hinds = popSize-no_of_males;



    vector<City>cityList; // Stores the initial list of cities
    City index[no_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);
        City c;
        c.x=x,c.y=y,c.id=i;
        cityList.pb(c);
        index[i]=c; // Assigning index to cities
    }

    auto begin = chrono::high_resolution_clock::now();

    RD_Algo(cityList, popSize, no_of_males, generations, alpha, beta, gamma, index);

    auto end = chrono::high_resolution_clock::now();
    auto duration = chrono::duration_cast<chrono::microseconds>(end-begin);
    cout<<fixed<<setprecision(5) <<"Time Taken by serial program: "<<duration.count()/1000000.0 <<" s "<<endl;

return 0;
```

```
}
```