Assignment 4

Genetic Algorithm for TSP

Parallel Computing(CS 309)

RUCHIR MEHTA

Roll No: 180001044

Date: 23-04-2021

Problem Statement

The problem that is taken into consideration is the Travelling Salesman Problem. “Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?”

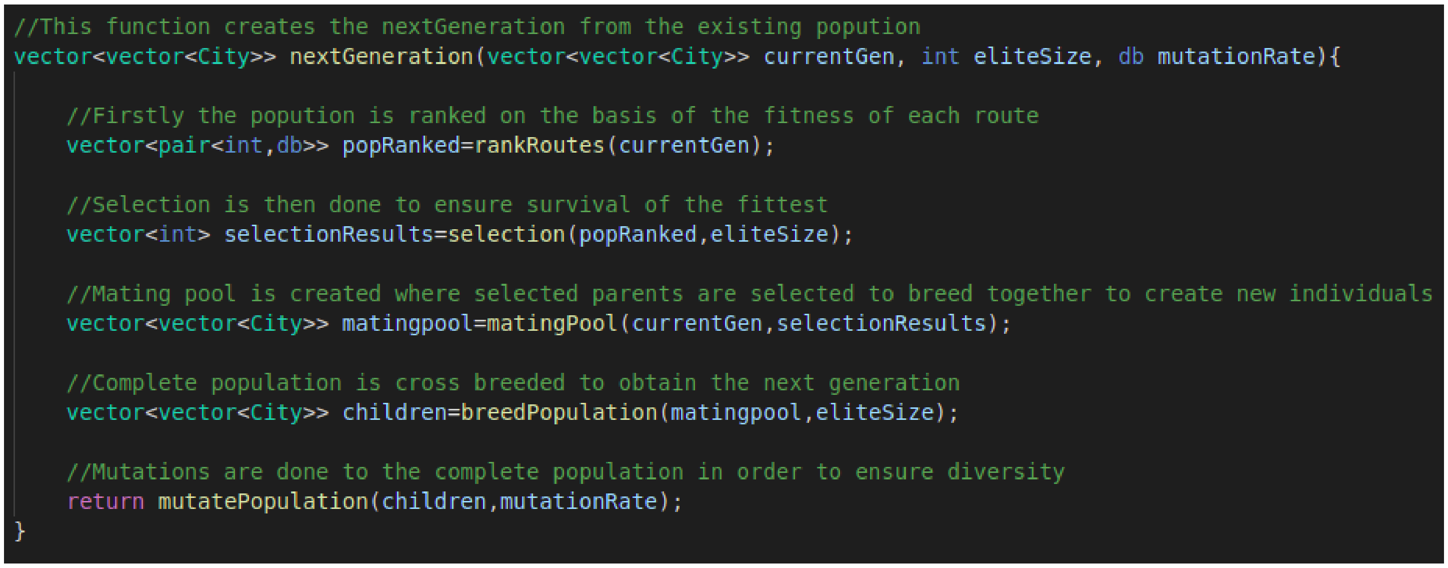
In this task, I have implemented a genetic algorithm that solves this famous travelling salesman problem. I have implemented both the serial as well as parallel versions of the algorithm. Parallel version is implemented in OpenMP that uses threads. References are here and here.

Code files for serial and parallel implementation: [Attached]

Approach:

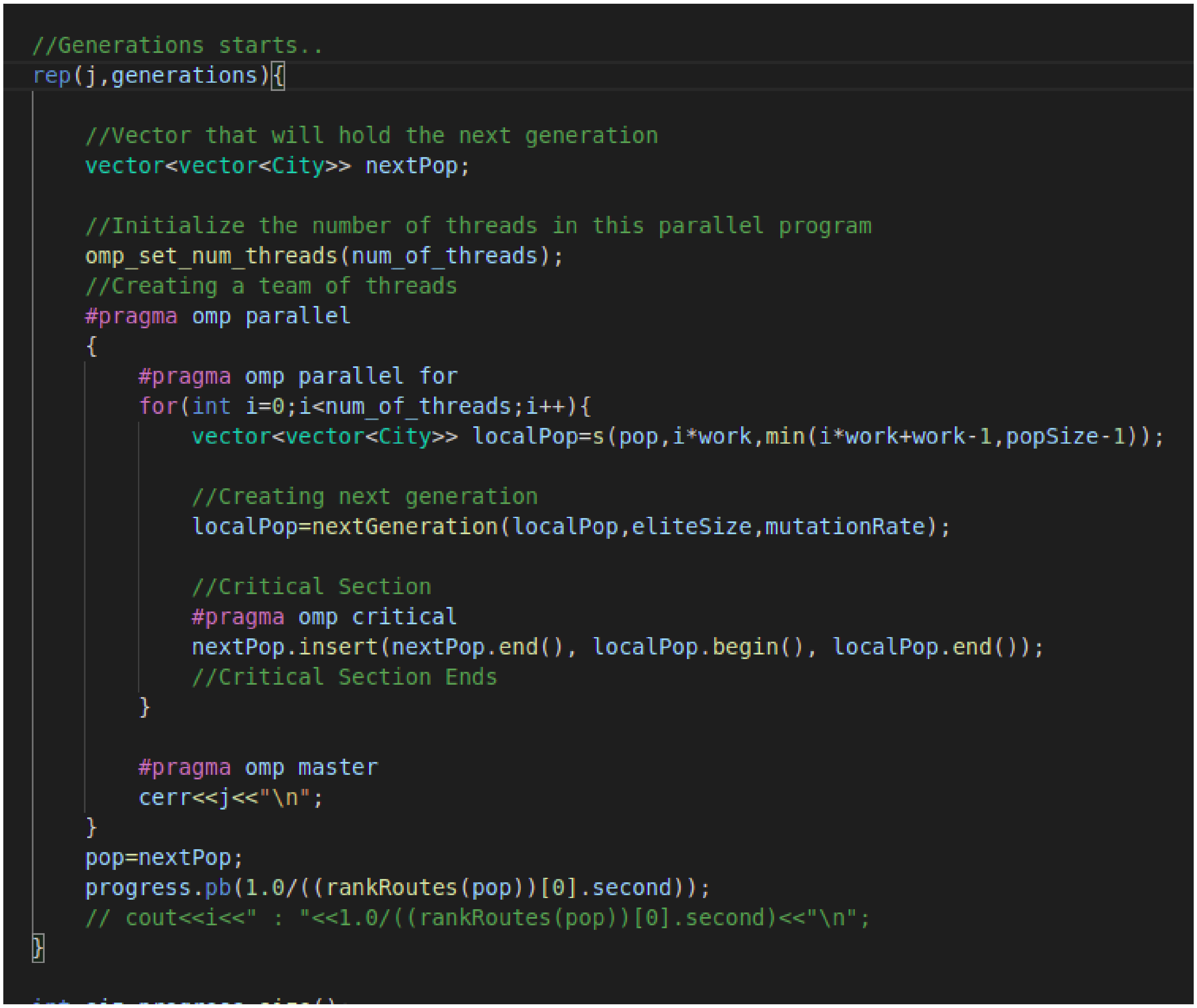
For this problem, I considered each route to be representing a solution where a route is more fit if its total path length is less. In approaching it via genetic algorithm, I firstly created a random population of routes and then in each generation, selection is done on the population to ensure survival of the fittest. Then, a mating pool is created where selected parents are selected to breed together to create new individuals. Then the complete population is cross breeded to obtain the next generation. Then mutations are done to the complete population in order to ensure diversity. Thus creating a brand new population and this process continues until n generations.

Main Code Snippet:



```
//This function creates the nextGeneration from the existing popution
vector<vector<City>> nextGeneration(vector<vector<City>> currentGen, int eliteSize, db mutationRate){

    //Firstly the popution is ranked on the basis of the fitness of each route
    vector<pair<int,db>> popRanked=rankRoutes(currentGen);

    //Selection is then done to ensure survival of the fittest
    vector<int> selectionResults=selection(popRanked,eliteSize);

    //Mating pool is created where selected parents are selected to breed together to create new individuals
    vector<vector<City>> matingpool=matingPool(currentGen,selectionResults);

    //Complete population is cross breeded to obtain the next generation
    vector<vector<City>> children=breedPopulation(matingpool,eliteSize);

    //Mutations are done to the complete population in order to ensure diversity
    return mutatePopulation(children,mutationRate);
}
```

Serial Execution



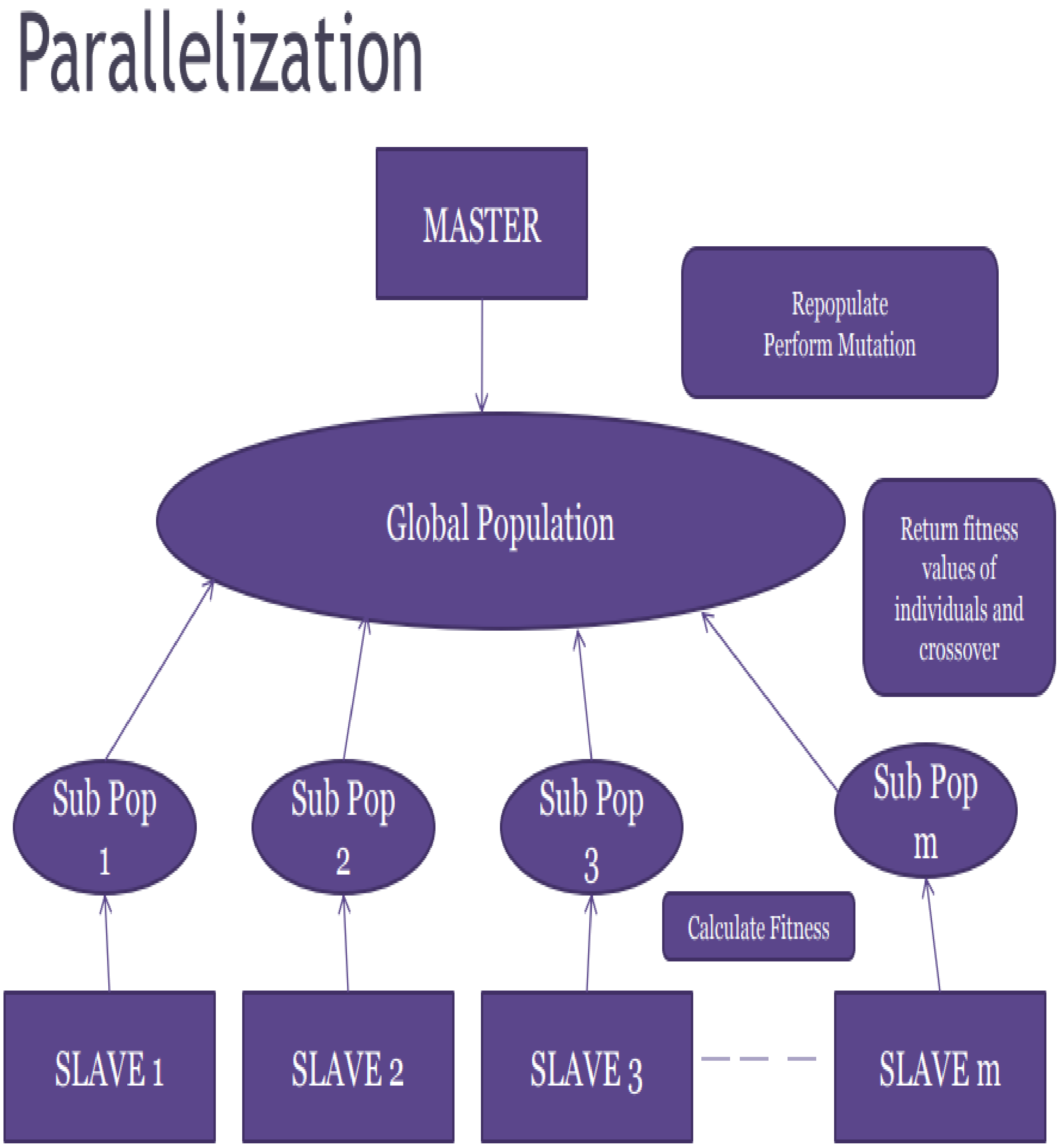
```cpp
//Generations starts..
rep(j,generations){

    //Vector that will hold the next generation
    vector<vector<City>> nextPop;

    //Initialize the number of threads in this parallel program
    omp_set_num_threads(num_of_threads);
    //Creating a team of threads
    #pragma omp parallel
    {
        #pragma omp parallel for
        for(int i=0;i<num_of_threads;i++){
            vector<vector<City>> localPop=s(pop,i*work,min(i*work+work-1,popSize-1));

            //Creating next generation
            localPop=nextGeneration(localPop,eliteSize,mutationRate);

            //Critical Section
            #pragma omp critical
            nextPop.insert(nextPop.end(), localPop.begin(), localPop.end());
            //Critical Section Ends
        }

        #pragma omp master
        cerr<<j<<"\n";
    }
    pop=nextPop;
    progress.pb(1.0/((rankRoutes(pop))[0].second));
    // cout<<i<<" : "<<1.0/((rankRoutes(pop))[0].second)<<"\n";
}
```

Parallel Execution

Here parallelization is done such that each thread gets a chunk of population and does selection, mating, breeding and mutating on that chunk and later merges it to the original but new population.

This approach uses OpenMP in C++.

Approach that I followed is represented in the figure below.

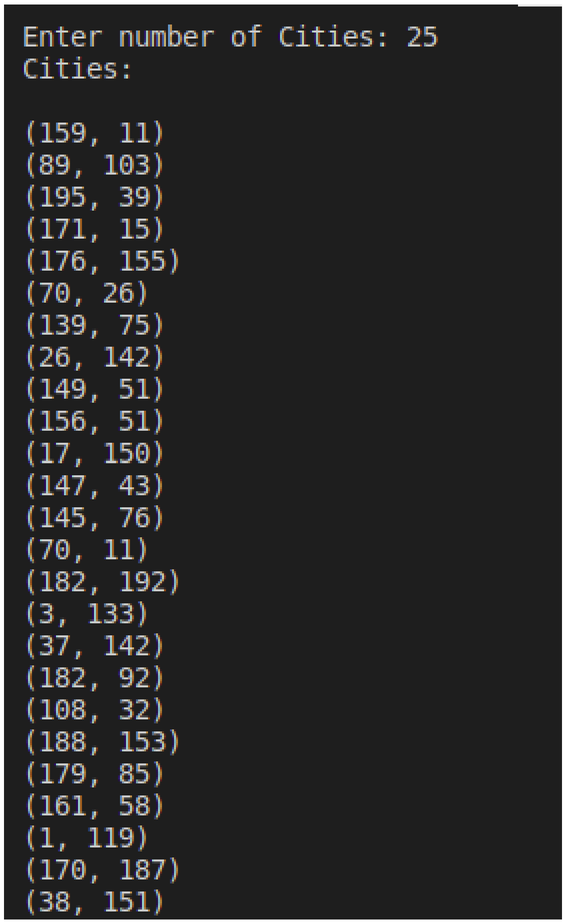


Source: https://cse.buffalo.edu/faculty/miller/Courses/CSE633/Sagar-Keer-F2010.pdf

Time Analysis with Test Cases(Time in seconds):

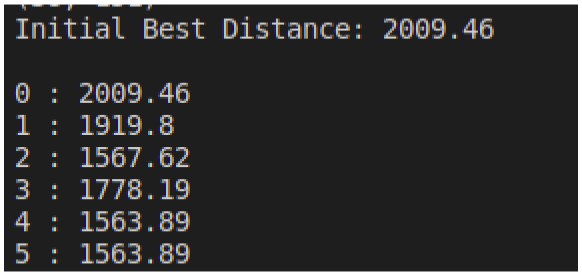
Serial Genetic Algorithm

1. **Test-Case**



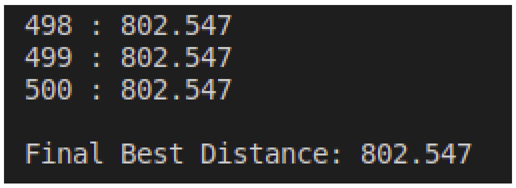
```
Enter number of Cities: 25
Cities:

(159, 11)
(89, 103)
(195, 39)
(171, 15)
(176, 155)
(70, 26)
(139, 75)
(26, 142)
(149, 51)
(156, 51)
(17, 150)
(147, 43)
(145, 76)
(70, 11)
(182, 192)
(3, 133)
(37, 142)
(182, 92)
(108, 32)
(188, 153)
(179, 85)
(161, 58)
(1, 119)
(170, 187)
(38, 151)
```

2. **Performance towards convergence to global minima**



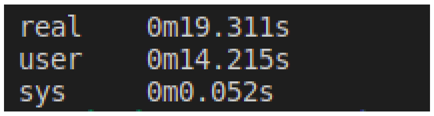
```
Initial Best Distance: 2009.46

0 : 2009.46
1 : 1919.8
2 : 1567.62
3 : 1778.19
4 : 1563.89
5 : 1563.89
```

…..after 500 generations



```
498 : 802.547
499 : 802.547
500 : 802.547

Final Best Distance: 802.547
```

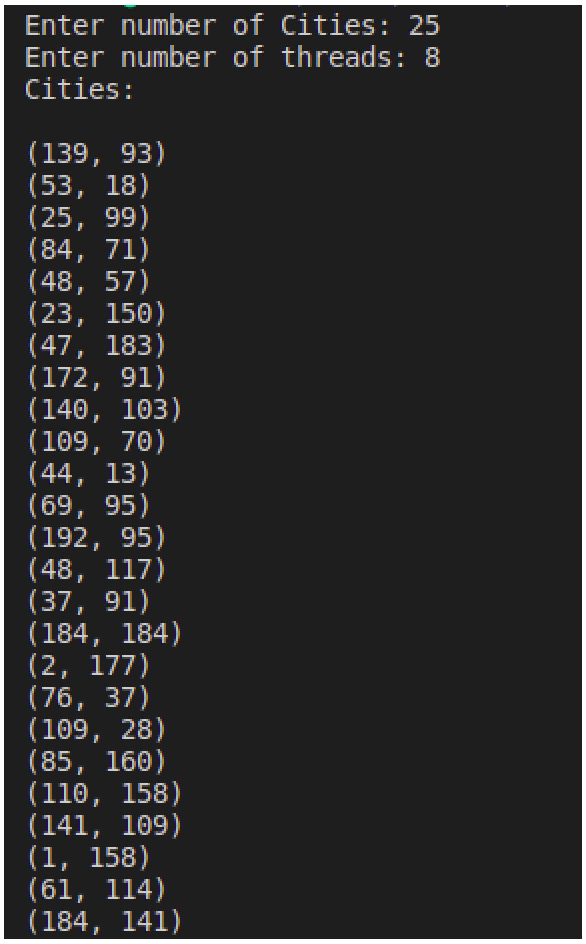
3. **Time Taken**



```
real    0m19.311s
user    0m14.215s
sys     0m0.052s
```

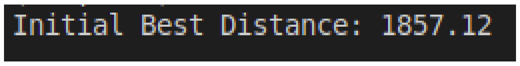
Parallel Genetic Algorithm

4. Test-Case

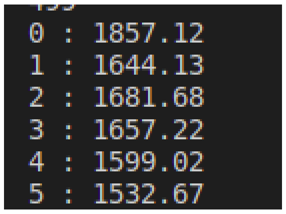


```
Enter number of Cities: 25
Enter number of threads: 8
Cities:

(139, 93)
(53, 18)
(25, 99)
(84, 71)
(48, 57)
(23, 150)
(47, 183)
(172, 91)
(140, 103)
(109, 70)
(44, 13)
(69, 95)
(192, 95)
(48, 117)
(37, 91)
(184, 184)
(2, 177)
(76, 37)
(109, 28)
(85, 160)
(110, 158)
(141, 109)
(1, 158)
(61, 114)
(184, 141)
```

5. Performance towards convergence to global minima

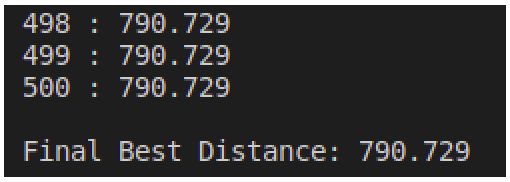


```
Initial Best Distance: 1857.12
```



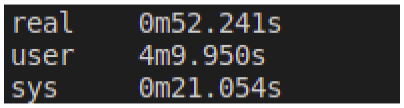
```
0 : 1857.12
1 : 1644.13
2 : 1681.68
3 : 1657.22
4 : 1599.02
5 : 1532.67
```

…..after 500 generations



```
498 : 790.729
499 : 790.729
500 : 790.729

Final Best Distance: 790.729
```

6. Time Taken



```
real    0m52.241s
user    4m9.950s
sys     0m21.054s
```

Observations:

I observed that, parallel algorithm works poorly than the serial algorithm. It is because the time it takes in creating the threads is larger in comparison to the time it takes to work parallely. I did not try for bigger test cases because the parallel algorithm took huge time to execute.