Genetic-Algorithms for TSP

Open-MP Parallelization

Shah Anwaar Khalid

CED18I048

Improvements since profiling

In the profiling experiment, we realized that there are a lot of optimizations that we can do in the code before parallelizing it. The following optimizations are done:

1. Fitness_score function: It is called by each genome in the population. Earlier, it would call the distance_to method to calculate the distance between two points. We realized that this leads to a lot of repetitive calculations. So, we stored the distances in a 2d vector & used this to update the fitness scores.

2. Sort Function: The standard library function sort is used to sort the population by fitness score values. We thought calling it just once & then using insetion_sort would reduce the time complexity from O(nlogn) at every iteration to O(K.n) but it turns out that k is always greater than logn. Therefore, we put the insertion_sort method in the naive implementation & sort method in the optimized implementation.

Just by doing these two updates, the following speed up was achieved:

1. Naive-Implementation:: Execution Time = 66.6776 for 500 cities
2. Optimized- Implementation: Execution time = 32.3772 for 500 cities

Great! We reduced the execution time by half by just making small & simple changes.

Parallelizing the execution with OPEN-MP

The following 4 for loops were parallezed:

1. **Get_fitness score function:**

```
/*
 * Computes the fitness score (path length) of the genome.
 */
double PathRepresentation::fitness_score(const vector<vector<double>> &distances) {
  size_t numPoints = genome.size();

  fitnessScore = 0;

#pragma omp parellel for default(shared) reduction(+:fitnessScore)
  for(size_t i = 0, j = 1; i < numPoints -1; ++i, ++j)
    fitnessScore += distances[genome.at(i)][genome.at(j)];

  // TS has to go back to the startting position.
  fitnessScore+= distances[genome.front()][genome.back()];

  return fitnessScore;
}
```

2. Filling the distances matrix:

```
#pragma omp parallel for num_threads(threads[thread])
    for(unsigned i = 0; i < numPoints ; ++i)
      for(unsigned j = 0; j < numPoints ; ++j)
        distances[i][j] = points[i].distance_to(points[j]);
```

3. Cross Linking code:

```
#pragma omp parallel for num_threads(threads[thread])

      for(unsigned i = keepPopulation; i < populationSize; ++i){
        unsigned indexA;
        unsigned indexB;

        // range [0 . . . keepPopulation - 1]
        indexA = rand() % keepPopulation;
        do {
          indexB = rand() % keepPopulation;
        } while( indexA == indexB);

        pair<PathRepresentation, PathRepresentation>  offsprings=
          CrossoverObject->crosslink(population[indexA], population[indexB]);

        // Get the fitness score of both the springs
        // & choose the fitter one.
        offsprings.first.fitness_score(distances);
        offsprings.second.fitness_score(distances);

        if (compare_paths(offsprings.first, offsprings.second) )
          population[i] = offsprings.first;
        else
          population[i] = offsprings.second;
      }
```

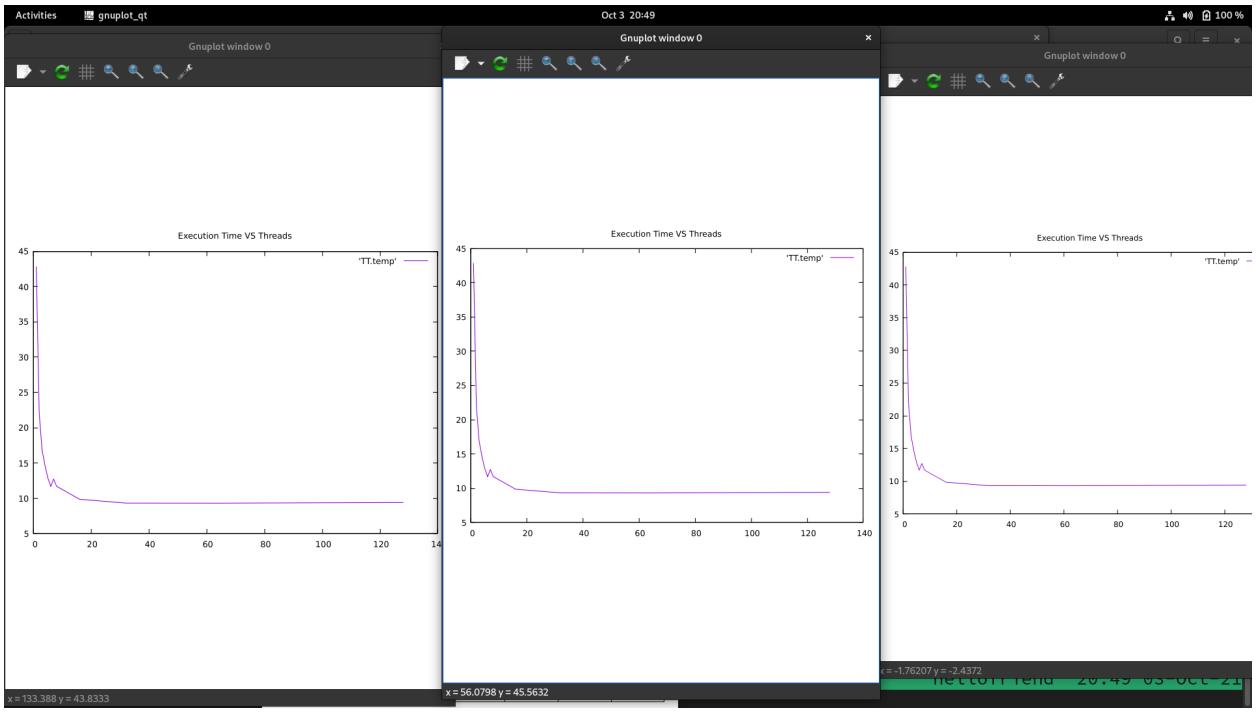
4. Mutation Code:

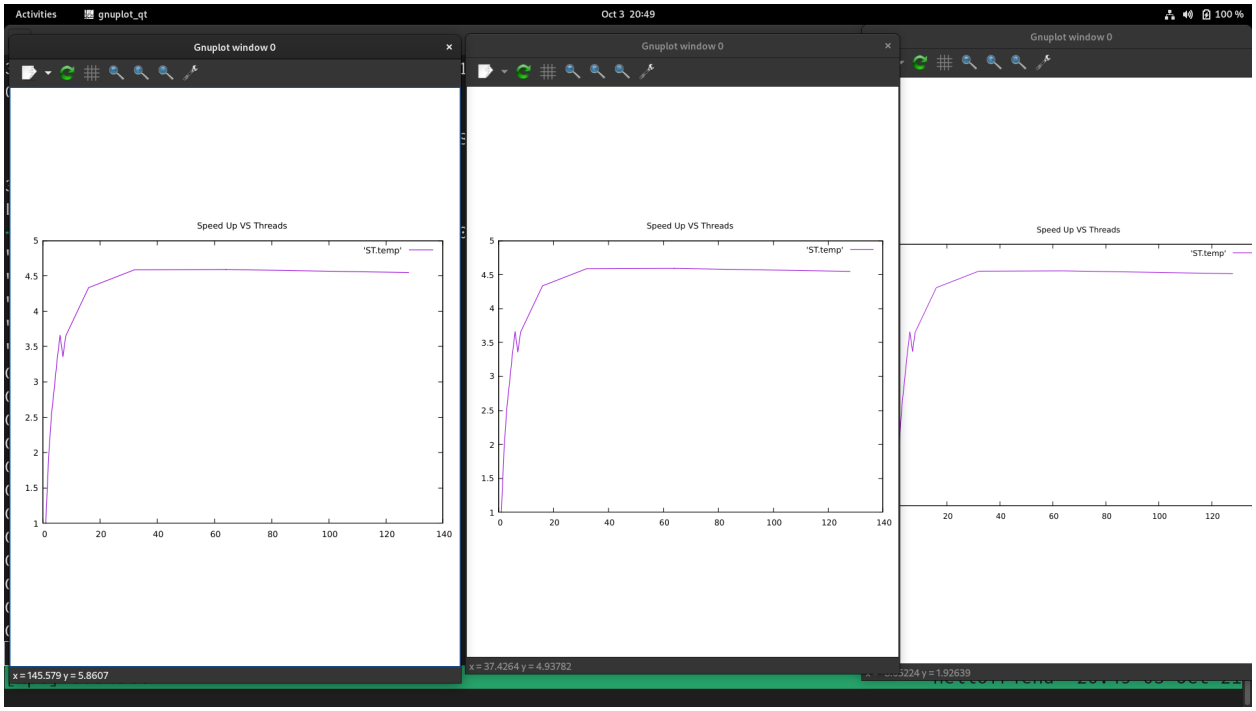
5.

```
    //Mutate random genes
#pragma omp parallel for num_threads(threads[thread])

    for(unsigned i = 0; i < numMutations; ++i){
      unsigned randomIndex;
      randomIndex = 1 + rand() %( populationSize - 1);
      population[randomIndex].mutate();
    }

    if(gen % 10 == 0){
      cout<<" Generation "<<gen/10<<" : shortest path found "
        <<population[0].get_fitness_score()<<endl;
    }
    }
```

Execution Time Vs Threads & Speed-Up Vs Threads:

The following graphs were obtained for the three crosslinking methods:




Activities
gnuplot_qt
Oct 3 20:49
100 %
Gnuplot window 0
Speed Up VS Threads
'ST.temp'
x = 145.579 y = 5.8607
Gnuplot window 0
Speed Up VS Threads
'ST.temp'
x = 37.4264 y = 4.93782
Gnuplot window 0
Speed Up VS Threads
'ST.temp'
y = 1.92639

Github Repository:

The changes are shared through the following Github repository:

https://github.com/hello-fri-end/Parallel-Implementation-of-Genetic-Algorithms-for-TSP

Conclusion/Inferences

We first reduced the execution time by half by doing small optimizations in the code. Further on parallelizing it, the best performance was 8 times faster than the naive implementation. Also, it can be observed from the graph that the performance doesn't improve after 16 threads.