Assignment 2

Profiling of Project

Shah Anwaar Khalid
Roll NO: CED18I048

Problem Statement

The travelling salesman problem, popularly known as TSP is an np-hard problem of finding a minimum Hamiltonian cycle on a complete graph with non-negative edges. A Hamiltonian cycle is a cycle that visits each node of the graph exactly once.

Why is the problem important?

The Travelling Salesman Problem is considered to be the holy-grail of computational problems. It has tremendous applications in real-life & therefore there's a need to solve the problem optimally.

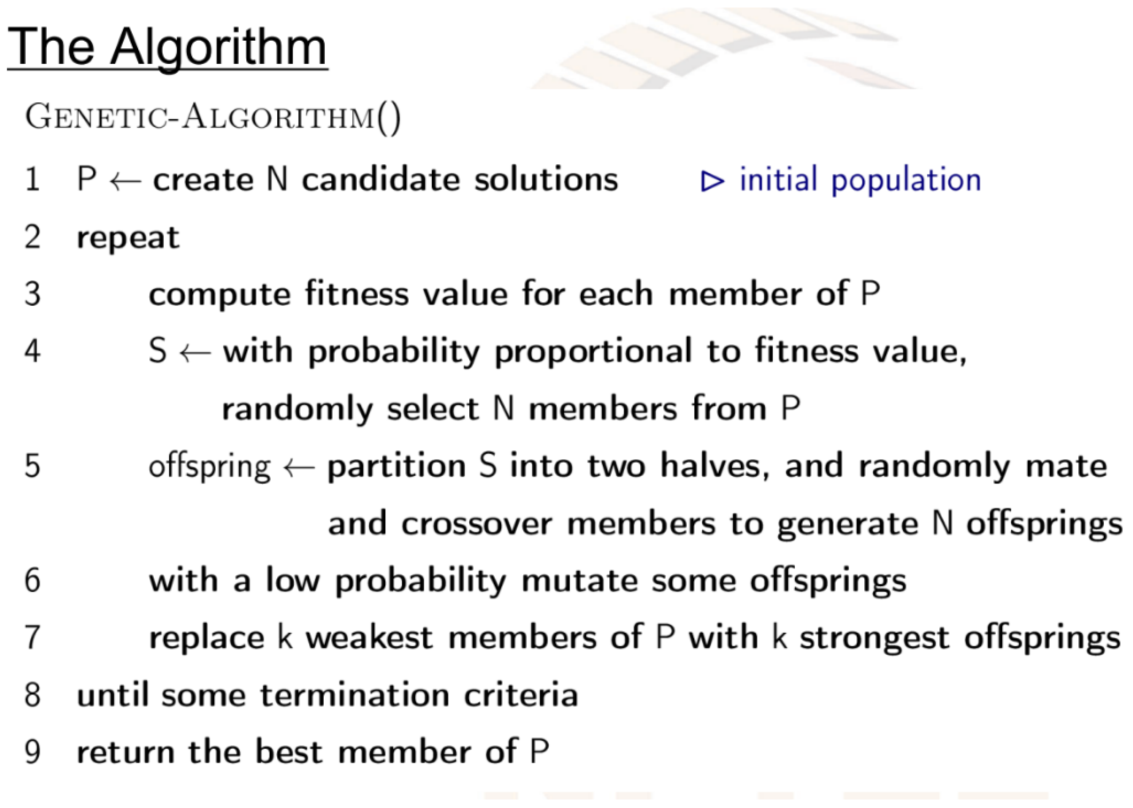
Computational Complexity

Given n cities, there exist as many as (n-1)! / 2 hamiltonian cycles which implies the no. of possible solutions to the TSP problem when no. of cities = 100 is of the order 10^157 which is much much larger than the no. of particles in the universe.

Genetic Algorithms

In computer science and operations research, a genetic algorithm (GA) is a metaheuristic inspired by the process of natural selection that belongs to the larger class of evolutionary algorithms (EA). Genetic algorithms are commonly used to generate high-quality solutions to optimization and search problems by relying on biologically inspired operators such as mutation, crossover and selection.

Pseudo - Code for Genetic Algorithm



The Algorithm

```
GENETIC-ALGORITHM()
1  P ← create N candidate solutions          ▷ initial population
2  repeat
3      compute fitness value for each member of P
4      S ← with probability proportional to fitness value,
           randomly select N members from P
5      offspring ← partition S into two halves, and randomly mate
                   and crossover members to generate N offsprings
6      with a low probability mutate some offsprings
7      replace k weakest members of P with k strongest offsprings
8  until some termination criteria
9  return the best member of P
```

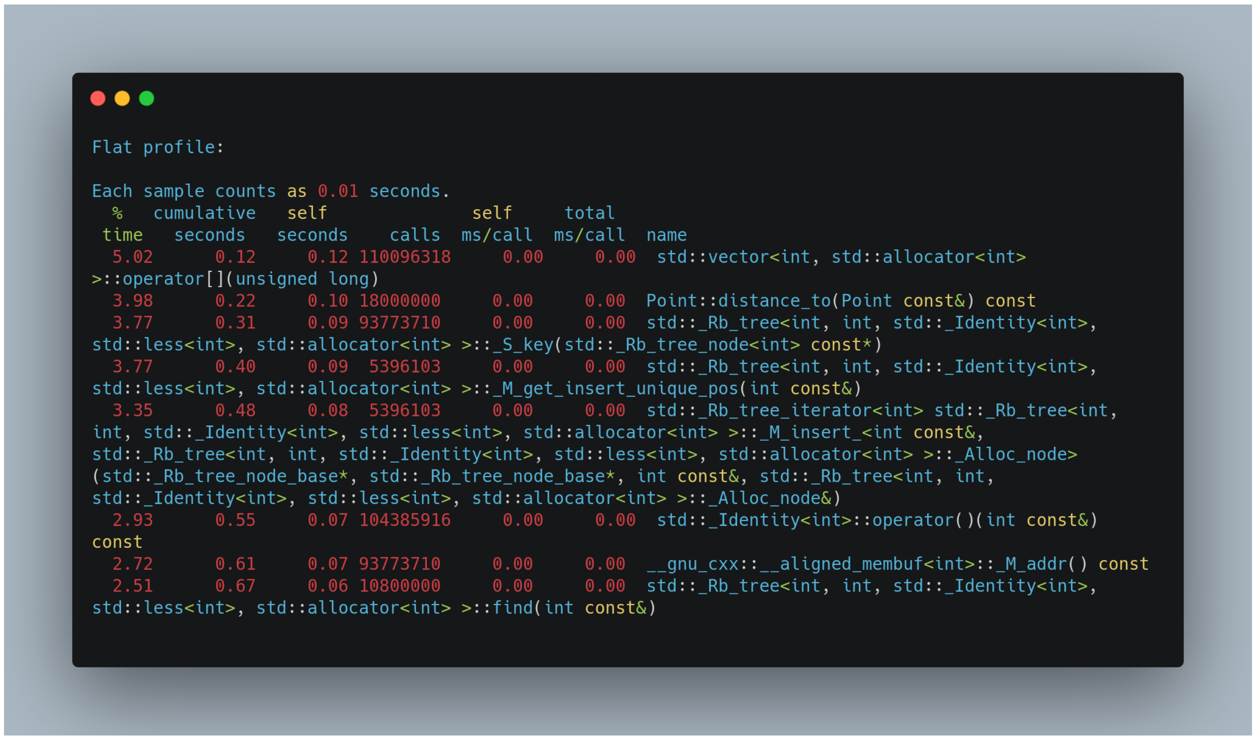
Serial Code:

The code is too long and has been omitted from the report. It can be accessed through the following github repository:

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

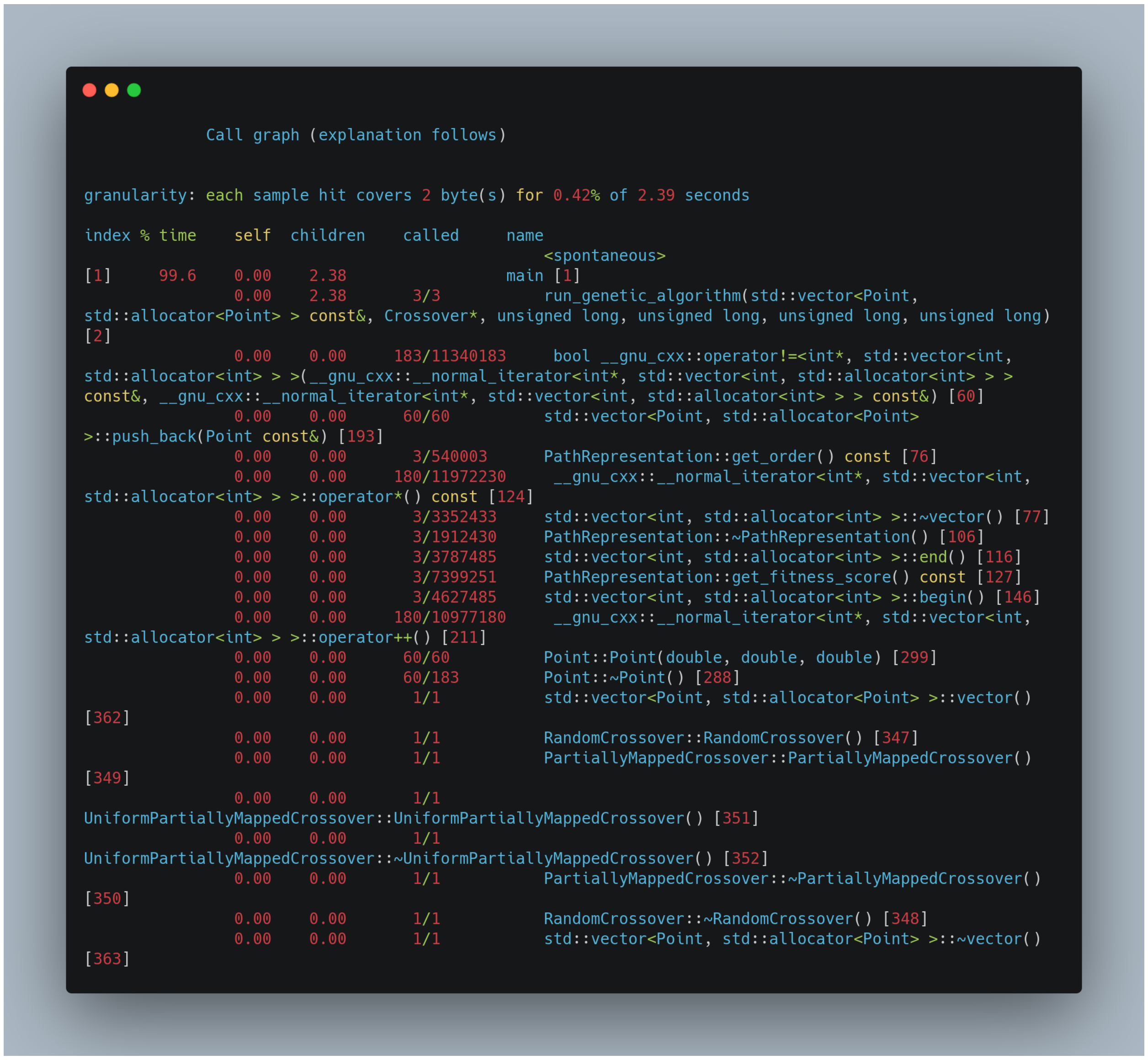
Functional Profiling Results

Showing only the top results. The file can also be accessed through the github repository.



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total           
 time   seconds   seconds    calls  ms/call  ms/call  name    
  5.02      0.12     0.12 110096318     0.00     0.00  std::vector<int, std::allocator<int>
>::operator[](unsigned long)
  3.98      0.22     0.10 18000000     0.00     0.00  Point::distance_to(Point const&) const
  3.77      0.31     0.09 93773710     0.00     0.00  std::_Rb_tree<int, int, std::_Identity<int>,
std::less<int>, std::allocator<int> >::_S_key(std::_Rb_tree_node<int> const*)
  3.77      0.40     0.09  5396103     0.00     0.00  std::_Rb_tree<int, int, std::_Identity<int>,
std::less<int>, std::allocator<int> >::_M_get_insert_unique_pos(int const&)
  3.35      0.48     0.08  5396103     0.00     0.00  std::_Rb_tree_iterator<int> std::_Rb_tree<int,
int, std::_Identity<int>, std::less<int>, std::allocator<int> >::_M_insert_<int const&,
std::_Rb_tree<int, int, std::_Identity<int>, std::less<int>, std::allocator<int> >::_Alloc_node>
(std::_Rb_tree_node_base*, std::_Rb_tree_node_base*, int const&, std::_Rb_tree<int, int,
std::_Identity<int>, std::less<int>, std::allocator<int> >::_Alloc_node&)
  2.93      0.55     0.07 104385916     0.00     0.00  std::_Identity<int>::operator()(int const&)
const
  2.72      0.61     0.07 93773710     0.00     0.00  __gnu_cxx::__aligned_membuf<int>::_M_addr() const
  2.51      0.67     0.06 10800000     0.00     0.00  std::_Rb_tree<int, int, std::_Identity<int>,
std::less<int>, std::allocator<int> >::find(int const&)
```

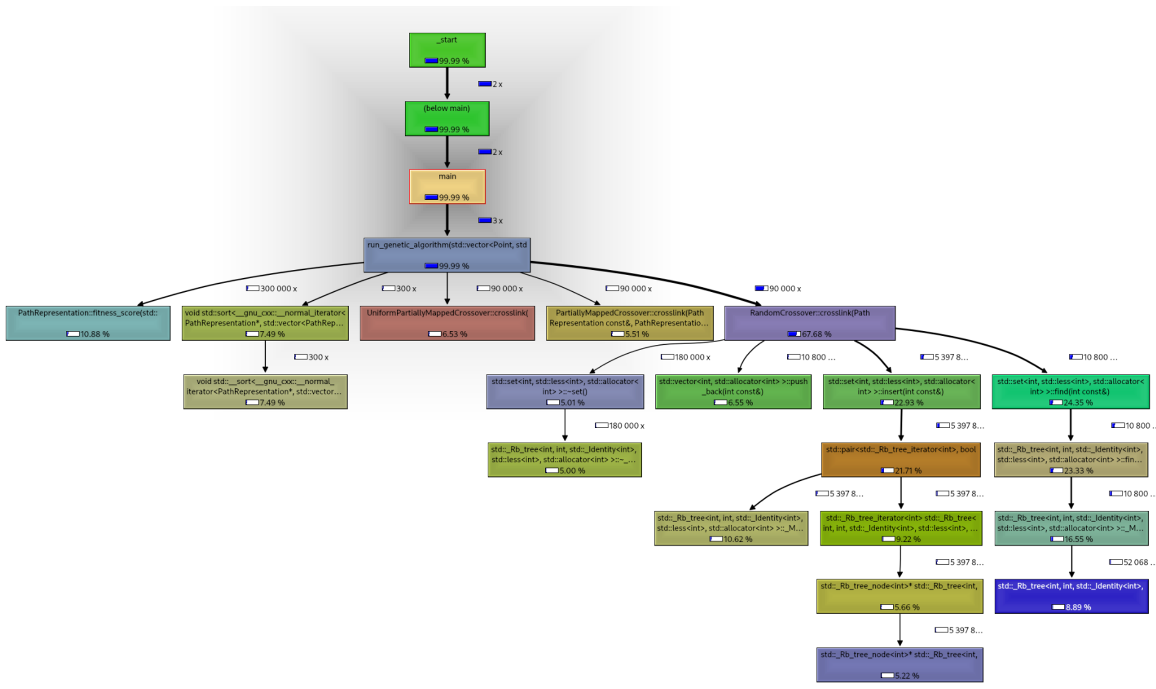
Call Graph:



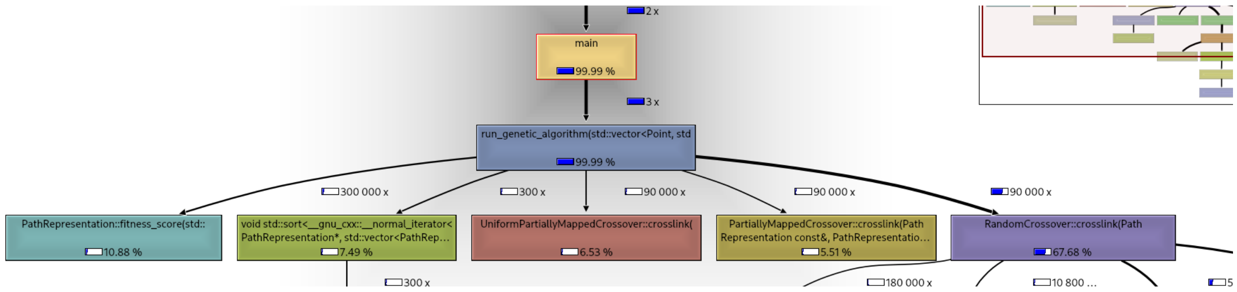
```
             Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 0.42% of 2.39 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]     99.6    0.00    2.38                 main [1]
                0.00    2.38       3/3           run_genetic_algorithm(std::vector<Point,
std::allocator<Point> > const&, Crossover*, unsigned long, unsigned long, unsigned long, unsigned long)
[2]
                0.00    0.00     183/11340183     bool __gnu_cxx::operator!=<int*, std::vector<int,
std::allocator<int> > >(__gnu_cxx::__normal_iterator<int*, std::vector<int, std::allocator<int> > >
const&, __gnu_cxx::__normal_iterator<int*, std::vector<int, std::allocator<int> > > const&) [60]
                0.00    0.00      60/60          std::vector<Point, std::allocator<Point>
>::push_back(Point const&) [193]
                0.00    0.00       3/540003      PathRepresentation::get_order() const [76]
                0.00    0.00     180/11972230     __gnu_cxx::__normal_iterator<int*, std::vector<int,
std::allocator<int> > >::operator*() const [124]
                0.00    0.00       3/3352433     std::vector<int, std::allocator<int> >::~vector() [77]
                0.00    0.00       3/1912430     PathRepresentation::~PathRepresentation() [106]
                0.00    0.00       3/3787485     std::vector<int, std::allocator<int> >::end() [116]
                0.00    0.00       3/7399251     PathRepresentation::get_fitness_score() const [127]
                0.00    0.00       3/4627485     std::vector<int, std::allocator<int> >::begin() [146]
                0.00    0.00     180/10977180     __gnu_cxx::__normal_iterator<int*, std::vector<int,
std::allocator<int> > >::operator++() [211]
                0.00    0.00      60/60          Point::Point(double, double, double) [299]
                0.00    0.00      60/183         Point::~Point() [288]
                0.00    0.00       1/1           std::vector<Point, std::allocator<Point> >::vector()
[362]
                0.00    0.00       1/1           RandomCrossover::RandomCrossover() [347]
                0.00    0.00       1/1           PartiallyMappedCrossover::PartiallyMappedCrossover()
[349]
                0.00    0.00       1/1
UniformPartiallyMappedCrossover::UniformPartiallyMappedCrossover() [351]
                0.00    0.00       1/1
UniformPartiallyMappedCrossover::~UniformPartiallyMappedCrossover() [352]
                0.00    0.00       1/1           PartiallyMappedCrossover::~PartiallyMappedCrossover()
[350]
                0.00    0.00       1/1           RandomCrossover::~RandomCrossover() [348]
                0.00    0.00       1/1           std::vector<Point, std::allocator<Point> >::~vector()
[363]
```

Call Graph:

Note that the graph obtained using gprof2dot wasn't readable. I used valgrind and & k-cache-grind to obtain the below call graph.


_start
99.99 %
2 x
(below main)
99.99 %
2 x
main
99.99 %
3 x
run_genetic_algorithm(std::vector<Point, std
99.99 %
300 000 x
300 x
90 000 x
90 000 x
90 000 x
PathRepresentation::fitness_score(std::
10.88 %
void std::sort<__gnu_cxx::__normal_iterator< PathRepresentation*, std::vector<PathRep…
7.49 %
UniformPartiallyMappedCrossover::crosslink(
6.53 %
PartiallyMappedCrossover::crosslink(Path Representation const&, PathRepresentatio…
5.51 %
RandomCrossover::crosslink(Path
67.68 %
300 x
void std::__sort<__gnu_cxx::__normal_ iterator<PathRepresentation*, std::vector…
7.49 %
180 000 x
10 800 …
5 397 8…
10 800 …
std::set<int, std::less<int>, std::allocator< int> >::~set()
5.01 %
std::vector<int, std::allocator<int> >::push _back(int const&)
6.55 %
std::set<int, std::less<int>, std::allocator< int> >::insert(int const&)
22.93 %
std::set<int, std::less<int>, std::allocator< int> >::find(int const&)
24.35 %
180 000 x
5 397 8…
10 800 …
std::_Rb_tree<int, int, std::_Identity<int>, std::less<int>, std::allocator<int> >::~_…
5.00 %
std::pair<std::_Rb_tree_iterator<int>, bool
21.71 %
std::_Rb_tree<int, int, std::_Identity<int>, std::less<int>, std::allocator<int> >::fin…
23.33 %
5 397 8…
5 397 8…
10 800 …
std::_Rb_tree<int, int, std::_Identity<int>, std::less<int>, std::allocator<int> >::_M…
10.62 %
std::_Rb_tree_iterator<int> std::_Rb_tree< int, int, std::_Identity<int>, std::less<int>, …
9.22 %
std::_Rb_tree<int, int, std::_Identity<int>, std::less<int>, std::allocator<int> >::_M…
16.55 %
5 397 8…
52 068 …
std::_Rb_tree_node<int>* std::_Rb_tree<int,
5.66 %
std::_Rb_tree<int, int, std::_Identity<int>,
8.89 %
5 397 8…
std::_Rb_tree_node<int>* std::_Rb_tree<int,
5.22 %

Using kcachegrind, let’s zoom into the call-graph & find where maximum percentage of the time is taken:


2 x
main
99.99 %
3 x
run_genetic_algorithm(std::vector<Point, std
99.99 %
300 000 x
300 x
90 000 x
90 000 x
90 000 x
PathRepresentation::fitness_score(std::
10.88 %
void std::sort<__gnu_cxx::__normal_iterator< PathRepresentation*, std::vector<PathRep...
7.49 %
UniformPartiallyMappedCrossover::crosslink(
6.53 %
PartiallyMappedCrossover::crosslink(Path Representation const&, PathRepresentatio...
5.51 %
RandomCrossover::crosslink(Path
67.68 %
300 x
180 000 x
10 800 ...

Let’s analyze the call graph:

1. Fitness_Score: This function is called for every new generation & computes the fitness_value (path length) of every genome in the population.

   Optimizations possible:

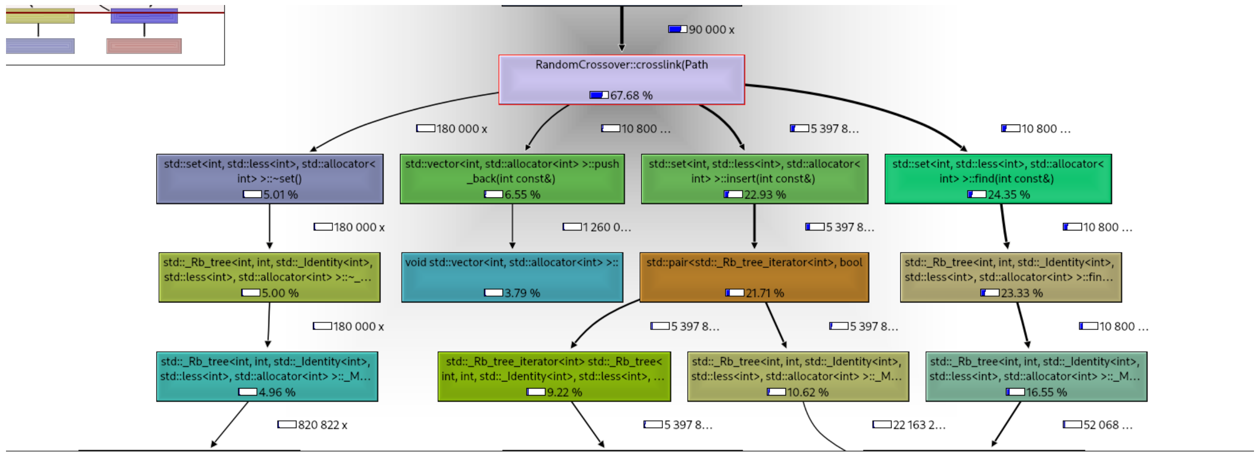
   1. The function uses distances_to method calculates the distance between the points. We can store the distances which have been calculated so that we won’t have to calculate them again.

2. Sort: This standard template library function is used to sort the fitness scores. It is called after every new generation.

Optimizations possible:

1. Instead of calling this function after every generation. It can be called only once during the first iteration & for every new iteration we can use insertion_sort to insert the crosslinked genome into the population. This will reduce the complexity from O(nlogn) to O(kn).

3. Crossovers: Among the crossovers, it can be seen that random_crossover takes the highest amount of time. Let's further visualize the call graph of random_crossover



90 000 x
RandomCrossover::crosslink(Path
67.68 %
180 000 x
10 800 …
5 397 8…
10 800 …
std::set<int, std::less<int>, std::allocator< int> >::~set()
5.01 %
std::vector<int, std::allocator<int> >::push_back(int const&)
6.55 %
std::set<int, std::less<int>, std::allocator< int> >::insert(int const&)
22.93 %
std::set<int, std::less<int>, std::allocator< int> >::find(int const&)
24.35 %
180 000 x
1 260 0…
5 397 8…
10 800 …
std::_Rb_tree<int, int, std::_Identity<int>, std::less<int>, std::allocator<int> >::~_…
5.00 %
void std::vector<int, std::allocator<int> >::
3.79 %
std::pair<std::_Rb_tree_iterator<int>, bool
21.71 %
std::_Rb_tree<int, int, std::_Identity<int>, std::less<int>, std::allocator<int> >::fin…
23.33 %
180 000 x
5 397 8…
5 397 8…
10 800 …
std::_Rb_tree<int, int, std::_Identity<int>, std::less<int>, std::allocator<int> >::_M…
4.96 %
std::_Rb_tree_iterator<int> std::_Rb_tree< int, int, std::_Identity<int>, std::less<int>, …
9.22 %
std::_Rb_tree<int, int, std::_Identity<int>, std::less<int>, std::allocator<int> >::_M…
10.62 %
std::_Rb_tree<int, int, std::_Identity<int>, std::less<int>, std::allocator<int> >::_M…
16.55 %
820 822 x
5 397 8…
22 163 2…
52 068 …

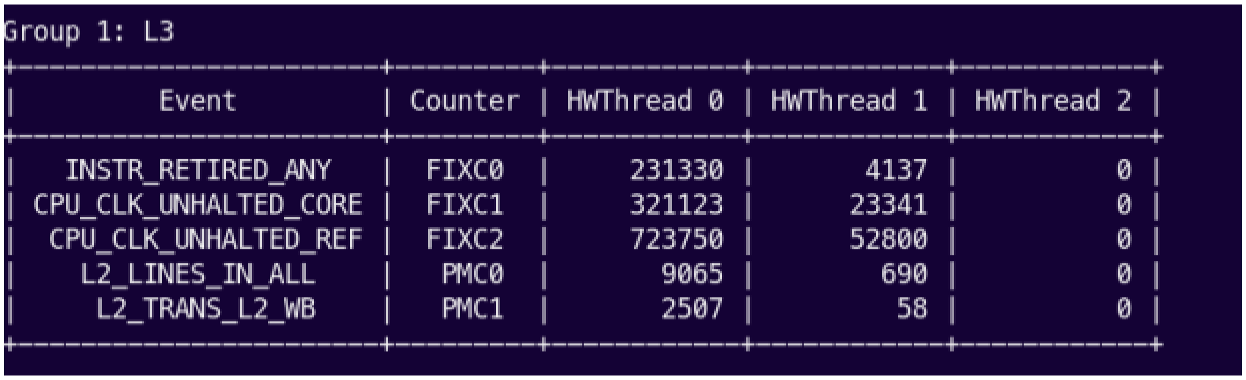
The highest amount of time is taken by vector.find() function which can be optimized by using a map.

Line - Based Profiling Results:

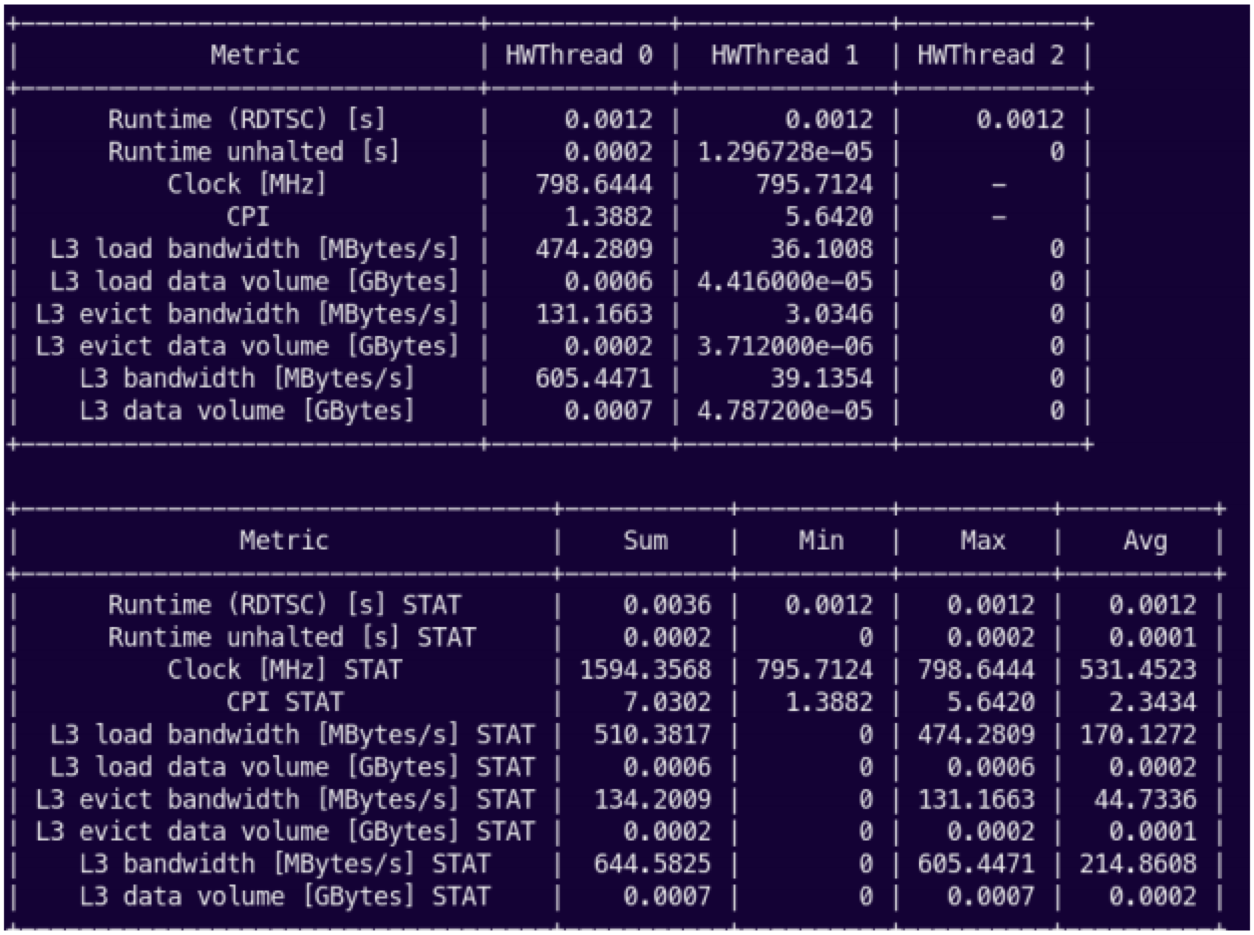
The output of line based profiling is shared in the link below:

https://github.com/hello-fri-end/Parallel-Implementation-of-Genetic-Algorithms-for-TSP/tree/master/Profiing%20Data

Process Resource Utilization Report:



```
Group 1: L3
+----------------------+---------+------------+------------+------------+
|        Event         | Counter | HWThread 0 | HWThread 1 | HWThread 2 |
+----------------------+---------+------------+------------+------------+
|   INSTR_RETIRED_ANY  |  FIXC0  |     231330 |       4137 |          0 |
| CPU_CLK_UNHALTED_CORE |  FIXC1  |     321123 |      23341 |          0 |
|  CPU_CLK_UNHALTED_REF |  FIXC2  |     723750 |      52800 |          0 |
|    L2_LINES_IN_ALL   |   PMC0  |       9065 |        690 |          0 |
|     L2_TRANS_L2_WB   |   PMC1  |       2507 |         58 |          0 |
+----------------------+---------+------------+------------+------------+
```



```
+-------------------------------+------------+--------------+------------+
|            Metric             | HWThread 0 |  HWThread 1  | HWThread 2 |
+-------------------------------+------------+--------------+------------+
|      Runtime (RDTSC) [s]      |     0.0012 |       0.0012 |     0.0012 |
|      Runtime unhalted [s]     |     0.0002 | 1.296728e-05 |          0 |
|          Clock [MHz]          |   798.6444 |     795.7124 |      -     |
|              CPI              |     1.3882 |       5.6420 |      -     |
|  L3 load bandwidth [MBytes/s] |   474.2809 |      36.1008 |          0 |
|  L3 load data volume [GBytes] |     0.0006 | 4.416000e-05 |          0 |
| L3 evict bandwidth [MBytes/s] |   131.1663 |       3.0346 |          0 |
| L3 evict data volume [GBytes] |     0.0002 | 3.712000e-06 |          0 |
|    L3 bandwidth [MBytes/s]    |   605.4471 |      39.1354 |          0 |
|    L3 data volume [GBytes]    |     0.0007 | 4.787200e-05 |          0 |
+-------------------------------+------------+--------------+------------+

+------------------------------------+-----------+----------+----------+----------+
|               Metric               |    Sum    |    Min   |    Max   |    Avg   |
+------------------------------------+-----------+----------+----------+----------+
|      Runtime (RDTSC) [s] STAT      |    0.0036 |   0.0012 |   0.0012 |   0.0012 |
|      Runtime unhalted [s] STAT     |    0.0002 |        0 |   0.0002 |   0.0001 |
|          Clock [MHz] STAT          | 1594.3568 | 795.7124 | 798.6444 | 531.4523 |
|              CPI STAT              |    7.0302 |   1.3882 |   5.6420 |   2.3434 |
|  L3 load bandwidth [MBytes/s] STAT |  510.3817 |        0 | 474.2809 | 170.1272 |
|  L3 load data volume [GBytes] STAT |    0.0006 |        0 |   0.0006 |   0.0002 |
| L3 evict bandwidth [MBytes/s] STAT |  134.2009 |        0 | 131.1663 |  44.7336 |
| L3 evict data volume [GBytes] STAT |    0.0002 |        0 |   0.0002 |   0.0001 |
|    L3 bandwidth [MBytes/s] STAT    |  644.5825 |        0 | 605.4471 | 214.8608 |
|    L3 data volume [GBytes] STAT    |    0.0007 |        0 |   0.0007 |   0.0002 |
```

Conclusion/Observations

The most useful observations are obtained from the call graph & have already been discussed in it’s analysis above. Once we’re done optimizing the code, we’ll parallelize it using openMP. We should expect significant improvement in the performance of the code since there isn't much dependency in the core algorithm.