COMS 4995 Parallel Functional Programming FInal Report

TSP - Traveling Salesman Problem

Submitted by - jcs2281, sb4443

Introduction

The Travelling Salesman Problem (TSP) is one of the best known NP-hard problems, that means that no exact algorithm can solve it in polynomial time. The method that would definitely obtain the optimal solution of TSP is the method of exhaustive enumeration and evaluation. This procedure begins by generating the possibility of all the tours and evaluating according length or cost of the tour. The tour with the smallest length or cost chosen as the best, and guaranteed to be optimal. TSP is prevalent in real-world scenarios and researchers and companies are working on resolving cases of TSP and finding an optimal solution. One example is the delivery service, a courier needs to deliver goods to customers with different destinations and time is of considerable concern in delivery of goods as it relates to the reputation of the company. To reach the target requires a system capable of providing an optimal travel route so that the travel time can be minimized. In this project, we are trying to use the parallelization supported by haskell to see if parallelization can speed up the algorithm or improve it over its sequential implementation. We are using brute-force technique where we are trying out all possible orders lexicographically and Genetic algorithm approximation and analysing the performance difference between parallel and sequential implementation.

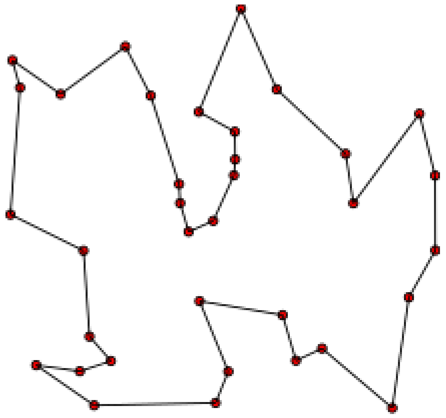
Problem formulation

The Travelling Salesman Problem (TSP) is the challenge that can be defined as follows: consider a number of cities which must be visited by a traveling salesman, only once, arriving once and departing once and starting and ending at the same city. Given the pairwise distances between cities, what is the best order in which to visit them, so as to minimize the overall distance traveled?

It is a well-known algorithmic problem in the fields of computer science. There are obviously a lot of different routes to choose, but finding the best one; the one that will require the least distance or cost is what researchers have spent decades trying to solve for.

It has commanded so much attention because it’s so easy to describe it yet difficult to solve. The complexity of calculating the best route will keep on increasing when we add more destinations to the problem. That’s why TSP belongs to the class of combinatorial optimization problems known as NP-complete. This implies that it is classified as NP-hard as it has no “quick” solution.

Example solution of a travelling salesman problem - the black line shows the shortest possible loop that connects every red dot:



Applications

1. **Overhauling gas turbine engines:** To guarantee a uniform gas flow through the turbines there are nozzle-guide vane assemblies located at each turbine stage. Such an assembly basically consists of a number of nozzle guide vanes affixed about its circumference. All these vanes have individual characteristics and the correct placement of the vanes can result in substantial benefits. The problem of placing the vanes in the best possible way can be modeled as a TSP with a special objective function.
2. **Order Picking problem:** This problem is associated with material handling in a warehouse. Assume that at a warehouse an order arrives for a certain subset of the items stored in the warehouse. Some vehicles have to collect all items of this order to ship them to the customer. The relation to the TSP is immediately seen. The storage locations of the items correspond to the nodes of the graph. The distance between two nodes is given by the time needed to move the vehicle from one location to the other. The problem of finding a shortest route for the vehicle with minimum pickup time can now be solved as a TSP.
3. **Vehicle Routing:** Suppose that in a city n mail boxes have to be emptied every day within a certain period of time, say 1 hour. The problem is to find the minimum number of trucks to do this and the shortest time to do the collections using this number of trucks.

Implementations

- Brute Force sequential
  In the brute force sequential approach we first enumerate all the possible permutations of the paths and calculate the distance of each possible path one after the other by traversing across the collected path one by one and picking the shortest one. This is an exhaustive search as we are searching over a large space. That's this algorithm has exponential time complexity.

- Bruteforce, calculate path distance in parallel
  In the brute force approach, we first enumerate all the possible permutations of the paths as we did in sequential and then create sparks for each one of them to get calculated in

parallel and pick the shortest one. This is still an exhaustive search as we are searching over a large space but with a small reduction in search space as we are involving more than one core to perform this parallelization.

- Bruteforce, calculate path distance in parallel with Chunk size
  In the parallel brute force approach, rather than enumerate all the possible path permutations and then create sparks for each one of them we first divide them into a fixed chunk and then run these chunks in parallel and pick the shortest one.

- Bruteforce for a batch of city groups
  In this approach, we read an input file, replicate it a user-specified number of times, and then randomize them. Once we have got b batches we run the sequential algorithm over these b batches in a similar fashion as the naive brute force approach.

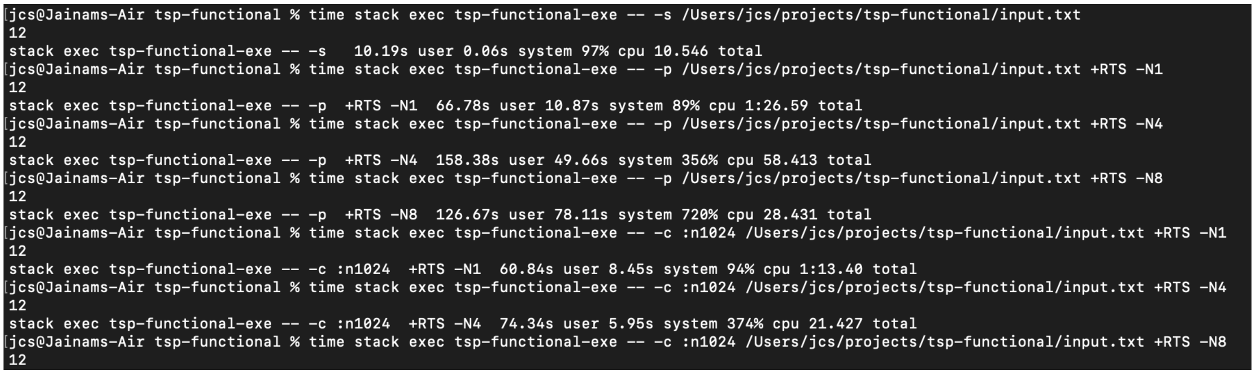
- Bruteforce for a batch of city groups, each group in parallel
  In this approach also, we first generate an infinite random number List between a list of empty length and the maximum number of cities as we did for the sequential batch algorithm. Once the random list is generated we pick the first b number from this random list and these b numbers for the batches from the city corpus. Now rather than running this algorithm for b batches in a sequential fashion we use Haskell parallelization to run them in parallel and choose the minimum cost path.

- Genetic Algorithm with Population Size and Number of Generations
  In the algorithm, we treat cities as genes, a single path that gets generated using these characters or problem constraints known as chromosomes, and a fitness score which is inversely proportional to the squared path length. The smaller the path length gene is, the fitter it is. The fittest of all the genes in the gene pool survive the population test and move to the next iteration. The number of iterations depends upon the value of a cooling variable. The cooling variable value keeps decreasing with each iteration and it reaches a threshold after a fixed number of iterations.

- Genetic Algorithm for a batch of city groups
  Here we replicate and randomize an input file to generate a batch of problems just like done before for the sequential batch processing. Once we have got b batches, we run the genetic algorithm defined earlier over these b batches in a sequential fashion and find the minimum cost path.

- Genetic Algorithm for a batch of city groups, each group in parallel
  This algorithm performs the same initial step as its sequential implementation defined earlier but it executes the genetic algorithm in batches parallelly similar to the brute force approach.

Performance Analysis

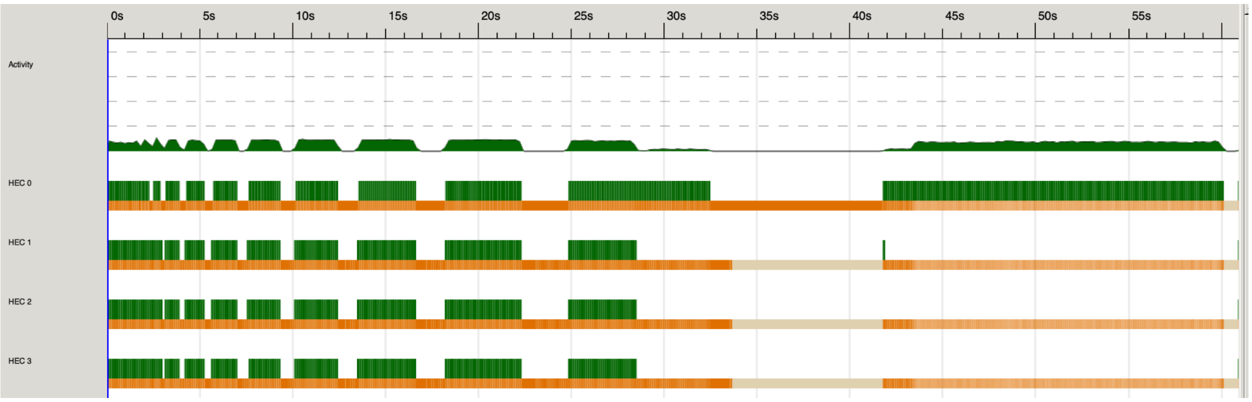


```
[jcs@Jainams-Air tsp-functional % time stack exec tsp-functional-exe -- -s /Users/jcs/projects/tsp-functional/input.txt
12
stack exec tsp-functional-exe -- -s   10.19s user 0.06s system 97% cpu 10.546 total
[jcs@Jainams-Air tsp-functional % time stack exec tsp-functional-exe -- -p /Users/jcs/projects/tsp-functional/input.txt +RTS -N1
12
stack exec tsp-functional-exe -- -p  +RTS -N1  66.78s user 10.87s system 89% cpu 1:26.59 total
[jcs@Jainams-Air tsp-functional % time stack exec tsp-functional-exe -- -p /Users/jcs/projects/tsp-functional/input.txt +RTS -N4
12
stack exec tsp-functional-exe -- -p  +RTS -N4  158.38s user 49.66s system 356% cpu 58.413 total
[jcs@Jainams-Air tsp-functional % time stack exec tsp-functional-exe -- -p /Users/jcs/projects/tsp-functional/input.txt +RTS -N8
12
stack exec tsp-functional-exe -- -p  +RTS -N8  126.67s user 78.11s system 720% cpu 28.431 total
[jcs@Jainams-Air tsp-functional % time stack exec tsp-functional-exe -- -c :n1024 /Users/jcs/projects/tsp-functional/input.txt +RTS -N1
12
stack exec tsp-functional-exe -- -c :n1024  +RTS -N1  60.84s user 8.45s system 94% cpu 1:13.40 total
[jcs@Jainams-Air tsp-functional % time stack exec tsp-functional-exe -- -c :n1024 /Users/jcs/projects/tsp-functional/input.txt +RTS -N4
12
stack exec tsp-functional-exe -- -c :n1024  +RTS -N4  74.34s user 5.95s system 374% cpu 21.427 total
[jcs@Jainams-Air tsp-functional % time stack exec tsp-functional-exe -- -c :n1024 /Users/jcs/projects/tsp-functional/input.txt +RTS -N8
12
```

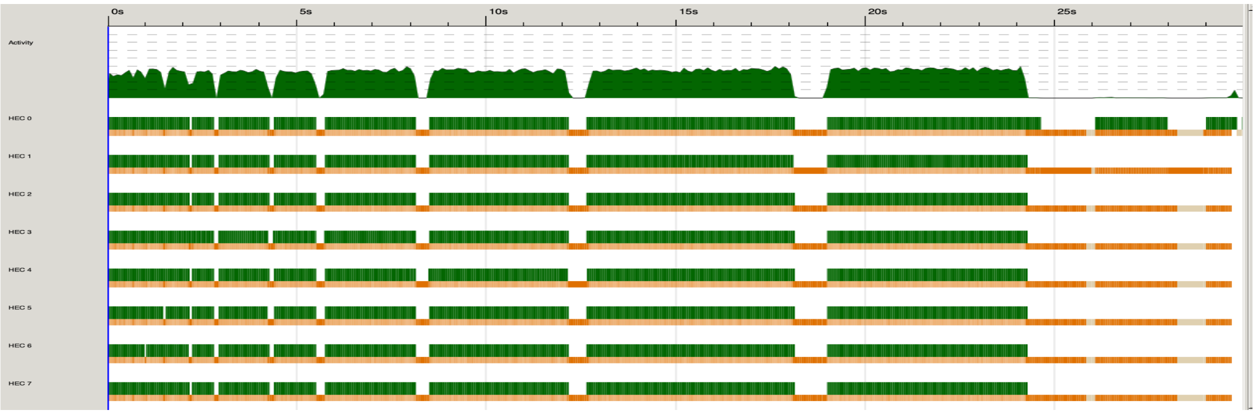
Approach	Number of Core	Time Taken(secs)
Sequential	1	10.546
Parallel	1	86.59
Parallel	4	58.413
Parallel	8	28.431

The sequential algorithm seems to be doing better than the parallel implementation. Therefore we look at threadscope and the number of sparks to see what the issue is:

4 core



8 Core



```
27,287,373,552 bytes allocated in the heap
 8,660,023,944 bytes copied during GC
 1,218,827,432 bytes maximum residency (27 sample(s))
 1,622,514,520 bytes maximum slop
        4915 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
Gen  0     14488 colls, 14488 par   29.908s  13.102s     0.0009s    0.0034s
Gen  1        27 colls,    26 par   12.141s   4.546s     0.1684s    1.1028s

Parallel GC work balance: 1.21% (serial 0%, perfect 100%)

TASKS: 18 (1 bound, 17 peak workers (17 total), using -N8)

SPARKS: 39916800 (39872098 converted, 44702 overflowed, 0 dud, 0 GC'd, 0 fizzled)

INIT    time    0.000s  (  0.003s elapsed)
MUT     time   85.721s  ( 11.171s elapsed)
GC      time   42.049s  ( 17.649s elapsed)
EXIT    time    0.000s  (  0.006s elapsed)
Total   time  127.771s  ( 28.829s elapsed)

Alloc rate    318,327,269 bytes per MUT second

Productivity  67.1% of total user, 38.7% of total elapsed
```

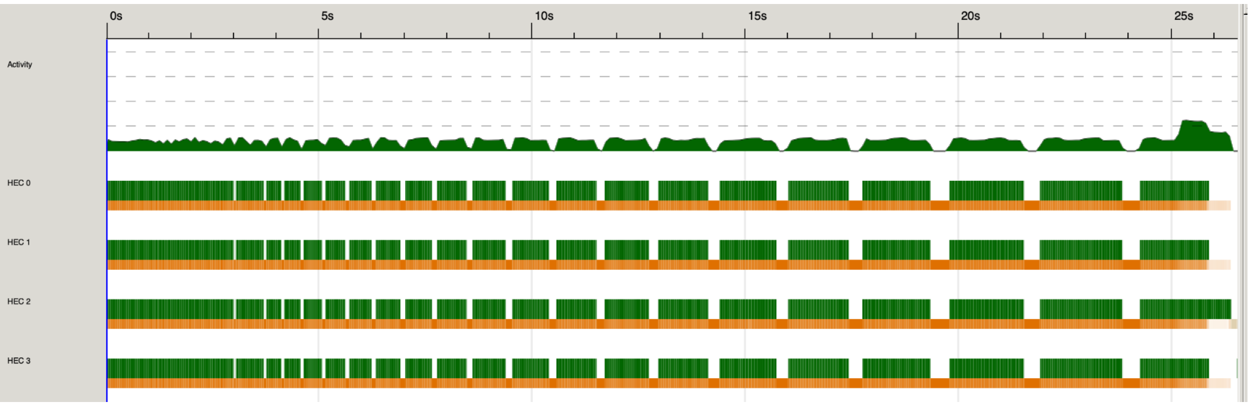
The run is spending a lot of time doing garbage collection which is overpowering any gain got by the parallelization. To overcome this, we divide the parallelization into chunks so as to not overwhelm the processors with too many sparks at once. This seems to improve performance over the normal parallel implementation.

Dividing path calculation in chunks:

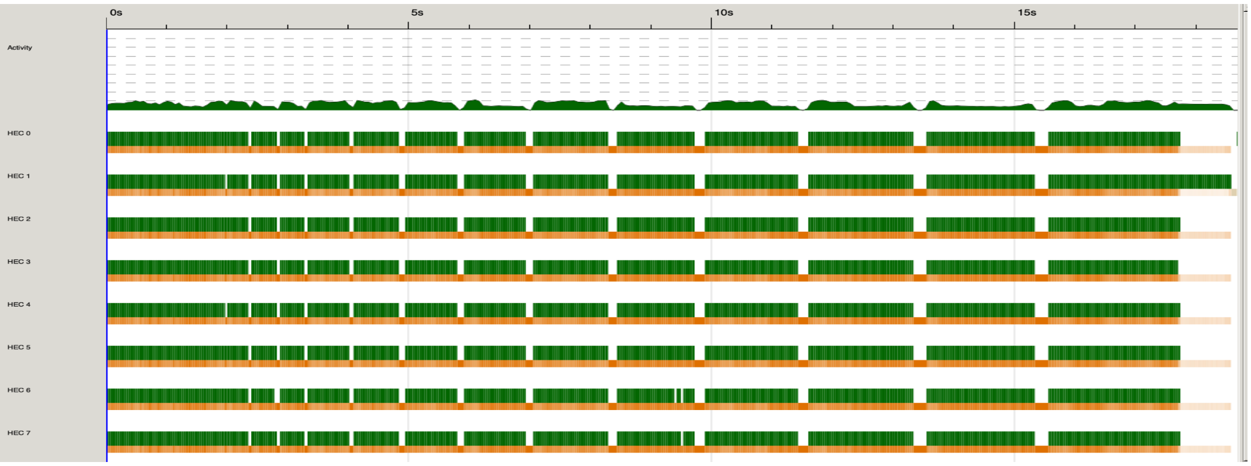
Method	Number of Core	Chunk	Time Taken(secs)

Parallel	1	1024	73.40
Parallel	4	1024	21.427
Parallel	8	1024	15.323

4 Core

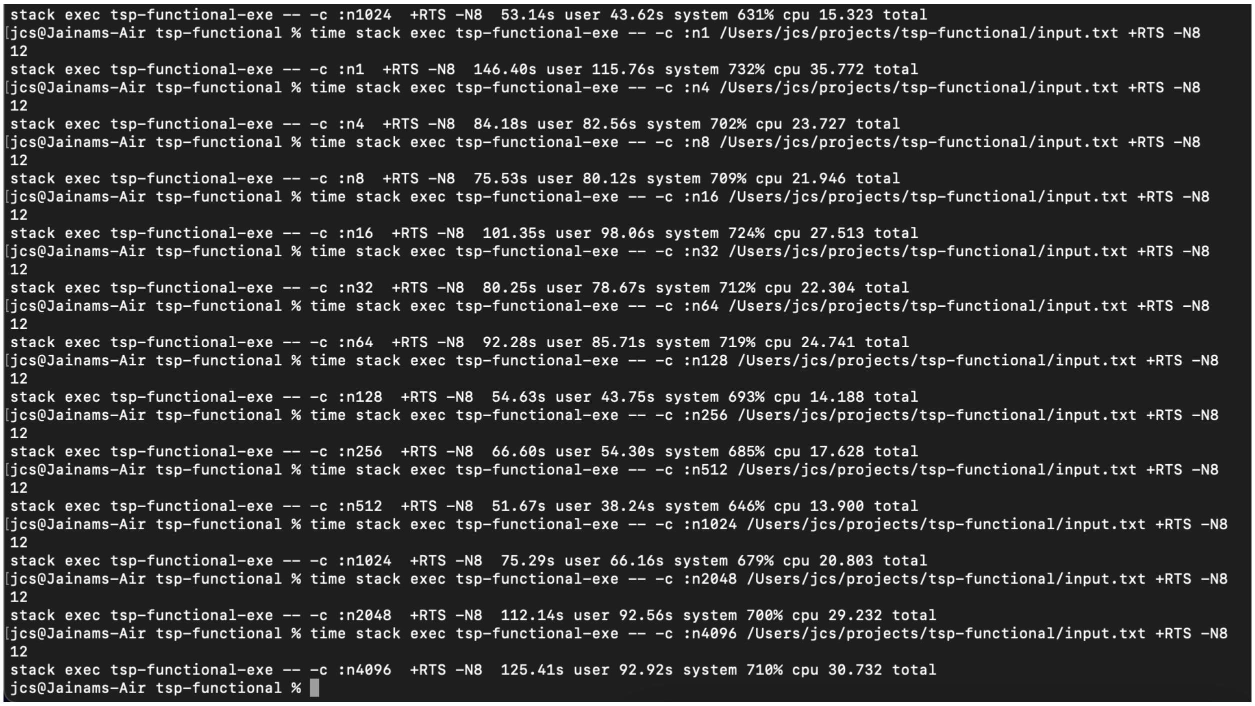


8 Core and 1024 chunk



```
34,964,115,760 bytes allocated in the heap
14,978,086,504 bytes copied during GC
   913,065,696 bytes maximum residency (37 sample(s))
    10,393,888 bytes maximum slop
          2416 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
Gen  0     21932 colls, 21932 par   39.448s  11.825s     0.0005s    0.0090s
Gen  1        37 colls,    36 par   11.952s   1.834s     0.0496s    0.2569s

Parallel GC work balance: 62.58% (serial 0%, perfect 100%)

TASKS: 18 (1 bound, 17 peak workers (17 total), using -N8)

SPARKS: 38982 (38982 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)
```

```
INIT    time    0.001s  (  0.003s elapsed)
MUT     time   15.361s  (  4.596s elapsed)
GC      time   51.400s  ( 13.658s elapsed)
EXIT    time    0.000s  (  0.009s elapsed)
Total   time   66.761s  ( 18.266s elapsed)

Alloc rate    2,276,218,926 bytes per MUT second

Productivity  23.0% of total user, 25.2% of total elapsed
```



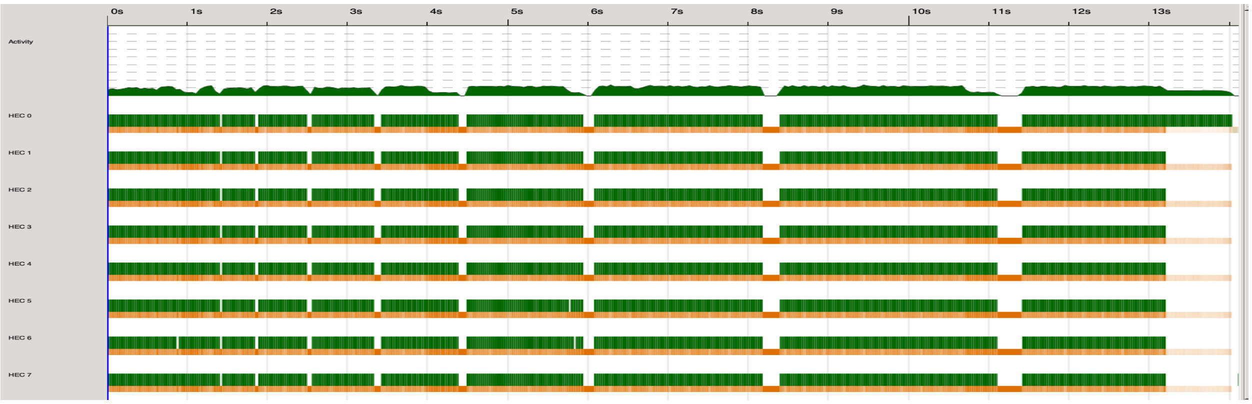
```
stack exec tsp-functional-exe -- -c :n1024  +RTS -N8  53.14s user 43.62s system 631% cpu 15.323 total
[jcs@Jainams-Air tsp-functional % time stack exec tsp-functional-exe -- -c :n1 /Users/jcs/projects/tsp-functional/input.txt +RTS -N8
12
stack exec tsp-functional-exe -- -c :n1  +RTS -N8  146.40s user 115.76s system 732% cpu 35.772 total
[jcs@Jainams-Air tsp-functional % time stack exec tsp-functional-exe -- -c :n4 /Users/jcs/projects/tsp-functional/input.txt +RTS -N8
12
stack exec tsp-functional-exe -- -c :n4  +RTS -N8  84.18s user 82.56s system 702% cpu 23.727 total
[jcs@Jainams-Air tsp-functional % time stack exec tsp-functional-exe -- -c :n8 /Users/jcs/projects/tsp-functional/input.txt +RTS -N8
12
stack exec tsp-functional-exe -- -c :n8  +RTS -N8  75.53s user 80.12s system 709% cpu 21.946 total
[jcs@Jainams-Air tsp-functional % time stack exec tsp-functional-exe -- -c :n16 /Users/jcs/projects/tsp-functional/input.txt +RTS -N8
12
stack exec tsp-functional-exe -- -c :n16  +RTS -N8  101.35s user 98.06s system 724% cpu 27.513 total
[jcs@Jainams-Air tsp-functional % time stack exec tsp-functional-exe -- -c :n32 /Users/jcs/projects/tsp-functional/input.txt +RTS -N8
12
stack exec tsp-functional-exe -- -c :n32  +RTS -N8  80.25s user 78.67s system 712% cpu 22.304 total
[jcs@Jainams-Air tsp-functional % time stack exec tsp-functional-exe -- -c :n64 /Users/jcs/projects/tsp-functional/input.txt +RTS -N8
12
stack exec tsp-functional-exe -- -c :n64  +RTS -N8  92.28s user 85.71s system 719% cpu 24.741 total
[jcs@Jainams-Air tsp-functional % time stack exec tsp-functional-exe -- -c :n128 /Users/jcs/projects/tsp-functional/input.txt +RTS -N8
12
stack exec tsp-functional-exe -- -c :n128  +RTS -N8  54.63s user 43.75s system 693% cpu 14.188 total
[jcs@Jainams-Air tsp-functional % time stack exec tsp-functional-exe -- -c :n256 /Users/jcs/projects/tsp-functional/input.txt +RTS -N8
12
stack exec tsp-functional-exe -- -c :n256  +RTS -N8  66.60s user 54.30s system 685% cpu 17.628 total
[jcs@Jainams-Air tsp-functional % time stack exec tsp-functional-exe -- -c :n512 /Users/jcs/projects/tsp-functional/input.txt +RTS -N8
12
stack exec tsp-functional-exe -- -c :n512  +RTS -N8  51.67s user 38.24s system 646% cpu 13.900 total
[jcs@Jainams-Air tsp-functional % time stack exec tsp-functional-exe -- -c :n1024 /Users/jcs/projects/tsp-functional/input.txt +RTS -N8
12
stack exec tsp-functional-exe -- -c :n1024  +RTS -N8  75.29s user 66.16s system 679% cpu 20.803 total
[jcs@Jainams-Air tsp-functional % time stack exec tsp-functional-exe -- -c :n2048 /Users/jcs/projects/tsp-functional/input.txt +RTS -N8
12
stack exec tsp-functional-exe -- -c :n2048  +RTS -N8  112.14s user 92.56s system 700% cpu 29.232 total
[jcs@Jainams-Air tsp-functional % time stack exec tsp-functional-exe -- -c :n4096 /Users/jcs/projects/tsp-functional/input.txt +RTS -N8
12
stack exec tsp-functional-exe -- -c :n4096  +RTS -N8  125.41s user 92.92s system 710% cpu 30.732 total
jcs@Jainams-Air tsp-functional %
```

Method	**Number of Core**	**Chunk**	**Time Taken(secs)**
Parallel	8	1	35.772
Parallel	8	4	23.727
Parallel	8	8	21.946
Parallel	8	16	27.513
Parallel	8	32	22.304
Parallel	8	64	24.741
Parallel	8	128	14.188
Parallel	8	256	17.628

Parallel	8	512	13.900
Parallel	8	1024	15.323
Parallel	8	2048	29.232
Parallel	8	4096	30.732

The best result is achieved for a chunk size of 512 on 8 cores.

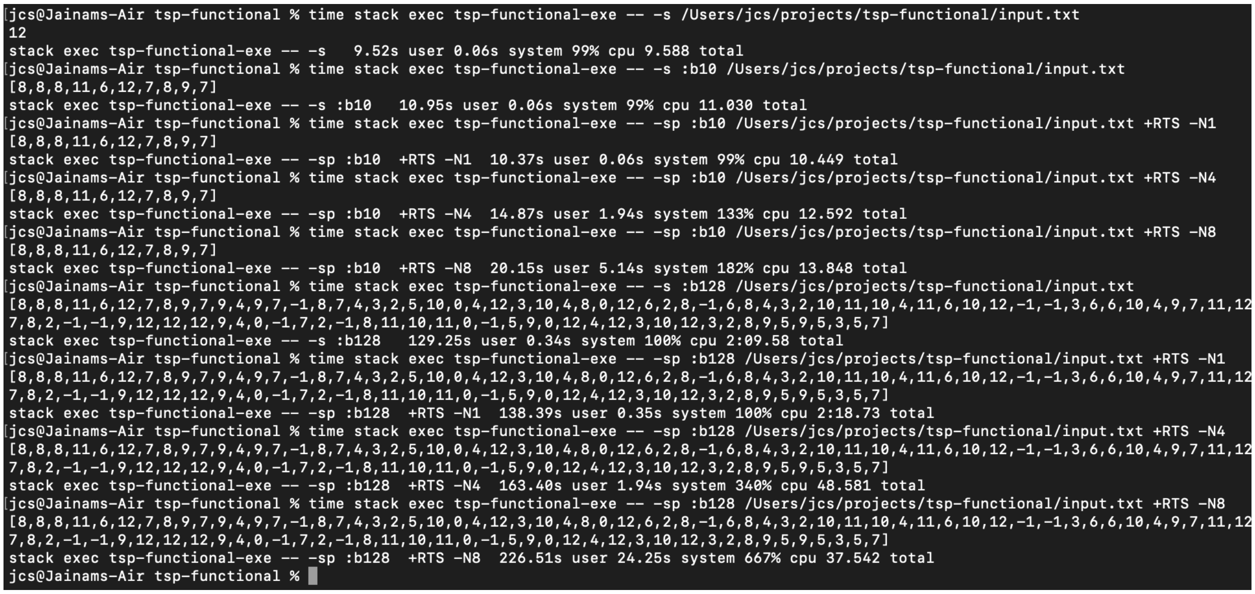
8 Core and 512 chunk



After this analysis, we realize that calculating the euclidean path distance for a set of coordinates is not a very heavy task by itself and therefore parallelizing the calculation of multiple paths at once does not benefit us much.

Therefore, next we try to calculate the minimum path distance for multiple city groups at once.

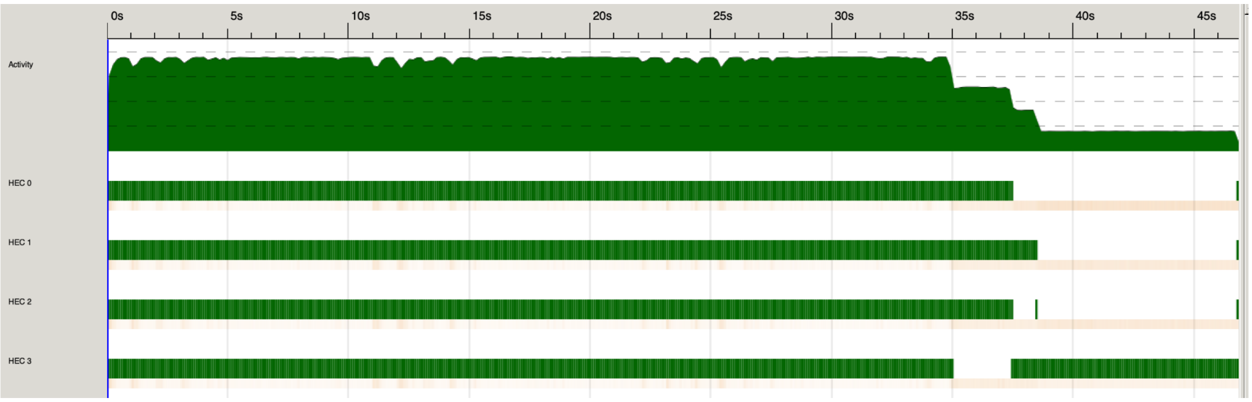
Analysis of City Groups in Batches



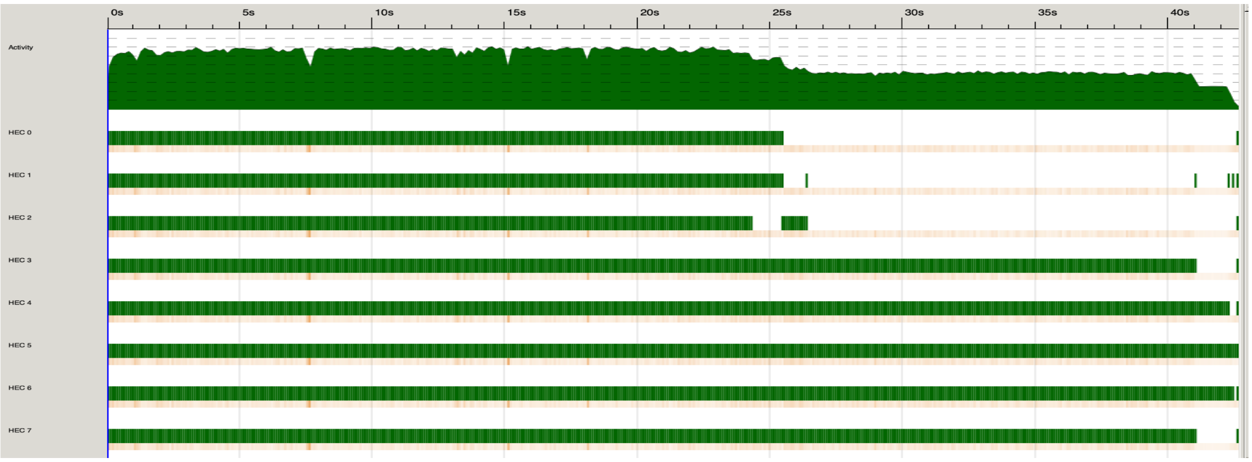
```
[jcs@Jainams-Air tsp-functional % time stack exec tsp-functional-exe -- -s /Users/jcs/projects/tsp-functional/input.txt
12
stack exec tsp-functional-exe -- -s   9.52s user 0.06s system 99% cpu 9.588 total
[jcs@Jainams-Air tsp-functional % time stack exec tsp-functional-exe -- -s :b10 /Users/jcs/projects/tsp-functional/input.txt
[8,8,8,11,6,12,7,8,9,7]
stack exec tsp-functional-exe -- -s :b10   10.95s user 0.06s system 99% cpu 11.030 total
[jcs@Jainams-Air tsp-functional % time stack exec tsp-functional-exe -- -sp :b10 /Users/jcs/projects/tsp-functional/input.txt +RTS -N1
[8,8,8,11,6,12,7,8,9,7]
stack exec tsp-functional-exe -- -sp :b10  +RTS -N1  10.37s user 0.06s system 99% cpu 10.449 total
[jcs@Jainams-Air tsp-functional % time stack exec tsp-functional-exe -- -sp :b10 /Users/jcs/projects/tsp-functional/input.txt +RTS -N4
[8,8,8,11,6,12,7,8,9,7]
stack exec tsp-functional-exe -- -sp :b10  +RTS -N4  14.87s user 1.94s system 133% cpu 12.592 total
[jcs@Jainams-Air tsp-functional % time stack exec tsp-functional-exe -- -sp :b10 /Users/jcs/projects/tsp-functional/input.txt +RTS -N8
[8,8,8,11,6,12,7,8,9,7]
stack exec tsp-functional-exe -- -sp :b10  +RTS -N8  20.15s user 5.14s system 182% cpu 13.848 total
[jcs@Jainams-Air tsp-functional % time stack exec tsp-functional-exe -- -s :b128 /Users/jcs/projects/tsp-functional/input.txt
[8,8,8,11,6,12,7,8,9,7,9,4,9,7,-1,8,7,4,3,2,5,10,0,4,12,3,10,4,8,0,12,6,2,8,-1,6,8,4,3,2,10,11,10,4,11,6,10,12,-1,-1,3,6,6,10,4,9,7,11,12
7,8,2,-1,-1,9,12,12,12,9,4,0,-1,7,2,-1,8,11,10,11,0,-1,5,9,0,12,4,12,3,10,12,3,2,8,9,5,9,5,3,5,7]
stack exec tsp-functional-exe -- -s :b128   129.25s user 0.34s system 100% cpu 2:09.58 total
[jcs@Jainams-Air tsp-functional % time stack exec tsp-functional-exe -- -sp :b128 /Users/jcs/projects/tsp-functional/input.txt +RTS -N1
[8,8,8,11,6,12,7,8,9,7,9,4,9,7,-1,8,7,4,3,2,5,10,0,4,12,3,10,4,8,0,12,6,2,8,-1,6,8,4,3,2,10,11,10,4,11,6,10,12,-1,-1,3,6,6,10,4,9,7,11,12
7,8,2,-1,-1,9,12,12,12,9,4,0,-1,7,2,-1,8,11,10,11,0,-1,5,9,0,12,4,12,3,10,12,3,2,8,9,5,9,5,3,5,7]
stack exec tsp-functional-exe -- -sp :b128  +RTS -N1  138.39s user 0.35s system 100% cpu 2:18.73 total
[jcs@Jainams-Air tsp-functional % time stack exec tsp-functional-exe -- -sp :b128 /Users/jcs/projects/tsp-functional/input.txt +RTS -N4
[8,8,8,11,6,12,7,8,9,7,9,4,9,7,-1,8,7,4,3,2,5,10,0,4,12,3,10,4,8,0,12,6,2,8,-1,6,8,4,3,2,10,11,10,4,11,6,10,12,-1,-1,3,6,6,10,4,9,7,11,12
7,8,2,-1,-1,9,12,12,12,9,4,0,-1,7,2,-1,8,11,10,11,0,-1,5,9,0,12,4,12,3,10,12,3,2,8,9,5,9,5,3,5,7]
stack exec tsp-functional-exe -- -sp :b128  +RTS -N4  163.40s user 1.94s system 340% cpu 48.581 total
[jcs@Jainams-Air tsp-functional % time stack exec tsp-functional-exe -- -sp :b128 /Users/jcs/projects/tsp-functional/input.txt +RTS -N8
[8,8,8,11,6,12,7,8,9,7,9,4,9,7,-1,8,7,4,3,2,5,10,0,4,12,3,10,4,8,0,12,6,2,8,-1,6,8,4,3,2,10,11,10,4,11,6,10,12,-1,-1,3,6,6,10,4,9,7,11,12
7,8,2,-1,-1,9,12,12,12,9,4,0,-1,7,2,-1,8,11,10,11,0,-1,5,9,0,12,4,12,3,10,12,3,2,8,9,5,9,5,3,5,7]
stack exec tsp-functional-exe -- -sp :b128  +RTS -N8  226.51s user 24.25s system 667% cpu 37.542 total
jcs@Jainams-Air tsp-functional % 
```

Method	Number of Core	Batch Size	Time Taken(secs)
Sequential	1	1	9.588
Sequential	1	10	11.030
Sequential	1	128	129.58
Parallel	1	10	10.449
Parallel	4	10	12.592
Parallel	8	10	13.848
Parallel	1	128	139.73
Parallel	4	128	48.581
Parallel	8	128	37.542

4 Core and 128 batches of city group

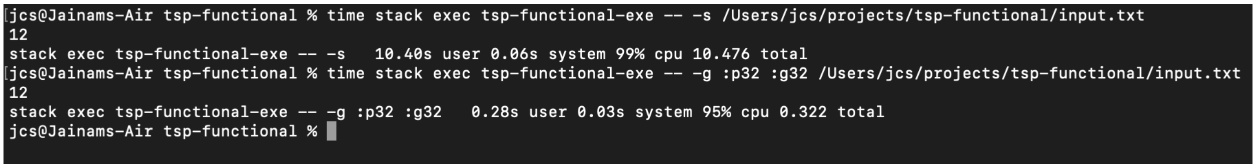


8 Core and 128 batch of city groups



Since calculating the entire tsp min distance path for a set of cities is a much more intensive task than calculating one euclidean path. Therefore we see that these batch computations greatly benefit by parallelization. For a batch of 128 city groups, we see a **speedup of 3.45x** when we move to batches of city.

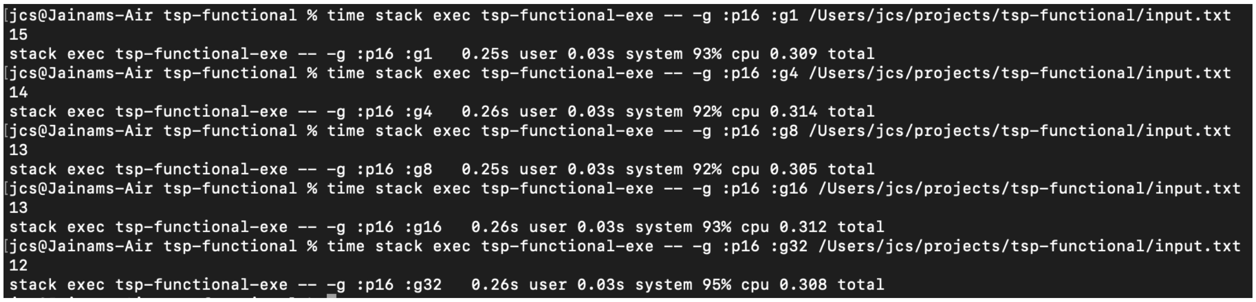
Genetic Algorithm and Parallelization Analysis:



```
[jcs@Jainams-Air tsp-functional % time stack exec tsp-functional-exe -- -s /Users/jcs/projects/tsp-functional/input.txt
12
stack exec tsp-functional-exe -- -s   10.40s user 0.06s system 99% cpu 10.476 total
[jcs@Jainams-Air tsp-functional % time stack exec tsp-functional-exe -- -g :p32 :g32 /Users/jcs/projects/tsp-functional/input.txt
12
stack exec tsp-functional-exe -- -g :p32 :g32   0.28s user 0.03s system 95% cpu 0.322 total
jcs@Jainams-Air tsp-functional % █
```

Method	Population Size	Generations	Input Size (Number of cities)	Min distance found (actual = 12)	Time Taken(secs)
Sequential	**-**	**-**	**12**	**12**	**10.476**
Genetic	**32**	**32**	**12**	**12**	**0.322**

Here we can see that the Genetic Algorithm has a speedup of **32.53x** over the sequential algorithm while giving the same answer.

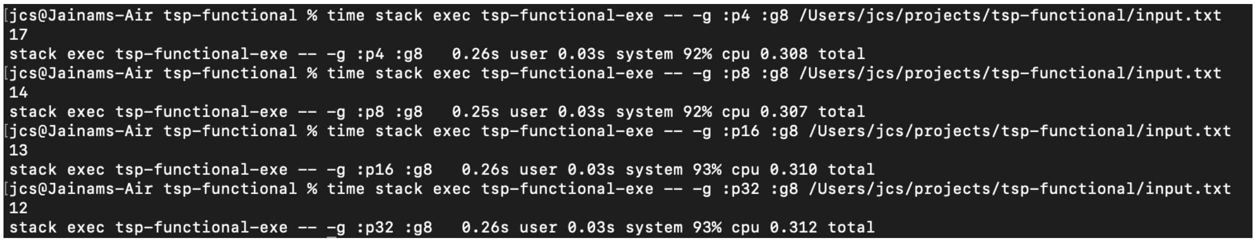


```
[jcs@Jainams-Air tsp-functional % time stack exec tsp-functional-exe -- -g :p16 :g1 /Users/jcs/projects/tsp-functional/input.txt
15
stack exec tsp-functional-exe -- -g :p16 :g1   0.25s user 0.03s system 93% cpu 0.309 total
[jcs@Jainams-Air tsp-functional % time stack exec tsp-functional-exe -- -g :p16 :g4 /Users/jcs/projects/tsp-functional/input.txt
14
stack exec tsp-functional-exe -- -g :p16 :g4   0.26s user 0.03s system 92% cpu 0.314 total
[jcs@Jainams-Air tsp-functional % time stack exec tsp-functional-exe -- -g :p16 :g8 /Users/jcs/projects/tsp-functional/input.txt
13
stack exec tsp-functional-exe -- -g :p16 :g8   0.25s user 0.03s system 92% cpu 0.305 total
[jcs@Jainams-Air tsp-functional % time stack exec tsp-functional-exe -- -g :p16 :g16 /Users/jcs/projects/tsp-functional/input.txt
13
stack exec tsp-functional-exe -- -g :p16 :g16   0.26s user 0.03s system 93% cpu 0.312 total
[jcs@Jainams-Air tsp-functional % time stack exec tsp-functional-exe -- -g :p16 :g32 /Users/jcs/projects/tsp-functional/input.txt
12
stack exec tsp-functional-exe -- -g :p16 :g32   0.26s user 0.03s system 95% cpu 0.308 total
```

Method	Population Size	Generations	Input Size (Number of cities)	Min distance found (actual = 12)	Time Taken(secs)
Genetic	**16**	**1**	**12**	**15**	**0.309**
Genetic	**16**	**4**	**12**	**14**	**0.314**
Genetic	**16**	**8**	**12**	**13**	**0.305**
Genetic	**16**	**16**	**12**	**13**	**0.312**

Genetic	**16**	**32**	**12**	**12**	**0.308**

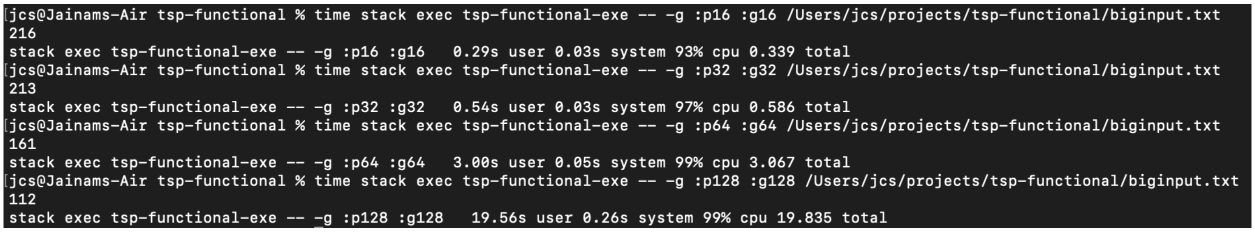
Keeping the population size constant at 16, we see that the accuracy increases as the number of generations increases and we get the optimal solution after 32 generations.



```
[jcs@Jainams-Air tsp-functional % time stack exec tsp-functional-exe -- -g :p4 :g8 /Users/jcs/projects/tsp-functional/input.txt
17
stack exec tsp-functional-exe -- -g :p4 :g8   0.26s user 0.03s system 92% cpu 0.308 total
[jcs@Jainams-Air tsp-functional % time stack exec tsp-functional-exe -- -g :p8 :g8 /Users/jcs/projects/tsp-functional/input.txt
14
stack exec tsp-functional-exe -- -g :p8 :g8   0.25s user 0.03s system 92% cpu 0.307 total
[jcs@Jainams-Air tsp-functional % time stack exec tsp-functional-exe -- -g :p16 :g8 /Users/jcs/projects/tsp-functional/input.txt
13
stack exec tsp-functional-exe -- -g :p16 :g8   0.26s user 0.03s system 93% cpu 0.310 total
[jcs@Jainams-Air tsp-functional % time stack exec tsp-functional-exe -- -g :p32 :g8 /Users/jcs/projects/tsp-functional/input.txt
12
stack exec tsp-functional-exe -- -g :p32 :g8   0.26s user 0.03s system 93% cpu 0.312 total
```

Method	**Population Size**	**Generations**	**Input Size (Number of cities)**	**Min distance found (actual = 12)**	**Time Taken(secs)**
Genetic	**4**	**8**	**12**	**17**	**0.308**
Genetic	**8**	**8**	**12**	**14**	**0.307**
Genetic	**16**	**8**	**12**	**13**	**0.310**
Genetic	**32**	**8**	**12**	**12**	**0.312**

Keeping the number of generations constant at 8, we see that the accuracy increases as the population size increases and we get the optimal solution with a population size of 32.



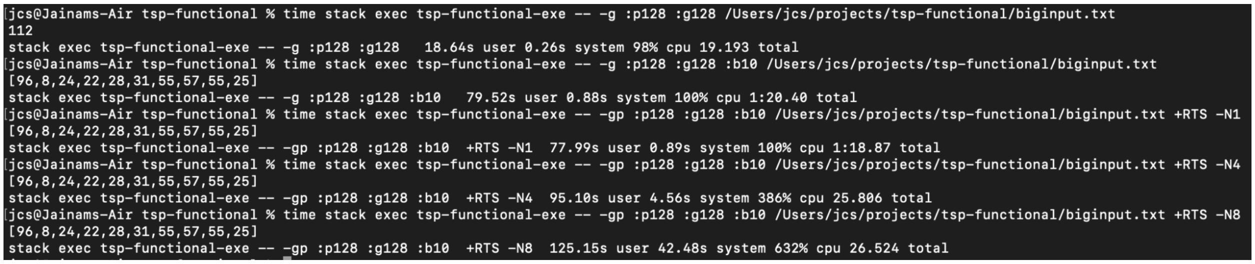
```
[jcs@Jainams-Air tsp-functional % time stack exec tsp-functional-exe -- -g :p16 :g16 /Users/jcs/projects/tsp-functional/biginput.txt
216
stack exec tsp-functional-exe -- -g :p16 :g16   0.29s user 0.03s system 93% cpu 0.339 total
[jcs@Jainams-Air tsp-functional % time stack exec tsp-functional-exe -- -g :p32 :g32 /Users/jcs/projects/tsp-functional/biginput.txt
213
stack exec tsp-functional-exe -- -g :p32 :g32   0.54s user 0.03s system 97% cpu 0.586 total
[jcs@Jainams-Air tsp-functional % time stack exec tsp-functional-exe -- -g :p64 :g64 /Users/jcs/projects/tsp-functional/biginput.txt
161
stack exec tsp-functional-exe -- -g :p64 :g64   3.00s user 0.05s system 99% cpu 3.067 total
[jcs@Jainams-Air tsp-functional % time stack exec tsp-functional-exe -- -g :p128 :g128 /Users/jcs/projects/tsp-functional/biginput.txt
112
stack exec tsp-functional-exe -- -g :p128 :g128   19.56s user 0.26s system 99% cpu 19.835 total
```

Method	**Population Size**	**Generations**	**Input Size (Number of cities)**	**Min distance found (actual = 102)**	**Time Taken(secs)**
Genetic	**16**	**16**	**102**	**216**	**0.339**
Genetic	**32**	**32**	**102**	**213**	**0.586**
Genetic	**64**	**64**	**102**	**161**	**3.067**

Genetic	**128**	**128**	**102**	**112**	**19.198**

For a very large input size (102 cities), solving it by brute force is infeasible as we would need to calculate (101)! possibile path distances. However we are able to calculate a reasonable approximation 112 to the actual min distance of 102 within reasonable time using the genetic algorithm. Above, we see that accuracy increases as population size and number of generations increases.

Genetic Algorithm in batch analysis over big input

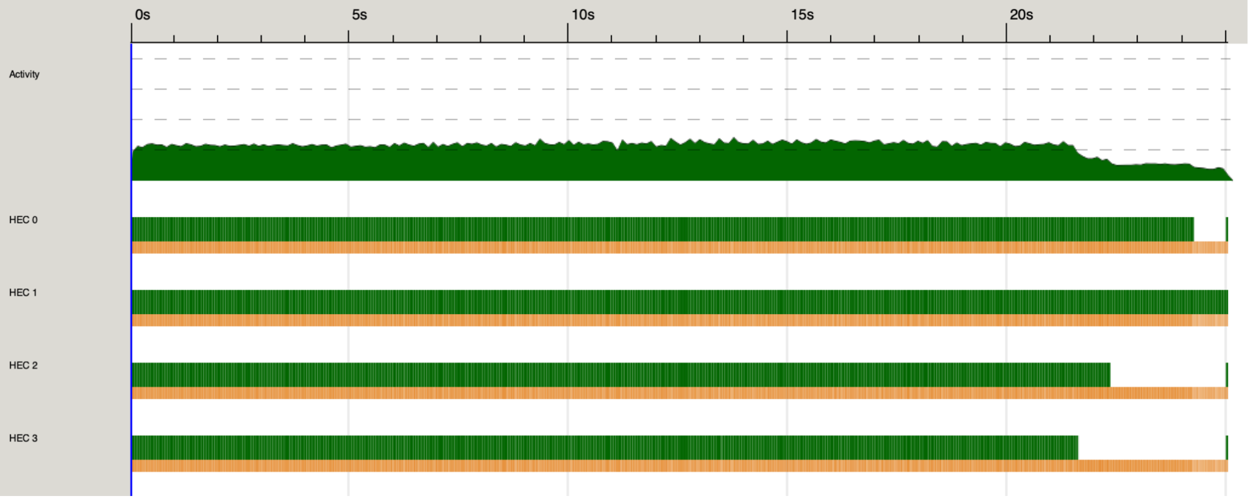


```
[jcs@Jainams-Air tsp-functional % time stack exec tsp-functional-exe -- -g :p128 :g128 /Users/jcs/projects/tsp-functional/biginput.txt
112
stack exec tsp-functional-exe -- -g :p128 :g128   18.64s user 0.26s system 98% cpu 19.193 total
[jcs@Jainams-Air tsp-functional % time stack exec tsp-functional-exe -- -g :p128 :g128 :b10 /Users/jcs/projects/tsp-functional/biginput.txt
[96,8,24,22,28,31,55,57,55,25]
stack exec tsp-functional-exe -- -g :p128 :g128 :b10   79.52s user 0.88s system 100% cpu 1:20.40 total
[jcs@Jainams-Air tsp-functional % time stack exec tsp-functional-exe -- -gp :p128 :g128 :b10 /Users/jcs/projects/tsp-functional/biginput.txt +RTS -N1
[96,8,24,22,28,31,55,57,55,25]
stack exec tsp-functional-exe -- -gp :p128 :g128 :b10  +RTS -N1  77.99s user 0.89s system 100% cpu 1:18.87 total
[jcs@Jainams-Air tsp-functional % time stack exec tsp-functional-exe -- -gp :p128 :g128 :b10 /Users/jcs/projects/tsp-functional/biginput.txt +RTS -N4
[96,8,24,22,28,31,55,57,55,25]
stack exec tsp-functional-exe -- -gp :p128 :g128 :b10  +RTS -N4  95.10s user 4.56s system 386% cpu 25.806 total
[jcs@Jainams-Air tsp-functional % time stack exec tsp-functional-exe -- -gp :p128 :g128 :b10 /Users/jcs/projects/tsp-functional/biginput.txt +RTS -N8
[96,8,24,22,28,31,55,57,55,25]
stack exec tsp-functional-exe -- -gp :p128 :g128 :b10  +RTS -N8  125.15s user 42.48s system 632% cpu 26.524 total
```

Method	Population Size	Generations	Batch Size	Num of Core	Time Taken(secs)
Genetic	128	128	1	1	19.193
Genetic	128	128	10	1	80.40
Genetic batch parallel processing	128	128	10	1	78.87
Genetic batch parallel processing	128	128	10	4	25.806
Genetic batch parallel processing	128	128	10	8	26.524

We parallelize the calculation of a batch of 10 groups using the genetic algorithm, each having about 102 cities. We see a speedup of **3.03x** by parallelizing the batch processing.

8 Core with batch size 10 over population size of 128 and 128 generations



Conclusion

Calculating the euclidean path distance for a set of coordinates is not a very heavy task by itself and therefore parallelizing the calculation of multiple paths at once does not benefit us much. Dividing the parallelization into chunks helps reduce the number of sparks and garbage collection.

Since calculating the entire tsp min distance path for a set of cities is a much more intensive task than calculating one euclidean path. Therefore we see that these batch computations greatly benefit by parallelization. For a batch of 128 city groups, we see a speedup of **3.45x** when we move from sequential to batch algorithms.

We also see that the Genetic Algorithm has a speedup of **32.53x** over the sequential algorithm while giving the same answer. The Genetic Algorithm also makes it feasible to solve very large problems which are not possible to be solved by brute force in reasonable time.

We also parallelize the calculation of a batch of 10 groups using the genetic algorithm, each having about 102 cities and see a speedup of **3.03x** by parallelizing the batch processing

Code Listing

Lib.hs

```
module Lib
  ( runMain,
  )
where


import Control.Parallel.Strategies
import Data.List (permutations)
```

```
import GeneticUtils
import System.Environment (getArgs, getProgName)
import System.Exit (die)
import System.IO (readFile)
import Types
import Utils

-- Consider all permutations while keeping starting point fixed
minPathDistance :: [Point] -> Int
minPathDistance [] = -1
minPathDistance (c : cities) =
 minimum $
   map (pathDistance . (c :)) $
     permutations cities

parallelMinPathDistance :: [Point] -> Int
parallelMinPathDistance [] = -1
parallelMinPathDistance (c : cities) =
 minimum $
   parMap rseq (pathDistance . (c :)) $
     permutations cities

chunkedParallelMinPathDistance :: [Point] -> Int -> Int
chunkedParallelMinPathDistance [] _ = -1
chunkedParallelMinPathDistance (c : cities) chunkSize =
 minimum $
   withStrategy (parListChunk chunkSize rdeepseq)
     . map (pathDistance . (c :))
     $ permutations cities

batchMinPathDistance :: [[Point]] -> [Int]
batchMinPathDistance = map minPathDistance

batchParallelMinPathDistance :: [[Point]] -> [Int]
batchParallelMinPathDistance = parMap rseq minPathDistance

geneticMinPathDistance :: Int -> Int -> [Point] -> Int
geneticMinPathDistance _ _ [] = -1
geneticMinPathDistance populationSize generations cities =
 minimum $ map pathDistance finalPop
 where
   population = replicate populationSize cities
```

```
   randomList = randomListInRange 0 (length cities - 1)
   finalPop =
     fst $
       foldr
         (\f (p, r) -> (f p r, tail r))
         (population, randomList)
         (replicate generations nextGen)

batchGeneticMinPathDistance :: Int -> Int -> [[Point]] -> [Int]
batchGeneticMinPathDistance p g =
 map
   (geneticMinPathDistance p g)

batchParallelGeneticMinPathDistance :: Int -> Int -> [[Point]] -> [Int]
batchParallelGeneticMinPathDistance p g =
 parMap
   rseq
   (geneticMinPathDistance p g)

runMain :: IO ()
runMain = do
 args <- getArgs
 case args of
   -- bruteforce sequential
   ["-s", filename] -> do
     corpus <- readFile filename
     print $ minPathDistance $ makeCities corpus

   -- bruteforce, calculate path distance in parallel
   ["-p", filename] -> do
     corpus <- readFile filename
     print $ parallelMinPathDistance $ makeCities corpus

   -- bruteforce, calculate path distance in parallel chunks
   ["-c", ':' : 'n' : n, filename] -> do
     corpus <- readFile filename
     print $ chunkedParallelMinPathDistance (makeCities corpus) (read n)

   -- bruteforce for batch of city groups
   ["-s", ':' : 'b' : b, filename] -> do
     corpus <- readFile filename
     let cities = makeCities corpus
```

```
         randomList = randomListInRange 0 (length cities)
      in print $ batchMinPathDistance [take r cities | r <- take (read b) randomList]

   -- bruteforce for batch of city groups, each group in parallel
   ["-sp", ':' : 'b' : b, filename] -> do
     corpus <- readFile filename
     let cities = makeCities corpus
         randomList = randomListInRange 0 (length cities)
      in print $ batchParallelMinPathDistance [take r cities | r <- take (read b)
randomList]

   -- genetic algorithm
   ["-g", ':' : 'p' : p, ':' : 'g' : g, filename] -> do
     corpus <- readFile filename
     print $ geneticMinPathDistance (read p) (read g) $ makeCities corpus

   -- genetic algorithm for batch of city gorups
   ["-g", ':' : 'p' : p, ':' : 'g' : g, ':' : 'b' : b, filename] -> do
     corpus <- readFile filename
     let cities = makeCities corpus
         randomList = randomListInRange 0 (length cities)
      in print $ batchGeneticMinPathDistance (read p) (read g) [take r cities | r <-
take (read b) randomList]

   -- genetic algorithm for batch of city gorups, each group in parallel
   ["-gp", ':' : 'p' : p, ':' : 'g' : g, ':' : 'b' : b, filename] -> do
     corpus <- readFile filename
     let cities = makeCities corpus
         randomList = randomListInRange 0 (length cities)
      in print $ batchParallelGeneticMinPathDistance (read p) (read g) [take r cities
| r <- take (read b) randomList]

   -- invalid running params
   _ -> do
     pn <- getProgName
     die $ "Usage: " ++ pn ++ " [-s|-p|-c :nN|-s :bN|-sp :bN|-g :pN :gN|-g :pN :gN
:bN|-gp :pN :gN :bN] <filename>"
```

Types.hs

```
module Types (Point) where
```

```haskell
type Point = (Int, Int)
```

Utils.hs

```haskell
module Utils
 ( distance,
   squaredDistance,
   makeCities,
   pathDistance,
   squaredPathDistance,
   randomListInRange,
 )
where

import System.Random (mkStdGen, randomRs)
import Types

squaredDistance :: Point -> Point -> Int
squaredDistance (x1, y1) (x2, y2) = ((x2 - x1) ^ 2) + ((y2 - y1) ^ 2)

distance :: Point -> Point -> Int
distance a b = floor . sqrt . fromIntegral $ squaredDistance a b

makeCities :: String -> [Point]
makeCities corpus = makePairs $ map read $ words corpus
 where
   makePairs [] = []
   makePairs [p] = [(p, p)] -- replicate last coordinate if odd numbers
   makePairs (p : q : r) = (p, q) : makePairs r

pathDistance :: [Point] -> Int
pathDistance cities = sum $ zipWith distance path (tail path)
 where
   path = last cities : cities

squaredPathDistance :: [Point] -> Int
squaredPathDistance cities = sum $ zipWith squaredDistance path (tail path)
 where
   path = last cities : cities

randomListInRange :: Int -> Int -> [Int]
randomListInRange s e = randomRs (s, e) rg
```

```
 where
   rg = mkStdGen 0
```

GeneticUtils.hs

```haskell
module GeneticUtils (nextGen) where

import Data.List (sortBy)
import qualified Data.Set as S
import Types
import Utils

crossover :: [Point] -> [Point] -> Int -> Int -> [Point]
crossover parentA parentB i j = c1 ++ c2
 where
   s = min i j
   e = max i j
   c1 = [x | (x, i) <- zip parentA [0 ..], s <= i && i <= e]
   c1Set = S.fromList c1
   c2 = [x | x <- parentB, not (x `S.member` c1Set)]

nextGen :: [[Point]] -> [Int] -> [[Point]]
nextGen pop randomList =
 take (length pop) $
   map fst $
     sortBy (\p1 p2 -> compare (snd p1) (snd p2)) $
       map (\p -> (p, squaredPathDistance p)) $
         [ crossover pa pb ri rj
           | (i, pa, ri) <- zip3 [0 ..] pop randomList,
             (j, pb, rj) <- zip3 [0 ..] pop (tail randomList),
             i < j
         ]
```

Main.hs

```haskell
module Main where

import Lib

main :: IO ()
main = runMain
```

Others

```
dependencies:
- base >= 4.7 && < 5
- parallel
- random
- containers
```

```
ghc-options:
- -O2
- -threaded
- -rtsopts
- -eventlog
```

References

[1] https://en.wikipedia.org/wiki/Travelling_salesman_problem
[2] https://www.schoolofhaskell.com/school/starting-with-haskell/libraries-and-frameworks/randoms
[3] https://stackoverflow.com/questions/40097116/get-all-permutations-of-a-list-in-haskell
[4] https://hackage.haskell.org/package/parallel-3.2.2.0/docs/Control-Parallel-Strategies.html
[5] https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3