CS546 “Parallel and Distributed Processing”

A comparison of the TSP solution in serial and parallel using Map Reduce

Chris Mathew Dani
A20372828

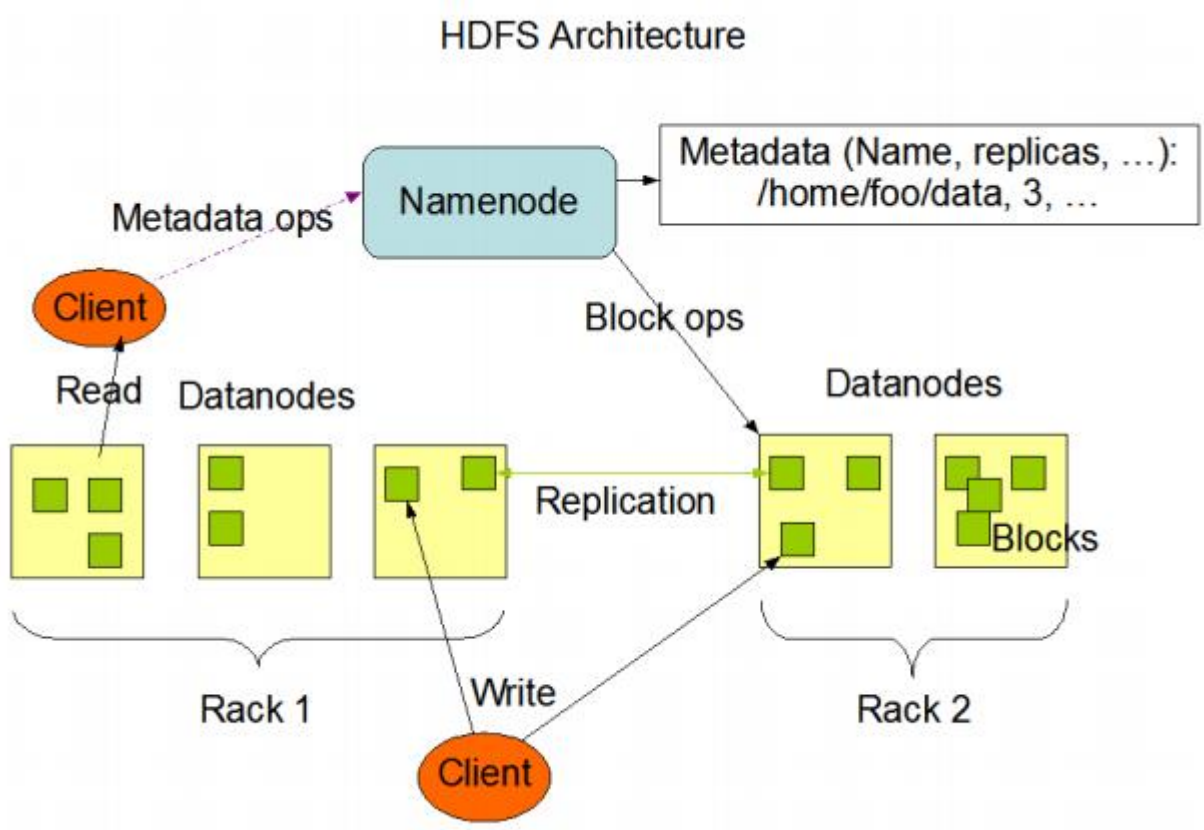
***Abstract**: We take the problem of finding the least cost path given a set of cities commonly referred to as the Travelling Salesman Problem and we identify an approach to solve it using Hadoop. This problem is a combinatorial optimization problem and is NP complete. So there is no known polynomial time algorithm to solve this. We implement a 2-opt solution and we identify the different challenges and results after parallelizing the solution. We conclude by checking if Map Reduce was the best platform for this task.*

The problem is that of helping a salesman travel n cities and return to the same starting point after covering all the cities. The goal over here is to find a route to travel all the cities with as minimal cost as possible. This problem is commonly referred to as the Travelling Salesman Problem (TSP). This is a combinatorial optimization problem (finding an optimal object from a finite set of objects).A naive approach is to compute all possible paths and then calculating the costs of all these paths and then selecting a path which has the least cost. The problem here is that computing all possibilities has a complexity of the factorial order and so it is not feasible for larger cities. In this particular case it is not yet proved that there is a polynomial time algorithm yet, it is a NP complete problem. Given a solution it can be verified if it's the optimal solution by going through the graph in linear time and computing the time taken. There are polynomial time approaches that can come close to solving the solution and they are called approximation algorithms and does not necessarily give the most optimal solution but can come close to the optimal solution. There are greedy algorithms that choose the best next city from the current city and then so on and so forth till the last city is reached (Nearest Neighbor). There is insertion based algorithm where you keep inserting cities to be sure that it has lesser cost (Greedy algorithm). There are heuristic based algorithms as well. Of these is a class of algorithms called Local Search algorithms. These algorithms look for a solution by searching in a localized area. There are 3 common local search algorithms, 2-opt, 3opt and Lin-Kernighan. 2-opt algorithm works by swapping edges that can reduce the overall tour cost until there are none left to swap. 3-opt works by swapping 3 edges at a time.

There are both asymmetric and symmetric variants of this problem, the asymmetric variant has a condition that the cost to travel to another city in one direction is not the same as that of the reverse direction, the cost can vary depending on the direction of the travel. Here for simplicity and because the focus is on parallelism and not the actual solution we use the symmetric variant.

Hadoop:

Hadoop is a framework for storage and computation of large data sets. The storage is using HDFS (Hadoop Distributed File System) and the computation uses the Map Reduce framework. Hadoop was introduced to bring about a platform where computation and storage can scale well to fulfill growing computational demands brought by larger data sizes. Scaling vertically has reached a limit (heating and size of chip) (Moore's law), so horizontal scaling was the solution. Hadoop provides that in terms of data storage as well as in computation and hence it is a popular solution. Few other reasons for Hadoop's popularity are getting computation cheaply was achieved here by using commodity hardware. All challenges that come with scaling horizontally and using commodity hardware were addressed as well. Also, parallelizing tasks is very easy and so converting a task into Hadoop compatible format is easy. Node failure is common and handled well in the hadoop architecture. Number of nodes can increase very easily with minimal change. Also, each application need not integrate, failure handling or parallelizing etc, Hadoop automatically takes care of these. It is also very efficient, reliable and open source.



Hadoop also provide faults tolerance mechanism by which system continues to function correctly even after some components fail's working properly. Faults tolerance is mainly achieved using data duplication and making copies of same data sets in two or more data nodes, heartbeats and recovery. Hence given all these advantages, Hadoop was chosen for parallelizing the TSP problem using opt.

The file system used by Hadoop, HDFS is very reliable and well suited for map reduce based tasks. HDFS is distributed and built in a way that the data is evenly spread across all nodes. So that map reduce tasks can process data where it is stored, so that the processing occurs on local data. The localized data processing gives speed so that there is little communication among nodes.

HDFS consists of datanodes and namenode. The namenode ensures coordination and the datanode is the storage units.

Solution:

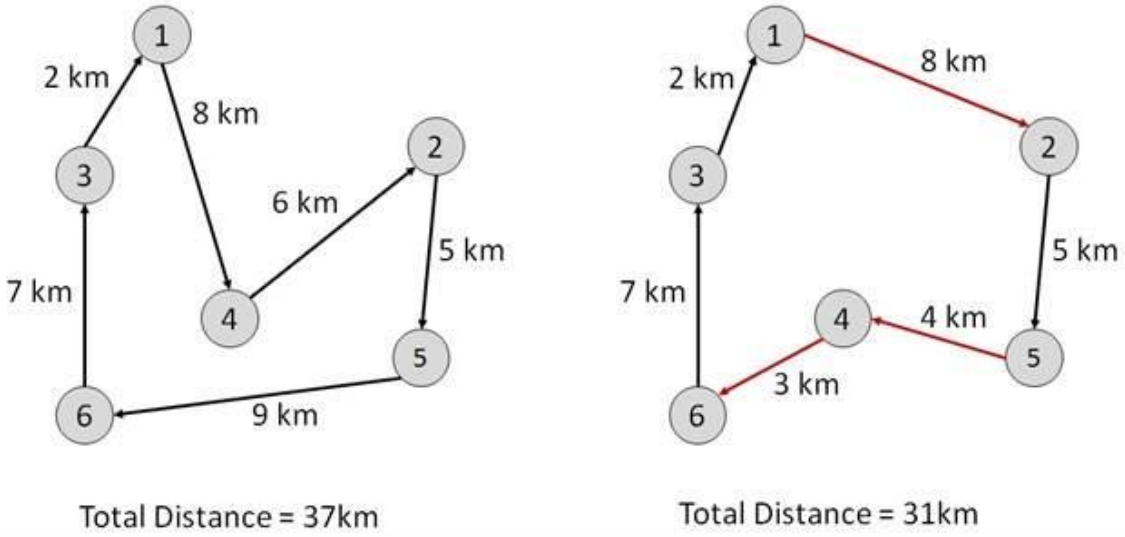
No work has been done on parallelizing the local search algorithms. So here we parallelize the 2-opt local search and see what results are obtained.

Brute force approach: We generate all the possible city permutations .Then we just iterate through cities adding the cost for each city to get the total cost.

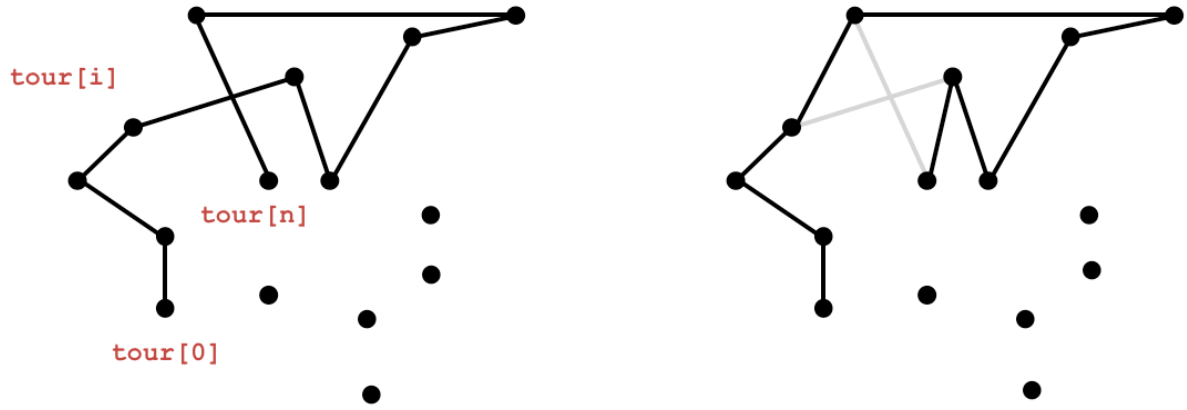
2-opt approach: We iterate through the cities pairwise and compute the most profitable city swap and then continue doing this until no more swaps can make the solution better.

Parallel 2-opt approach: We split the single input of several cities into several groups of few cities each, and then run the 2-opt algorithm on each section and find the local optimum. Then all the groups are merged and then the algorithm is run again, this time to find the global optimum solution.

2-opt was chosen because of its ability to divide the data and process as well as the simplicity in implementing the algorithm in serial and in parallel. In most other algorithms the task dependencies allow for very little concurrency because of the implicit dependencies between successive operations. Here we have split the data to achieve concurrency.



In the image above, two paths are illustrated each with different length, we need to find out a path that is overall minimal. The right one is cheaper than the left.



In the image above, we see how 2-opt works, the crossed edge shown on the right (greyed out) is selected to be swapped and then is swapped.

Two approaches, complexity:

1) Brute force approach: All cities are examined individually, from the start to each city and the distance is measured. Serial execution took only a little amount of time.
2) Local search: A random path is taken as the starting point and then better ones are searched by exchanging few edges. Using 2-opt, 2 edges are changes to see if a better one can be obtained.

Approach:	Serial time	Parallel time
Brute force (each vertex is checked)	O(n!)	O(1)
Local search – 2 opt	O(n^2) * j-iterations	O(n^2) * k-iterations

Where n is the number of cities to be searched. j and k are the number of iterations to reach the optimal solutions. k<<j because lot of computations are covered in the parallel portion, and the rest are executed serially.

Complexity for the parallel portion: O(m^2) , where m = n/g, where n is the number of cities and g is the number of groups. So m is the number of cities in each group. Since each group is run in parallel we achieve time complexity of O(m^2) instead of O(m) * g if these were computed serially.
Complexity for the serial portion: O(n^2) * k-iterations. Here n is the number of cities. K is the number of iterations needed to reach the optimal solution.
We know that k is much less than j because lot of local optimizations have already occurred in the sub groups.

One major drawback in the predictability for the 2-opt performance is the number of iterations. This is heavily dependent on the distance of the starting solution from the optimal solution (by distance we mean the number of swaps needed). Hence the algorithm performs much better when the starting solution is closer to the optimal one. Several approaches can be taken to achieve this, for example a branch and bound approach to pick the starting path and then improve from there using the 2-opt. (This can be executed for some time and then the output) another way is to randomly pick n paths and then select the cheapest one amongst them. Neither can guarantee a better starting solution, but we only focus on having a better starting point. In this project, we fix the starting path to be a straightforward order the cities appear in the file where the set of cities is stored. This is done because when we compare the serial and parallel versions, we only want to compare the performance differences and not necessarily want the best solution in the shortest amount of time.

Implementation:

The 2-opt algorithm was implemented in serial and in parallel. The distance between cities is taken as the Euclidian distance for its location x and y.

Serial execution:
We read the input from a file, a set of cities and its location as coordinates x and y. We set the starting route as the route in the order the cities were stored in the file. We go over the entire list and compare pairs of edges to see the best exchange choice and keep iterating till there is no more exchanges possible.

Pseudocode for 2-opt (the core component):

```
1.  do {
2.      mini = 0;
3.      mink = 0;
4.      minchange = 0;
5.      for(int i = 1; i < noOfCities-2; i++)
6.      {
7.          for(k = i+2; k< noOfCities-1 ; k++)
8.          {
9.              change = calcDist(i, k, Route, distancematr)
10.             + calcDist(i+1, k+1, Route, distancematr)
11.             - calcDist(i, i+1, Route, distancematr)
12.             - calcDist(k, k+1, Route, distancematr);
13.         if(minchange > change){
14.             minchange = change;
15.             mini = i; mink = k;
16.             }
17.         }
18.     }
19.      Route = swap(Route, mini, mink);
20.  } while(minchange < 0);
```

This is the core portion of the 2-opt algorithm, the part that takes the most time. In a high level, this loop just contains two loops which allows comparison with all pairs of edges to find the most profitable swap in the path (loops in line 5 and 7). This is repeated as long as there are edges that are needed to be swapped that can make the total cost lesser (do-while loop).

Functions that are called, swap () returns the new path obtained by exchanging the cities at index mini and mink. This swap is done because after comparisons it was found that swapping these would provide a cheaper path. That is, going from mini to mink directly is better than going to its next city in the current route. This swap is implemented by reversing the path from i to k.

calcDist() simply takes in the city i and j and the route and using information in the distancematrix computes the cost of going from city i to j.

Parallel execution:
In parallel execution, we have to modify the input, such that the work is split equally amongst the mappers and we get the desired output. In order to achieve that the input was split into 25

approximately equal parts. So that each mapper will work on achieving the local optimum within that mappers data set. After one iteration with the split data set, the map reduce code is once again run on the output from the first iteration. This time we compute the global optimum and produce the result.

For the mappers we assign all of the mappers the same key value and using that the reducer will combine each mappers output to a single value by simple append operation.

For the first iteration each mapper will work on a subset of the entire input data whereas in the second iteration, only one mapper will execute as there is only one input line. So in the first iteration we will obtain parallelism with as much nodes there is, in the second it effectively runs serially.

Pseudo code for parallel in Hadoop:

```
    Mapper:
1.  Set key = 1
2.  //value is the line of the input file with city(s) information
3.  do {
4.      mini = 0;
5.      mink = 0;
6.      minchange = 0;
7.      for(int i = 1; i < noOfCities-2; i++)
8.      {
9.          for(k = i+2; k< noOfCities-1 ; k++)
10.         {
11.             change = calcDist(i, k, Route, distancematr)
12.             + calcDist(i+1, k+1, Route, distancematr)
13.             - calcDist(i, i+1, Route, distancematr)
14.             - calcDist(k, k+1, Route, distancematr);
15.         if(minchange > change){
16.             minchange = change;
17.             mini = i; mink = k;
18.             }
19.         }
20.     }
21.     Value = swap(value, mini, mink);
22. } while(minchange < 0);
```

23. **return key, Value**

Reducer:

1. **for key, values from mapper:**
2. String a = *concatenate(values)*
3. **return key, a**

Each mapper gets a line of input data. It does the exact same work as that of the serial code, but here it works on a subset of the data and not the whole data. So the operations are the same but the data is different. All data lines will have the same key and in the mapper it will combine all optimized groups of cities and merge them as one.

The input for this is a line having m rows. Each row having k cities. This is equally divided. Each mapper takes one row and optimizes it. After one iteration of a map and reduce process, the algorithm is run again to take the input of the previous iteration's output. This then produces the final solution.

Test execution and Results:

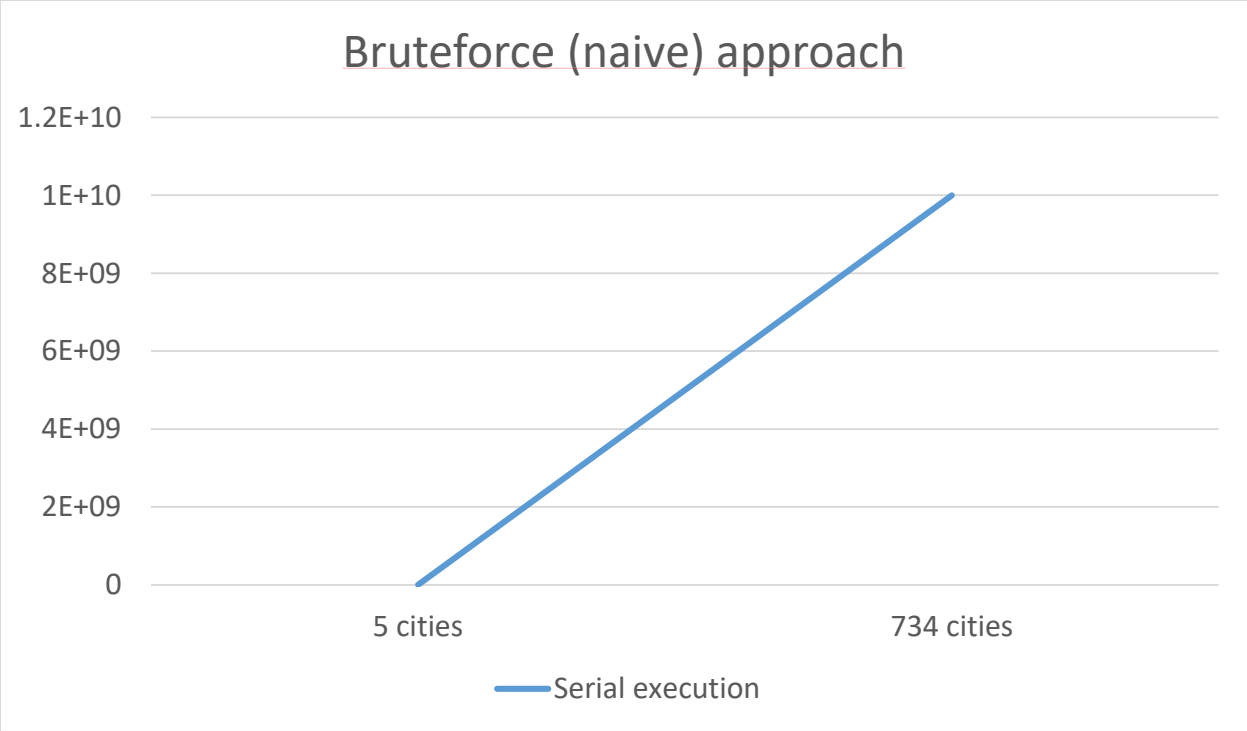
Set - up:

Jarvis failed multiple times both connection wise and storage wise. The setup instead was done on Amazon EC2 instances. 4 instances were used, 1 namenode and 3 datanodes. Each node contains 1 GB of RAM and runs on Ubuntu 16.04.1. Each instance has a single CPU: Intel(R) Xeon(R) CPU E5-2676 v3 @ 2.40GHz. The hadoop execution was done on this machine while the serial execution was on one of these instances. Throughout the execution and tests the machines were reliable and no job had to be re run.

The data used was from university of waterloo website, the cities uruguay, sahara and dijibouti were taken. The algorithms were implemented in Java and java compiler version 1.8 was used.

Result:

For the bruteforce execution, it took 30 seconds to calculate the best path and for the inupt of 734 cities, it did not finish executing. This was expected as the number of computations grew too large to be executed on commodity hardware.



Created the 2-OPT Serial algorithm for TSP and implemented it

For the 2-opt algorithm, the execution for a data size of 29, 38 and 734 cities were done. The execution was done on 3 modes, Serial execution on a single system in java, serial execution by using a single mapper in Hadoop and parallel execution by using 25 mappers in the first iteration and a single mapper in the second iteration.

As can be seen, the serial execution both in plain java as well as in Hadoop has negligible difference. This is due to the fact that no parallelization of Hadoop is utilized during this time, thus we are not making use of the available resources fully.

The execution that uses 25 mappers performed much better (whether 3 instances or 4, here instances are the data nodes available for execution). Here the parallelism was taken advantage of. When we breakdown the time taken for the execution by the parallel and serial phase we see that the parallel phase took only 2 minutes while the serial phase took 150 minutes. That is, the part where we utilized parallelism ran very fast and comes as negligible when compared to the total time it takes to execute. For the city size of 734, each mapper had around 29 cities to find the local optimum within it. Taking 2 minutes to execute that is expected since a serial implementation took around the same amount of time. Also, the effects of data movement was not noticed because of the extremely small size of the data and the fast network that connects these nodes. The serial phase took much time because it had to run 734^2 executions for several iterations. When compared to a linear execution, the parallel execution allows the solution computation to reach several iterations ahead in the parallel phase. Which is then executed serially to reach the optimal solution. The parallel phase can be thought of as an accelerator to reach the computations be ahead by several iterations in a very short amount of time. For the case of 734 cities, the initial cost was close to 438k units, in serial execution, we keep iterating over it to reach

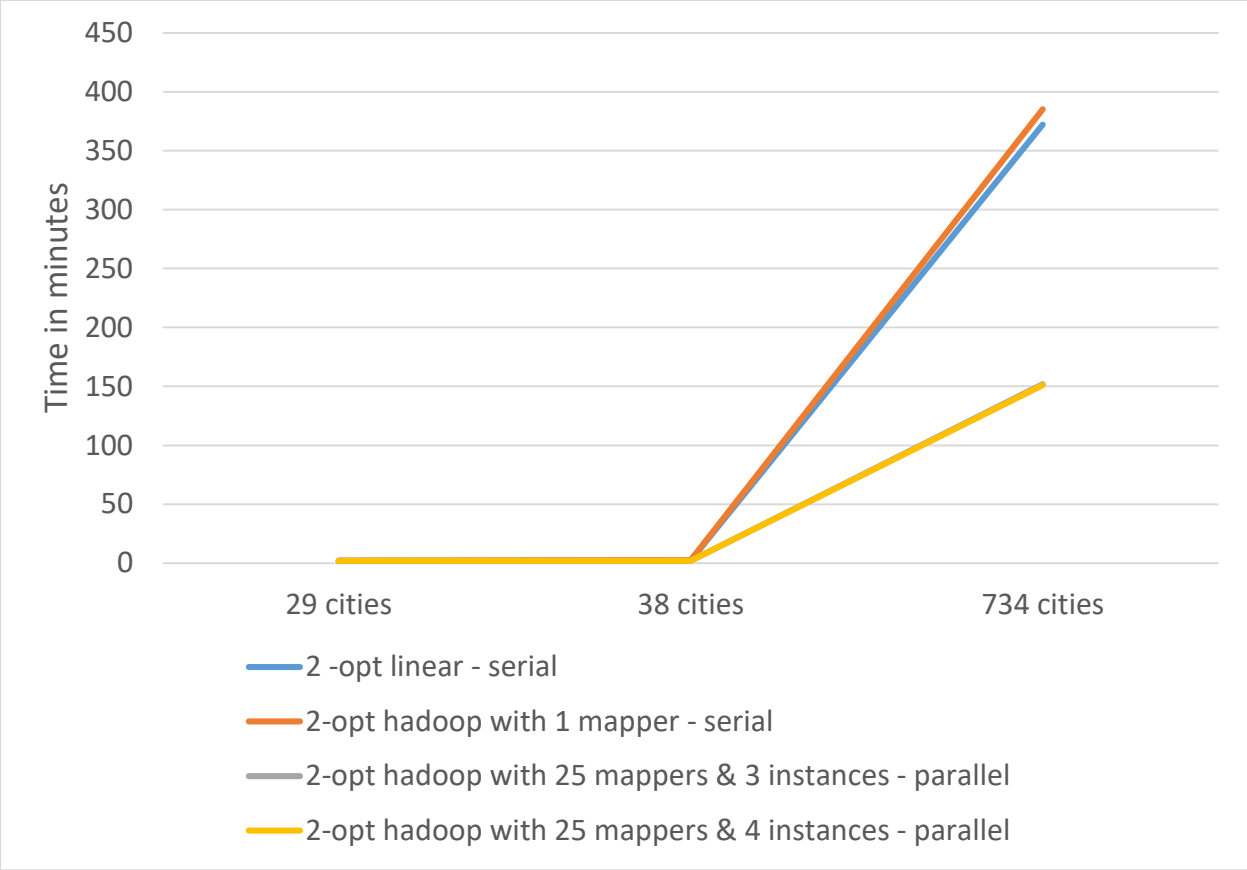
85k cost solution. In the parallel phase, in just 7 (average) iterations, the cost was brought down to close to 232k units. This took close to 4 hours to reach in the serial only execution. Here is where we got a huge advantage. We were able to cut down 4 hours into 2 minutes. This is because of the available local swaps that can be done without comparing it globally.

Further analysis of the parallel phase: We see a superlinear speedup here. This is for two reason, the computation per mapper is much lesser than if we take the total computation and the number of iterations is much lesser because the search space is less (Only so much local optimization can be done) . This is because the computation per mappers is of the order O(m^2) + k-iterations when compared, if this was done linearly, it would have been O(n^2) + p-iterations. Here m is the number of cities per mapper and k is the number of iterations. N is the total number of cities and p is the number of iterations. Here we also know that k << p because in s smaller search space we require only fewer changes.

Another aspect of load distribution is that the load distribution in each mapper. Some had as much as 7 iterations and some had only 2. So because the number of instances available is less than the number of mappers, the ones that get over early give way to others that are not yet done. So overall there's no performance hit. If there were more or equal number of instances than mappers, then the performance would have been affected.

Another aspect in solving this problem using 2-opt is that it is an approximation solution and does not guarantee that the solution it produces is the best. This can be evident in the calculated cost below. In the test cases, we used the data from u of waterloo. They have computed cost based on the Lin-Kernighan heuristic. As can be seen in the results, we see that in smaller cities the calculated city cost is better than that of Lk approach. But in the larger cities the computed cost is not good compared to the provided solution. This seems to be within acceptable limits and so we do not give much importance to this. But it is evident that cutting down time using 2-opt can affect the optimality of the result. Also, the closeness to optimality depends on starting route as well as the number of swaps that occurred which caused a global optimal not to be achieved. Which is why the quality of the solution decreased when we used parallelism.

- Results:
    - 6 hours 12 minutes for 734 TSP instance. Calculated cost: 85347.95 , Optimal: 79114.
    - 2.1 minutes for 38 TSP instance. Calculated cost: 6402 Optimal cost: 6656.
    - 2 minutes for 29 TSP instance. Calculated cost: 26852 Optimal: 27603.
    - 2 hours 32 minutes for 734 TSP instance for 25 mappers and 3 datanodes in hadoop. Calculated cost: 96354.75 , Optimal: 79114.
    - 2 hours 31 minutes for 734 TSP instance for 25 mappers and 4 datanodes in hadoop. Calculated cost: 96354.75 , Optimal: 79114.



Approaches and tools that did not work:

- Infrastructure:
    - The Jarvis system ran out of storage and there were random connection drops this caused several days delay in execution as the environment had to be set up again. This shows the importance of high availability of nodes in a cluster. Amazon cluster were reliable and there were 0% connection drops.
- Approaches:
    - Bruteforce approaches: The execution took n factorial time to execute serially. Hence this is not usable in computing cities larger than ~25 as commodity hardware runs out of memory. Unless it is run using out-of-core execution.
    - Local Search : 2- opt An approach where several mappers take on the same data and each of them compute globally best, second best, third best and so on and the reducer can then combine swaps that do not overlap . This allows to parallel compute globally optimal solution. Unfortunately, unlike the MPI of pthreads approach, we are not able to identify which mappers do what task. Or even identify a mapper by its ID. This is because the number of mappers is implicit. Even if we set it, there is no guarantee that there would be so many mappers in the execution. This is because of the automatic

parallelizing feature of Hadoop. Thus data parallelization is implicit but task parallelization cannot be done using Hadoop. We have to separate the tasks and figure out what can be data parallelized in it and then feed it to Hadoop. This is a common trend in combinatorial optimization problems, that there are only task parallelism and less data parallelism. Which is why map reduce cannot be used effectively in most cases.

Conclusion:

The TSP is a computationally hard combinatorial optimization. Here we have shown how it can be executed serially and in parallel in Hadoop. The idea is to find the data parallelism here so that it can be executed on Hadoop. We have shown that because of the nature of the problem we could improve the solution greatly using parallelism but at the same time sacrificed on the quality of the solution. Hence we conclude that Map Reduce helps in achieving a fast solution to an extent and can be used in cases where decreasing the quality of the result a bit does not affect the overall execution as long as we are able to compute a reasonably short path quickly.

References:

[1] J. Dean and S. Ghemawat, “MapReduce : Simplified Data Processing on Large Clusters,” Commun. ACM, vol. 51, no. 1, pp. 1–13, 2008.

[2] Giovanni Cesari. Divide and conquer strategies for parallel TSP heuristics. Computers and Operations Research, Vol. 23, Issue 7, Pages 681–694, 1996.

[3] IZZATDIN A. AZIZ, NAZLEENI HARON, MAZLINA MEHAT, LOW TAN JUNG, AISYAH NABILAH “Solving Traveling Salesman Problem on High Performance Computing using Message Passing Interface” , Proc. of the 7th WSEAS Int. Conf. on COMPUTATIONAL INTELLIGENCE, MAN-MACHINE SYSTEMS and CYBERNETICS (CIMMACS '08)

[4] Prof. Dr. Nadia Erdoğan , Harun Raşit Er “Parallel Genetic Algorithm to Solve Traveling Salesman Problem on MapReduce Framework using Hadoop Cluster” The International Journal of Soft Computing and Software Engineering [JSCSE], Vol. 3, No. 3, Special Issue

[5]Matt Heavner “Massively Parallel Travelling Salesman Genetic Algorithm “

[6] MMManjurul Islam, Md Waselul Haque Sadid, S. M Mamun Ar Rashid, Mir Md Jahangir Kabir “AN IMPLEMENTATION OF ACO SYSTEM FOR SOLVING NP-COMPLETE PROBLEM; TSP”, 4th International Conference on Electrical and Computer Engineering

[7] Leaornardo Zambito “The Traveling Salesman Problem: A Comprehensive Survey”

[8] Sanchit Goyal “A Survey on Travelling Salesman Problem “

[9] Data Source : http://www.math.uwaterloo.ca/tsp/world/countries.html

[10] “HDFS Architecture Guide” by Dhruba Borthakur

[11] Sagar Keer, “A Parallelized Solution for the Traveling Salesman Problem using Genetic Algorithms”