EVALUATION OF A SPARK-ENABLED GENETIC ALGORITHM APPLIED TO THE TRAVELLING SALESMAN PROBLEM

A Paper
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Amritha Raveendran

In Partial Fulfillment of the Requirements
for the Degree of
MASTER OF SCIENCE

Major Department:
Computer Science

April 2020

Fargo, North Dakota

North Dakota State University
Graduate School

Title

EVALUATION OF A SPARK-ENABLED GENETIC ALGORITHM APPLIED TO THE TRAVELLING SALESMAN PROBLEM

By

Amritha Raveendran

The Supervisory Committee certifies that this ***Paper*** complies with North Dakota State University's regulations and meets the accepted standards for the degree of

MASTER OF SCIENCE

SUPERVISORY COMMITTEE:

Dr. Simone Ludwig
Chair

Dr. Pratap Kotala

Dr. Annie Tangpong

Approved:

April 17, 2020
Date

Dr. Kendall Nygard
Department Chair

ABSTRACT

Traveling salesman problem aims to find the shortest route. A salesman travels to each of the cities once. Genetic Algorithm is used for solving this problem, which returns the best solution found showing the distance that can be covered with the minimum cost (shortest path). A Spark-enabled parallel implementation was investigated in terms of performance. The aim of the study was to show the effect of parallelization of a Genetic Algorithm applied to the Travelling salesman problem. Experiments are run using different numbers of processors and the performance of the algorithm is evaluated based on the execution speed. To identify the best performing number of processors to be used, we made a comparison measuring the execution time of the algorithm for different numbers of cities using different numbers of cores.

ACKNOWLEDGEMENTS

Foremost, I would like to express my sincere gratitude to my advisor Dr. Simone Ludwig for the continuous support of my MS Paper, for her patience, motivation, enthusiasm and immense knowledge. Her guidance helped in all the time of research and writing of my paper. Besides my advisor, I would like to thank my MS paper committee members Dr. Pratap Kotala and Dr. Anne Tangpong for their encouragement and support. A special thanks to Computer Science department System Administrator, Guy Hokanson, for his time and patience in answering my queries. Last but not the least, I would like to thank my family and friends for all the strength and support.

DEDICATION

I would like to dedicate this project to my family, friends and teachers who supported me throughout my life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION AND RELATED WORK

The traveling salesman problem aims to find the shortest distance (optimal solution) that can be travelled from one city to another city. One algorithm that can be applied to the problem is the Genetic algorithm. The Genetic algorithm is a search heuristic theory based on the idea of natural selection inspired by Charles Darwin [20]. The process of natural selection begins by choosing the fittest individuals from a population. The individuals produce offspring which inherit the characteristics of their parents and will be added to the next generation. The fitter parents will have a better chance of producing better offspring. This process continues to iterate and at the end, a generation with the fittest individual is selected. There are different genetic operators that play a major role in the performance of the genetic algorithm.

We applied a Genetic Algorithm to solve the Travelling Salesman Problem using Spark, which is a parallel implementation, to find the optimum solution. We ran the program for different numbers of cities on different number of processors (cores), calculated the shortest distance, the time taken to execute it and calculated the speedup. The main objective of the project is to:

- Show the effect of the parallelization using a Spark-based implementation of a genetic algorithm applied to the travelling salesman problem.
- Evaluate the performance of running the genetic algorithm on different numbers of processors in terms of execution time.
- Focus on the parallelization aspect of the Spark-based GA implementation.

Parallelization is the act of intending a computer program to process data in parallel instead of running sequentially. If a system is parallelized, it breaks the problem into parts and runs them independently at the same time by discrete processors. Earlier, computer software was written

for serial computing [25]. In order to solve a problem, an algorithm is constructed and is run sequentially. Advantages of parallel computing are that it saves processing time and solves larger problems in a shorter time compared to serial computing [26]. Many operations can be done simultaneously using parallel computing. In this Travelling Salesman problem, the reason why we chose Genetic Algorithm is because it is a population-based method in which each of the individual solutions will be given attention and we work on each of them simultaneously. Due to this reason, Genetic Algorithm is the best fit for a parallel implementation. Spark is used as the tool to enable the computation. Spark is an open-source distributed cluster computing framework. It was developed at UC Berkeley on 2014. One of the main building blocks of Spark is Resilient Distributed Dataset [RDD]. Spark uses Resilient Distributed Datasets to perform parallel processing across processors. In the Travelling salesman problem using Genetic Algorithm with Spark, we ran the solutions on different number of cores parallelly. Then, obtained the execution time of each of the processors. Later, the speedup is calculated to compare the performance of different number of processors. Thus, we can show the effect of parallelization of a genetic algorithm applied to Travelling salesman problem using Spark.

The application of the genetic algorithm in Urban Bus Route Network Design [1], emphasized on urban public transport system involving determining a route configuration with a set of transit routes and associated frequencies that achieves the desired objective. The authors formulated the scenario as an optimization problem by minimizing the overall cost of users and the operator. The main focus is on the Genetic Algorithm. The design is done in two phases, first a set of potential routes called the candidate route set is generated using the genetic algorithm. Then, an optimum set of routes are selected from the candidate route set using the genetic algorithm. The reason for using this approach is because the genetic algorithm can provide a

robust search as well as a near optimal solution in a reasonable time (for small problems). In another paper [2], the author used Genetic Algorithm using Spark. The authors used parallelism to enhance the computation performance and to improve the quality of the solutions. The authors proposed a two-phase parallelization algorithm that involves the fitness evaluation parallelization and the genetic operation parallelization. Their experimental result clearly depicts the fact that this method is an improvement of the genetic algorithm for pairwise test suite generation. In the Powerful Genetic Algorithm for Traveling Salesman Problem [3], they added edge swapping with a local search procedure to filter better combinations of building blocks of parent solutions for producing best offspring. We know that the performance of the genetic algorithm mainly depends on the type of genetic operators; selection, crossover, mutation and replacement are used. In the Study of Various Mutation Operators in Genetic Algorithms [4], the authors mainly discussed about various mutation methods that can be implemented in a genetic algorithm to solve the problem that involves large population sizes. The authors discussed about nearly nine mutation techniques starting from insert mutation to creep mutation. In a Study of Various Crossover Operators in Genetic Algorithm [5], the authors discussed the different crossover operators that are used to solve large population size optimization problems using a genetic algorithm. The authors mainly discussed the eight types of crossover techniques beginning from single point crossover to cycle crossover. To analyze the performance of mutation operators applied to solve the Travelling Salesman Problem [6], a comparative analysis of different mutation operators was presented, surrounded by a discussion that justifies the relevance of genetic operators chosen to solve the travelling salesman problem. A powerful machine with best performing processors is needed to make a comparison of modern CPUs and GPUs running the Genetic Algorithm under the Knapsack Benchmark as done by Jiri Jaros and team from Brno

University [7]. The authors introduced an optimized multicore CPU implementation of the genetic algorithm and compared its performance with a fine-tuned GPU version. Their goal was to show the true performance relation between modern CPUs and GPUs and eradicate some of the myths surrounding GPU performance. In the Travelling Salesman Problem using Genetic Algorithm [9], the authors mainly compared the efficiency of the new crossover operator with some existing crossover operators. In Genetic Algorithms and Parallel Processing in Maximum-Likelihood Phylogeny Inference [10], the authors investigated the usefulness of parallel genetic algorithm for phylogenetic inference under the maximum likelihood optimality criterion. The authors explored various ways of optimizing and tuning the parameters of the genetic algorithm. In their analysis, the authors could not find the best-known solution using the genetic algorithm approach before terminating each run. The authors said that even if the process speeds up by the addition of processors resulting in a decrease of problems of size variation, they still have issues with processors when they duplicate each other's calculations resulting in redundancy. As communication among processors if often slow relative to computational time, this transmission of duplicate data may significantly slow the process. Researchers from University of Bucharest researched about a distributed implementation using Apache Spark of a genetic algorithm applied to test data generation [12]. The authors mainly used the parallelized genetic algorithm test data generation for executable programs. GenRBFNSpark: A first implementation in Spark of a genetic algorithm to RBFN design is mentioned in [13], where Spark is discussed, which is a powerful cluster computing platform based on Map-Reduce. Due to dependencies between data that involves a multi-pass computing mode, there are no implementations of artificial networks models in well-known libraries of machine learning. In their paper the authors tried to develop Radical Basis Function Networks and GenRBFNSpark by implementing Spark with the genetic

method. The authors ran the algorithm with different configurations, and at the end they demonstrated that GenRBFNSpark achieves a remarkable speed up with respect to the local version.

2. APPROACH

2.1. Genetic Algorithm

Genetic Algorithm is a process for solving both forced and free optimization problems and is centered on natural selection, the process that pushes biological evolution. The genetic algorithm repeatedly modifies a population of individual solutions [21]. Genetic algorithms were developed to mimic some of the processes observed in natural evolution, a process that functions on chromosomes. The genetic algorithm method differs from other search methods in that it searches among a population of points and work with a coding of parameters, rather than the parameter values themselves. The old methods of genetic algorithm used gradient information, but the evolution scheme is probabilistic [24]. Due to these characteristics, genetic algorithms are being used as general-purpose optimization algorithms. The working principles of a genetic algorithm is as follows:

- Randomly initialize the population
- Repeat
- Find objective function value
- Find fitness function value
- Generate next population using operators
- Reproduction
- Crossover
- Mutation
- Until convergence

The flow chart of genetic algorithm is shown in Fig 4. The main step involved in a genetic algorithm is generating the population of solutions, finding the objective function and

fitness function, and the application of the genetic operators. Coding of variables is the important characteristic of a genetic algorithm. If the problem has more than one variable, a multivariable coding is constructed by concatenating as many single-variable coding as the number of variables in the problem. GAs process a number of solutions simultaneously. The first step is to generate a random population whose individuals represent feasible solutions. It starts from a wide range of solutions in the solution space, so the search is robust and unbiased. Next, individual members of the population are evaluated to find the objective function value. Then, the objective function is mapped onto a fitness function that computes a fitness value for each member of the population. The fitness function is used to allocate reproductive trails to the individuals in the population and thus acts as some measure of goodness to be maximized. Thus, individuals with a higher fitness value will have a higher probability of being selected as candidates for further examination. A new population is generated from the current population by performing three distinct genetic operators, namely reproduction, crossover and mutation.

Reproduction is an operator that makes more copies of better individuals in a new population. Healthier individuals should be from the fittest individuals of the preceding population; this is attained by calculating the reproduction count associated with each individual and then removing or copying the individual depending on the values of the reproduction count. The reproduction count is computed based on the fitness value by various methods; the present work uses stochastic sampling without replacement, because it is both superior to other schemes and widely used [22].

The advantages of a genetic algorithm compared to a classical optimization algorithm [6] are it uses only the evaluation of the objective function regardless of its nature. In fact, we do not require any special property of the function to be optimized, which gives it more flexibility and a

wide range of applications. The work is done in parallel by working on several solutions at once instead of using a single iteration in a classical algorithm. The use of probabilistic alteration rules compared to deterministic algorithms where the evolution between two individuals is required by the structure and nature of the algorithm.

2.1.1. Crossover

In the crossover operation, a recombination process creates diverse individuals in the next generation by combining material from two individuals of the earlier generation. A crossover operator is used to recombine two chromosomes to get a better chromosome (solution). Crossover is usually performed with a probability called crossover probability to preserve some of the good chromosomes found previously [16]. Crossover is done at the gene (individual part of a chromosome) level by randomly selecting two chromosomes for the crossover operation. It is a process of taking more than one parent solution and producing a child solution from them. There are many types of crossover techniques.

- Single Point Crossover: Single point crossover in which a single crossover point on both parents' organism strings (chromosomes) is selected [16]. All data beyond that point in either organism string is swapped between the two-parent organisms in the resulting organisms are the children as shown in Figure 1.

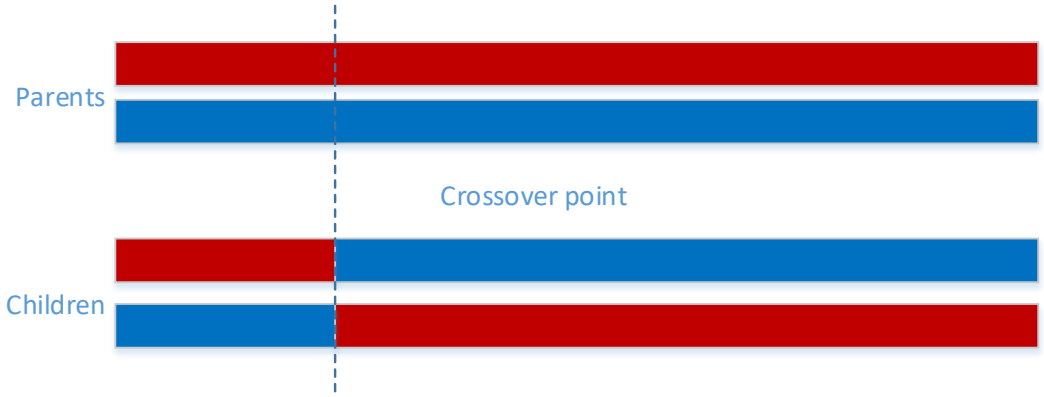

Figure 1. Single Point Crossover

- Double-Point Crossover: It calls for two points to be selected on the parent organism strings. Everything between the two points is swapped between the parent organisms, rendering two child organisms as shown in Figure 2.

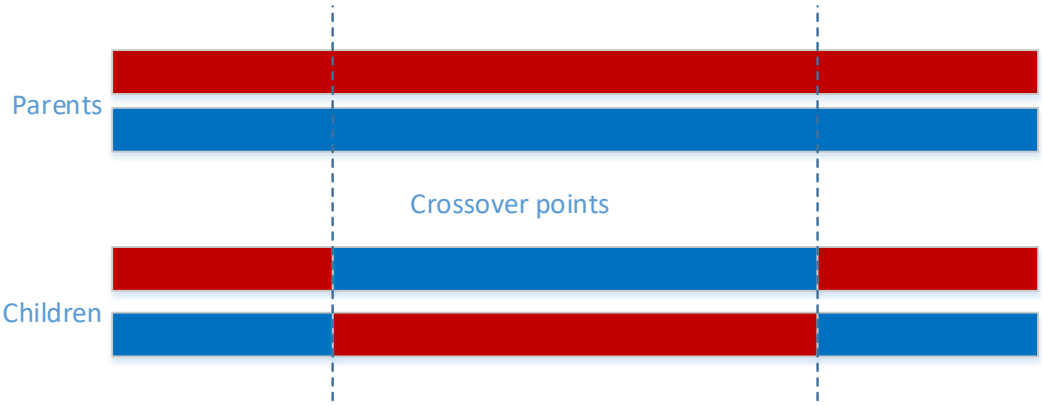

Figure 2. Double Point Crossover

- Uniform Crossover: Uniform and half uniform crossover is another operator. Uniform crossover follows a step that applies a fixed mixing ratio between both the parents. Unlike single and double point crossover, the uniform crossover enables the parent

chromosomes to contribute the gene level rather than the segment level as shown in Figure 3. Even if the crossover points can be randomly chosen as seen in the figure, suppose the mixing ratio is 0.5, the offspring has approximately half of the genes from the first parent and the next half from the second parent. It evaluates each bit in the parent strings for exchange with a probability of 0.5. Empirical evidence suggests that it is a more investigative approach to crossover than the traditional manipulative approach that maintains longer plans. This results in a more complete search of the design space with maintaining the exchange of good information [5].

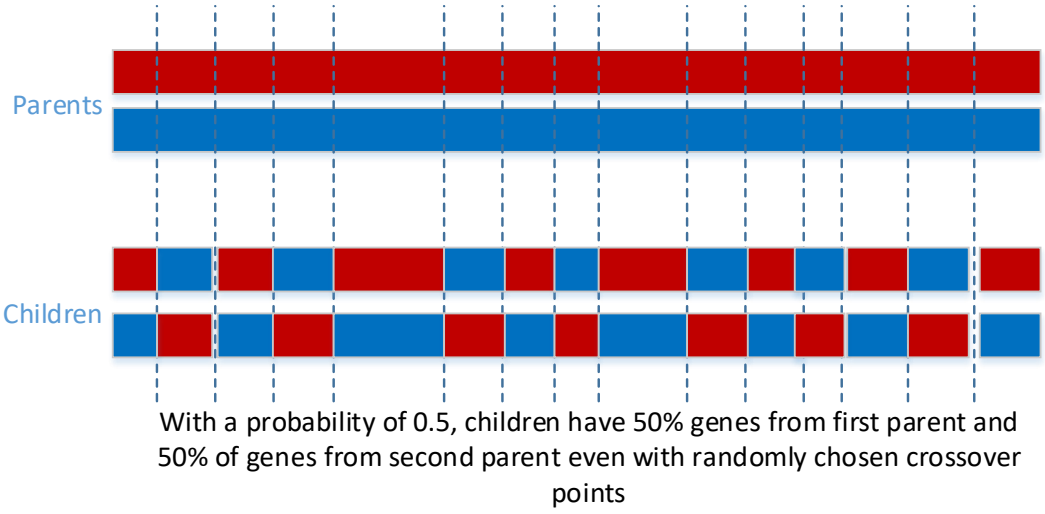

Figure 3. Uniform Crossover

- N-point Crossover: This was first implemented by De Jong in 1975. It has many cross over sites but the rule used is the same as in single point crossover. The significance of crossover sites is 2 in 2-point crossover [5]. Adding more crossover disrupts the building blocks of the chromosomes and that would affect the performance of the genetic algorithm. Here the author accepts both the head and the tail section of the chromosome.
- Three Parent Crossover: Here three parents are chosen arbitrarily. Each gene of the first parent is compared with the equivalent gene of the second parent [5]. If both genes are

similar, then the gene is used as the offspring or else the gene from the third parent is taken as the offspring.It is typically used in case of binary encoded chromosomes.

- Arithmetic Crossover: It is used in the case of real-value encoding. The arithmetic crossover operator linearly combines the two parent chromosomes [5]. Two chromosomes are chosen randomly for crossover to create two offspring which are a linear mixture of their parents.
- Partially mapped Crossover: This is the most often used crossover operator. It was proposed by Goldberg and Lingle for the Travelling Salesman Problem. Two chromosomes are combined, and two crossover sites are chosen randomly [5]. The portion of chromosomes between the two crossover points gives an equivalent selection that undergoes the crossover process through position-by-position exchange operations. This crossover method tends to respect the absolute positions.
- Order Crossover: It was proposed by Davis and also used for chromosomes with permutation encoding. The process starts in a way similar to partially mapped crossover by choosing two crossover points [5]. But in its place of using point-by-point exchanges as in the case of partially mapped crossover, order crossover applies a sliding motion to fill up the left out spaces by sending the mapped positions. It copies the portion of permutation elements between the crossover points from the cut string directly to the offspring, inserting them in the same absolute position.

- Cycle Crossover: If there are chromosomes with permutation encoding, this crossover is used. During recombination in cyclic crossover there is a limitation that each gene either comes from the one parent or the other [5]. The fundamental model at the back-cycle crossover is that each gene either comes from the one parent or the other. To construct a cycle of genes from parent1 we must start with the first gene of parent1. Then look at the gene at the equal position in parent 2 and go to the position with the same gene in parent 1. Insert this gene to the cycle in the first child on the positions they have in the first parent and the remaining genes of first child come from the second parent along with their position. Produce next cycle from parent 2.

2.1.2. Mutation

Mutation is a genetic operator used to conserve the genetic diversity from one generation of a population of chromosomes to the next generation of a population of genetic algorithm chromosomes. It is similar to biological mutation. It alters one or more gene values in a chromosome from its initial state [4]. There are different types of mutation techniques:

- Insert Mutation: It is used in Permutation encoding. First, pick two gene values at random [4]. Then move the second gene to follow the first, shifting the rest along to accommodate. This conserves most of the symmetry and the adjacency information.
- Inversion Mutation: This is used for chromosomes with permutation encoding. To do inversion, pick two genes randomly and then invert the substring between them. It preserves most adjacency information and only breaks two links but it leads to the disruption of order information [4].

- Scramble Mutation: It is also used with permutation encoded chromosomes. In this mutation, one must pick a subset of genes at random and then randomly rearrange the genes in those positions. Subset does not have to be contiguous [4].
- Swap Mutation: It is also in permutation encoding. The method is, select two genes at random and swap their positions. It preserves most of the adjacency information, but the links broken disrupts the order more [4].
- Flip Mutation: According to a generated mutation chromosome, flipping of a bit involves altering 0 to 1 and 1 to 0. A parent is studied, and a mutation chromosome is randomly produced. For a 1 in mutation chromosome, the corresponding bit in the parent chromosome is flipped and the child chromosome is produced. It is commonly used in binary encoding [4].
- Interchanging Mutation: Two random positions of the string are chosen and the bits corresponding to those positions are interchanged [4].
- Reversing Mutation: This is applied for binary encoded chromosomes. Here they choose a random spot and the bits next to that spot are reversed and the child chromosome is generated [4].
- Uniform Mutation: The value of a chosen gene with a uniform random value selected between the user specified upper and lower bound for that gene will be changed by the mutation operator. It is used in case of real and integer representation [4].
- Creep Mutation: In creep mutation, a random gene is chosen, and its value is modified with a random value between lower and upper bound. It is used in the case of real representation [4].

For solving the Traveling salesman problem using a genetic algorithm, the following basic steps can be used:

- Encoding: Permutation encoding is used to solve traveling salesman problem. We symbolize cities with an integer value, and later we initialize the population.
- Distance matrix: Distance matrix is an **NxN** matrix of point to point distance.
- Selection based on fitness function: It will be the total cost of the tour represented by each chromosome. The lesser the sum, the fitter the solution represented by that chromosome. Figure 4 shows the flow chart of the genetic algorithm steps.

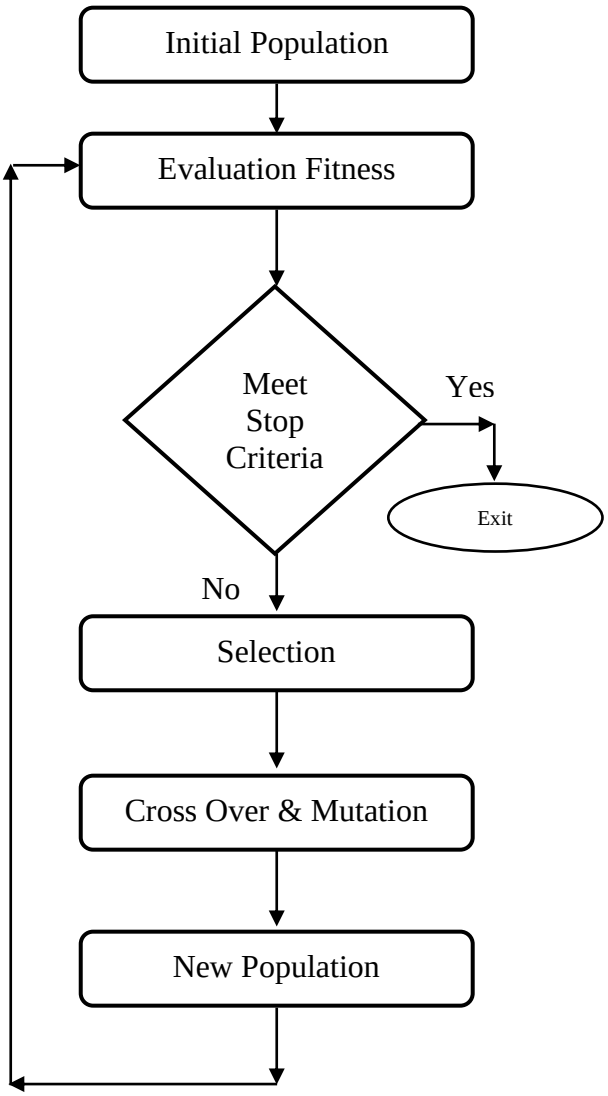

Figure 4. Flow Chart of Genetic Algorithm

2.2. Spark

Spark is an open source cluster computing system whose aim is to fasten data analytics [15]. Spark is fast to run as well as write. Spark offers high-level APIs in Java, Scala, Python and R, and to support general execution graphs, an optimized engine is used. It helps a find set of higher-level tools including Spark SQL for SQL and structured data processing, GraphX for graph processing, and Spark streaming, MLib for machine learning.

At a high level, every Spark application contains a driver program that runs the user's main function and executes several parallel operations on a cluster. The main concept spark provides is a *resilient distributed database (RDD), which* is a collection of elements assigned across the nodes of the cluster that can be operated on in parallel [19]. RDDs are created by starting with a file in the Hadoop file system or any other Hadoop-supported file system, or an existing Scala collection in the driver program, and transforming it. Users may also ask Spark to continue an RDD in memory, allowing it to be reused efficiently across parallel operations. Finally, RDDs automatically recover from node failures. The second concept in Spark is shared variables that can be used in parallel operations. By default, when Spark runs a function in parallel as a set of tasks on different nodes, it sends a copy of each variable used in the function to each task. Sometimes across tasks and driver program, a variable need to be shared. There are two types of shared variables in spark: broadcast variables, which can be used to cache a value in memory on all nodes, and accumulators, which are variables that are only added to, like counters and sums.

3. ALGORITHM SETUP

We are investigating the performance of the parallel Spark implementation of the Traveling Salesman Problem as outlined in [17]. The program finds the shortest distance while travelling many cities while starting and ending in the same city. To do this, it uses a Genetic algorithm which generates random routes between cities, finds the best ones and then recombines and mutates the best ones to find even better ones. This is done in several iterations.

The algorithm runs in parallel when we split the work between threads, after they will start to do a lot of double work. Therefore, it works in a fork-solve-recombine fashion where after each iteration a new work package is constructed from the best routes so far. This process continues either until the max number of iterations is reached or there is no change in distance anymore between iterations.

The steps that were followed was, initiate an initial population of chromosomes P. Assess the fitness of each chromosomes. Through proportional selection, choose P/2 parents from the current population. To create offspring using crossover operator, randomly select two parents. For minor changes apply the mutation operator. Repeat random selection, cross over and mutation until all parents are chosen and mated. Replace old population of chromosomes with new population. Evaluate the fitness of each chromosome in the newly created population. Terminate if the number of generations meets some upper bound, otherwise again choose P/2 parents from the current population via proportional selection and repeat the steps.

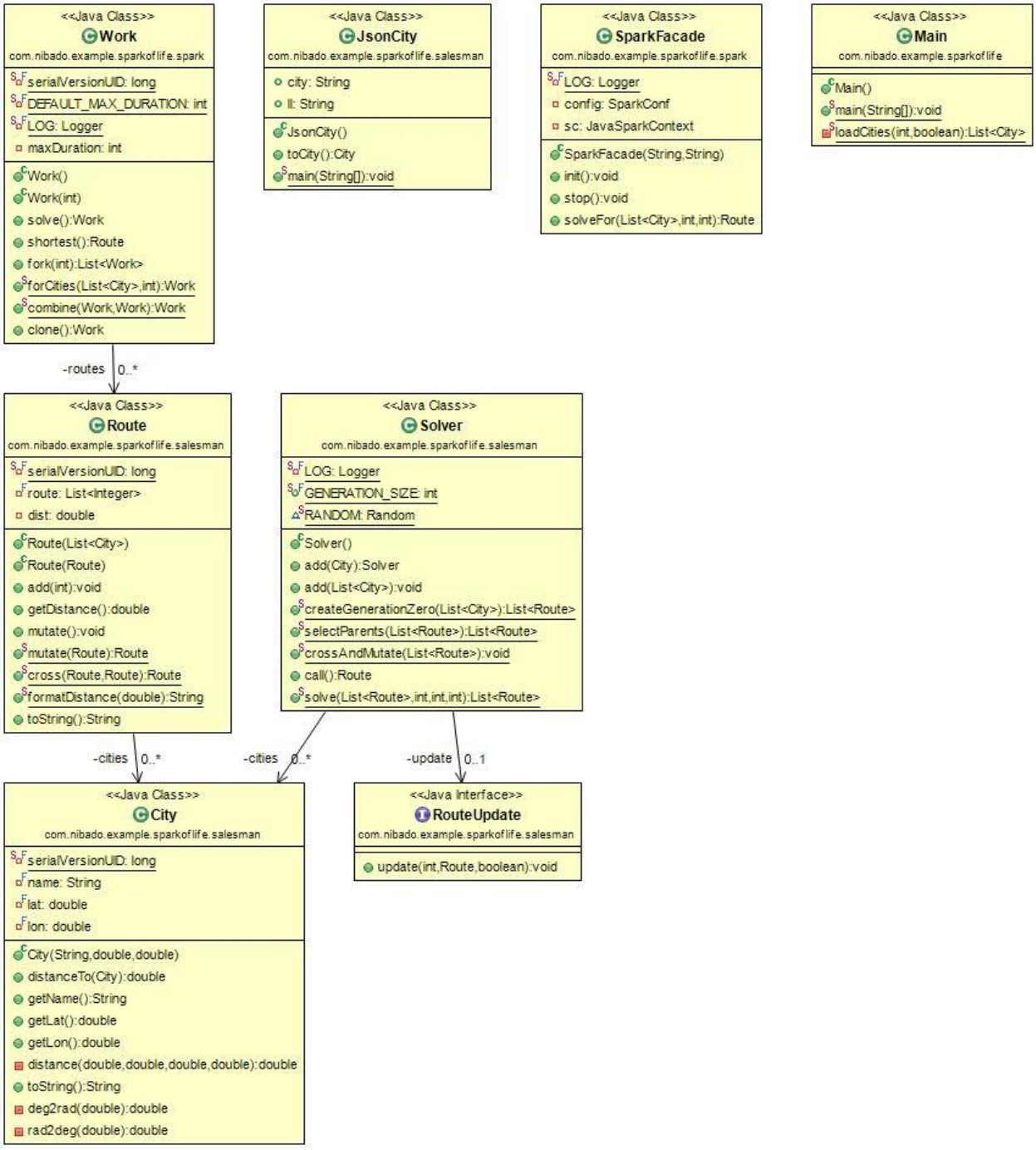

Figure 5. Class Diagram of the Process

The spark-of-life project [17] has been used. The following steps are used in the project, there is a Salesman folder in which classes named City and Route are listed that give the list of cities and size of route, respectively. This is used as the input for creating the population. A class named Solver does the actual genetic algorithm steps. It creates the generation size, finds the list of cities, it sets the route update, creates a random function to generate a random population and new array list of cities, later they make use of the genetic algorithm operators like Crossover and Mutation. There is another folder called Spark that has two classes named SparkFacade and Work. Both have shared the task to make the process easier. SparkFacade access the cities and routes and they help the solver to do initialization followed by iterations. Before they begin the iteration, each set of population should be sent to different processors parallel. The Work Class represents the unit of work that is executed by the Spark Executors. Work Class directs the population into different processors to do parallelization. For this purpose, the Work class help the solver and continue the execution of the population and begin the iteration process to get the final optimum result as the output. The end result would give the best solution, execution time of each processors and SpeedUp.

A class named City was created with many variables, ***name***, ***lat1***, ***lat2*** and ***lon1***, ***lon2*** for name, latitude and longitude. This calculates the number of cities and distance between each cities. ***Theta*** value is calculated by finding the difference between ***lon1*** and ***lon2***.

Distance = sin(deg2rad(lat1))*sin(deg2rad(lat2)+cos(deg2rad(lat2))*cos(deg2rad(theta)) (1)

- ***Deg2rad***:-Converts degrees to radians. The input ***deg*** must be a scalar, vector, or ***N***-dimensional array of double or single floating point values. Deg may be complex in which case the real and imaginary components are converted separately. The output rad is the same size and shape as ***deg*** with degrees converted to radians using the conversion constant ***pi/180***.

- ***Rad2deg***:- This is to convert radians to degrees. The conversion constant is ***180/pi***. What was done is to parallelize the task so that we can utilize the efficiency of multiple worker threads. By approximating ***Pi***, a code uses many workers to do a CPU related task. To do fork-solve-recombine, the RDD is parallelized first, then the CPU bound calculation is done in the map and then reduce the values to print the approximation of ***Pi***.

Figure 5 shows the Class diagram of the process. In the Solve method, the algorithm runs the parallelize-map-reduce code inside the ***for*** loop. A work class is created, it has the list of routes that can be a task for the Worker to solve. The Solve method is called in the map function and it runs until it reaches a maximum time, repeated value, or maximum number of generations. The loop ends if there are no improvements in the distance or when the maximum number of iterations are done.

In the class Route, the city list is collected in which they put the number of cities as the route size. The distance should always be greater than 1. The used ***for*** loop is to show the distance of the route. The crossover and mutation are done by dividing the route into half and appending it with the other one. Mutation is done randomly on any index values and it returns a new route. The StringBuilder method is used to append the string representation of the argument.

In the Solver class, the generation size, final list of cities, and route update is set. Then a random function is created for generating a random population. A new array list of cities is created. The ***for*** loop is used in all creations. Then, the parent routes are selected. They are sorted and compared by distance. Then, the final route is selected and added in a new array list. An ***if*** statement is used to check if the distance of the routes is same or not. If they are different, the ***crossover*** is done between them by obtaining a child route. In the child route, ***mutation*** is done by making small change in index. Then, they create the new population.

The next step is the update for the next generation. The new generation set is taken and considered as parents for the next generation and then the process is continued. The process continues until it reaches an optimal solution that gives the best solution. As they are run in different cores, we kept track of the execution time of each of the cores. The execution time is calculated by computing the current time and start time in milliseconds.

$$T = \text{current time} - \text{start time} \quad (2)$$

Then, the execution speedup is calculated by dividing the time taken for running the algorithm in one core and the time taken for running the algorithm in different numbers of cores.

We can run the program from IDE by running the com.nibado.example.sparkoflife.Main.class. It has sensible defaults for the different command line options [17].

We can run it using the command line "**–c 20 –s –I 10 –d 4 –m local[2] –a MyApp"** This will run the example with 20 cities, shuffled, in 10 iterations max with an iteration duration of 4 seconds max. It runs in Spark in local mode with 2 workers and label the Spark application as MyApp. But we do not shuffle the cities in this case.

Table 1. Command Line Options

Option	Type	Default	Description
-c	int	100	Number of cities to limit to
-i	in	4	Number of iterations
-d	int	8	Iteration max duration in seconds
-s		False	Shuffle city input
-m	String	local[4]	Spark Master
-a	String	SparkOfLife	Application name

Here in this GA, the more time you let it spend the closer it will get to an optimal solution, but there is no guarantee that the found solution is optimal [17].

4. RESULTS

After sending the population to the different cores for execution, we obtained the optimal solution which is the shortest distance. We used 70 iterations for each of the experiments. The results of the experiments are the shortest distance, execution time for each of the cores, and the calculated speedup.

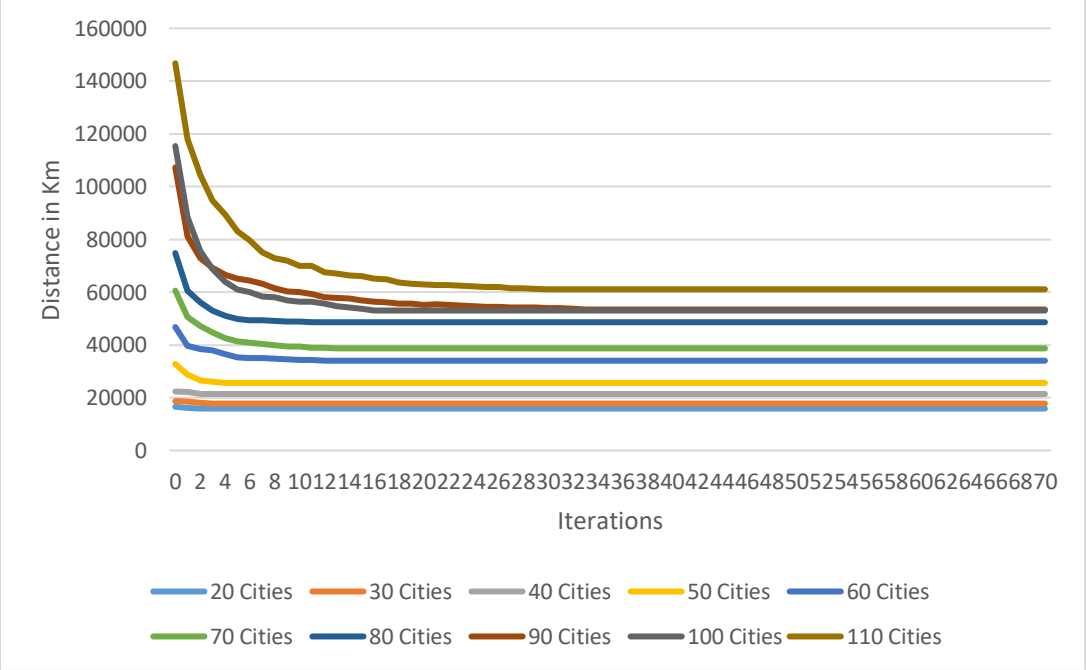

Figure 6. 1-core machine

Figure 6 shows the result when the program was run on 1 core with 70 iterations. For 20 cities the optimum distance obtained was 15285.75 Km. For 30 Cities, the initial distance obtained in 0^{th} iteration was 18745.65 Km and at the end, the final optimized distance was 17793.67 Km. When run with 40 cities, the distance was 22368.18 Km in the 0^{th} iteration and 21437.1 Km in the 70^{th} iteration. For 50 Cities, the initial distance was 32783.73 Km and the final optimized distance was 25621.57 Km. 46785.49 Km was the distance in the 0^{th} iteration and 34042.24 Km in the 70^{th} iteration for 60 cities. For 70 Cities, 60594.62 Km was the distance obtained in 0^{th} iteration and 38712.94 Km for the 70^{th} iteration. The distance obtained when the

program was run with 80 cities was 74828.97 Km and 48615.4 Km as the initial distance and the final optimized distance obtained. For 90 cities, in the 0^{th} iteration, the distance obtained was 107283.6 Km and for 70^{th} iteration, the final distance obtained was 53436.76 Km. For 100 cities, the distance in the 0^{th} iteration was 115387 Km and for the 70^{th} iteration, 53066.59 Km was the result. When the program was run with 110 cities, the initial value obtained was 146729.3 Km and the final optimized value was 61098.32 Km.

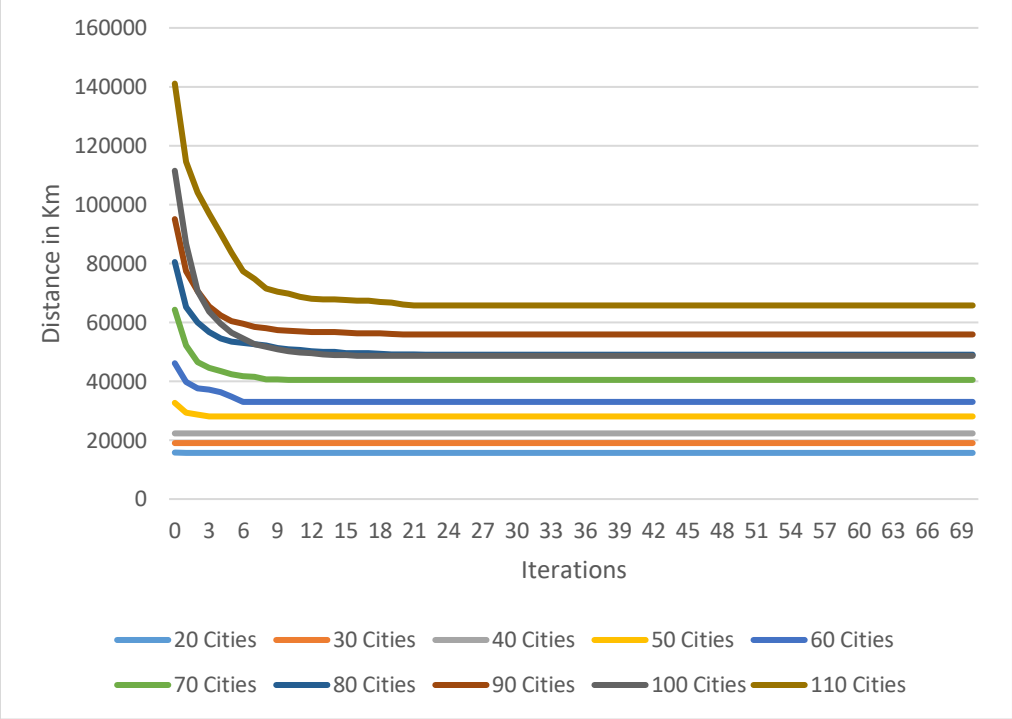

Figure 7. 2-core machine

Figure 7 shows the result when the program was run on 2 cores with 70 iterations. For 20 cities, the initial distance was 15798.85 Km and the optimum distance obtained was 15684.06 Km. For 30 Cities, the initial distance obtained in 0^{th} iteration was 19021.73 Km and at the end, the final optimized distance was also the same. When run with 40 cities, the distance was 22302.03 Km in the 0^{th} iteration as well as 70^{th} iteration. For 50 Cities, the initial distance was 32678.03 Km and the final optimized distance was 28060.47 Km. 46785.49 Km was the distance

in the 0th iteration and 32996.03 Km in the 70th iteration for 60 cities. For 70 Cities, 64305.55 Km was the distance obtained in 0th iteration and 40468.17 Km for the 70th iteration. The distance obtained when the program was run with 80 cities was 80488.95 Km and 49028.83 Km as the initial distance and the final optimized distance obtained respectively. For 90 cities, in the 0th iteration, the distance obtained was 95072.45 Km and for 70th iteration, the final distance obtained was 55900.47 Km. For 100 cities, the distance in the 0th iteration was 111485 Km and for the 70th iteration, 48644.79 Km was the result. When the program was run with 110 cities, the initial value obtained was 141086.2 Km and the final optimized value was 65746.72 Km.

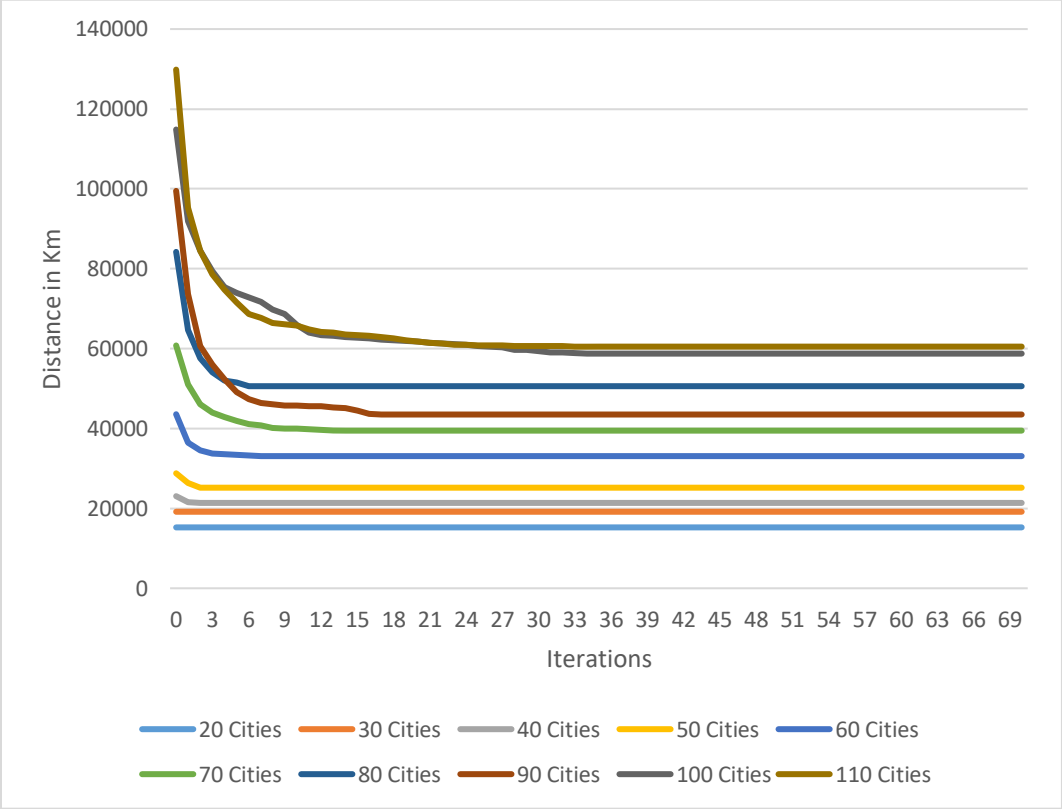

Figure 8. 3-core machine

Figure 8 shows the result when the program was run on 3 cores with 70 iterations. For 20 cities the optimum distance obtained was 15243.56 Km. For 30 Cities, the initial distance obtained in 0th iteration was 19160.22 Km and at the end, the final optimized distance was

19160.22 Km. When run with 40 cities, the distance was 23057.85 Km in the 0^{th} iteration and 21366.98 Km in the 70^{th} iteration. For 50 Cities, the initial distance was 28787 Km and the final optimized distance was 25182.21 Km. 46785.49 Km was the distance in the 0^{th} iteration and 39469.2 Km in the 70^{th} iteration for 60 cities. For 70 Cities, 60786.76 Km was the distance obtained in 0^{th} iteration and 38712.94 Km for the 70^{th} iteration. The distance obtained when the program was run with 80 cities was 84219.67 Km and 50592.18 Km as the initial distance and the final optimized distance obtained. For 90 cities, in the 0^{th} iteration, the distance obtained was 99489.08 Km and for 70^{th} iteration, the final distance obtained was 43498.64 Km. For 100 cities, the distance in the 0^{th} iteration was 114855.3 Km and for the 70^{th} iteration, 58751.5 Km was the result. When the program was run with 110 cities, the initial value obtained was 129853 Km and the final optimized value was 60499.12 Km.

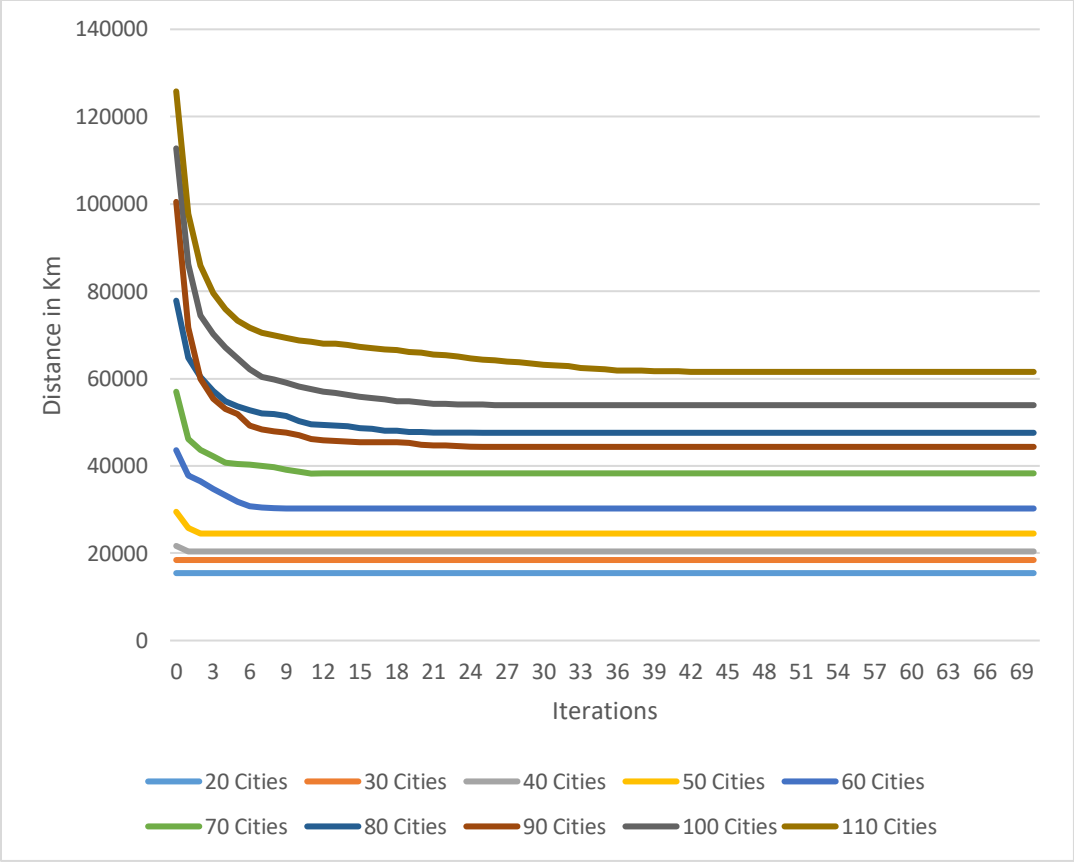

Figure 9. 4-core machine

Figure 9 shows the result when the program was run on 4 cores with 70 iterations. For 20 cities the optimum distance obtained was 15437.06 Km. For 30 Cities, the initial distance obtained in 0^{th} iteration was 18441.62 Km and at the end, the final optimized distance was 18441.62 Km. When run with 40 cities, the distance was 21668 Km in the 0^{th} iteration and 20399.14 Km in the 70^{th} iteration. For 50 Cities, the initial distance was 29487.22 Km and the final optimized distance was 24505.71 Km. 43595.48 Km was the distance in the 0^{th} iteration and 30241.51 Km in the 70^{th} iteration for 60 cities. For 70 Cities, 56988.09 Km was the distance obtained in 0^{th} iteration and 38287.77 Km for the 70^{th} iteration. The distance obtained when the program was run with 80 cities was 77849.49 Km and 47583.3 Km as the initial distance and the final optimized distance obtained respectively. For 90 cities, in the 0^{th} iteration, the distance

obtained was 100471.6 Km and for 70th iteration, the final distance obtained was 44357.03 Km. For 100 cities, the distance in the 0th iteration was 112731.2 Km and for the 70th iteration, 53907.01 Km was the result. When the program was run with 110 cities, the initial value obtained was 125787.4 Km and the final optimized value was 61532.21 Km.

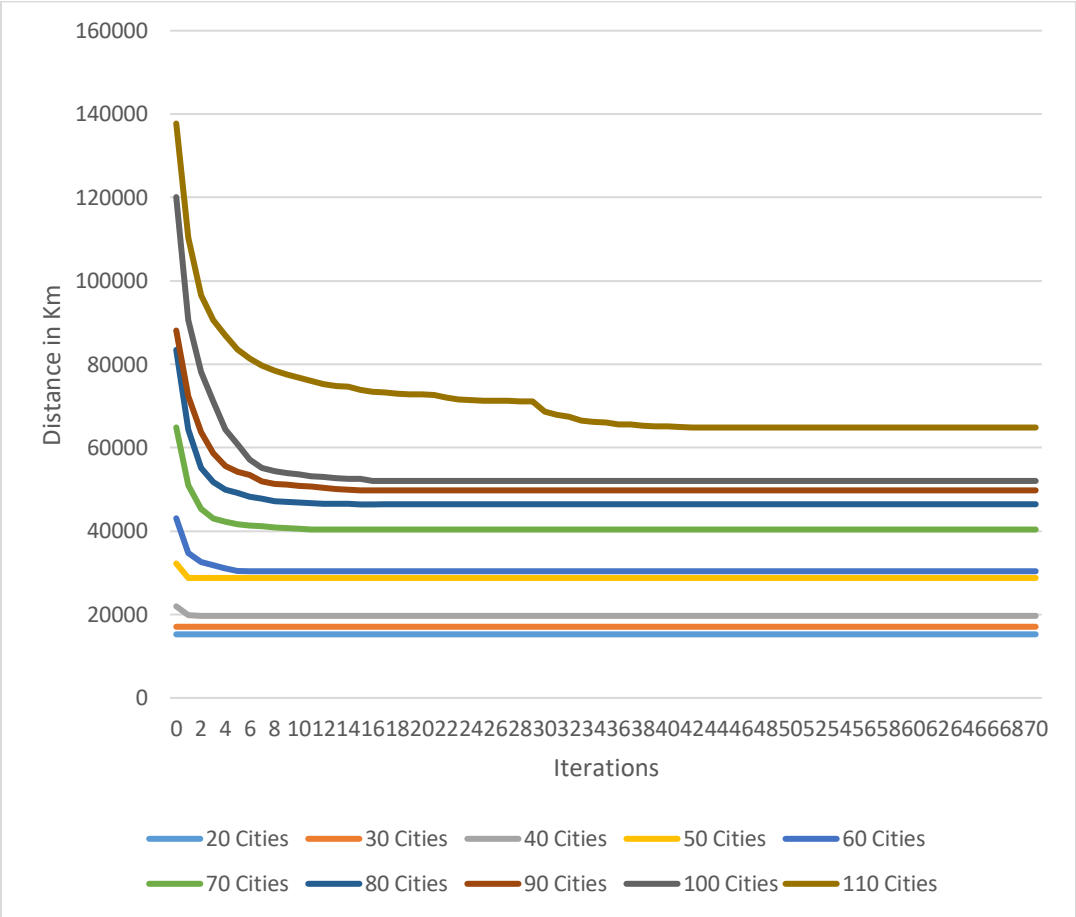

Figure 10. 5-core machine

Figure 10 shows the result when the program was run on 5 cores with 70 iterations. For 20 cities the optimum distance obtained was 15243.56 Km. For 30 Cities, the initial distance obtained in 0th iteration was 17058.89 Km and at the end, the final optimized distance was 17058.89 Km. When run with 40 cities, the distance was 21966.1 Km in the 0th iteration and 19667.46 Km in the 70th iteration. For 50 Cities, the initial distance was 32249.88 Km and the

final optimized distance was 28760.3 Km. 43072.59 Km was the distance in the 0th iteration and 30368.92 Km in the 70th iteration for 60 cities. For 70 Cities, 64863.6 Km was the distance obtained in 0th iteration and 40373.83 Km for the 70th iteration. The distance obtained when the program was run with 80 cities was 83459.52 Km and 46438.45 Km as the initial distance and the final optimized distance obtained respectively. For 90 cities, in the 0th iteration, the distance obtained was 88103.75 Km and for 70th iteration, the final distance obtained was 49762.59 Km. For 100 cities, the distance in the 0th iteration was 120105.3 Km and for the 70th iteration, 52025.7 Km was the result. When the program was run with 110 cities, the initial value obtained was 137744.5 Km and the final optimized value was 64835.04 Km.

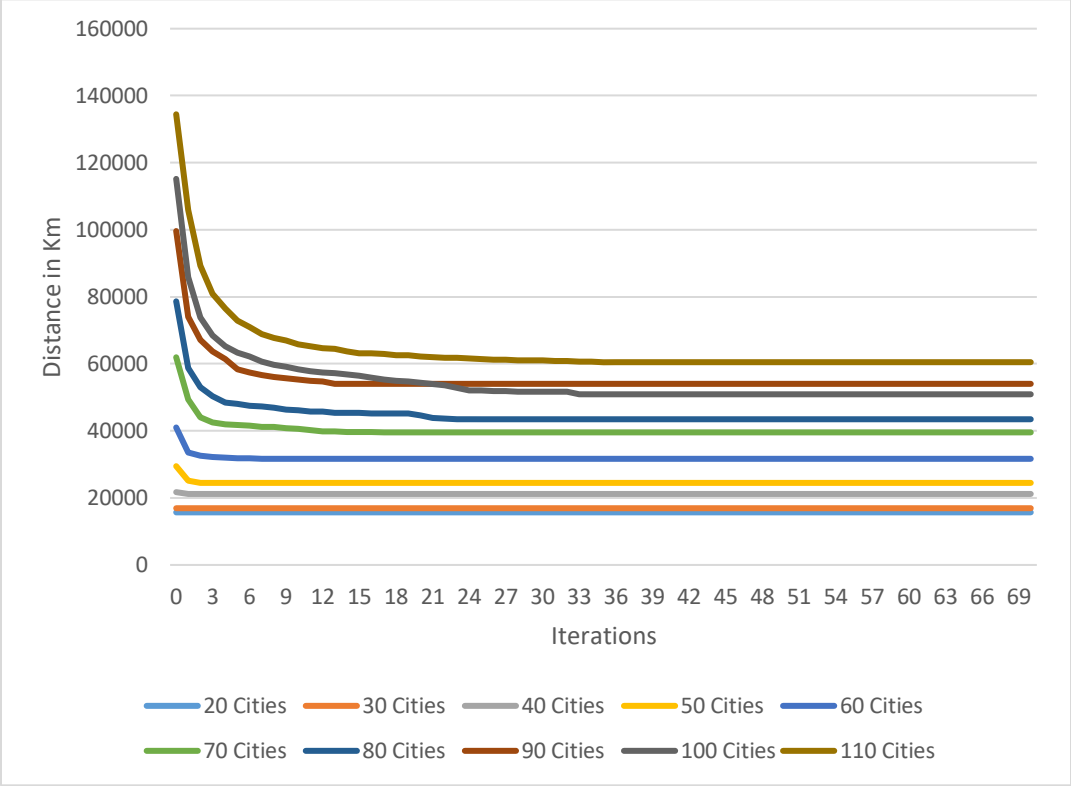

Figure 11. 6-core machine

Figure 11 shows the result when the program was run on 6 cores with 70 iterations. For 20 cities the optimum distance obtained was 15684.06 Km. For 30 Cities, the initial distance

obtained in 0^{th} iteration was 16916.49 Km and at the end, the final optimized distance was 16916.49 Km. When run with 40 cities, the distance was 21717.85 Km in the 0^{th} iteration and 21176.29 Km in the 70^{th} iteration. For 50 Cities, the initial distance was 29469.77 Km and the final optimized distance was 24476.94 Km. 41008.34 Km was the distance in the 0^{th} iteration and 31660.57 Km in the 70^{th} iteration for 60 cities. For 70 Cities, 61956.58 Km was the distance obtained in 0^{th} iteration and 39546.51 Km for the 70^{th} iteration. The distance obtained when the program was run with 80 cities was 78642.66 Km and 43463.88 Km as the initial distance and the final optimized distance obtained respectively. For 90 cities, in the 0^{th} iteration, the distance obtained was 99620.4 Km and for 70^{th} iteration, the final distance obtained was 54027.96 Km. For 100 cities, the distance in the 0^{th} iteration was 115178.1 Km and for the 70^{th} iteration, 50906.56 Km was the result. When the program was run with 110 cities, the initial value obtained was 134454.2 Km and the final optimized value was 60485.33 Km.

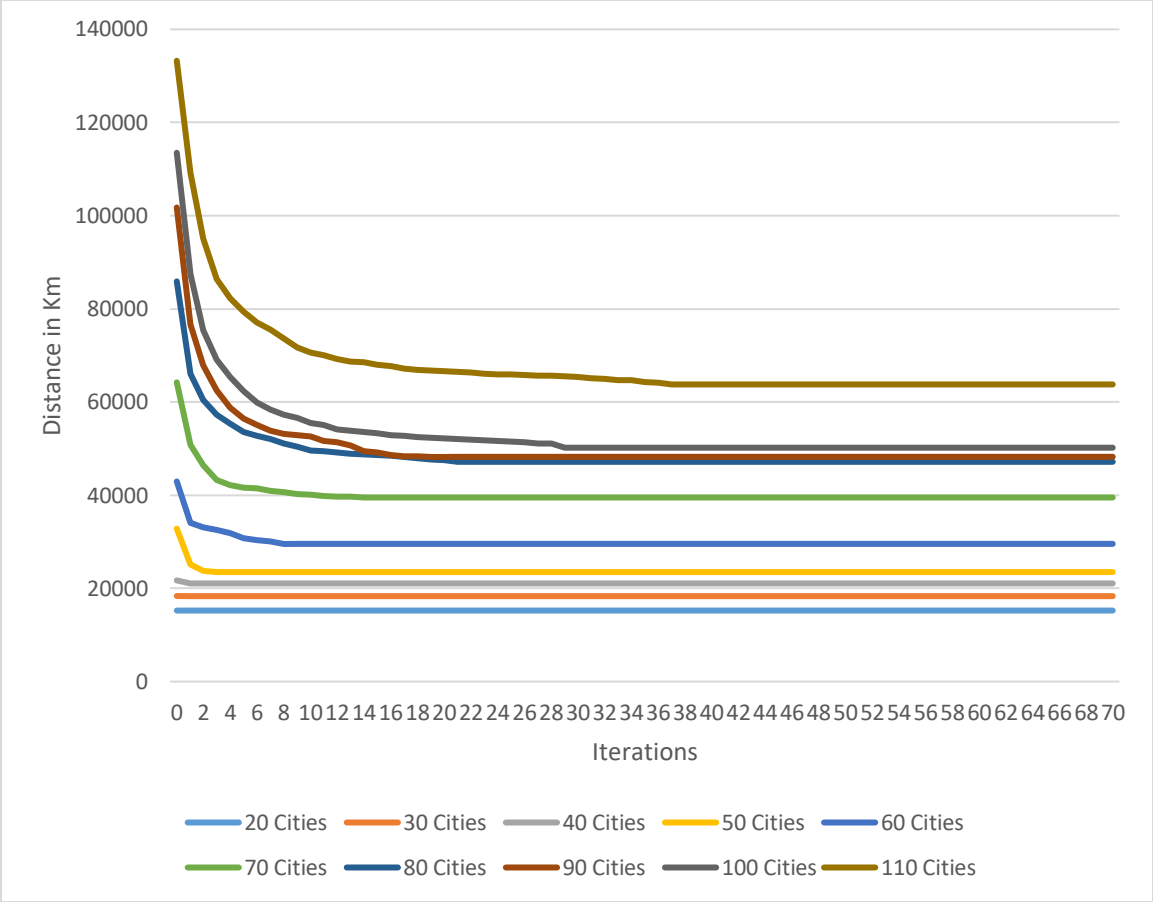

Figure 12. 7-core machine

Figure 12 shows the result when the program was run on 7 cores with 70 iterations. For 20 cities the optimum distance obtained was 15243.56 Km. For 30 Cities, the initial distance obtained in 0^{th} iteration was 18362.75 Km and at the end, the final optimized distance was 18337.75 Km. When run with 40 cities, the distance was 21719.42 Km in the 0^{th} iteration and 21060.81 Km in the 70^{th} iteration. For 50 Cities, the initial distance was 32826.89 Km and the final optimized distance was 23506.55 Km. 42960.95 Km was the distance in the 0^{th} iteration and 29560.79 Km in the 70^{th} iteration for 60 cities. For 70 Cities, 64233.2 Km was the distance obtained in the 0^{th} iteration and 39515.49 Km for the 70^{th} iteration. The distance obtained when the program was run with 80 cities was 85897.04 Km and 47171.54 Km as the initial distance and the final optimized distance obtained respectively. For 90 cities, in the 0^{th} iteration, the

distance obtained was 101760 Km and for 70th iteration, the final distance obtained was 48242.71 Km. For 100 cities, the distance in the 0th iteration was 113507.1 Km and for the 70th iteration, 50199.86 Km was the result. When the program was run with 110 cities, the initial value obtained was 133256.2 Km and the final optimized value was 63777 Km.

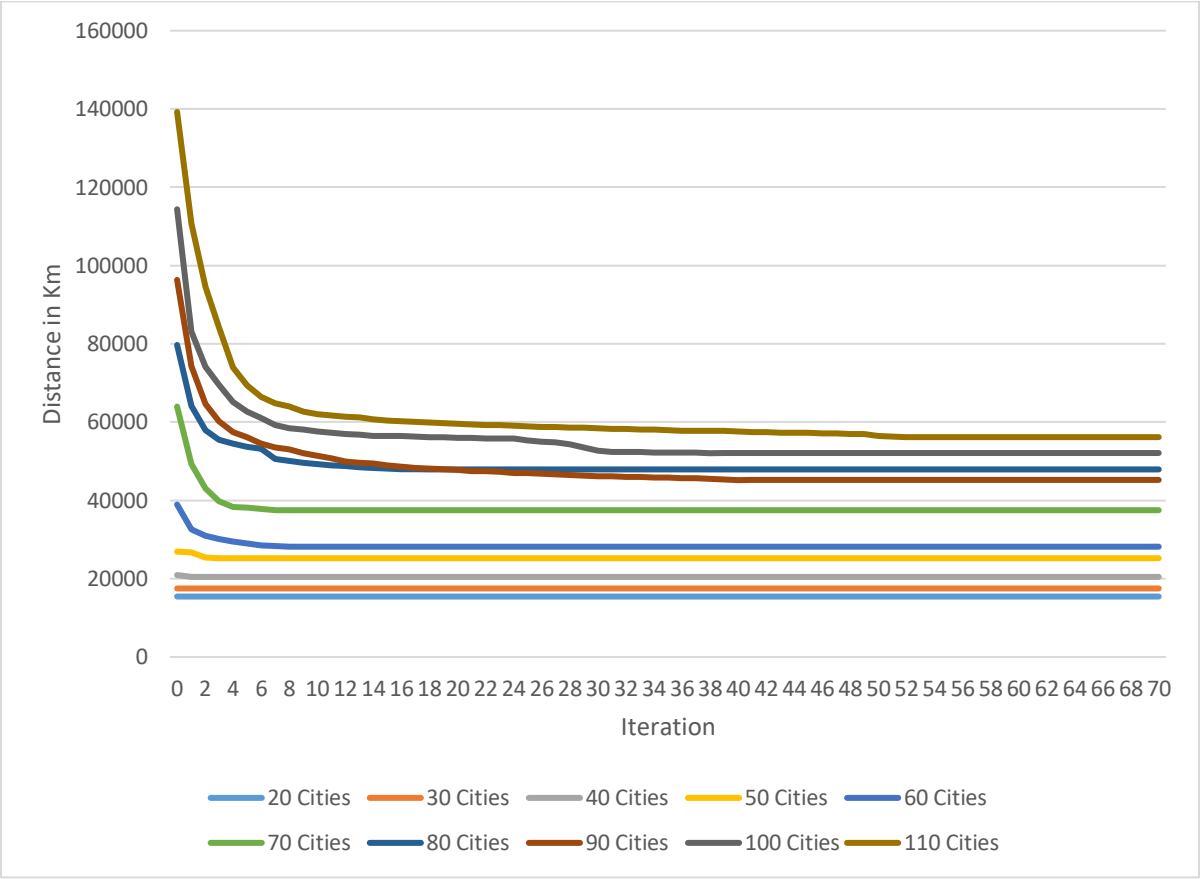

Figure 13. 8-core machine

Figure 13 shows the result when the program was run on 8 cores with 70 iterations. For 20 cities the optimum distance obtained was 15437.06 Km. For 30 Cities, the initial distance obtained in 0th iteration was 17497.76 Km and at the end, the final optimized distance was 17497.76 Km. When run with 40 cities, the distance was 20924.37 Km in the 0th iteration and 20457.59 Km in the 70th iteration. For 50 Cities, the initial distance was 26924.37 Km and the final optimized distance was 25229.49 Km. 38938.81 Km was the distance in the 0th iteration

and 28167.17 Km in the 70th iteration for 60 cities. For 70 Cities, 63962.2 Km was the distance obtained in 0th iteration and 37496.83 Km for the 70th iteration. The distance obtained when the program was run with 80 cities was 79712.04 Km and 47910.33 Km as the initial distance and the final optimized distance obtained respectively. For 90 cities, in the 0th iteration, the distance obtained was 96334.46 Km and for 70th iteration, the final distance obtained was 45233.52 Km. For 100 cities, the distance in the 0th iteration was 114380.4 Km and for the 70th iteration, 52101.65 Km was the result. When the program was run with 110 cities, the initial value obtained was 139235.1 Km and the final optimized value was 56157.38 Km.

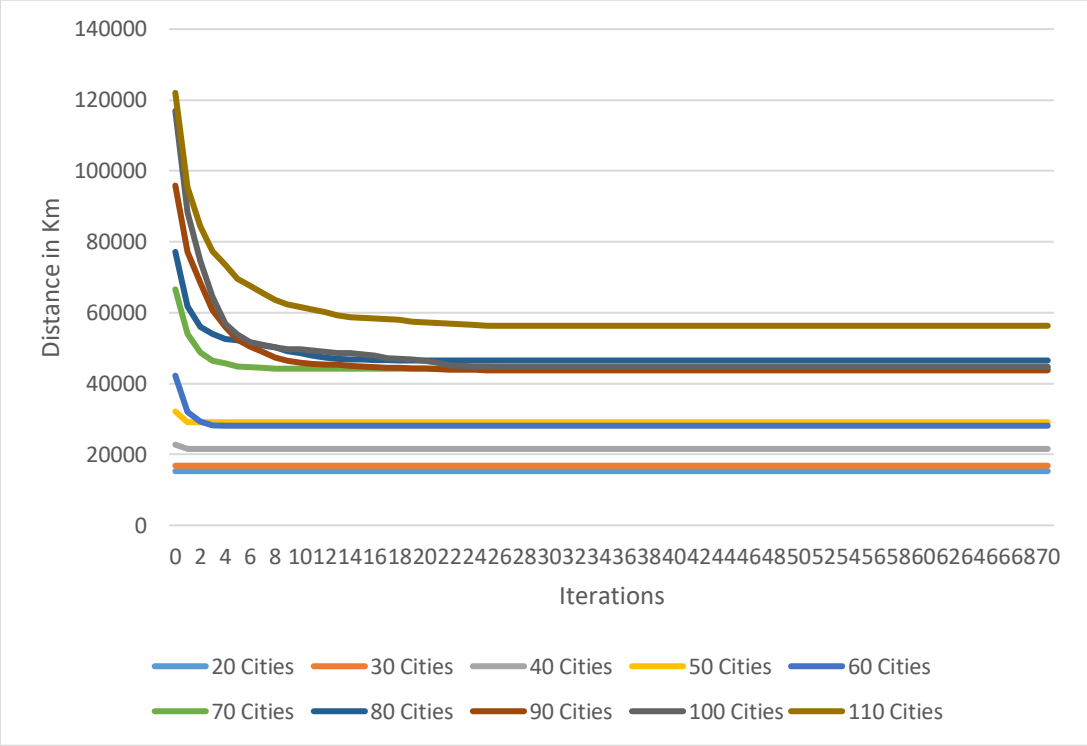

Figure 14. 9-core machine

Figure 14 shows the result when the program was run on 9 cores with 70 iterations. For 20 cities the optimum distance obtained was 15285.75 Km. For 30 Cities, the initial distance obtained in 0th iteration was 16820.22 Km and at the end, the final optimized distance was 16820.22 Km. When run with 40 cities, the distance was 22766.5 Km in the 0th iteration and

21555.95 Km in the 70th iteration. For 50 Cities, the initial distance was 32123.62 Km and the final optimized distance was 29081.49 Km. 42232.41 Km was the distance in the 0th iteration and 28106.16 Km in the 70th iteration for 60 cities. For 70 Cities, 66582.35 Km was the distance obtained in 0th iteration and 44194.25 Km for the 70th iteration. The distance obtained when the program was run with 80 cities was 77140.23 Km and 46500.35 Km as the initial distance and the final optimized distance obtained respectively. For 90 cities, in the 0th iteration, the distance obtained was 95853.69 Km and for 70th iteration, the final distance obtained was 43718.64 Km. For 100 cities, the distance in the 0th iteration was 116931.6 Km and for the 70th iteration, 44748.78 Km was the result. When the program was run with 110 cities, the initial value obtained was 122007.5 Km and the final optimized value was 56303.68 Km.

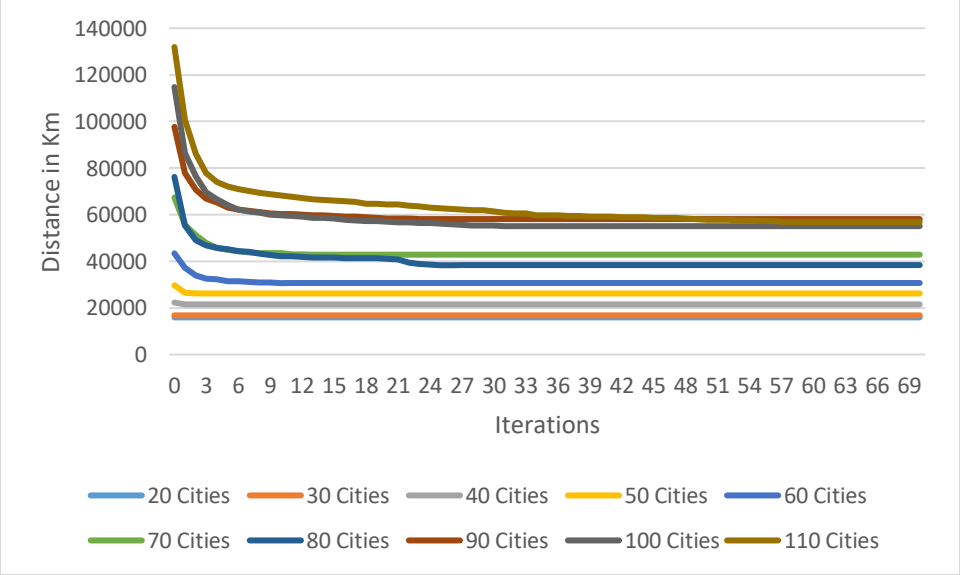

Figure 15. 10-core machine

Figure 15 shows the result when the program was run on 10 cores with 70 iterations. For 20 cities the optimum distance obtained was 15940.64 Km. For 30 Cities, the initial distance obtained in 0th iteration was 16796.97 Km and at the end, the final optimized distance was 16796.97 Km. When run with 40 cities, the distance was 22321.06 Km in the 0th iteration and

21500.2 Km in the 70th iteration. For 50 Cities, the initial distance was 29728.66 Km and the final optimized distance was 26206.03 Km. 43402.39 Km was the distance in the 0th iteration and 30696.95 Km in the 70th iteration for 60 cities. For 70 Cities, 67400.69 Km was the distance obtained in 0th iteration and 42848.72 Km for the 70th iteration. The distance obtained when the program was run with 80 cities was 76214.42 Km and 38382.86 Km as the initial distance and the final optimized distance obtained respectively. For 90 cities, in the 0th iteration, the distance obtained was 97703.96 Km and for 70th iteration, the final distance obtained was 58117.92 Km. For 100 cities, the distance in the 0th iteration was 114779.4 Km and for the 70th iteration, 55076.18 Km was the result. When the program was run with 110 cities, the initial value obtained was 131947.1 Km and the final optimized value was 56879.42 Km.

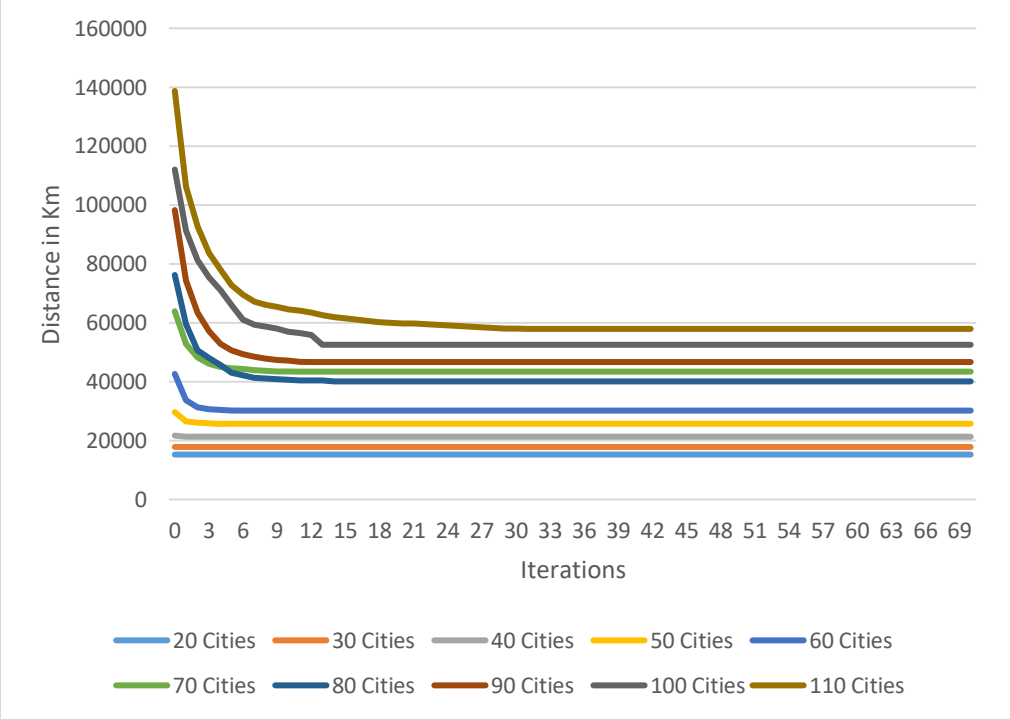

Figure 16. 11-core machine

Figure 16 shows the result when the program was run on 11 cores with 70 iterations. For 20 cities the optimum distance obtained was 15243.56 Km. For 30 Cities, the initial distance

obtained in 0^{th} iteration was 17819.8 Km and at the end, the final optimized distance was 17819.8 Km. When run with 40 cities, the distance was 21683.92 Km in the 0^{th} iteration and 21312.13 Km in the 70^{th} iteration. For 50 Cities, the initial distance was 29646.72 Km and the final optimized distance was 25711.21 Km. 42634.72 Km was the distance in the 0^{th} iteration and 30176.97 Km in the 70^{th} iteration for 60 cities. For 70 Cities, 63912.72 Km was the distance obtained in 0^{th} iteration and 43403.25 Km for the 70^{th} iteration. The distance obtained when the program was run with 80 cities was 76218.83 Km and 40082.96 Km as the initial distance and the final optimized distance obtained respectively. For 90 cities, in the 0^{th} iteration, the distance obtained was 98256.86 Km and for 70^{th} iteration, the final distance obtained was 46664.44 Km. For 100 cities, the distance in the 0^{th} iteration was 112024.6 Km and for the 70^{th} iteration, 52519.73 Km was the result. When the program was run with 110 cities, the initial value obtained was 138715.2 Km and the final optimized value was 57919.82 Km.

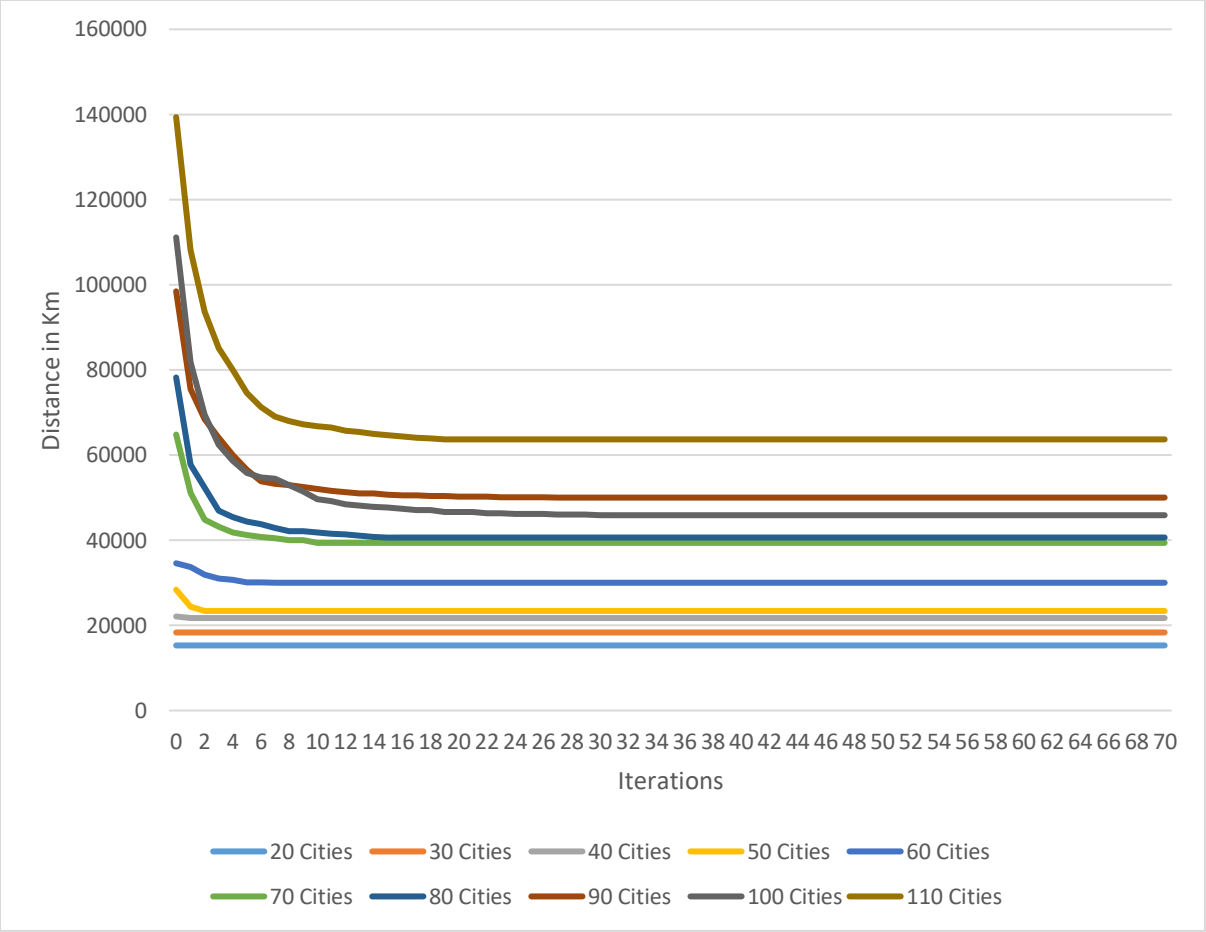

Figure 17. 12-core machine

Figure 17 shows the result when the program was run on 12 cores with 70 iterations. For 20 cities the optimum distance obtained was 15285.75 Km. For 30 Cities, the initial distance obtained in 0^{th} iteration was 18328.05 Km and at the end, the final optimized distance was 18328.05 Km. When run with 40 cities, the distance was 22088.14 Km in the 0^{th} iteration and 21722.21 Km in the 70^{th} iteration. For 50 Cities, the initial distance was 28385.77 Km and the final optimized distance was 23392.49 Km. 34586.67 Km was the distance in the 0^{th} iteration and 30014.63 Km in the 70^{th} iteration for 60 cities. For 70 Cities, 64827.07 Km was the distance obtained in 0^{th} iteration and 39376.31 Km for the 70^{th} iteration. The distance obtained when the program was run with 80 cities was 78226.16 Km and 40633.98 Km as the initial distance and

the final optimized distance obtained respectively. For 90 cities, in the 0th iteration, the distance obtained was 98459.06 Km and for 70th iteration, the final distance obtained was 50014.62 Km. For 100 cities, the distance in the 0th iteration was 111134.2 Km and for the 70th iteration, 45872.82 Km was the result. When the program was run with 110 cities, the initial value obtained was 139371.9 Km and the final optimized value was 63686.06 Km.

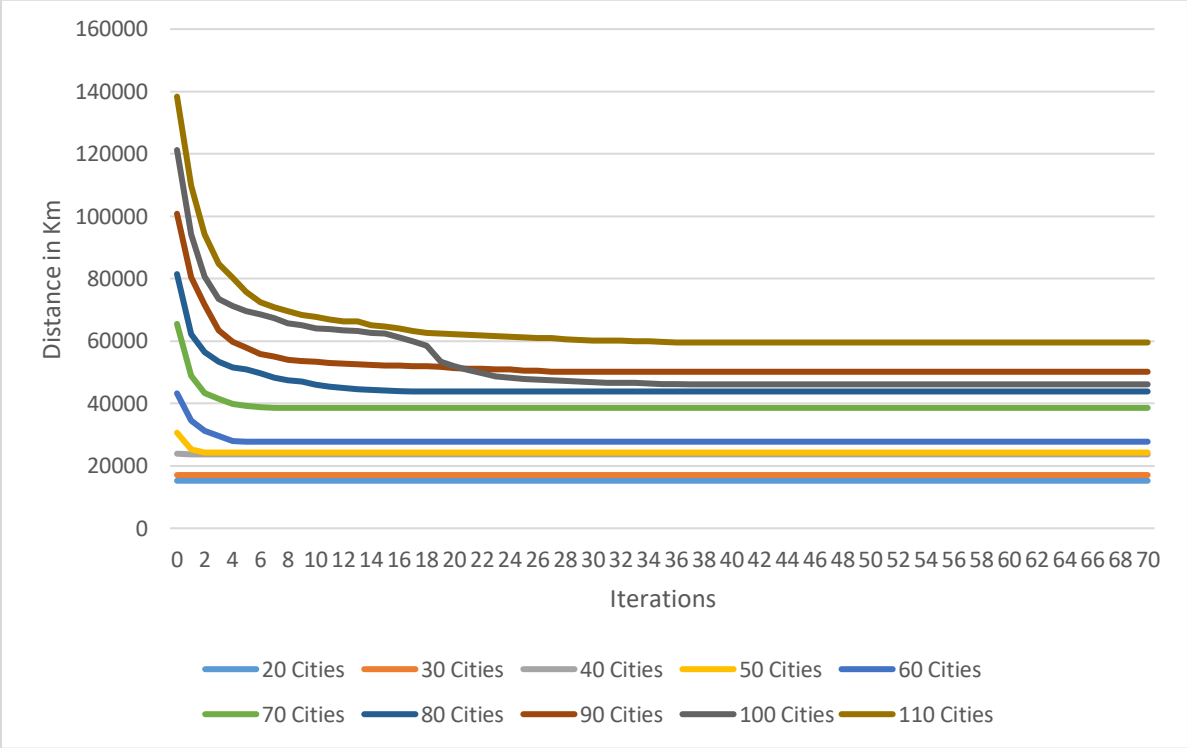

Figure 18. 13-core machine

Figure 18 shows the result when the program was run on 13 cores with 70 iterations. For 20 cities the optimum distance obtained was 15243.56 Km. For 30 Cities, the initial distance obtained in 0th iteration was 17060.84 Km and at the end, the final optimized distance was 17060.84 Km. When ran with 40 cities, the distance was 23906.48 Km in the 0th iteration and 23676.3 Km in the 70th iteration. For 50 Cities, the initial distance was 30638.16 Km and the final optimized distance was 24284.6 Km. 43254 Km was the distance in the 0th iteration and

27751.75 Km in the 70th iteration for 60 cities. For 70 Cities, 65510 Km was the distance obtained in 0th iteration and 38598.62 Km for the 70th iteration. The distance obtained when the program was run with 80 cities was 81476.19 Km and 43852.64 Km as the initial distance and the final optimized distance obtained respectively. For 90 cities, in the 0th iteration, the distance obtained was 100820.5 Km and for 70th iteration, the final distance obtained was 50163.08 Km. For 100 cities, the distance in the 0th iteration was 121232.3 Km and for the 70th iteration, 46136.33 Km was the result. When the program was run with 110 cities, the initial value obtained was 13835.3 Km and the final optimized value was 59554.48 Km.

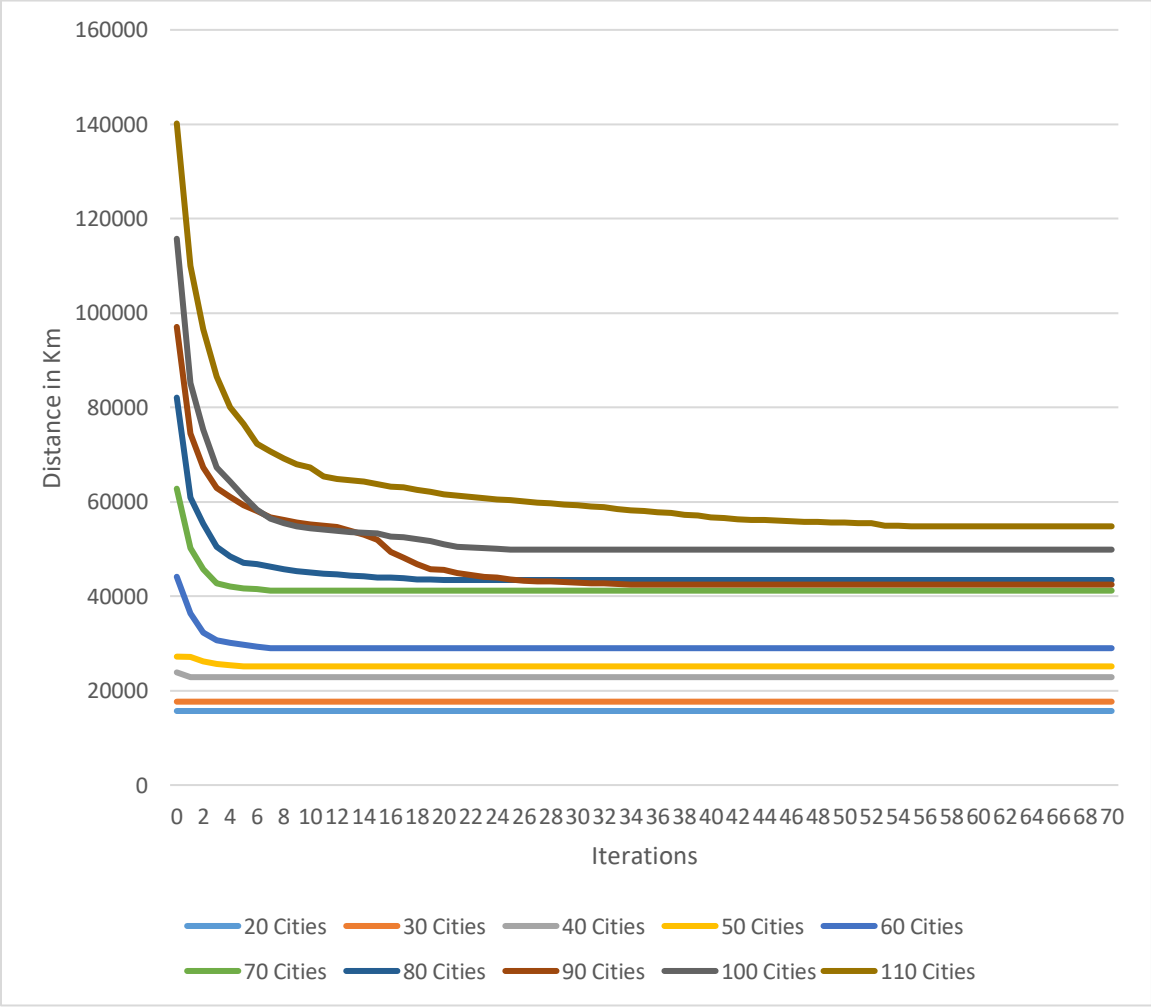

Figure 19. 14-core machine

Figure 19 shows the result when the program was run on 14 cores with 70 iterations. For 20 cities the optimum distance obtained was 15691.85 Km. For 30 Cities, the initial distance obtained in 0^{th} iteration was 17669.36 Km and at the end, the final optimized distance was 17669.36 Km. When run with 40 cities, the distance was 23875.2 Km in the 0^{th} iteration and 22863.8 Km in the 70^{th} iteration. For 50 Cities, the initial distance was 27214.56 Km and the final optimized distance was 25144.48 Km. 44132 Km was the distance in the 0^{th} iteration and 28997.03 Km in the 70^{th} iteration for 60 cities. For 70 Cities, 62794.71 Km was the distance obtained in 0^{th} iteration and 41188.93 Km for the 70^{th} iteration. The distance obtained when the program was run with 80 cities was 82090.38 Km and 43446.11 Km as the initial distance and the final optimized distance obtained respectively. For 90 cities, in the 0^{th} iteration, the distance obtained was 97070.08 Km and for 70^{th} iteration, the final distance obtained was 42471.44 Km. For 100 cities, the distance in the 0^{th} iteration was 115747.6 Km and for the 70^{th} iteration, 49880.21 Km was the result. When the program was run with 110 cities, the initial value obtained was 140174.4 Km and the final optimized value was 54816.97 Km.

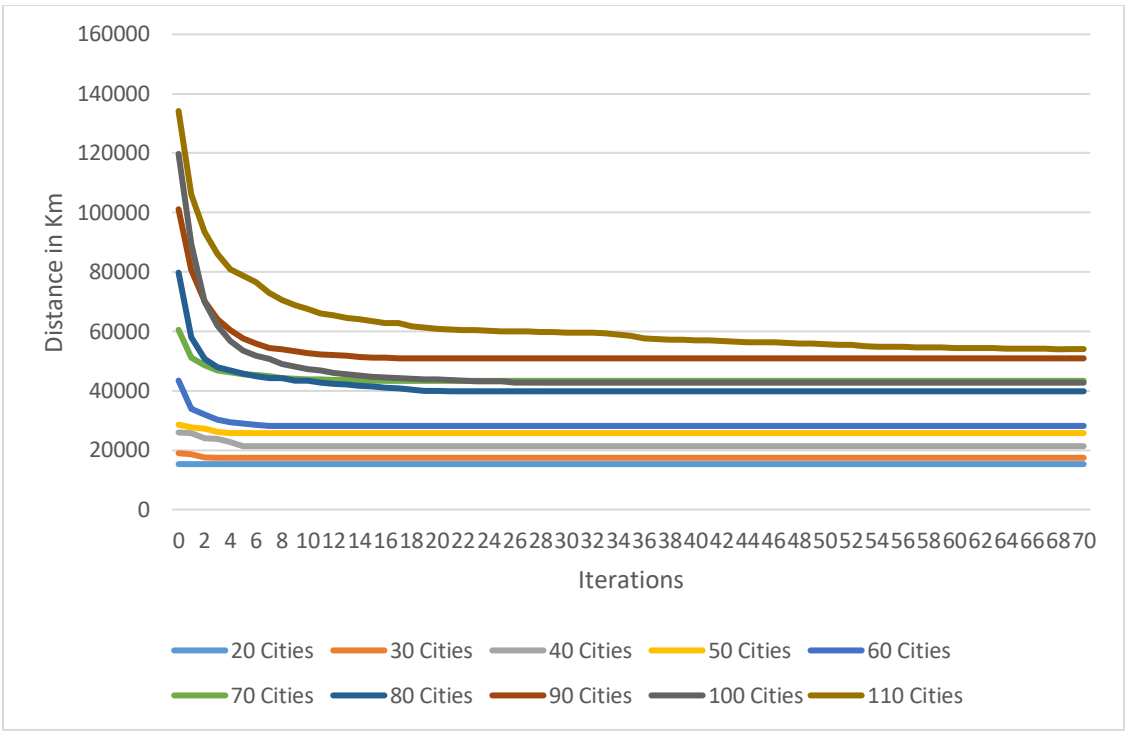

Figure 20. 15-core machine

Figure 20 shows the result when the program was run on 15 cores with 70 iterations. For 20 cities the optimum distance obtained was 15285.75 Km. For 30 Cities, the initial distance obtained in 0^{th} iteration was 18964.75 Km and at the end, the final optimized distance was 17424.65 Km. When run with 40 cities, the distance was 25937.87 Km in the 0^{th} iteration and 21324.32 Km in the 70^{th} iteration. For 50 Cities, the initial distance was 28563.85 Km and the final optimized distance was 25726.34 Km. 43418.21 Km was the distance in the 0^{th} iteration and 28144.14 Km in the 70^{th} iteration for 60 cities. For 70 Cities, 60495.19 Km was the distance obtained in 0^{th} iteration and 43320.05 Km for the 70^{th} iteration. The distance obtained when the program was run with 80 cities was 79767.92 Km and 39805.33 Km as the initial distance and the final optimized distance obtained respectively. For 90 cities, in the 0^{th} iteration, the distance obtained was 101109.1 Km and for 70^{th} iteration, the final distance obtained was 50893.58 Km. For 100 cities, the distance in the 0^{th} iteration was 119741.4 Km and for the 70^{th} iteration,

42722.52 Km was the result. When the program was run with 110 cities, the initial value obtained was 134173 Km and the final optimized value was 54009.78 Km.

Figure 21. 16-core machine

Figure 21 shows the result when the program was run on 16 cores with 70 iterations. For 20 cities the optimum distance obtained was 15285.75 Km. For 30 Cities, the initial distance obtained in 0^{th} iteration was 17381.5 Km and at the end, the final optimized distance was 17247.83 Km. When run with 40 cities, the distance was 21651.29 Km in the 0^{th} iteration and 20280.08 Km in the 70^{th} iteration. For 50 Cities, the initial distance was 31069.69 Km and the final optimized distance was 24852.36 Km. 45062.47 Km was the distance in the 0^{th} iteration and 29574.28 Km in the 70^{th} iteration for 60 cities. For 70 Cities, 67220.34 Km was the distance obtained in 0^{th} iteration and 41289.24 Km for the 70^{th} iteration. The distance obtained when the program was run with 80 cities was 77324.28 Km and 39592.02 Km as the initial distance and the final optimized distance obtained, respectively. For 90 cities, in the 0^{th} iteration, the distance obtained was 97016.57 Km and for 70^{th} iteration, the final distance obtained was 48891.82 Km.

For 100 cities, the distance in the 0th iteration was 120369.7 Km and for the 70th iteration, 50054.93 Km was the result. When the program was run with 110 cities, the initial value obtained was 142125.5 Km and the final optimized value was 49751.01 Km.

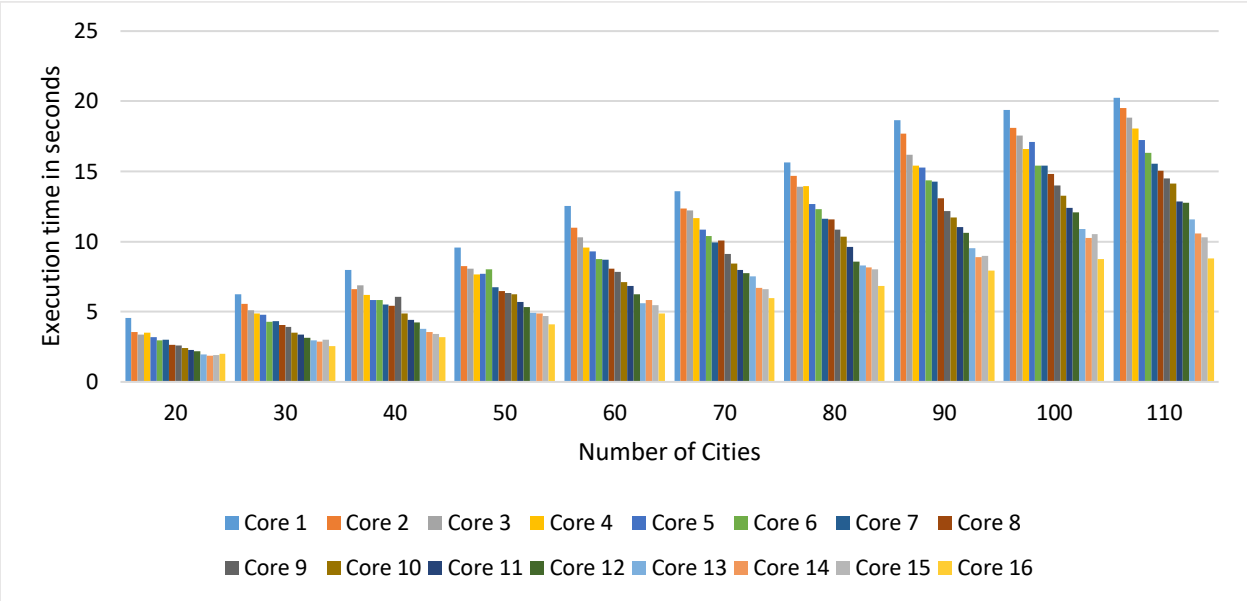

Figure 22. Time Taken

In Figure 22, the time taken by the different cores for different numbers of cities are shown. Time is directly proportional to the number of cities. Core 1 took more time than Core 16. The final optimum distance taken by each cores with different numbers of cities are shown in Table 2. The optimized result varies based on the number of cities as well as the number of cores. Looking at the figure, we can also see that at some point, even the difference in the number of cities does not vary much in execution time when the program is run in different cores. So when it is going too high, the difference is less sometimes. For example, in the figure if we take the execution time of Core 5, after reaching 100 cities, the execution time is similar even for 110 cities. Core 16 has similar execution time for 80 cities, 90 cities, and so on. It has only very little variation.

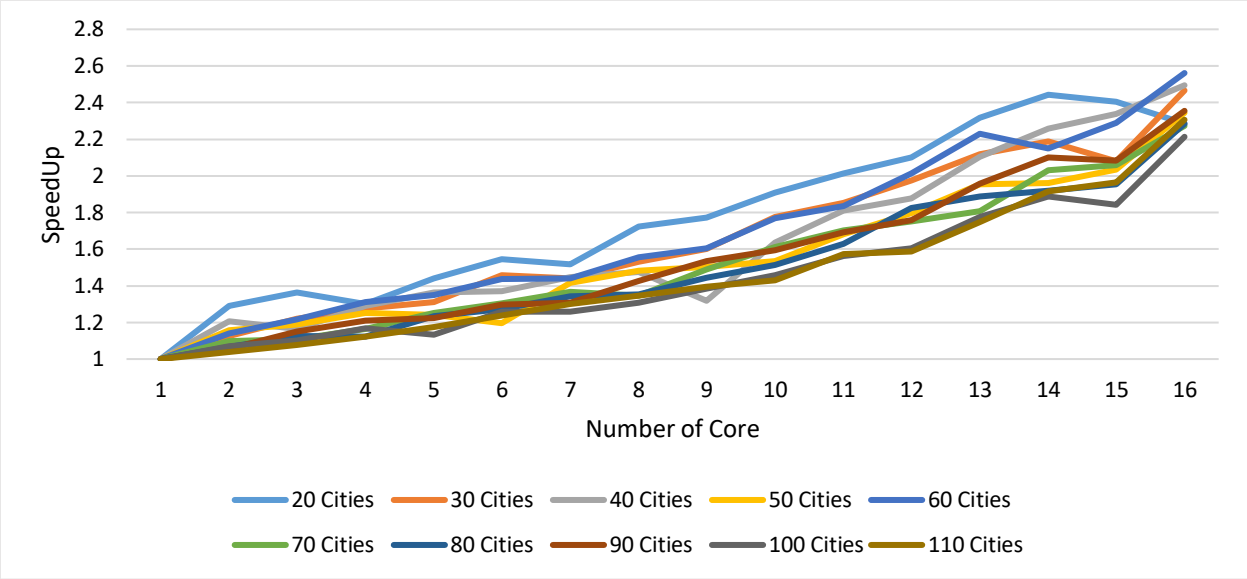

Figure 23. SpeedUp

In Figure 23, it shows the execution speedup, which is calculated by dividing the time taken for running the algorithm on one core and the time taken for running the algorithm on several cores. When the speedup value of the different numbers of cores were compared, the highest number of cores had the highest performance.

Table 2. Final Optimum Distance

Core #	20 Cities	30 Cities	40 Cities	50 Cities	60 Cities	70 Cities	80 Cities	90 Cities	100 Cities	110 Cities
1	15910.43	17793.67	21437.10	25621.57	34042.24	38712.94	48615.40	53436.76	53066.59	61098.32
2	15684.06	19021.73	22302.03	28060.47	32996.03	40468.17	49028.83	55900.47	48644.79	65746.72
3	15243.56	19160.22	21366.98	25182.21	33088.20	39469.20	50592.18	43498.64	58751.50	60499.12
4	15437.06	18441.62	20399.14	24505.71	30241.51	38287.77	47583.30	44357.03	53907.01	61532.21
5	15243.56	17058.89	19667.46	28760.30	30368.92	40373.83	46438.45	49762.59	52025.70	64835.04
6	15684.06	16916.49	21176.29	24476.94	31660.57	39546.51	43463.88	54027.96	50906.56	60485.33
7	15243.56	18337.75	21060.81	23506.55	29560.79	39515.49	47171.54	48242.71	50199.86	63777.00
8	15437.06	17497.76	20457.59	25229.49	28167.17	37496.83	47910.33	45233.52	52101.65	56157.38
9	15285.75	16820.22	21555.95	29081.49	28106.16	44194.25	46500.35	43718.64	44748.78	56303.68
10	15940.64	16796.97	21500.20	26206.03	30696.95	42848.72	38382.86	58117.92	55076.18	56879.42
11	15243.56	17819.80	21312.13	25711.21	30176.97	43403.25	40082.96	46664.44	52519.73	57919.82
12	15285.75	18328.05	21722.21	23392.49	30014.63	39376.31	40633.98	50014.62	45872.82	63686.06
13	15243.56	17060.84	23676.30	24284.60	27751.75	38598.62	43852.64	50163.08	46136.33	59554.48
14	15691.85	17669.36	22863.80	25144.48	28997.03	41188.93	43446.11	42471.44	49880.21	54816.97
15	15285.75	17424.65	21324.32	25726.34	28144.14	43320.05	39805.33	50893.58	42722.52	54009.78
16	15285.75	17247.83	20280.08	24852.36	29574.28	41289.24	39592.02	48891.82	50054.93	49751.01

5. CONCLUSION

The travelling salesman problem applying genetic algorithm using spark is a good approach for optimization. The problem addressed mainly focused on how to show the effect of parallelization of a genetic algorithm using the travelling salesman problem as the benchmark. Thus, the genetic algorithm based travelling salesman problem parallelized with Spark was investigated.

The algorithm was run on different numbers of cores with different numbers of cities using 70 iterations. The result was an optimized final distance. The execution time was also recorded to calculate and compare the speedup value of the different experiments. The conclusion from the results of the experiments are that the speedup obtained using the highest numbers of cores is higher than the others, i.e., the performance of using 16 cores is better than when lesser numbers of cores are used.

As for future work, a comparison of the genetic algorithm with other optimization methods could be made by implementing the other optimization methods using the Spark framework so that the speedup could be compared.

REFERENCES

[1] S. B. Pattnaik, S. Mohan, V. M. Tom. July 2008. Urban Bus Route Network Design Using Genetic Algorithm. *Journal of Transportation Engineering*. 124 4. doi: https://doi.org/10.1061/(ASCE)0733-947X(1998)124:4(368).

[2] R. Qi, Z. Wang, S. Li, IEEE. March 2016. A Parallel Genetic Algorithm Based on Spark for Pairwise Test Suite Generation. *Journal of Computer Science and Technology* 31(2):417-427. doi: 10.1007/s11390-016-1635-5.

[3] S. Liu, Sun Yat-sen University, Guangzhou China. February 2014. A Powerful Genetic Algorithm for Traveling Salesman Problem. *eprint arXiv: 1402.4699.* doi: 2014arXiv1402.4699L.

[4] N. Soni, T. Kumar. Lingaya's University, Faridabad. 2014. Study of Various Mutation Operators in Genetic Algorithms. *International Journal of Computer Science and Information Technologies, Vol. 5(3), 2014, 4519-4521.* Issn: 0975-9646.

[5] N. Soni, T. Kumar. Lingaya's University, Faridabad. 2014. Study of Various Crossover Operators in Genetic Algorithms. *International Journal of Computer Science and Information Technologies, Vol. 5(6), 2014, 7235-7238.* Issn:0975-9646.

[6] O. Abdoun, J. Abouchabaka, C. Tajani, Ibn Tofail University, Kenitra, Morocco. March 2012. Analyzing the Performance of Mutation Operators to Solve the Travelling Salesman Problem. *IJES, International Journal of Emerging Sciences , 2(1), 61-77.* arXiv:1203.3099v1 [cs.NE]. Issn: 2222-4254.

[7] J. Jaros, P. Pospichal. April 2012. A Fair Comparison of Modern CPUs and GPUs Running the Genetic Algorithm under the Knapsack Benchmark. *Applications of Evolutionary Computation.* doi: 10.1007/978-3-642-29178-4_43.

[8] S. Gupta, G. Agarwal, V. Kumar. April 2013. An Efficient and Robust Genetic Algorithm for Multiprocessor Task Scheduling. *IJCTE 2012 Vol.5(2): 377-382 ISSN: 1793-8201.* doi: 10.7763/IJCTE.2013.V5.713.

[9] V. Dwivedi, T. Chauhan, S. Saxena, P. Agarwal. 2012. Travelling Salesman Problem using Genetic Algorithm. *National Conference on Development of Reliable Information systems, Techniques and Related Issues (DRISTI). Proceedings published in International Journal of Computer Applications (IJCA).* https://www.ijcaonline.org/proceedings/dristi/number1/5926-1007.

[10] M. J. Brauer, M. T. Holder, L. A. Dries, D. J. Zwickl, P. O. Lewis, D. M. Hillis. October 2002. Genetic Algorithms and Parallel Processing in Maximum-Likelihood Phylogeny Inference. *Oxford Journals* https://doi.org/10.1093/oxfordjournals.molbev.a003994.

[11] S. Forrest. August 1993. Genetic Algorithms: Principles of Natural Selection Applied to Computation. New Series. *American Association for the Advancement of Science.* Volume 261, Issue 5123(Aug. 13, 1993), 872-878. https://www.jstor.org/stable/2882113?seq=1#metadata_info_tab_contents.

[12] C. Paduraru, M. Melemciuc, A. Stefanescu, University of Bucharest and Electronic Arts Bucharest, Romania. July 2017. A distributed implementation using Apache Spark of a genetic algorithm applied to test data generation. *The Genetic and Evolutionary Computation Conference Companion.* doi: 10.1145/3067695.3084219.

[13] A. J. Rivera, M.D. Perez-Godoy, F. Pulgar and M. J. Jesus. GenRBFNSpark: A first implementation in Spark of a genetic algorithm to RBFN design. http://simd.albacete.org/actascaepia15/papers/01011.pdf.

[14] The Apache Software Project. Introduction to the POM.
http://maven.apache.org/guides/introduction/introduction-to-the-pom.html.

[15] Apache Spark Introduction.
https://www.tutorialspoint.com/apache_spark/apache_spark_introduction.htm

[16] Crossover_(genetic_algorithm)
https://en.wikipedia.org/wiki/Crossover_(genetic_algorithm)

[17] Example of Running Genetic Algorithm
https://github.com/nielsutrecht/spark-of-life

[18] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, Ion Stoica University of California, Berkeley.Resilient. 2012. Distributed Dataset: A Fault-Tolerant Abstraction for In-Memory Clister Computing. *Usenix: The Advanced Computing Systems Association.*
https://cs.stanford.edu/~matei/papers/2012/nsdi_spark.pdf.

[19] RDD — Resilient Distributed Dataset
http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.rdd.RDD.

[20] Genetic Algorithms – Darwin's influence in computer science
https://blogs.unimelb.edu.au/sciencecommunication/2013/10/14/genetic-algorithms-darwins-influence-in-computer-science/

[21] What is the Genetic Algorithm?
https://www.mathworks.com/help/gads/what-is-the-genetic-algorithm.html

[22] Genetic Algorithm.
T.V. Mathew, Department of Civil Engineering, Indian Institute of Technology Bombay.
https://datajobs.com/data-science-repo/Genetic-Algorithm-Guide-[Tom-Mathew].pdf

[23] Genetic Algorithms.
https://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol1/hmw/article1.html

[24] Genetic Algorithms
https://datajobs.com/data-science-repo/Genetic-Algorithm-Guide-[Tom-Mathew].pdf

[25] Parallel computing
https://en.wikipedia.org/wiki/Parallel_computing

[26] Parallel computing and its advantage and disadvantage
https://www.geekboots.com/story/parallel-computing-and-its-advantage-and-disadvantage