Parallel genetic algorithm for Travelling Salesman Problem

Lokesh Agrawal

December 09, 2016

Abstract

Travelling salesman problem is a problem which belongs to the class of NP- hard problems which states that worst case timing for any algorithm for solving TSP increases exponentially by increasing number of cities. This problem is a special case of vehicle routing problem and travelling purchaser problem. There have been many heuristic approaches and Brute Force methods that have been used to try solving Travelling Salesman Problem. But even, the heuristics approach takes a lot of time for larger cities when implemented sequentially. So, in our research investigation we plan to solve Travelling Salesman Problem with Parallel Genetic Algorithm on the cluster. Genetic Algorithm is a heuristic approach which belongs to the class of Evolutionary Algorithms. It involves the process of evolution of solution relying on several operators such as Crossover, Mutation and Selection. So, after huge number of evolutions there is high probability of getting a solution which will be near to the optimum solution. For implementing Parallel Genetic Algorithm on the cluster, we will use Parallel PJ2 library developed by Professor Alan Kaminsky.

1. Introduction to the computation problem

Travelling Salesman Problem is a computation problem which is defined as: "*Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city*"[1]. Travelling salesman problem is a problem which belongs to the class of NP- hard problems which states that worst case timing for any algorithm for solving TSP increases exponentially by increasing number of cities. This problem can be modeled as a graph in which each city can be taken as a node in the graph and path between each pair of cities can be replaced by the edges in the between corresponding nodes and weight can be replaced by the distance between those nodes.

The most direct approach for solving this problem is trying all combination which can also be called as Brute Force Method. The running time for this approach is O(n!), where n is the number of cities. For only 15 cities, this problem has total "1.3e+12" combinations which is huge for even modern computers. From several researches the complexity of this problem has

been reduced to O(n^2*n^2) by Held-Karp Algorithm, which is also exponential. This is impractical, even when the above approaches are implemented in Parallel on clusters or even on GPUs. So, we have used Genetics Algorithm to solve this problem which is a heuristics approach and belongs to the class of Evolutionary Algorithms. This approach gives near to optimum solution in practical period of time.

2. Related Work

2.1 Research Paper 1

Title: Performance enhancement in solving Traveling salesman problem using hybrid genetic algorithm

2.1.1 Problem addressed

This paper discusses the hybrid genetic algorithm as compared to pure genetic algorithm. It also discusses why hybrid genetic algorithm gives better results and converges earlier as compared to pure genetic algorithm. Other evolution parameters such as cross-over and mutation in also discussed.

2.1.2 Novel contributions

This paper discusses both Pure Genetic Algorithm and Hybrid Genetic Algorithm and other evolution parameters such as crossover and mutation. As per the discussion in the paper pure and hybrid genetic algorithm only differs on how is the initial population generated.

- **Hybrid Genetics Algorithm**

Using Hybrid Genetic Algorithm better solution and better convergence rate is achieved in most cases as compared to Pure Genetics Algorithm. Hybrid Genetics Algorithm differs from Pure Genetics Algorithm only on the basis of initial population generation. In this paper, they have chosen Nearest Neighbor algorithms to compare performance enhancement in solving TSP with Hybrid Approach over pure approach.

Nearest Neighbor Algorithm consist of following steps:

1. Move all the cities in a list.
2. Make any city as the current city and remove it from the list and store it in result list.
3. Find nearest city to the current city from the cities list. Remove this city from the city list, make this city as the current city and add to the result list.

4. Repeat above step till the initial city list becomes empty.

In Hybrid approach, only 10% of the total population is calculated by using nearest neighbor algorithm and the remaining 90% population is calculated randomly. This hybrid approach helps to evade local maximum. Below graph shows the comparison between the best tour distances achieved in pure GA and Hybrid GA. It shows that hybrid GA is far better than pure GA even if the Hybrid GA have an addition complexity because of NN algorithm for generating better population. Moreover, the NN algorithm solution depends on the starting city but Hybrid GA is independent of starting city.

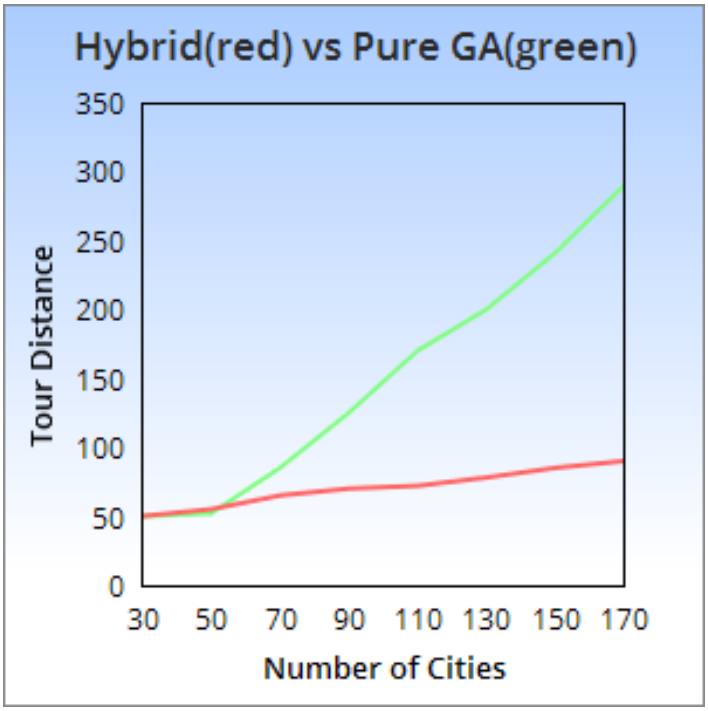

Fig1[2]. Best tour distance comparison for Hybrid vs Pure GA

Below is the table with data of Number of convergence generation and best tour distance for Hybrid GA and Pure GA for different number of cities. It can be easily observed that for Hybrid GA both tour distance and convergence generations used are much lower than Pure GA.

Cities	Distance(Pure GA)	Convergence Generation(Pure GA)	Distance(Hybrid GA)	Convergence Generation(Hybrid GA)
30	43.7765	150	4.4814	40
50	67.9719	350	61.4153	20
70	78.6589	500	69.9534	150
90	98.6679	500	81.4888	190

Table1: Hybrid GA vs Pure GA on the basis of convergence rate and tour distance

- **Evaluation Function/Operation**

To evaluation function is applied on each tour to find its fitness value based on which it will or will not be selected for further evolutions. This paper assigns a fitness value to each tour based on the tour distance.

- **Crossover**

Crossover is one of the evolution operations. In this paper, they have used the most basic crossover. Tours for crossover are selected on the basis of the fitness value assigned to them. For every crossover operation, 2 chromosomes are randomly selected using roulette wheel selection. Tours with higher fitness have higher probability of selection for crossover. Also, tours with higher probability have higher probability of generating better tours than previous ones.

In the above figure shown 2 different parents are chosen randomly, parent1 and parent2. Now 2 indexes are selected randomly such that index2 is always greater than index1. These index2 and index1 make a window. Now two empty childs are created, Child1 and Child2 with the same size as parents. The window made is copied from 1st parent to child 2 and from 2nd parent to child 1. Now we start traversing from 1st index in parent 1 and keep on checking if element at that index is present in child1 or not. If not then that element is copied to child 1 on the next available spot. Similarly, child 2 is also created.

- **Mutation**

The mutation operation is performed on the single tours at a time. The percentage of the mutation is generally kept very small which as near to 10% of the the total population size. Mutation is very critical for the Genetics Algorithm as it helps in evading local maxima. In mutation, a window is selected by generating 2 random indices which has been

optimized during the evolution of generations. Now, by changing the start and end city of the sub-route, the solution keeps on getting better.

- **Useful ideas**

In our implementation, we have used the Hybrid Genetic Algorithm approach.

2.2 Research Paper 2

Title: Enhanced Traveling Salesman Problem by Genetic Algorithm Technique

2.2.1 Problems Addressed

This research paper proposes a software system to find the optimal route for the Travelling Salesman Problem using Genetic Algorithm Technique. This paper also discusses the sequential approach and the evolution operations such as crossover and mutation. This paper also discusses the approaches to create initial population and finding fitness.

2.2.2 Novel Contributions

Current generation in a genetic algorithm is evolved using 3 operations selection, crossover and mutation. This paper discusses the approaches for fitness, selection, crossover and mutation operation.

- **Fitness Function**

Fitness function assigns a value to each member in the population (routes in our case) which tells that how fit or good this path is. Fitness function is applied to all the routes in the population to assign a numerical value to each path. For finding the fitness of each tour, first a route distance is calculated of each path which is euclidean distance in our case. So, each tour in our program has a distance associated with it noted as Di.

Now to find the fitness of each tour, below formula is used:

$Fi = D_{max} - D_i$ where

Fi = Fitness of tour i.

D_{max} = Maximum tour distance among all tours

Di = Tour distance of i^{th} tour.

So, fitness assigns a fitness value to each tour in such a way that tours having larger route distance will have lower fitness value and tours having lower distances will have large fitness value.

- **Selection**

Selection operation chooses 2 routes from the current population to participate in crossover and mutation. The idea behind this operation is selecting 2 routes from current population in such a way that these paths will have high probability to give better results after crossover and mutation.

Selection operation is the continuation of fitness function in which probability and sampling rate is defined on the basis of the fitness value calculated:

Probability is computed as the proportion using the following equation:

Pi = Fi / summation of Fi

Now sampling rate which is also called as deterministic sampling denoted by Si is evaluated using following relation

Si = ROUND(Pi*POPSIZE) + 1

Where: ROUND means rounding off to integer and POPSIZE means the population size.

i	Route	Di	fi	Pi	Si
1	ABCDEF	122	0	0.01	1
2	BCDEFA	101	21	0.10	2
3	CDEFAB	80	42	0.19	2
4	DEFABC	50	72	0.31	3
5	EFABCD	30	92	0.40	3

Fig2[]. Table which shows fitness, probability and deterministic sampling rate for 5 tours

- **Cross-Over**

Crossover is one of the important operations as this operation evolves the 2 selected routes by selection and new routes are created which has high probability to have better fitness than parents. This research paper discusses several approaches of crossover: Order Crossover, Partially Matched Crossover and Cycle Crossover. But the author states that Order crossover gives better results as compared to the other 2 approaches and hence we have implemented Order Crossover and our project.

The steps in crossover algorithm are as follows:

1. Two parents are selected randomly which are parent1 and parent 2 in the below example.
2. Now, 2 indexes are selected randomly diving any tour in 3 parts.
3. Now, 2 child are initialized with the same size as of parents.
4. Copy the part 2 (middle one) from parent1 to child2 and from parent 2 to child 1 at the same location.
5. This example shows child2, so GHB is copied from parent 1 to child2 at the same location i.e part2.
6. Now, start iterating in parent 2 in the order of: part3 > part2 > part1 which is BA>HDE>FGC in our case and keep on checking in child 2 if that city is present in the path or not.
7. If that city is present then do nothing and if that city is not present then update this city in child2 at the next available location.
8. At the end, parents have been evolved to obtain new child.

Fig3: Example of Crossover

- **Mutation**

Mutation is a process of swapping 2 genes in an DNA. In our case, 2 cities are swapped in a randomly selected tour. Mutation helps in evading local maxima since it provides more diversity in the population. Also, mutation preserves the path evolved during several generations since it only swaps 2 cities and all other city locations in the tour are preserved.

Mutation is done by following the below steps:

1. A random tour is selected from the mating pool.
2. 2 indexes are selected randomly.
3. Cities at these 2 locations are swapped with each other.

In the below example 2 index are: 2 and 6 and it can be observed that B and G are swapped to get the mutated child.

- **Useful Ideas**

In our paper, we have used fitness function, selection criteria to make new mating pool based on discrete sampling rate, ordered crossover and mutation. This paper was very useful for us to understand the sequential approach and implement it using all the parameters and operations required.

2.3 Research Paper 3

Title: Island migration model with parallel mutation strategies for computing TSP on multicomputer platform.

2.3.1 Problems Addressed

This authors states that TSP is an NP hard problem. Even trying solving it with Genetic Algorithm takes a lot of time to find an optimal solution as it contains a lot of operations like crossover, mutation, selection and after crossover or mutation, fitness needs to be calculated again. This paper proposes migration strategies to implement the genetic algorithm on a cluster which is based on the concept of sharing evolved tours with others at a rate which is defined as migration rate. This is done to bring more diversity in the population present.

This paper also aims at studying the efficiency of the migration strategies discussed and studying their scalability when implemented on a cluster. The results in the paper points that migration strategies for implementing Genetic Algorithm in parallel is powerful and useful.

2.3.2 Novel Contributions

An efficient parallel genetic model with periodic bidirectional migration of circular topology with parallel mutation rates for solving TSP on a multi-computer platform has been proposed in this paper which is very powerful.

The migration topology constitutes a logical ring of processors. Each processor evolves it population by running sequential Genetic Algorithm in each processor and then each processor shares its best routes with other processors. Each processor shares its best routes with a processor having processor id: (Pi+1)modN and receives only from (Pi-1+N)modN

where, Pi = Processor id of current processor and N = total number of processors. The receiving process replaces its tours with least fitness with the best tours received from other process.

- **Useful Ideas**

The most useful idea in the above paper that we have used in our project is Parallel migration strategy.

3. Implementation

In our implementation of both sequential and parallel approach we have used Parallel Java PJ2 library.

Sequential Implementation:

Steps in our sequential algorithm are as follows:

1. The first step is to create the population. To create population of size N, initial tour is shuffled N times with different random key such that a different path will be generated each time shuffle function is called.
2. Fitness function is applied on each tour in the population to find the fitness of each tour.
3. In selection step, Discrete Sampling Rate is calculated based on the fitness value of the tours and now based on the calculated Sampling Rate a new mating pool is created. To create this mating pool, each tour is copied into the mating pool the number of times which is equal to its sampling rate. Such that probability of getting tours for crossover or mutation will be higher.
4. Cross-over operation is performed x times on randomly selected 2 routes and evolved child are added into the initial population pool.
5. Mutation operation is performed y times which is 10% of total population size. By increasing mutation rate it has been observed that performance degrades and then evolved routes are added into initial population pool.
6. Now initial population pool is sorted based on fitness value of tours and all tours which are present at indexes greater than population size are deleted in order to maintain population

size at each iteration of Genetic Algorithm. Otherwise, size of population will keep on increasing.

7. Steps from step 2
8. Best tour at the end of final iteration and details related to it are returned.

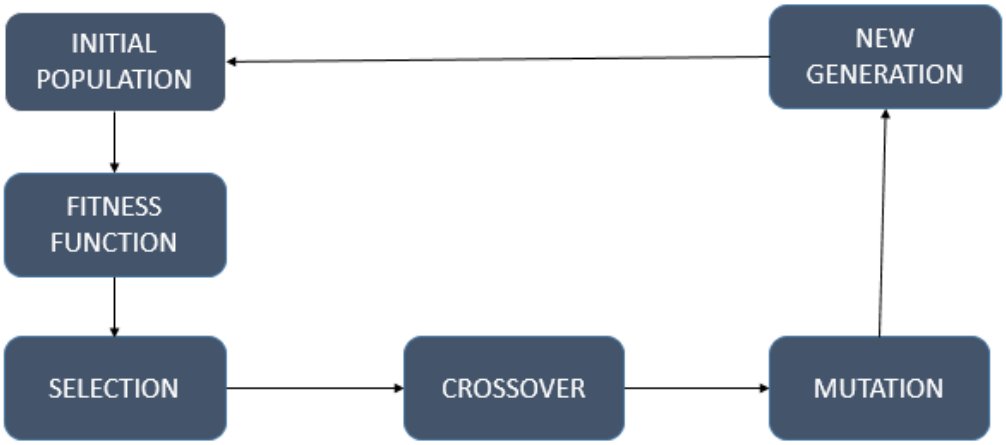

Fig 4. Sequential Approach Block Diagram

Parallel Implementation

In parallel implementation of our program, sequential Genetic Algorithm runs on each core of each node present in the cluster. But in parallel implementation, migration occurs after x number of iterations which is defined by the user. During migration, each core shares its best routes with some particular core to increase population diversity at each location. This approach also helps the Genetic Algorithm to converge early.
In the below diagram, Master creates the initial tour and puts this initial tour in the tuple space. Now, each worker reads this initial tour. Now each core creates this initial tour and runs the sequential version of the Genetic Algorithm.

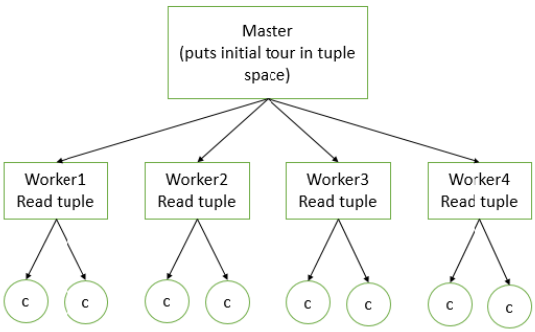

Fig 5. Master-Worker Pattern for parallel implementation on cluster

Now, after x iterations which is defined by the user, migration occurs between each core. Now for migration in between the cores in the same node, cores do not need to put their best routes in the tuple space and read from tuple space as cores can communicate with each other in the same node. But for cores which need to pass its best routes to the core which is present in some other node, it has to put these cores in the tuple space. And the receiver reads this tours from the tuple space.

After completing all iterations and migrations, Reduction happens. Reduction happens in 2 stages:

1. In between cores of the same nodes.
2. In between nodes.

For reduction in between same nodes, the best route is selected out of the evolved population on each core and then this route and its fitness is put in tuple space by each node. So, final reduction happens on the basis of the fitness. The tour with the highest fitness is selected.

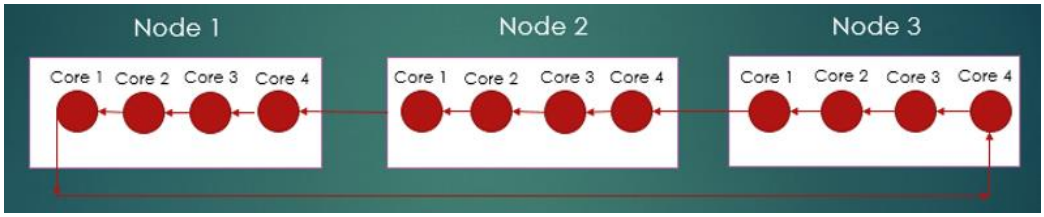

Fig 6: Migration Topology with Logical Ring of Processors

4. Developer's Manual

Our program is a cluster parallel program. For this program to run a cluster of multi-core machines are required. We have tested this program in a similar environment which is available at RIT's computer science department and the name of the server is TARDIS.

The program can be run in sequential and parallel mode. A parallel pj2 library is required to compile and run this project.

Following are the steps required to compile the source code for sequential version of the program on Tardis cluster:

1. Set Java classpath to include the PJ2 distribution.
   $ export CLASSPATH=.:/var/tmp/parajava/pj2/pj2.jar

2. Copy all the Java source files of this project (City.java, PointgGroup.java, Population.java, RandomPointGroup.java, SeqTSPMain.java, TravelingPath.java) in one folder.

3. Compile all these files using the following Java command - javac *.java

4. Create jar file by executing the following command - jar cf <jarfilename>.jar *.class

Following are the steps required to compile the source code for parallel version of the program on Tardis cluster:

1. Set Java classpath to include the PJ2 distribution.
   $ export CLASSPATH=.:/var/tmp/parajava/pj2/pj2.jar
2. Copy all the Java source files of this project (City.java, PointgGroup.java, Population.java,
   RandomPointGroup.java, SmpTSPMain.java, TravelingPath.java) in one folder.
3. Compile all these files using the following Java command - javac *.java
4. Create jar file by executing the following command - jar cf <jarfilename>.jar *.class

5. User's Manual

Sequential Version:
Below are the steps to run the sequential version of the program:

1. Create the jar file using steps mentioned in the Developer's manual.
2. java pj2 jar=<jarfile> workers=1 seqTSPMain <ctor> <populationsize> <GAiterations>

Where <ctor> is a constructor expression which uses "RandomPointGroup(40,500,987134)", the first parameter represents the number of cities, second parameter is maximum absolute coordinate value for a city and the last parameter is a seed for the RandomPointGroup class.

Parallel Version:
Below are the steps to run the sequential version of the program:

1. Create the jar file using steps mentioned in the Developer's manual.
2. java pj2 jar=<jarfile> workers=<K> seqTSPMain <ctor> <populationsize> <GAiterations>

Where K is the number of workers, <ctor> is a constructor expression which uses "RandomPointGroup(40,500,987134)", the first parameter represents the number of cities, second parameter is maximum absolute coordinate value for a city and the last parameter is a seed for the RandomPointGroup class.

6. Performance

The performance of this program is obtained by calculating Strong Scaling and Weak Scaling.

a. Strong Scaling

The Strong scaling is measured by running the program on different dataset of cities. We have run this program on 30 cities, 40 cities, 60 cities, 70 cities and 80 cities. As per the requirement of the strong scaling we have kept the problem size as constant and increased the number of cores. For our program the problem size is the number of genetic algorithm iterations to evolve the population at each iteration.

The graphs shows that we have achieved a good efficiency between 0.8 to 0.9. The efficiency is quite close to ideal but not perfect. One reason for non ideal efficiency is the parallel program does a little bit of extra work than a sequential version of the program. The parallel program has to create tuples for inter node migration. There is an extra processing involved to extract the top 10 fitter population from the node and then send this population to the neighbouring node. The receiving node also need to do extra processing to collect this migrated population and add it to its population list. Because of this extra processing we are not getting ideal efficiency. Also if you look at the graphs and table of the efficiency and speed up, you will notice that as the number of cores increases the efficiency slightly degrades. The reason behind this is as we are increasing the number of cores we are also increasing the tuples in the system. Also at the time of reduction more tuples are extracted by the front node to do the reduction. The processing at the reduction step also increases. Because of all these reasons we see a slight decrease in performance as we increase the number of cores.

No. of Cities	No. of Cores	Time(in msec)	Speed-up	Efficiency
30	1(Sequential)	99048		
	4	29422	3.366	0.841
	8	15149	6.538	0.817
	12	11007	8.998	0.749
	16	8882	11.151	0.896
40	1(Sequential)	144948		
	4	39412	3.677	0.919
	8	20434	7.093	0.886
	12	14770	9.813	0.817
	16	11634	12.458	0.778
60	1(Sequential)	251527		
	4	67700	3.715	0.928
	8	36745	6.845	0.855
	12	24458	10.284	0.857
	16	19472	12.917	0.807
70	1(Sequential)	311898		
	4	87780	3.553	0.888
	8	44827	6.957	0.892
	12	31704	9.837	0.895
	16	21.395	13.354	0.907
80	1(Sequential)	399053		
	4	108734	3.609	0.917
	8	58426	6.830	0.853
	12	39462	10.112	0.842
	16	29698	13.437	0.839

Fig: Strong Scaling Data

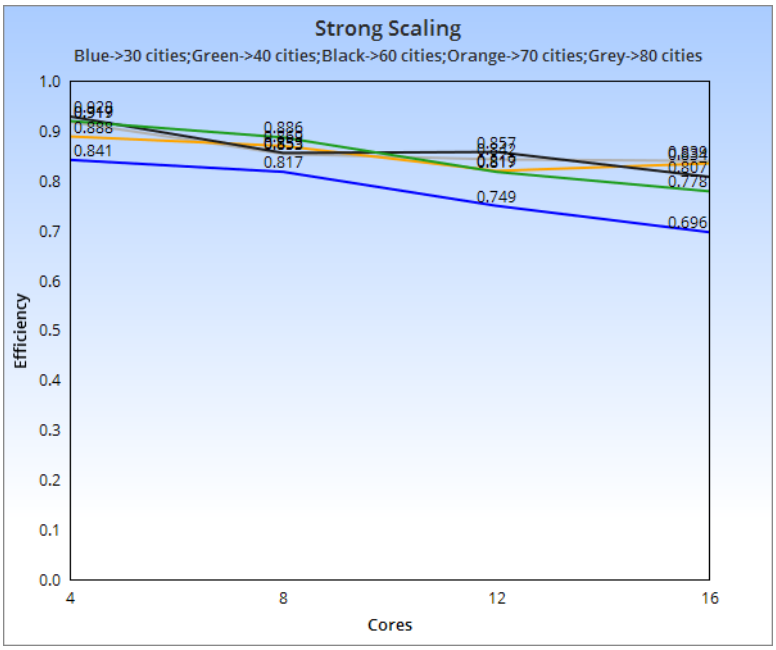

Fig: Number of cores vs Efficiency

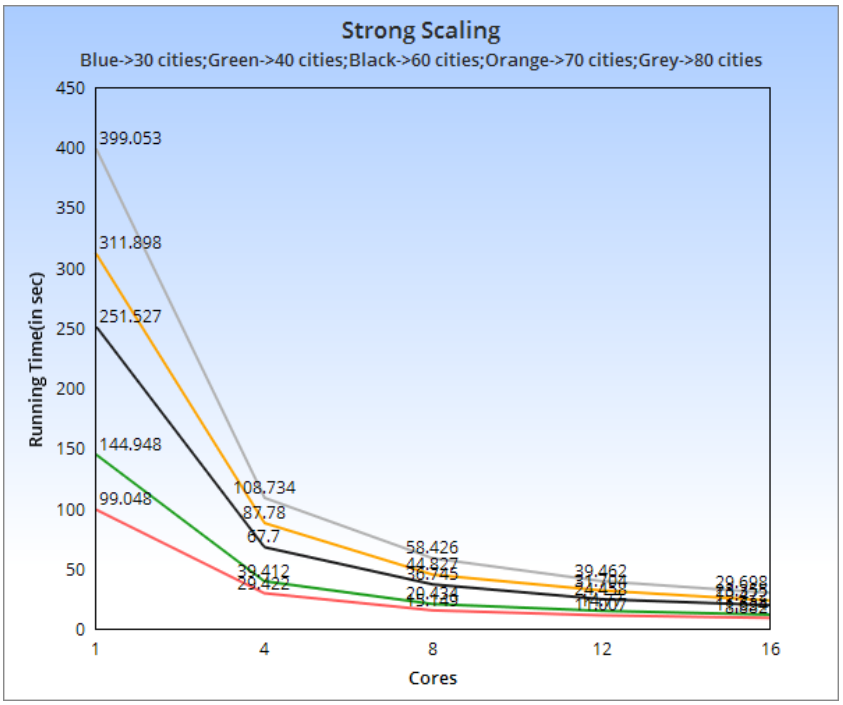

Fig: Number of cores vs Running time

b. **Weak Scaling**

In weak scaling we increase the problem size proportional to increase in the number of cores. The weak scaling is measured by running the program on different dataset of cities. We have run this program on 30 cities, 40 cities, 60 cities, 70 cities and 80 cities. The graphs shows that we have achieved a good efficiency between 0.8 to 0.9. The efficiency is quite close to ideal but not perfect. One reason for non ideal efficiency is the parallel program does a little bit of extra work than a sequential version of the program. The parallel program has to create tuples for inter node migration. There is an extra processing involved to extract the top 10 fitter population from the node and then send this population to the neighbouring node. The receiving node also need to do extra processing to collect this migrated population and add it to its population list. Because of this extra processing we are not getting ideal efficiency. Also if you look at the graphs and table of the efficiency and speed up, you will notice that as the number of cores increases the efficiency slightly degrades. The reason behind this is as we are increasing the number of cores we are also increasing the tuples in the system. Also at the time of reduction more tuples are extracted by the front node to do the reduction. The processing at the reduction step also increases. Because of all these reasons we see a slight decrease in performance as we increase the number of cores.

No. of Cities	No. of Cores	Problem Size	Time(in msec)	Size-up	Efficiency
30	1(Sequential)	160000	49212		
	4	640000	56208	3.502	0.875
	8	1280000	56188	7.006	0.876
	12	1920000	57277	10.310	0.859
	16	2560000	55611	14.158	0.885
40	1(Sequential)	160000	72205		
	4	640000	77205	3.740	0.935
	8	1280000	75991	7.093	0.950
	12	1920000	77896	7.601	0.927
	16	2560000	79087	14.067	0.912
60	1(Sequential)	160000	126961		
	4	640000	139304	3.645	0.911
	8	1280000	141567	7.174	0.896
	12	1920000	145431	10.475	0.873
	16	2560000	142849	14.220	0.889
70	1(Sequential)	160000	156323		
	4	640000	170964	3.66	0.915
	8	1280000	175172	7.139	0.892
	12	1920000	174650	10.740	0.895
	16	2560000	172275	14.158	0.907
80	1(Sequential)	160000	203202		
	4	640000	211919	3.835	0.959
	8	1280000	219395	7.409	0.926
	12	1920000	223942	10.888	0.907
	16	2560000	213876	15.201	0.950

Fig: Weak Scaling Data

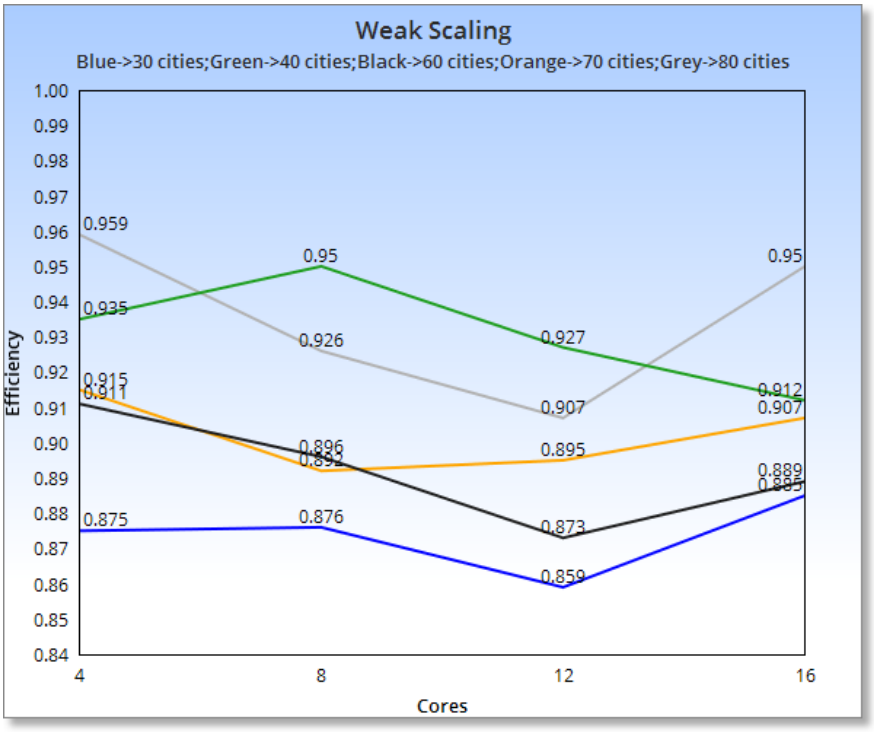

Fig: Number of Cores vs Efficiency

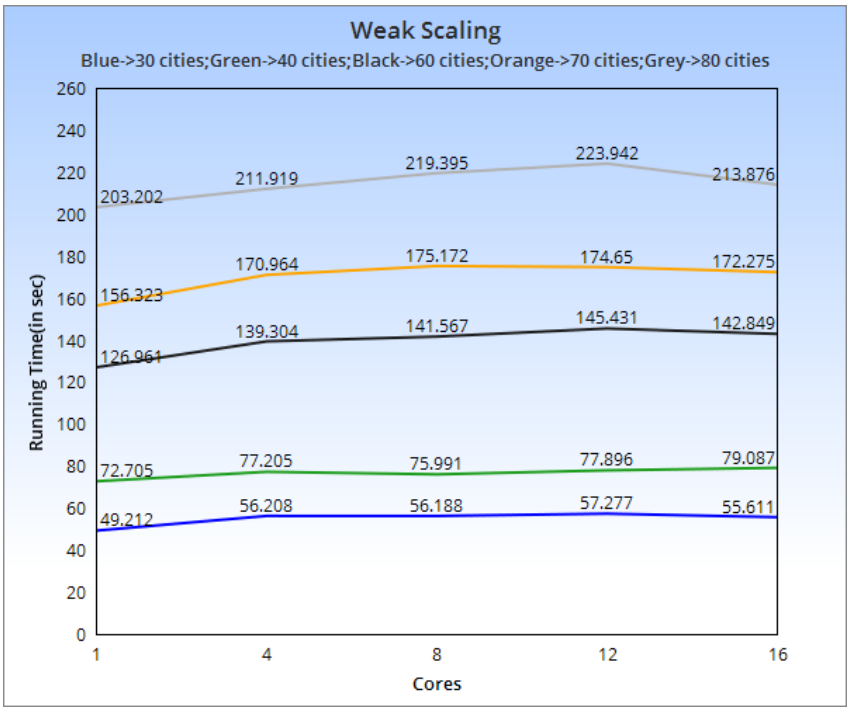

Fig: Running Time vs Number of Cores

7. Future Work

Lot of researches are going in the field of Travelling Salesman Problem using Genetic Algorithm and there are lot of factors which can be implemented in near future to improve the efficiency, quality and convergence rate of the solutions.

1. GPU Implementation:

The same approach can be implemented on GPU so that results will be obtained in a shorter time. Then results of GPU can be compared with Cluster results to decide which approach is

better. We believe, Cluster approach will be more efficient because in GPU's due to high number cores, overhead will be very high because of migration.

2.GUI:

Making a GUI for this project is one of the important thing which we should be implemented so that users who are not familiar with the command-line can use this application with ease.

3.Crossover and Mutation strategies:

Different crossover, mutation and selection combinations can be tried to check if some combination gives better results as compared to the approach that we tried in our implementation.

Learnings:

This project was a great learning experience for both of us. We were familiar with Travelling Salesman Problem but not with Genetic Algorithm which we found very interesting after learning the concept behind it and how it can give optimal results in a very short time as compared to the brute force approach.

We learned that efficiency of parallel programs really degrades if it contains intra node communication due to overhead. Also, concept of migration model was very interesting to learn.

Our understanding of Parallel PJ2 library developed by Professor Alan Kaminsky got excelled after implementing TSP using Parallel PJ2 library. We came across several features while implementing this project which we did not know existed.

Roles of Team Members:

Both members in our contributed equally towards the design, implementation of the problem, presentations and the final report. Few modules of the code were developed independently but while integrating our modules we both discussed each other work to make sure that the module developed by other person is working as per the requirement.

References:

1. http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=4531202

2. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.193.6087&rep=rep1&type=pdf
3. http://www.computingonline.net/index.php/computing/article/viewFile/431/401