Parallel Genetic Algorithm for Traveling Salesman Problem

Satyajeet Shahane

Ajeeth Kannan

July 22, 2018

Abstract

The Traveling Salesman Problem (TSP) is a classic problem belonging to the field of combinatorial optimization's and belongs to the class of NP-Hard problems. Since it's an NP-Hard problem, the exact solution to this problem is not known. However, the processing power of cluster computers or GPU's can be put to use, to try and find a solution that is very close to the ideal one. There are a number of approaches that can be employed to solve the TSP using cluster computers. Our solution is based on genetic algorithm which mimics the biological process of evolution. The algorithm produces fitter individuals (solutions), with the theory that, after an extremely large number of evolution's we would arrive at a near perfect solution. In our team research investigation, we have planned to design and implement a parallel genetic algorithm for the traveling salesperson problem on a cluster computer.

1 Computational Problem

The Traveling Salesman Problem is a classic problem belonging to the field of combinatorial optimizations and belongs to the class of NP-Hard problems. The traveling salesman problem asks the question that - '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]. This problem can also be modeled as a graph problem, where the cities represent vertices and the distances between each pair of vertices represent the distance between the corresponding cities. The problem can then be rephrased as finding a Hamiltonian path with minimal distance in this graph.

Since this is a NP-Hard problem, the exact solution to this problem is not known. Modern computers have tried running through all the combinations in order to get to the right answer. This approach can give us the right answer for a smaller number of cities. But as the number of cities increases, the total number of solutions to run through increases exponentially and so does the time needed to do this. For example, even with just 10 cities, we have to run through a total of 10! that is 37 lakh candidate solutions and for 20 cities it becomes 2.4 in 10 to the power of 15, a huge number indeed!

Thus the main computational problem is to traverse this huge solution space with some heuristic, which in our case is provided by the genetic algorithm.

2 Related Work

2.1 Research Paper One

2.1.1 Problems Addressed

The paper Al-Dulaimi et al [2] discusses a sequential design to solve TSP using genetic algorithm. This paper discusses the various techniques pertaining to all the stages in the genetic algorithm to solve TSP like fitness function evaluation, selection, crossover, and mutation.

2.1.2 Novel Contributions

To enhance the genetic algorithm solution for TSP, this paper proposes unique algorithms for selection, crossover and mutation.

- **Selection:** Selection is the process of selecting two tours of the present generation to participate in the next operations of crossover and mutation. This paper proposes two approaches for implementing selection. First is roulette selection, where each individual is assigned a probability computed using the following formula, $P_i = f_i + \Sigma f_j$, where P_i represents probability value of each tour(individual) and f_i is the fitness value of the ith individual. [2]

  The second approach, called deterministic sampling, assigns to each organism i, a value S_i evaluated by the relation - $S_i = ROUND(P_i * POPSIZE) + 1$, where ROUND means rounding off to integer and POPSIZE means the population size.[2] An example table with fitness values, calculated probability and sampling values is shown in Figure 1

Individual	D_i	f_i	P_i	S_i
BEFCAD	122	0	0	1
FEBACD	101	21	0.156	2
CDAFBE	80	42	0.311	2
BDAEFC	50	72	0.533	3

Figure 1: Illustrates sample values for probabilities in selection

  The authors claim that the above two approaches where performing poor in the sense that they were taking longer time to converge, so the authors have suggested a new approach hybrid approach which is the combination of the above mentioned two approaches[2].

- **Crossover:** Crossover is the most important operations among all the operations where two tours from the population are selected to form a new tour for the next generation. The paper discusses three types of crossover techniques named ordered crossover, partially matched crossover

and cycle crossover. But, we choose the ordered crossover technique in our implementation.

To apply ordered crossover, two random cross points are selected. All elements from first parent are copied to the offspring into the same position. Second Parent determines the remaining elements. Non-duplicative elements are copied form second parent to the offspring beginning at the position following the second cross point. From that point, both, the second parent and the offspring are traversed circularly.[2] An example of ordered crossover is given in the figure 2. If 3 and 6 are two random cross points, the elements between the cross points from first parent (GHB) are copied directly to the same position in the offspring. The elements after second cross point from second parent (BA), B is skipped, as B already exists in offspring. So, A is copied to position 7 in offspring. Traverse parent2 circularly and copy D in position 8. Traverse offspring circularly to fill the first 3 positions as shown in the example. [2]

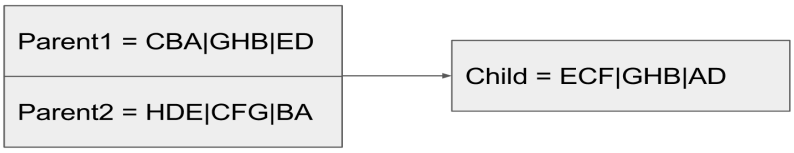

Figure 2: Illustrates result of crossover

- Mutation: Mutation is a process of changing single gene that does not preserve uniqueness. While mutation, in a particular tour, two randomly selected cities will be interchanged, thus preserving gene uniqueness. For example, as shown in the figure 3, If old child is a tour and it need to be mutated, two random cities B and G are selected and they are interchanged to form a new child. According to this paper mutation just occurs with very few tours of the total population in a generation.[2]

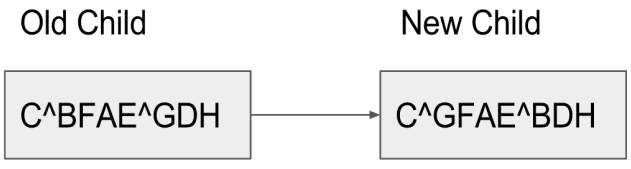

Figure 3: Illustrates result of mutation

2.1.3 Useful Ideas:

In our implementation we have used the same fitness evaluation method, hybrid approach for selection, ordered crossover for crossover and the mutation process

that were discussed above.

2.2 Research Paper 2

2.2.1 Problems addressed

The paper Pospichal et al [4] primarily aims at solving the traveling talesman problem using parallel genetic algorithms on CUDA architecture. The authors of this paper are aware that there are multiple ways a genetic algorithm can be paralleled, like masterslave, island model, hybrid approach etc.

The authors have done considerable research to find out that the CUDA toolkit for nvidia gives the best possible performance and speedup. They were also aware of the likely bottlenecks data migration can cause while executing a genetic algorithm.

Given the above considerations, this paper aims at designing a genetic algorithm for the TSP accelerated by a GPU with efficient mapping to CUDA software model. While designing this algorithm, the authors intend to specially focus on massive parallelism and using shared memory within multiprocessors to avoid system bus bottleneck.[4]

2.2.2 Novel contribution

As this is an island based approach, different cores carry out the evolution process independently. Better results can be achieved if there is some level of migration between the different cores. This paper proposes a unique asynchronous unidirectional migration strategy using the GPU main memory.

Different islands (cores) are interconnected to form a logical ring structure. After a set amount of time, a number of pre-determined individuals are migrated in one direction. This migration happens asynchronously, that is, while a core is working on evolution, it sends/received data from another core in a different thread. The results published in this paper clearly show that this migration strategy significantly improves the value to which the algorithm converges. [4]

2.2.3 Useful ideas

The main useful idea which we took from this paper was the ring topology of arranging the nodes in the cluster. However the authors implemented their approach on a GPU and expressed concerns regarding how efficient it would actually be on a cluster. The third research paper which we found assured us that the ring topology with a certain modifications in the migration strategy would work well on cluster computer.

2.3 Paper 3

2.3.1 Problems addressed

The paper Borovska et al [3] primarily investigates the efficiency of parallel genetic computation of the traveling salesman problem with periodic circular migration while applying parallel variable mutation rates. This paper thus aims at measuring the efficiency of the novel migration strategy proposed for cluster computers, while considering variable and constant mutation rates.

The authors also measure the impact of the scalability of the application and the parallel machine size over the efficiency of the parallel system. They wrote this paper with the intention of comparing their results with those of other migration strategies, hence the paper also give a summary of these comparisons, and proves that the novel migration strategy is indeed effective. [3]

2.3.2 Novel Contribution

The novel contribution of this paper is the bi-directional circular migration strategy. The authors mention that the motivation behind having a ring topology is that having a larger diameter for the ring means a greater diversity, that is the individuals get spread across and mix evenly.[3] A process Pi shares its individuals only with process (Pi+1)modN and receives only from (Pi-1+N)modN, where N is the total number of nodes in the system. The number of evolution's after which the migration happens and the number of individuals sent over are fixed in advance.

When a node receives some individuals, it replaces its weakest individuals with the newly received ones. This paper also introduces the novel concept of elitism, which means that a certain percentage of individuals which are the fittest or elite are not replaced. [3]

2.3.3 Useful ideas

The main useful ideas which we took from this paper were the modifications to the migration strategy performed on the nodes arranged in ring topology. This papers bidirectional migration approach is what we have used. We have also implemented the idea of elitism as introduced in this paper.

3 Implementation

We have implemented both sequential and parallel implementation using Parallel Java 2 library. This section describes our sequential and parallel design of the genetic algorithm. For both sequential and parallel design we have kept the population size of each generation as 30.

3.1 Sequential program: Design and Operation

The sequential genetic algorithm which we have used is as follows (refer figure 4)-

Step 1 (Initial tour): Initially tours are selected in random from the given cities and we form an initial population. The initial generation comprises this population.

Step 2 (Start GA): With the formed population, the stages of genetic algorithms will be performed.

Step 3 (Fitness function): Euclidean distance of each tour in the population is evaluated. All the tours in the population are sorted using the fitness function.

Step 4 (Selection): The selection process consists of 2 steps as follows

1. The first 20 percent from the sorted order of tours is directly selected for the next generation (enforcing elitism) i.e. 6 tours out of 30 tours are directly selected in our approach.

2. The probability value all the 30 tours have to be calculated using the formula. The sampling value for each tour has to be found using the formulas mentioned in section 2.

Step 5 (Crossover): Step 5 will be running half the times to the number of tours in perIterPopulation. Ordered crossover will be performed as the crossover method in all the iterations. The newly formed tours by crossover will also be copied to next generation

Step 6 (Mutation): 5 percent of the randomly selected tour in perIterPopulation will be mutated and copied for next generation.

Step 7 (End): If all the iterations are completed, step 8 will be continued If there are some more iterations left, step 2 will be continued

Step 8 (Stop): The best tour that has the smallest Euclidian distance among the population in final generation will be shown as the result.

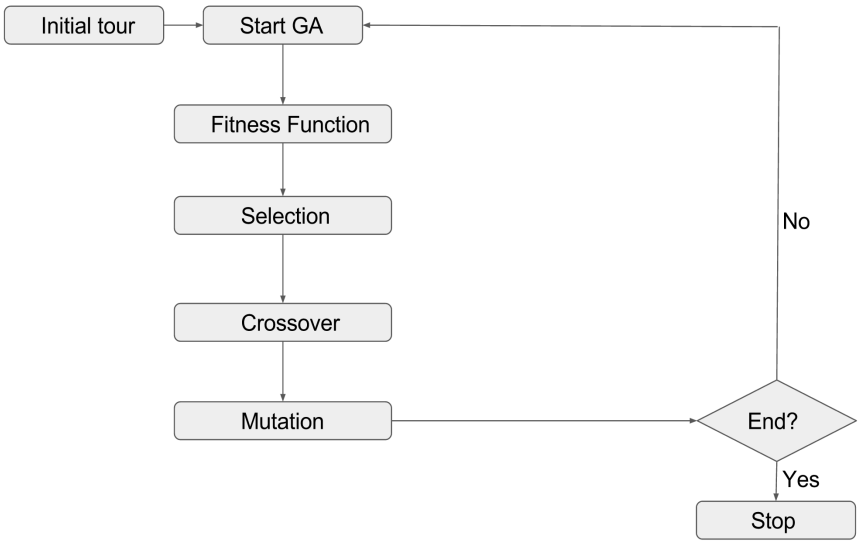

Figure 4: Inter-node migration scheme

3.2 Parallel Program: Design and Operation

The parallel genetic algorithm we have implemented is an island model with migration after certain generations. The parallel design is a cluster parallel design where each node is an island. Inside a node, genetic algorithm would be

running independently in each core. Figure 6 shows the working inside a node. Inside each core all the stages in genetic algorithm will be running independently. But after certain number of iterations all the cores will do a migration with one of the other cores i.e. if suppose there are four cores in a node, tours in core 0 will migrate to core 1, tours in core 1 will migrate to core 2, tours in core 2 will migrate to core 3 and tours in core 3 will migrate to core 0.

The genetic algorithm would stop after completing the predetermined number of evolution's in all the cores. Our complete design for parallelizing genetic algorithm is explained in figure 5, which is a cluster parallel design. All the nodes run independently. The above-explained parallelization within cores happens inside each node. After a certain number of iterations inside each node, the best 6 tours inside a node (after comparing all the cores) will be transferred to the next node. The next node will be identified in the same way how next core is identified with a node. The migration in our design happens in two levels were first level migrations are an intra-node migration and second level migrations are an inter-node migration. After all the iterations, the best tour will be sent from all the nodes to front end node. The front end node will reduce the best tour out of all the tours to result tour.

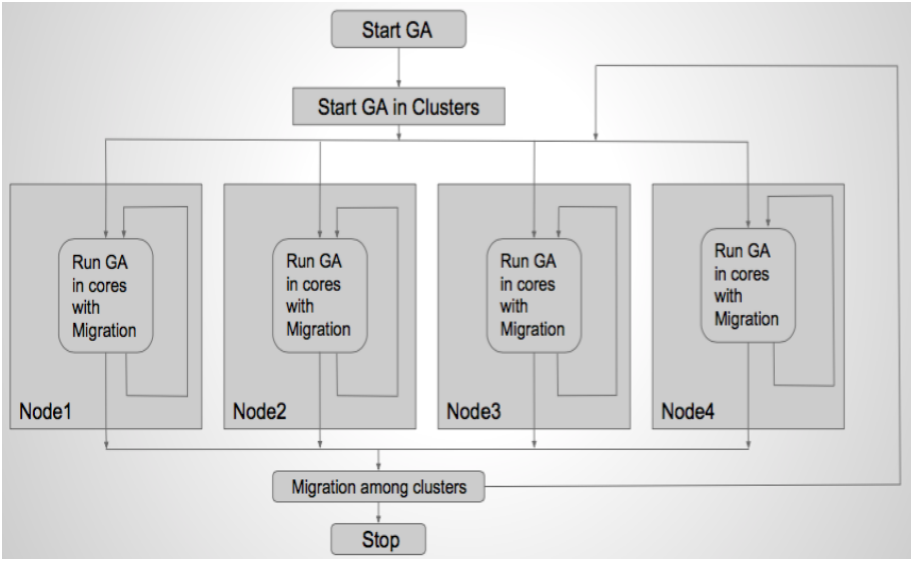

Figure 5: Inter-node migration scheme

3.3 Developers Manual

The program for sequential and parallel versions of our project have been tested on the Computer Science departments cluster computer - TARDIS. Following are the steps to compile the source code for sequential version of the project -

1. order to compile the source code with the pj2.jar, execute the following commands to set the necessary class paths -

   ```
   export CLASSPATH=.:/var/tmp/parajava/pj2/pj2.jar
   ```

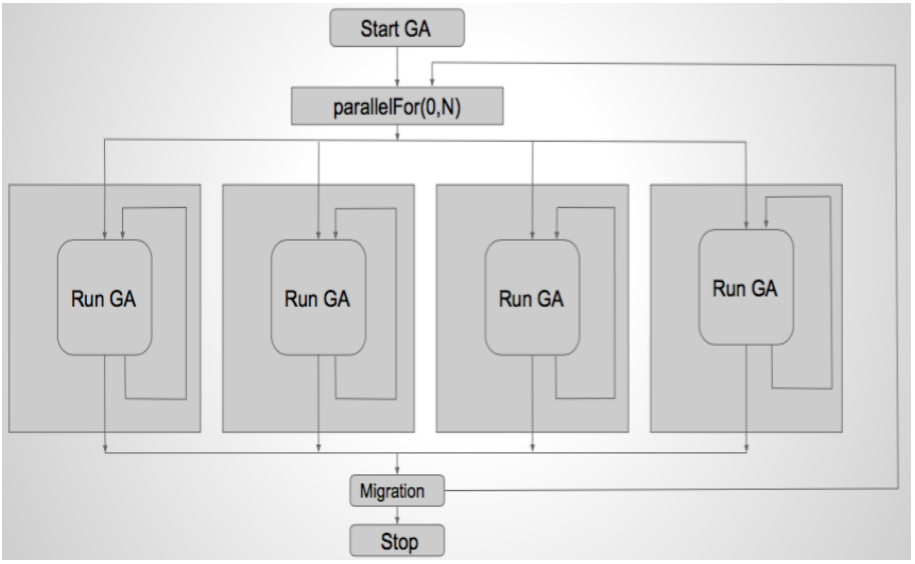

Figure 6: Intra-node migration scheme

```
export PATH=/usr/local/cuda/bin:$PATH
export LD_LIBRARY_PATH=/usr/local/cuda/lib:/usr/local/cuda/lib64
:/var/tmp/parajava/pj2:\$LD_LIBRARY_PATH
```

2. Put all the java souce files (City.java, Tour.java, Population.java, GeneticAlgorithm.java and SeqTSPGA.java) in one directory.
3. Compile all these files by running the following command - javac *.java
4. Create jar file by running the command - jar cf JarFileName.jar *.class

Following are the steps to compile the source code for parallel version of the project -

1. order to compile the source code with the pj2.jar, execute the following commands to set the necessary class paths -

```
export CLASSPATH=.:/var/tmp/parajava/pj2/pj2.jar
export PATH=/usr/local/cuda/bin:$PATH
export LD_LIBRARY_PATH=/usr/local/cuda/lib:/usr/local/cuda/lib64
:/var/tmp/parajava/pj2:\$LD_LIBRARY_PATH
```

2. Put all the java souce files (City.java, Tour.java, Population.java, GeneticAlgorithm.java and SmpTSPGA.java) in one directory.
3. Compile all these files by running the following command - javac *.java
4. Create jar file by running the command - jar cf JarFileName.jar *.class

3.4 Users Manual

3.4.1 Sequential version

Steps to run the sequential version of the code are

1. Create the jar for using the steps mentioned above for the sequential version
2. java pj2 jar= <jarFile>.jar workers=1 SeqTSPGA <inputFile.txt><E>

3.4.2 Parallel version

Steps to run the parallel version of the code are

1. Create the jar for using the steps mentioned above for the parallel version
2. java pj2 jar=<jarFile>workers=<K>SmpTSPGA <inputFile.txt><E><O><I>

where K is the number of worker, E is number of evolution's, O is rate of inter node migrations and I is rate of intra node migration

4 Performance

The performance of the parallel program can be expressed in terms of strong and weak scaling as follows -

4.1 Strong Scaling

We have measured the strong scaling performance on five different data sets viz 30 cities, 40 cities, 50 cities, 55 cities and 60 cities. For each of these data sets, the problem size was kept constant. The problem size in this case is the number of times the initial population is evolved with periodic migrations. This problem size was kept constant and the number of cores was increased. The speedup , efficiencies and executions times are can be seen in the table shown in figure 7. The graphs of execution time Vs number of cores and execution time Vs efficiency are also shown in figures 8 and 9 respectively and they show that we have achieved a good efficiency between 0.7 and 0.9.

Although we are achieving a quite good efficiency of about 0.8 to 0.9 on an average, it is not 1 and hence there is non-ideal scaling. The main reason for non-ideal scaling is the fact that some nodes finish their share of evolution before others and have to wait at the barrier while other nodes still evolve their population. Even though all nodes have the same size of initial population and run the same amount of evolution's, this problem is seen. The reason for this is, during crossover if the new child produced has cities present in the same order as an individual in the current population set then it is not treated as a new individual and simply discarded. Thus the number of such discarded duplicate children will be different in each node and hence the time needed for each generate a completely new population will be different.

This is the main reason behind different nodes finishing their share of evolution at different times and causing non-ideal strong scaling.

4.2 Weak Scaling

Weak scaling measures the sizeup of the problem. That is we increase the problem size as well along with the increase in the number of cores used to run the program. We have measured weak scaling for 5 different data sets - 30 cities, 40 cities, 50 cities, 55 cities and 60 cities. For each data set, we ran the program with 4800 evolution on one core and then increased the number of evolution's in direct proportion to the increase in number of cores. The detailed results of sizeup and efficiency are present in the table shown in figure 10. The figures 11 and 12 show the graphs of number of cores Vs program running time and number of cores Vs Efficiency respectively.

In weak scaling also since we are not getting efficiency of 1, it means that the scaling is non-ideal. The reasons for this are same as the ones explained in the strong scaling subsection. The main reason for non-ideal scaling is the fact that some nodes finish their share of evolution before others and have to wait at the barrier while other nodes still evolve their population. Even though all nodes have the same size of initial population and run the same amount of evolution's, this problem is seen. The reason for this is, during crossover if the new child produced has cities present in the same order as an individual in the current population set then it is not treated as a new individual and simply discarded. Thus the number of such discarded duplicate children will be different in each node and hence the time needed for each generate a completely new population will be different.

This is the main reason behind different nodes finishing their share of evolution at different times and causing non-ideal weak scaling.

5 Future work

The Traveling Salesman problem is a very interesting and old problem in the field of combinatorial optimizations. There a number of areas in this project where quality future work can be done to get a better understand of the problem. A few of them are as follows -

- **Migration strategies:** there can be a number of other innovative migration strategies to try out. Implementing some of those migrating strategies and comparing the accuracies and performance would reveal a lot about the nature of TSP and under which conditions a particular migration strategy should be used.
- **Crossover and mutation algorithms:** better crossover and mutation algorithms can be designed to produce more accurate results.
- **Massively Parallel approach:** this is another interesting approach which can be used to attempt a parallel solution to the TSP. This approach involves evolving separate populations on independent nodes with little or no migration between them. It would indeed be very interesting to compare the accuracy of the results we get from the massively parallel approach and the genetic algorithm with circular migration technique.

6 Learnings from the project

This project was a great learning experience in terms of programming on a cluster computer as well as the area of genetic algorithms. Before this project we were not aware of the fundamentals behind genetic algorithms. This project served as good case for understanding the fundamentals behind genetic algorithms and also how to implement one for the travelling salesman problem.

We also learned the fact that a lot of research can be done on a parallel approach to the travelling salesman problem. This is due to the fact that there are various approaches like massively parallel, genetic algorithms, master-slave etc. to solve this problem in the context of parallel computing.

Another subtle aspect that we picked up was regarding the migration rate. This factor needs to be varied before in order to find a value which gives us the best solution. However having a high rate of migration tends to be a communication overhead in cluster computing and reduces the accuracy of the result.

References

[1] Traveling salesman problem, wikipedia. `https://simple.wikipedia.org/wiki/Travelling_salesman_problem`.

[2] B. F. Al-Dulaimi and H. A. Ali. *International Journal of Mathematical, Computational, Physical, Electrical and Computer Engineering*, 2(2):123 – 129, 2008.

[3] P. Borovska, S. Bahudaila, and M. Lazarova. Island migration model with parallel mutation strategies for computing the traveling salesman problem on multicomputer platform. *International Journal of Computing*, 6(1):96–102, 2014.

[4] P. Pospichal, J. Jaros, and J. Schwarz. Parallel genetic algorithm on the cuda architecture. In *Proceedings of the 2010 International Conference on Applications of Evolutionary Computation - Volume Part I*, EvoApplicatons'10, pages 442–451, Berlin, Heidelberg, 2010. Springer-Verlag.

Problem size	Cores (K)	T (msec)	Speedup	Efficiency
30 cities	seq	70550		
	4	22608	3.121	0.78
	8	12424	5.679	0.71
	12	9165	7.698	0.642
	16	7115	9.916	0.62
40 cities	seq	184133		
	4	52826	3.486	0.871
	8	29185	6.309	0.789
	12	19985	9.213	0.768
	16	15283	12.048	0.753
50 cities	seq	389820		
	4	108884	3.582	0.895
	8	59208	6.584	0.823
	12	39916	9.766	0.813
	16	29969	13.007	0.813
55 cities	seq	609473		
	4	167053	3.648	0.912
	8	87385	6.975	0.872
	12	59680	10.212	0.851
	16	44920	13.567	0.847
60 cities	seq	856784		
	4	240854	3.557	0.889
	8	118524	7.22	0.903
	12	83767	10.228	0.852
	16	62589	13.689	0.855

Figure 7: Stong Scaling

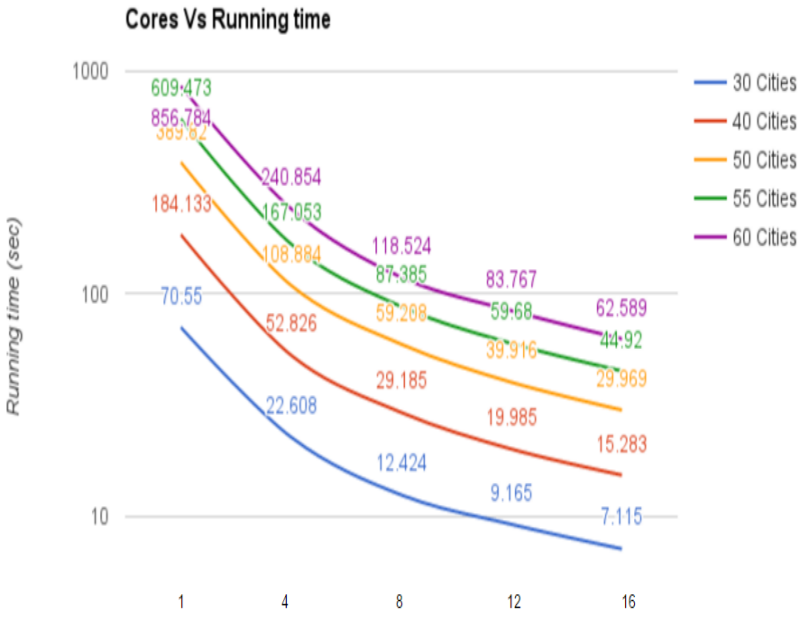

Figure 8: Cores Vs Running time for Strong Scaling

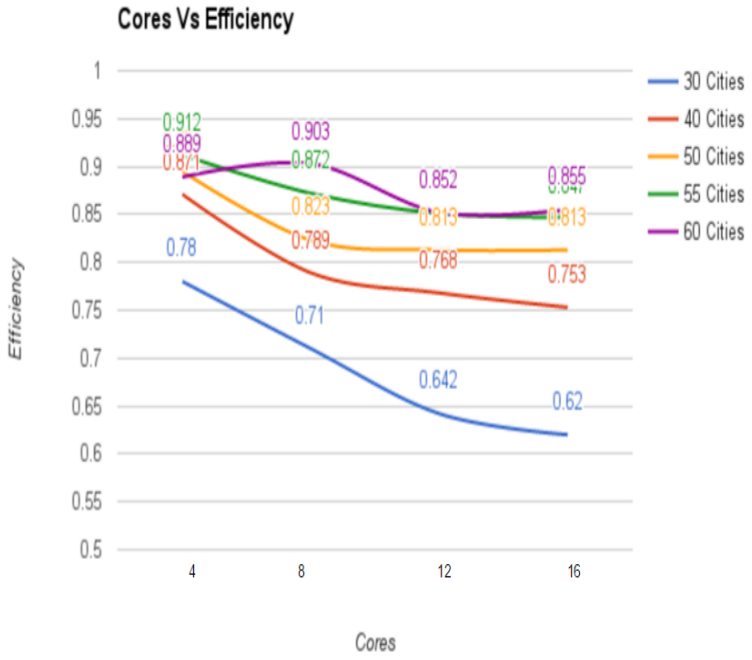

Figure 9: Cores Vs Efficiency for Strong Scaling

Cities	Problem size (iterations)	Cores (K)	Time (ms)	Sizeup	Efficiency
30	4800	seq	72988		
	19200	4	83048	3.515	0.87
	38400	8	84579	6.903	0.862
	57600	12	86846	10.085	0.84
	76800	16	87857	13.333	0.833
40	4800	seq	168709		
	19200	4	183868	3.67	0.917
	38400	8	185348	7.281	0.91
	57600	12	188315	10.75	0.895
	76800	16	189803	14.221	0.888
50	4800	seq	312025		
	19200	4	332088	3.75	0.939
	38400	8	353213	7.067	0.883
	57600	12	356849	10.492	0.874
	76800	16	358363	13.931	0.87
55	4800	seq	410536		
	19200	4	437632	3.75	0.938
	38400	8	453823	7.236	0.904
	57600	12	456609	10.789	0.899
	76800	16	477367	13.76	0.86
60	4800	seq	520780		
	19200	4	578925	3.598	0.899
	38400	8	590005	7.061	0.882
	57600	12	572378	10.918	0.882
	76800	16	596841	13.96	0.872

Figure 10: Weak Scaling

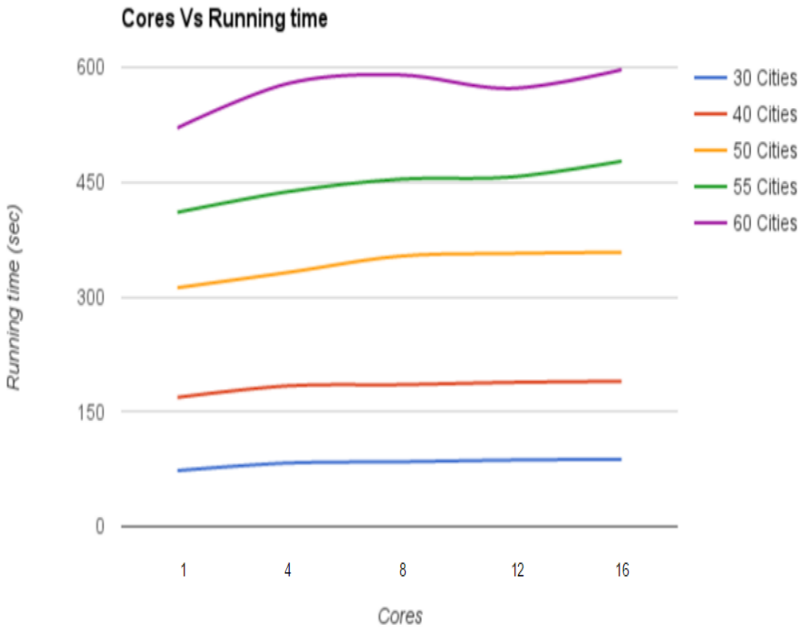

Figure 11: Cores Vs Running Time for Weak Scaling

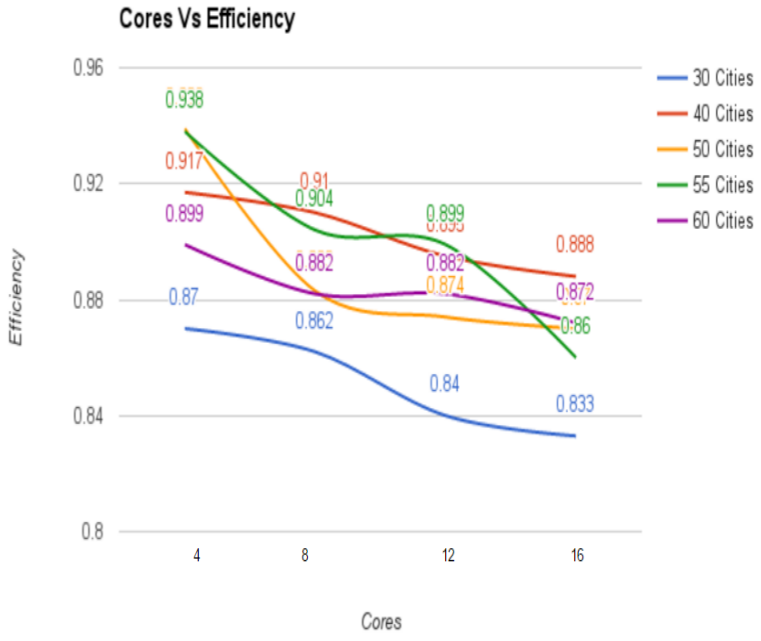

Figure 12: Cores Vs Efficiency for Weak Scaling