Parallel Genetic Algorithm for Traveling Salesman Problem

Project Report

CSE 6230 : High Performance Parallel Computing – Tools and Applications

Team Members

Harshit Mehrotra

Anuprem Chalvadi

Nishith Agarwal

1.0 PROBLEM DESCRIPTION

The **Traveling Salesman Problem,** or TSP for short, is one of the most intensively studied optimization problems in computational mathematics. Given a collection of cities and the cost of travel between each pair of them, the traveling salesman problem is to find the cheapest way of visiting all of the cities and returning to your starting point. We study and implement and analyze a simple version of TSP in which the *travel costs are symmetric* in the sense that traveling from city X to city Y costs just as much as traveling from Y to X and also the *graph is dense*, that is there is a way to go directly from any city to any other city.

There can be many interesting variations to TSP

a) **Asymmetric TSP :** In this problem the cost of going from city X to Y is not the same as going from City Y to X

b) **Sparse Graph :** It is possible that not all cities are connected to each other.

Genetic Algorithms (GA) belong to the larger class of evolutionary algorithms (EA), which generate solutions to optimization problems using techniques inspired by natural evolution, such as inheritance, mutation, selection, and crossover. The algorithm starts with a set of potential solutions and after successive stages of improvement, converge to an optimal solution. GA's are best suited for optimization and search base problems.

Why is Genetic Algorithm a good approach for TSP?

TSP is an **NP hard problem** and so there is no algorithm available to solve it in polynomial time on deterministic machine. Genetic Algorithm starts with some random initial tours (although in our approach we have not randomly generated initial tours but used some intelligent heuristic to design them) and work to improve these tours. The improvement techniques are drawn from biology and natural evolution and so the algorithm converges much faster as compared to O(n!) for a brute force technique. Also the solution does not deviates from the most optimal solution by more than 2-3%

1.1 GENETIC ALGORITHM GLOSSARY

Below is an overview of GA terminology in context of Traveling Salesman Problem

Global Population : A subset set of all possible solution of TSP i.e. subset of all possible permutations of cities

Sub-Population : A small subset of global Population

Fitness Function : A numerical measure of tour length. It is sum of distance between all neighboring cities in a tour + Distance between first and last city in tour (since TSP starts and ends at same node)

Mutation / Swap : Exchange position of two cities in a tour with aim of improving tour fitness

2.0 ALGORITHM DESIGN

Genetic algorithms provide an interesting potential for obtaining near-optimal solutions to TSP in short time. Our solution approach is inspired from [1] which presents insight for tour optimization at both global and local level (i.e. optimizing complete tour and parts of tour). We improved the solution proposed in [1] by using heuristics like **Nearest Neighbor algorithm** for global population generation rather than starting with random tours. Below is a brief overview of our approach:

- Generate an initial set of tours using *Nearest Neighbor algorithm.*
- Split this set into subsets each subset having a few tours, one subset for each MPI node
- On each MPI node create OpenMP threads, one thread for each tour
- Compute the fitness of each tour
- Divide a number of sub-tours of length less than or equal to 32.
- Improve each sub-tour on CUDA kernel using genetic algorithm for a fixed number of local iterations
    - CUDA kernel has 32 X 32 threads for each block.
    - **Swap mutation** - Thread (x,y) swaps cities at x and y indices respectively in the tour.
    - Accept this pair only if it increases fitness
- Combine these optimized sub-tours to form a complete tour.
- On each MPI node compute the two best tours
- Collect these two best tours from each MPI node, this form the *elite group*
- Use the *elite group* to regenerate a fitter global population using Genetic Algorithm
- Redistribute the global population and repeat above steps for a fixed number of global iterations.

Fig 1: Result of swap mutation

3.0 SYSTEM ARCHITECTURE

Divide the solution set on MPI nodes : An ideal approach is to consider a large number of potential tours for improvement. So we use a larger global population size and distribute it across MPI nodes for concurrent improvement. We construct initial tours *by starting from a different city* each time and moving to its nearest neighbor.

Spawn OpenMP thread for each tour : Operations on each tour at local level are independent of each other. So we parallelize local tour improvement by creating OpenMP thread for each tour.

Exhaust local search space using CUDA : To obtain the most optimal sub-tours, we execute a CUDA kernel in which threads within a thread block perform swap mutations improve sub-tours.

Fig 2. shows a visual representation of system design with the functions performed by each sub-system

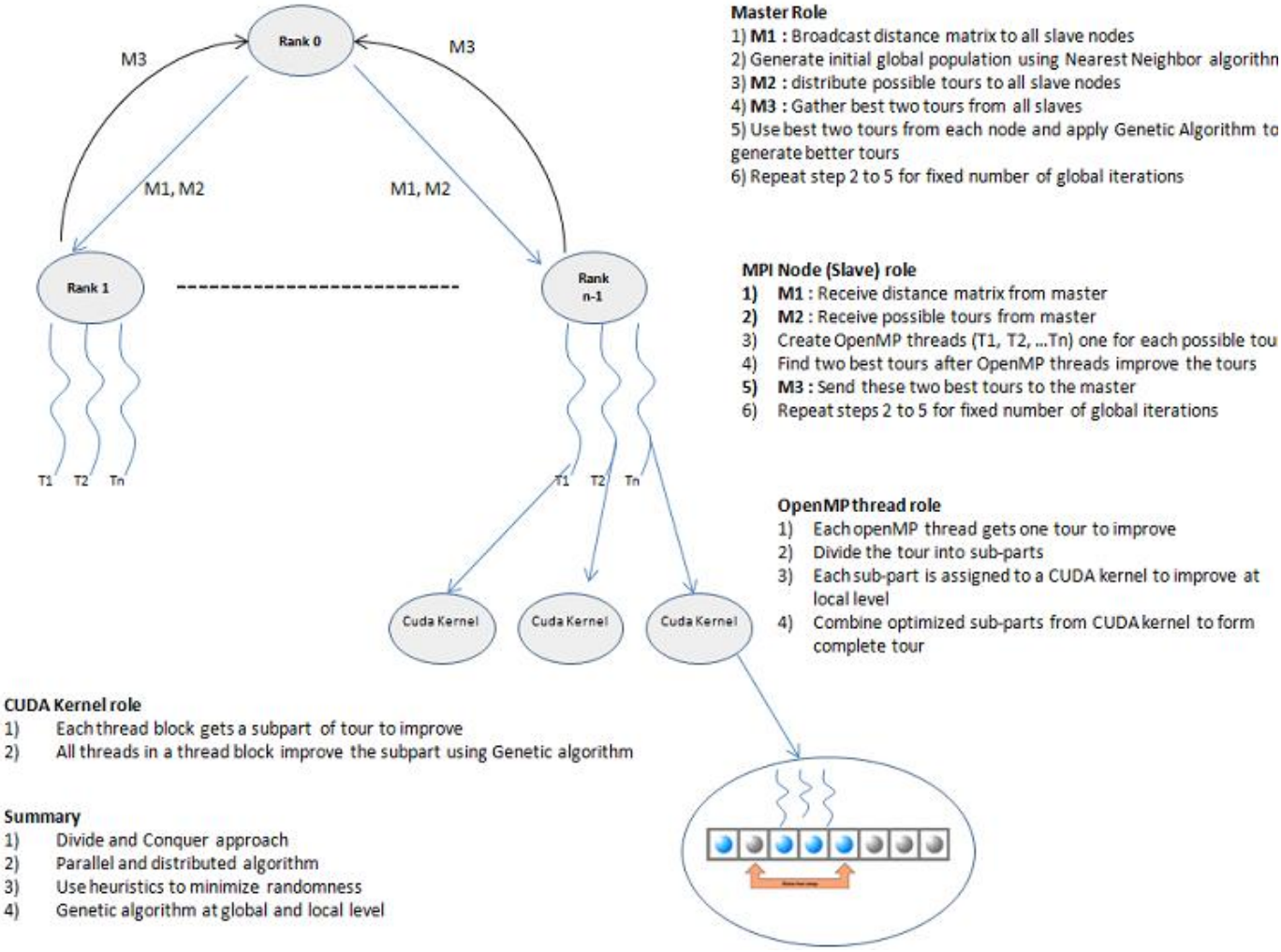

Fig 2: Visual representation of system design

4.0 MEMORY MODEL OF MPI BUFFERS

We analyzed two models for dynamically allocating buffers for message passing. These buffers are used to store the initial population (a set of tours) and city co-ordinates

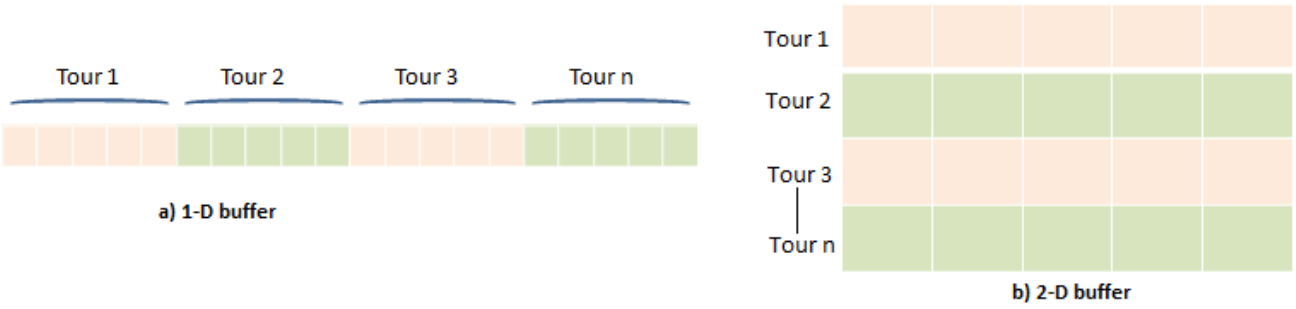

Fig 3: Memory model of set of n tours to be sent to each MPI node

a) 1-D buffers : All elements occupy *contiguous memory* locations and we can pass all tours in a single message.

Number of messages = 1 *Length of message* = n^2 *Total elements transferred* = n^2

b) 2-D buffers : The buffers are dynamically allocated and so rows of the matrix are *not contiguous*. We need one message per tour.

Number of messages = n *Length of message* = n *Total elements transferred* = n^2

Our implementation uses 1-D buffers so that *message length increases* while *number of messages decreases*. We chose to send one long 1-D array over a number of small arrays because it is quicker to send one large array compard to sending a number of smaller arrays.

5.0 EXPERIMENTS AND ANALYSIS

We have performed all the experiments on jinx cluster. The dataset for this project has been obtained from **TSPLib** (http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/tsp/), a library of sample instances for TSP consisting of *Euclidean coordinates* of cities and the length of most optimum tour. These optimum tours are the best known solutions (till date) for these problem instances and obtained using different algorithmic approaches. We compare the accuracy of our solution against these solutions.

5.1 WORK DEPTH ANALYSIS

In this section we present a broad overview of work and depth of proposed algorithm. Due to the nature of genetic algorithm, we can only specify the major components without diving into their minute details

Work

Total Work = Work done in communication + Work global and local level tour optimization

= Number of Global Iterations * (Work done in sending messages by
rank 0 + N units of operation across all OpenMP threads * (
L CUDA BLOCKS for each thread * (Number of swaps per thread Block)
)
)

Depth

As shown in the Fig 4, Rank 0 acts as the master node and all the remaining ranks act as slaves.

- Rank 0 sends across messages to other Ranks in a round robin manner. This marks the first transfer of control (to other Ranks) signifying depth = 1. (**Note** : Although the Sends happen in round - robin manner, they are overlapped with the execution on the ranks to which the Rank 0 has already sent the message , thus we consider it as depth = 1 and no more)

- Each Rank in turn spawns N number of OpenMP threads in parallel. The number of OpenMP threads spawned is equal to the number of Tours that each Rank receives from Rank 0 (after partitioning). All OpenMP threads can be spawned in parallel (although some threads can be re-used (sequentialized) thus not providing us full parallelism , we have ignored this case). So, this stage contributes to +1 depth, increasing our depth to 2. (simultaneously across all nodes)

- Each OpenMP thread in turn makes a CUDA Kernel call , spawning L CUDA Blocks. The number of Blocks depends on the size of the Tour (number of Cities) and number of blocks that can run in parallel over the all the SM's (multiprocessors) on Jinx. This step adds +1 to depth, depth = 3. (simultaneously across all nodes)

- Blocks run threads in Warps (32 threads). We have tried to avoid divergent branches by using simple reduction

techniques (avoiding 'else' statements). Again , all warps run in parallel, hence +1 depth , depth = 4. (simultaneously across all nodes)

- CUDA threads perform limited number of swaps and then let the block exit. This stage is overlapped with the running of Warps in the previous stage, hence depth = 4. (simultaneously across all nodes)
- The thread blocks return and synchronize. This is analogous to a return from a recursive call (after computation) so +1 depth, depth = 5. (simultaneously across all nodes)
- After thread blocks synchronize , the control return to the stalled OpenMP thread so +1 depth , depth = 6. (similarly for all other OpenMP threads simultaneously across all nodes)
- The thread control returns to the Ranks so +1 depth, depth = 7.
- Finally the Rank 0 (waiting asynchronously) receives updated values from all other Ranks so +1 depth , depth = 8.

This process happens for *I* global iterations. But we see the max depth (with possible assumptions as stated) = 8.

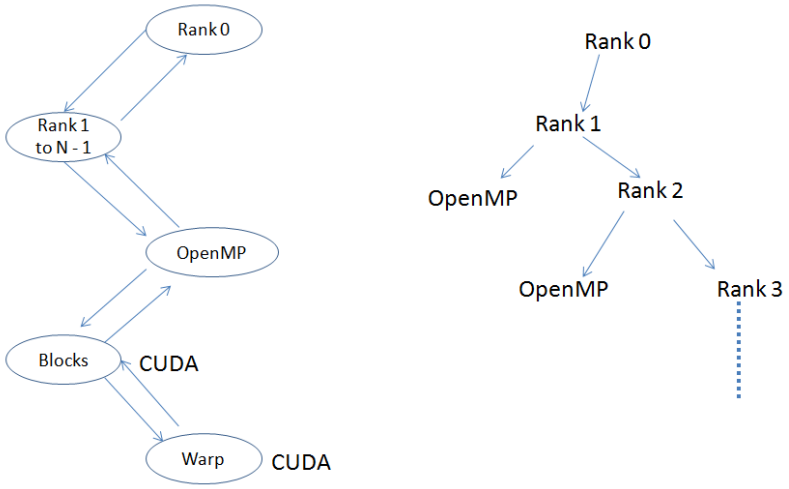

Fig 4 : Work and Depth of proposed algorithm

5.2 α - β MODEL

Time in terms of latency and bandwidth can be expressed as below where n is dimension of the task. (n = NUM_CITIES)

$$T = \alpha + \frac{n}{\beta}$$

In our proposed method we have implemented a star topology to send messages

$$\text{Total size of task} = \text{NUM\_CITIES} * \text{NUM\_CITIES}$$

$$\text{Width of message} = \frac{\text{NUM\_CITIES}}{\text{P}}$$

Thus, the bandwidth can be calculated as follows, where P is the number of MPI nodes

$$\text{Bandwidth} = \text{NUM\_CITIES} * \left(\frac{\text{NUM\_CITIES}}{\text{P} * \beta}\right)$$

$$\text{Latency} = \alpha \text{ units per processor}$$

$$T = \alpha + \left(\frac{NUM\_CITIES * NUM\_CITIES}{P * \beta}\right)$$

5.3 PARALLEL SPEEDUP

We measured the execution time of Serial Vs Parallel implementation on 10 MPI processes with varying number of cities from 48 to 666 for total 100 iterations. Our parallel implementation has 3 loops with the outermost loop iterating over total number of iterations (global * local). The two inner loops (i, j) iterate over the tour swapping cities at indices i and j and keeping track of the improvements, if there are any. In one of our later experiments we show that the proposed approach attains an accuracy of ~70% within 100 iterations. Hence we use iterations as a stopping condition in this experiment rather than accuracy. Since the algorithm has high degree of parallelism inherently, it can be easily seen from Fig 6 that parallelization increases speedup significantly. For larger problem sizes, the speedup is more pronounced since the sequential algorithm has time complexity of $O(n^3)$ and the utilization of the nodes is high in parallel algorithm.

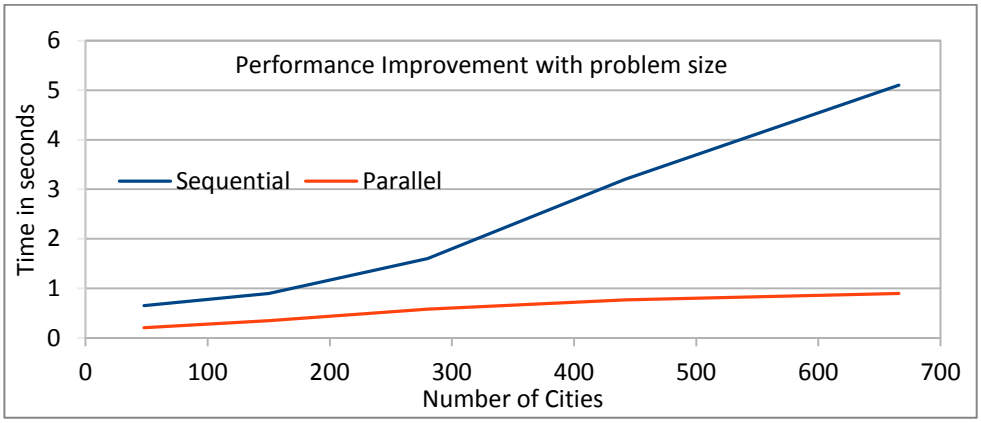

Fig 5 : Speedup - Time for serial Vs parallel execution

5.4 SCALIBILITY OF DESIGN

The proposed distributed algorithm has a scalable design which is shown by speedup in Fig 5. As the problem size increases, we can increase the number of nodes to perform the optimization with observable speedup. However the scalability of sub-tour optimization on CUDA is constrained by GPU resources like

- Number of threads per thread block
- Total size of shared memory for thread block etc

5.5 SIMD INTRINSICS FOR FITNESS CALCULATION

Fitness calculation attributes to significant computation in the program because the function is called frequently to evaluate a potential tour. We compared the performance of fitness evaluations using *SIMD intrinsic* with an ordinary implementation.

Observation: From Fig 6 we can see that SIMD giver better performance when number of cities is less. However since access to distance matrix is scattered and not contiguous, we found ordinary implementation to give better results for higher problem size. This is because SIMD implementation required extra memory read to store scattered reads into contiguous memory before loading in 128 bit registers. These extra memory reads

degrade the performance.

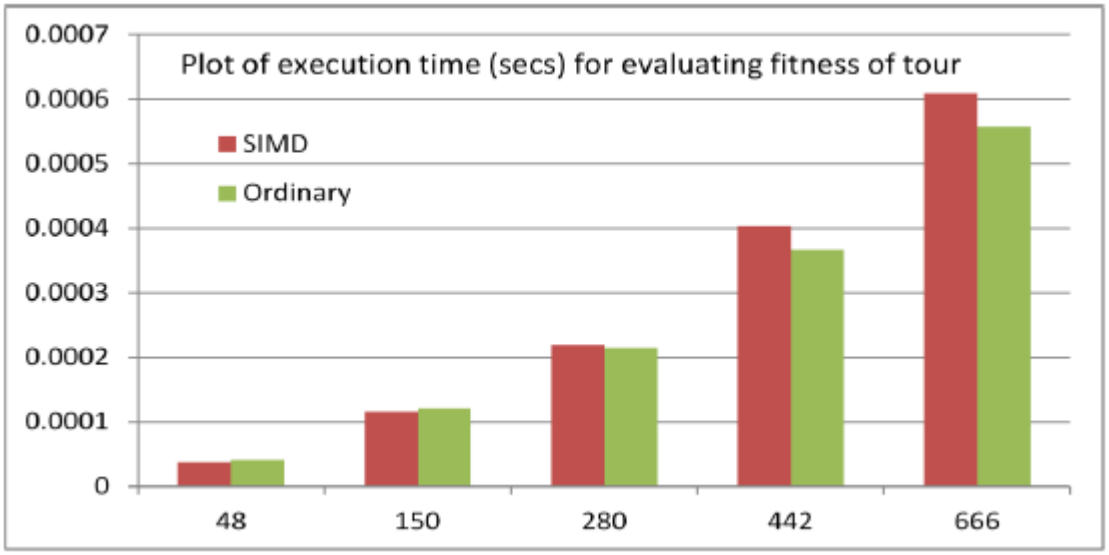

Fig 6 : Time taken for computing fitness using SIMD and ordinary implementation over 100 iterations

5.6 Parameter Tuning : Experiments with number of iterations, number of cities and MPI Nodes

In this section we present the experiments we performed and the analysis of the results we obtained. Note that we varied one parameter while keeping all other parameters constant to have a clearer understanding of the dynamics. In each section we mentioned only the parameters that are varied.

Iterations : In Fig 10 we show how accuracy varies with iterations. The solution attains an accuracy of 65-70% in ~100 iterations and irrespective of the problem size, this trend remains the same. This is because we improve tours at two levels:

a) At local levels we focus upon improving sub-tours

b) While optimizing the tour at global level, we keep common links in the elite group intact and generate the remaining tour using Nearest Neighbor. We observed that common links persist across all tours throughout the end resulting in slow increase in accuracy with further increase in iterations.

Number of Cities : We ran our experiments on number of cities varying from 48 to 666 keeping all other parameters fixed. In each case we compare our suggested tour with the most optimal tour given in library to evaluate performance. As shown in Fig 10, an accuracy of 70% is reached quickly within 100 iterations for all problem sizes. However, the increase in efficiency increases slowly with the number of iterations after that. This is because the chunks of tour that persist across all the tours are formed initially and improvement is much harder is to achieve after the chunks are formed.

MPI Nodes : The number of physical nodes on JINX which have a GPU is limited which restricted our experiments. Though we varied the number of MPI processes from 5 to 25 and measured the time to achieve 70% fit tour (Figure 9). It can be seen that the time taken decreases almost linearly with increase in number of MPI processes. This is because of increase in parallelization at MPI level instead of at CUDA level. The amount or work done by each MPI node is decreased and hence time taken decreases. We did not have sufficiently large data set to run the experiments the exhaust CUDA resources.

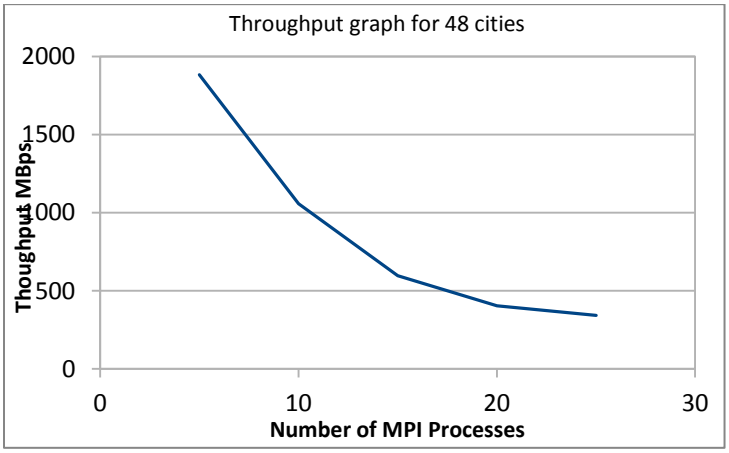

Fig 7: Throughput Vs No of MPI Process

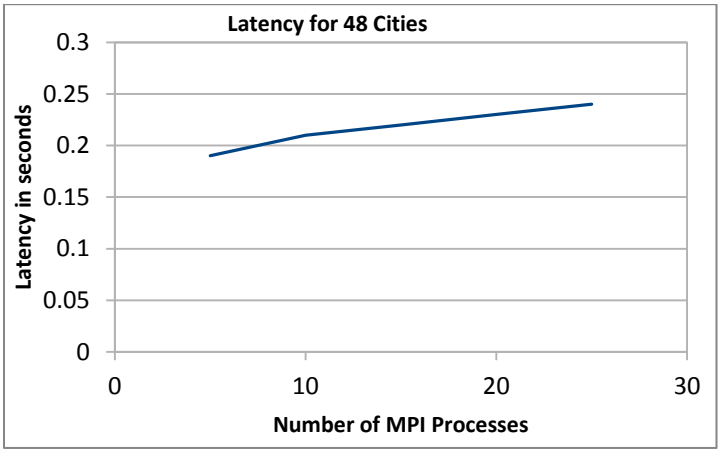

Fig 8: Latency Vs No of MPI Process

For a given problem size, when we increase the number of MPI processes the message size decreases and the number of messages increases. The message size decreases because the same work is being distributed among more number of processors. Since large number of smaller messages take more communication time as compared to small number of large messages, latency increase (observed in Fig 8) as expected. The time taken here is sum of communication and computation time

From Fig 7 we observe that throughput decreases with increasing the number of MPI processes. This is because while the problem size remains fixed, we transfer the same amount of elements in more number of messages which takes more time.

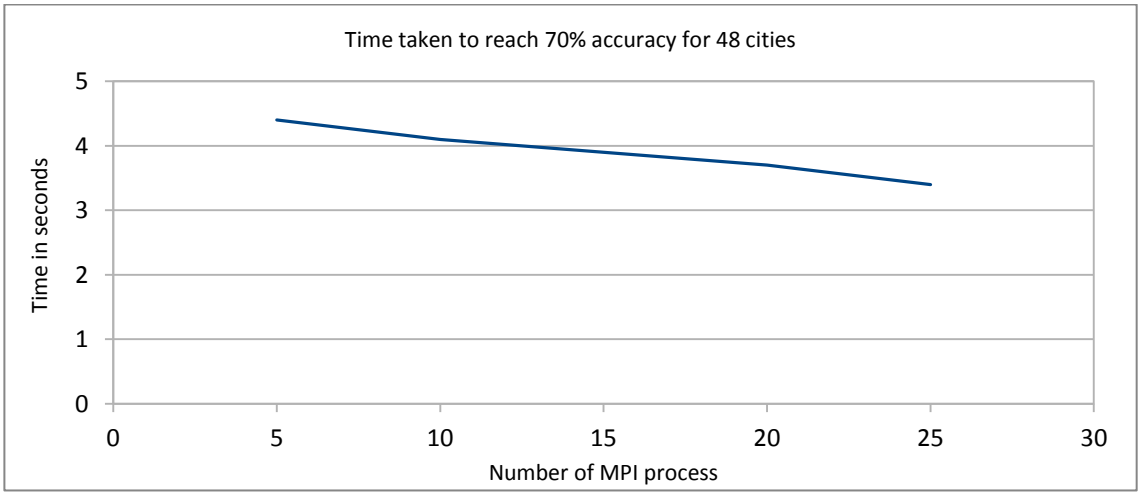

Fig 9: Time taken to achieve 70% accuracy with increasing the number of MPI processes

These results are easy to perceive, more the number of worker processes less is the time taken to achieve targets. Again in this experiment we define target as 70% accuracy. The time measured here is just the computation time.

5.7 CONVERGENCE OF GENETIC ALGORITHM

Finding the convergence point for Genetic algorithm is difficult to prove because of the iterative approach. The algorithm tries to improve solution without knowing what the most optimal solution is. In such a blindfold iterative approach, we experimented with following stopping criteria :

a) Execute fixed number of global and local iterations - We observe our solution design attains accuracy levels of 65 - 70% in very few iterations (~100 iterations : 10 global * 10 local) beyond which the improvement is very slow. High accuracy in initial iterations as compared with results in [1] is because of using Nearest Neighbor heuristic for constructing and improving global population.

b) Execute till we improve above a certain threshold - This approach did not work for us because achieving accuracy of more than 90 % almost always exhausted the wall time.

c) Execute till end of wall time - Did not work if the number of cities is too less because actual convergence happens much before wall time expires. Beyond executing this point wastes execution resources without improvement in performance.

Conclusion:

- Our choice termination condition gives satisfactory results given resource constraints (wall time).
- Using intelligent heuristics give a moderate accuracy in very few iterations.
- Our proposed solution attains this accuracy with a much smaller global population size (same as number of cities) as compared to experiments in [1] whose minimum population size is 480 for 48 cities.

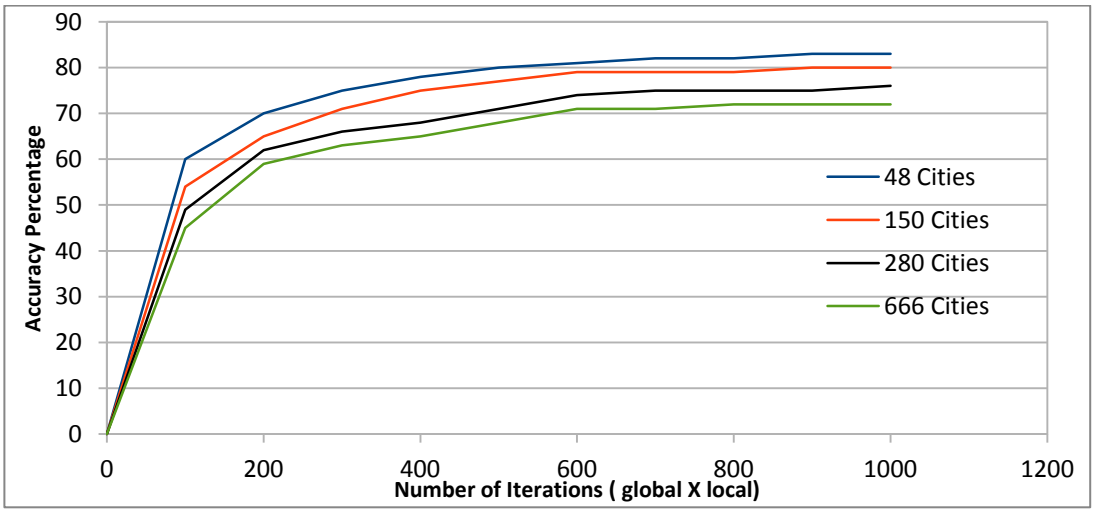

Fig 10 : How accuracy improves with iteration for Genetic Algorithm

6.0 ORGANIZATION OF CODE BASE

driver.c	Contains the main() function and the setup environment for the MPI nodes is defined here. Each rank calls the function which creates OpenMP threads for further optimization.
globalPopGen.c	Contains code for Initial Global Population generation, Global Population improvement and fitness calculation.
IndMPINode.c	Contains code for OpenMP threads which in turn call Cuda Kernel.
TSPCuda.c	Contains the code for Cuda processing of the route.
globalData.h	Defines all the static parameters like MPI nodes , number of cities etc.
readFromFile.c	Reads all the necessary data from the file TSPData.txt , computes the distance matrix

References

1) Massively Parallel Genetic Algorithm for Travelling Salesman Problem by Matt Heavner. http://www.cse.buffalo.edu/faculty/miller/Courses/CSE710/710mheavnerTSP.pdf

2) New Genetic Local Search Operators for the Traveling Salesman Problem, Bernd Freisleben and Peter Merz

3) Genetic Local Search Operators for the Traveling Salesman Problem - New Results, Bernd Freisleben and Peter Merz

4) Solving Traveling Salesman Problem on Cluster Compute Nodes by Izattdin A. Aziz, Nazleeni Haron, Mazlina Mehat, Low Tan Jung, Aisyah Nabilah Mustapa, Emelia Akashah Patah Akhir Computer and Information Sciences Department, Universiti Teknologi, Petronas.

5) Solving TSP problem by using Genetic Algorithm by Fozia Hanif Khan, Nasiruddin Khan, Syed Inayatullah and Shaikh Tajuddin Nizami International Journal of Basic & Applied Sciences IJBAS Vol: 9 No: 10

6) Datasets and other useful link: http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/tsp/

* * *