Team Research Investigation

Topic: Travelling Salesman Problem using Branch and Bound Algorithm

Team Members:
Pavan Prabhakar Bhat (pxb8715@rit.edu), Siddharth Tarey (st2476@rit.edu)

Introduction

The travelling salesman problem can be defined as a path in a directed/undirected graph *G* such that the path traversed includes every node exactly once with minimum cost. The travelling salesman problem is considered as an NP-hard problem. The most direct solution is, to try all permutations and obtain the most optimal solution, brute force implementation. The computational complexity of such a method would be O (n!) or O (2^n). This research investigation report discusses solution to the travelling salesman problem using one such brute force method called the branch and bound approach. The traditional branch and bound (B&B) algorithmic computations can be performed as follows:

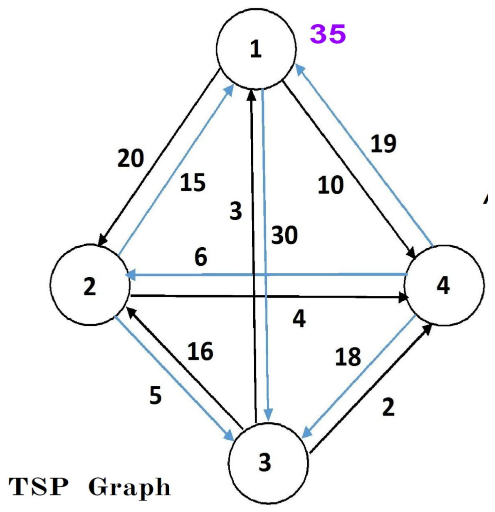


Fig1: A weighted directed TSP graph with start node = 1

In the above complete weighted graph, the travelling salesman problem needs to be solved using the traditional branch and bound algorithm where the starting node is 1. The reduced cost at node 1 can be calculated as a combination of subsequent row and column operations are to be performed. In order to generate a row reduced matrix, each of the rows in the adjacency matrix computes the least cost available on the row which are the numbers circled in red. After obtaining the row reduced matrix each of the columns is checked in a similar fashion in order to obtain the column reduced matrix. Here, it is important to notice that the rows and columns containing 0 are ignored for computation of the reduced matrices. Then these circles values with the shortest path from node 1 to node 1 is considered at depth 0 which has a cost of 0. Thus, the reduced cost value for node 1 can be computed as 35 as shown in the image illustrated below.

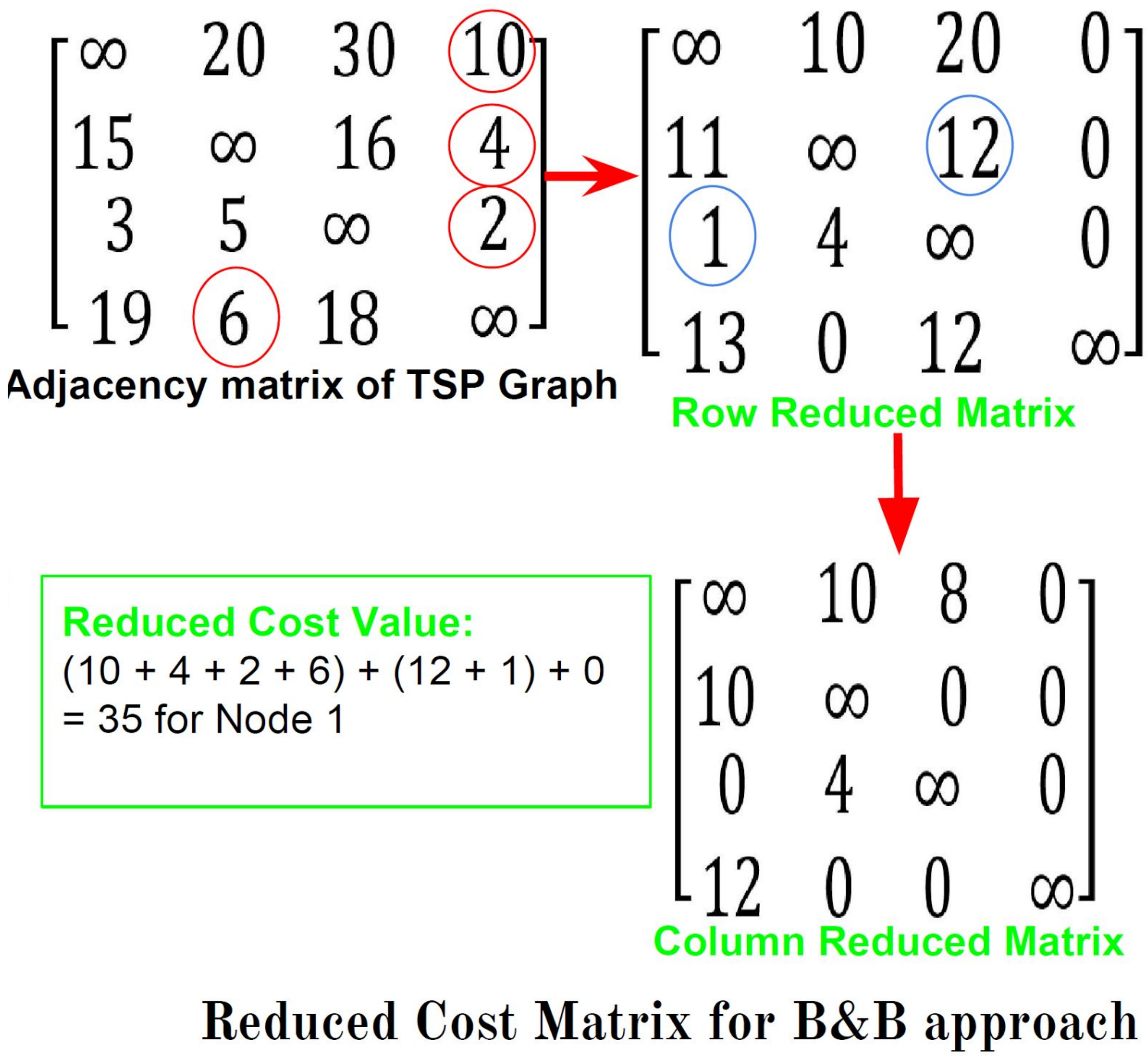


Fig2: Computation of Reduced Cost Matrix for a traditional B&B approach

This computation is performed at each of the node. A summation of this cost at each depth on each node such that every node is traversed once provides a path from the starting node in a cycle.

Analysis of the first paper:

[1] Discusses the best first search strategy for searching a graph. A branch and bound solution consists of constantly enumerating the states of a graph and pruning the states which are not desirable.

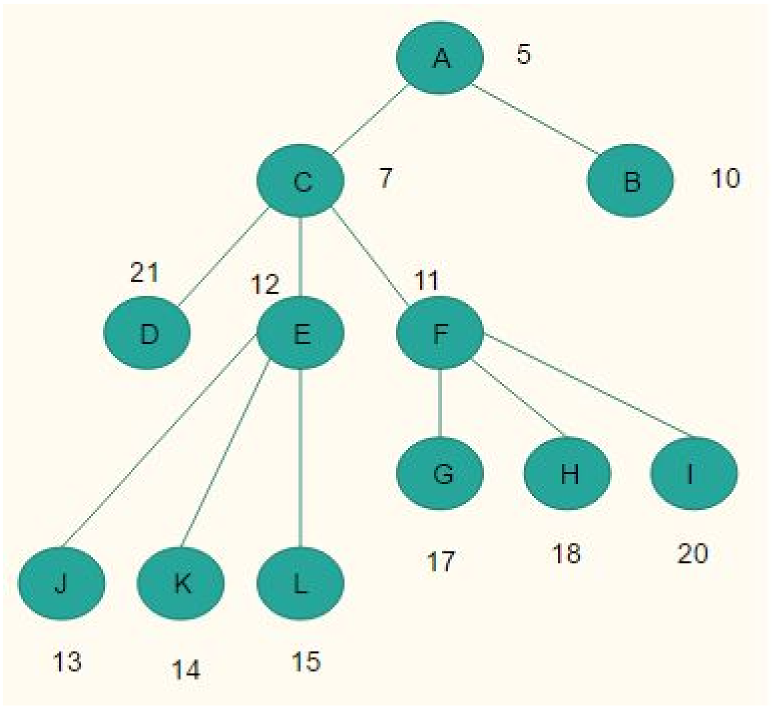


Fig 3: State space tree

Consider the state space in the above diagram. The first node to be explored will be the A as it is the root node consequently the children of A will then be explored. Out of the two nodes the node C has a lesser cost than node B. Thus the child that will be explored next will be the node C. The next node to be explored will be B, since it has a lesser cost than D, E and F nodes. Thus the best first search picks the node with best (least) cost.

The nodes to be explored can be put on a priority queue, with the node with the least cost at the front of the queue.

There are two strategies of parallelism for the branch and bound techniques. The first is a Node based strategy where the operations on the nodes are parallelized. The second is a Tree based strategy where the exploration of the state space tree is parallelized. This paper focuses on the latter. The exploration of the tree is done using a master slave architecture as shown below.

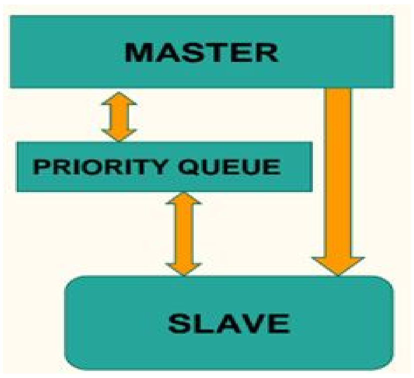


Fig 4: Master Slave Architecture

In this master slave architecture, the slave is responsible for exploring the tree, the nodes explored by the slave is then populated in the priority queue. The role of the master is to allocate tasks to the slave, assigning slaves the nodes to be explored.

Analysis of the second paper:

[2] introduces local search in conjunction with the branch and bound algorithm. The local search techniques that are used are *Variable Neighbourhood Search, Random Neighbourhood Search, and Multi-Neighbour search.*

Variable neighbourhood search incorporates searching the nodes according to some heuristic criteria. If while exploring the nodes the heuristic determines that the path is not optimal, the nodes are backtracks and explores other possibilities, this is similar to the best first searches.

Random neighbourhood search suggests to randomize the selection of the nodes. The randomly selected node will be explored and if it does not turn out to be an optimal path, that path is pruned and the random search then explores other possibilities by again randomly selecting nodes.

Analysis of the third paper:

[3] Introduces GPU to enhance the performance of the branch and bound algorithm. Once a node has been explored, it is passed to a GPU. therefore each initial node is given to a block in a GPU. The threads inside the blocks now will be used to explore the children of the nodes in a parallel way.

The GPU parallel execution can thus be summarized in the following way:

1. Explore the initial nodes of the search tree.
2. Send these nodes to the GPU blocks.
3. The GPU then explores the children of these nodes in parallel.
4. The most optimal solution will be reduced and sent back to the CPU from the GPU

Sequential Approach:

Traditional Branch and Bound

Given a directed or undirected graph, the branch and bound can be explained as follows:

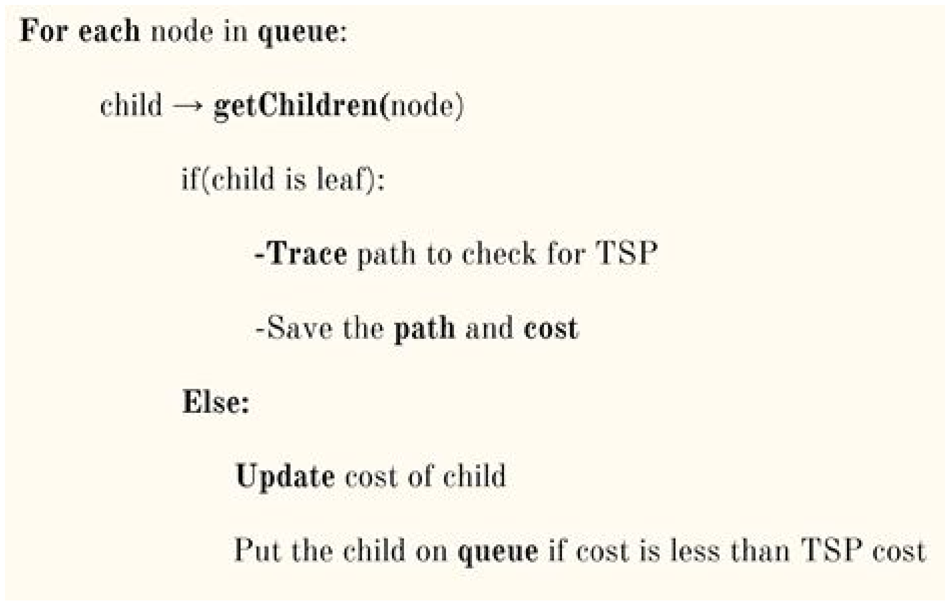
Select: Select a node in the graph based on the search criterion.

Branch: Explore the children of the selected node.

Bound: Prune the nodes based on a bounding criterion.

Repeat the above three steps till an optimal solution is reached or when until all the paths in the graph have been explored

Pseudo Code:



```
For each node in queue:
    child → getChildren(node)
        if(child is leaf):
            -Trace path to check for TSP
            -Save the path and cost
        Else:
            Update cost of child
            Put the child on queue if cost is less than TSP cost
```

The above pseudo code uses a queue to organize the state space node that is to be explored. For each node that is dequeued from the *queue,* the children of the node are explored. If there are no children nodes present it means that the node is a leaf node, and we can trace a path to see if the path satisfies the TSP. If it satisfies the TSP the *path* and *path cost* are saved. If the node is not a terminal node, the cost of its children are evaluated. If the cost of these children is less than the current TSP path cost, the children nodes are put on the queue else they are discarded. This algorithm finishes after all the possible paths are explored. This is because the current solution present may not be the most ideal solution, there can be other unexplored nodes that may contain a better path.

Parallel Approach:

There can be different sources of parallelism depending upon the strategy used for approaching the TSP problem such as a node based strategy or a tree based strategy.

A node based strategy is encompasses the lower level information i.e. computations that are performed within the node The second is a Tree based strategy where the exploration of the state space tree is parallelized. The tree based algorithm works very similarly like the sequential program. The only difference is that in the sequential algorithm there was only one thread that was enqueuing the state nodes on the queue, in the parallel algorithm there will be other threads that would be enqueuing and dequeuing from the work queue.

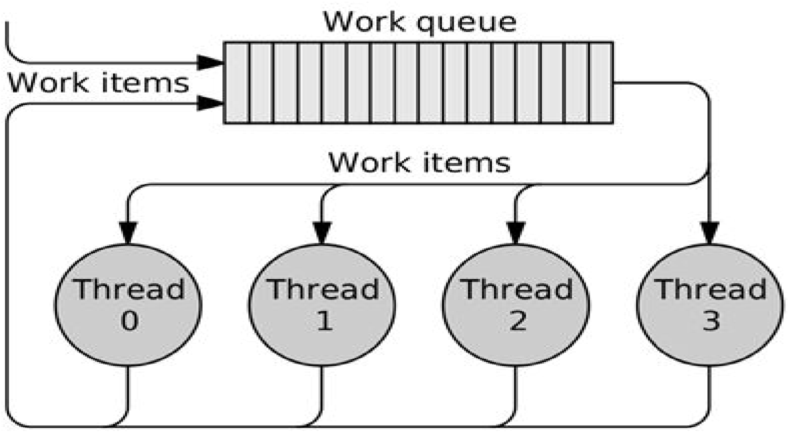


Fig 5: Parallel Work Queue *

Initially, during the early phases of our research investigation we had tried implementing the Master Slave architecture in order to provide for parallelism as a cluster parallel model.

Later, a multi-core implementation was sought for parallelism. As shown in the above figure, a multicore program will explore different paths simultaneously. Each thread will evaluate the cost of the nodes and update the *work queue.*

```
ParallelFor node in queue:
    child → getchildren(node)
        if(child is leaf):
            -Trace path to check for TSP
                -Save the path and cost
        Else:
                Update cost of child
                    Put the child on queue if cost is less than TSP cost
```

**Referenced from text book: https://www.cs.rit.edu/~ark/bcbd/*

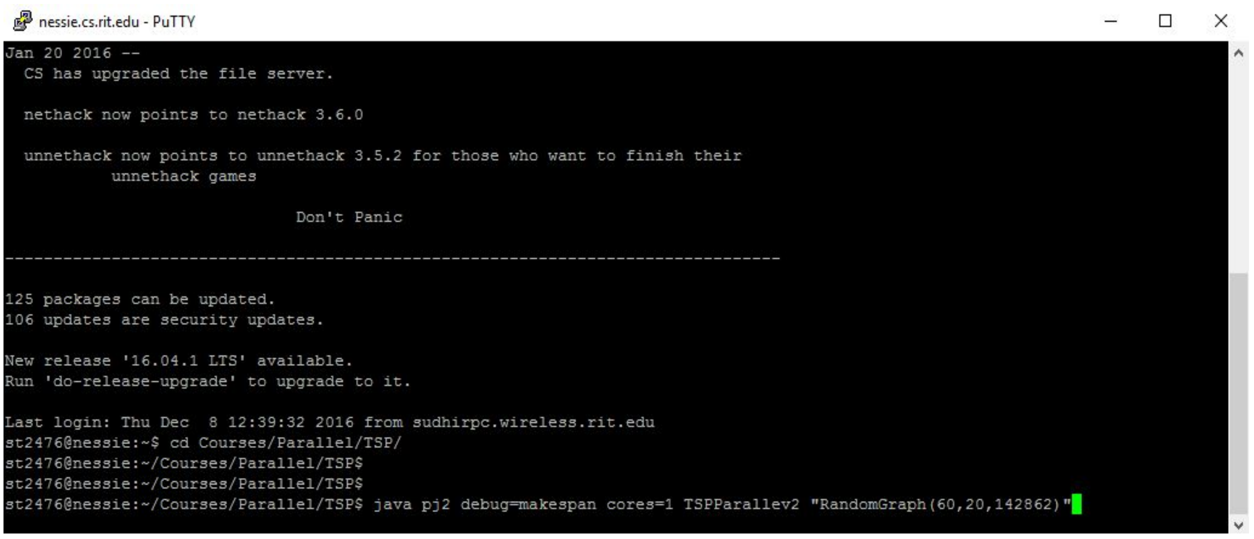
The pseudo code for the parallel is almost similar to the sequential algorithm, the only difference is that we use a *Parallel For* instead of a regular for loop.

Developer's Manual:

1. Open a terminal connecting to a multicore parallel computer. e.g *nessie or champ.*
2. Set the classpath of this terminal to the pj2 library using the *export CLASSPATH* command. e.g. *export CLASSPATH=.:/var/tmp/parajava/pj2/pj2.jar*
3. compile the program using using the javac command *e.g javac Execute.java*

User Manual:

1. Open a terminal connecting to a multicore parallel computer. e.g *nessie or champ*
2. Navigate to the folder where the class files are present.
3. Execute the file as given below



```
Jan 20 2016 --
  CS has upgraded the file server.

  nethack now points to nethack 3.6.0

  unnethack now points to unnethack 3.5.2 for those who want to finish their
           unnethack games

                              Don't Panic

--------------------------------------------------------------------------------

125 packages can be updated.
106 updates are security updates.

New release '16.04.1 LTS' available.
Run 'do-release-upgrade' to upgrade to it.

Last login: Thu Dec  8 12:39:32 2016 from sudhirpc.wireless.rit.edu
st2476@nessie:~$ cd Courses/Parallel/TSP/
st2476@nessie:~/Courses/Parallel/TSP$
st2476@nessie:~/Courses/Parallel/TSP$
st2476@nessie:~/Courses/Parallel/TSP$ java pj2 debug=makespan cores=1 TSPParallev2 "RandomGraph(60,20,142862)"
```

4. The *pj2* tells Java to use the pj2 library, *cores* specify the number of cores to be used while executing the program.

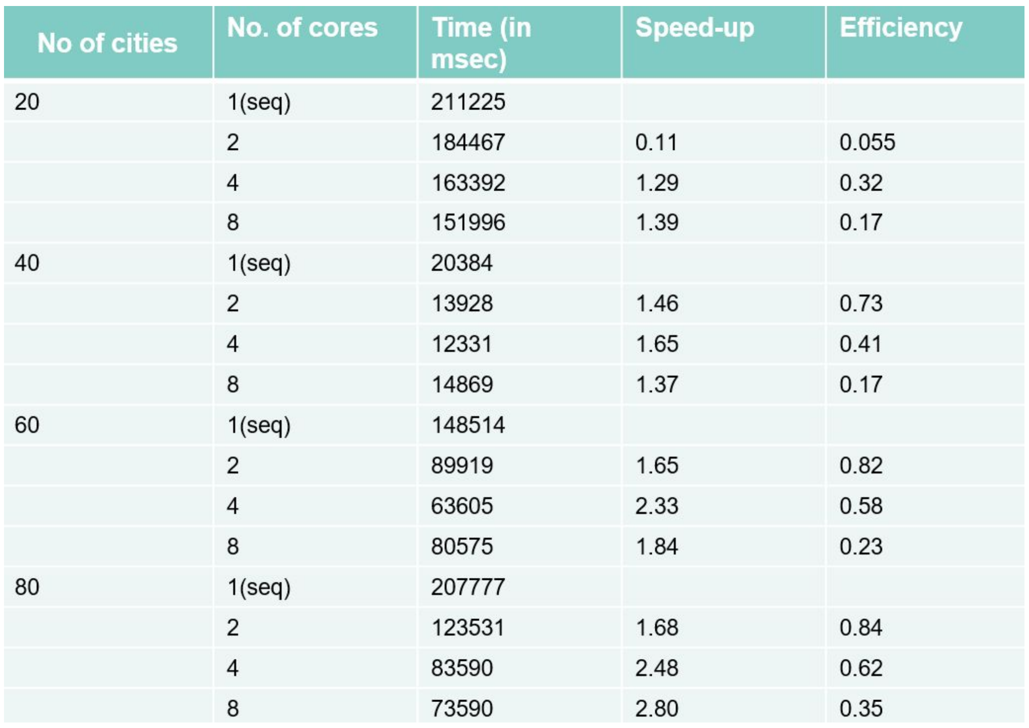
5. TSParallev2 specifies the class file that will begin the execution.

6. The program's command line argument is a constructor expression for a class that implements interface Graph. The program uses the Instance.newInstance() to construct a graph object.

7. The *RandomGraph* constructor takes 3 arguments. The number of vertices in a graph, the number of edges, and a seed to randomly generate edges between nodes.

Strong Scaling:

The strong scaling was measured using a graph having 20,40,60 and 80 vertices. The strong scaling performance was measured as below.



No of cities	No. of cores	Time (in msec)	Speed-up	Efficiency
20	1(seq)	211225		
	2	184467	0.11	0.055
	4	163392	1.29	0.32
	8	151996	1.39	0.17
40	1(seq)	20384		
	2	13928	1.46	0.73
	4	12331	1.65	0.41
	8	14869	1.37	0.17
60	1(seq)	148514		
	2	89919	1.65	0.82
	4	63605	2.33	0.58
	8	80575	1.84	0.23
80	1(seq)	207777		
	2	123531	1.68	0.84
	4	83590	2.48	0.62
	8	73590	2.80	0.35

Fig 6: Strong scaling metrics

The graphs were tested for their speed up on 1 core(i.e. sequential), 2 cores, 4 cores and 8 cores. The *Time* column represents the running time of the graph for each number of cores. The speed up is calculated using the following formula:

$$\text{Speed up} = \frac{Tseq}{Tparallel}$$

where *Tseq* represents the time taken for the program to run sequentially and *Tparallel* represents the time taken when the program is run parallely. The efficiency is calculated by the following formula:

$$\text{Efficiency} = \text{speedup}/\text{k}$$

where k is the number of cores used when running the program in parallel.

The graphical representation of the strong scaling is given below:

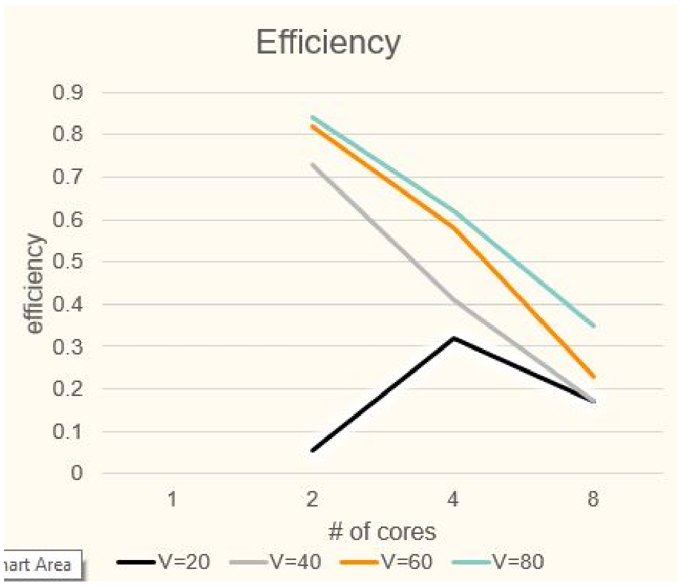


Fig 7: Efficiency vs # of cores

The graph shows that the efficiency decreases as the number of cores increases. The only case where the efficiency increases is when the number of vertices are 20.

Weak Scaling:

The strong scaling was measured using a graph having 20,40,60 and 80 vertices. The strong scaling performance was measured as below.

No of cities	No. of cores	Problem size	Time (in msec)	Speed-up	Efficiency
20	1(seq)	200	95		
	2	400	102	1.86	0.93
	4	800	3142	0.12	0.03
	8	1600	774996	0.01	0.01
40	1(seq)	400	110		
	2	800	109	2.01	1.00
	4	1600	209	2.10	0.525
	8	3200	900869	0.01	0.001
60	1(seq)	525	112		
	2	1050	110	2.02	1.01
	4	2100	98	4.57	1.14
	8	4200	12397	0.07	0.01
80	1(seq)	880	98		
	2	1760	118	1.66	0.84
	4	3520	119	3.24	0.81
	8	7040	17724	0.04	0.005

Fig 8: Weak scaling metrics

The graphs were tested for their speed up on 1 core(i.e. sequential), 2 cores, 4 cores and 8 cores. The problem size for each of the graphs were increased by increasing the number edges in the graph in proportion to the number of cores used. For eg. the problem size for a graph with 20 vertices was increased by increasing the number of vertices proportional to the number of cores.

The *Time* column represents the running time of the graph for each number of cores. The speed up is calculated using the following formula:

$$\text{Speed up} = \frac{Nparallel * Tseq}{Nsquential * Tparallel}$$

where *Tseq* represents the time taken for the program to run sequentially and *Tparallel* represents the time taken when the program is run parallely. The Nparallel and Nsequential represent the problem size for the parallel and sequential execution.The efficiency is calculated by the following formula:

$$\text{Efficiency} = \text{speedup}/k$$

where k is the number of cores used when running the program in parallel.

The graphical representation of the strong scaling is given below:

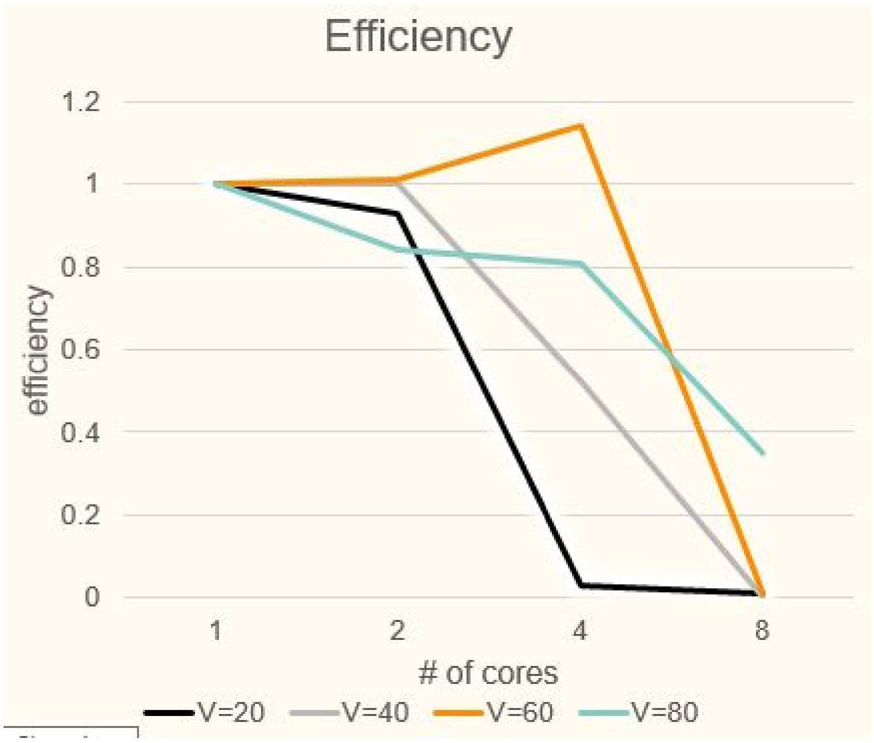


Fig 9: Efficiency vs # of cores

The efficiency generally increases when we move up to 4 cores, but as we move towards 8 cores the efficiency drops drastically.

The reasons for such anomalies for the strong and weak scaling are discussed in the next section.

Future Work

There can be several add ons to the current version of the implemented branch and bound algorithm in order to solve a TSP. One such Incorporation would be Hybridization schemes, where the Branch and Bound Algorithm is used in conjunction with Local Search Algorithms such as Variable Neighborhood Search, Random Neighborhood Search, Multi-Neighborhood Search etc. for pruning nodes and thereby avoiding unnecessary computations and branching which would significantly reduce the running time of the algorithm when higher number of cores are being used. Next would be to implement a GPU based approach in order to achieve a better speedup and to attain an expected strong and weak scaling.

Applications of TSP

A few applications which we found that could employ the usage of TSP would be Routing in Google Maps, drilling a Printed Circuit Board, Navigating a self driving car, Sonet Rings etc.

Learning

While implementing the sequential algorihm for the TSP problem we were able to interpret a lot of unnecessary computations being performed by the traditional method and thereby we tried to exploit different constraints in order to perform a bounding operation. During this exploration, we came across different strategies that led us to several techniques and heuristics for optimizing the computation of the TSP problem which could provide a speedup to the traditional implementation of the Branch and Bound Algorithm even after a sizeup. While scaling the output we noticed behavior of different graphs which introduced anomalies with the increase in the number of cores due to the bounding condition not being placed efficiently. Lastly, one other area which we felt could have improved is an efficient use of a data structure that could introduce an ease of operation by means of access at the node level.

References

[1]Crainic,T.G.,LeCun,B.andRoucairol,C.(2006)ParallelBranch-and-BoundAlgorithms, inParallelCombinatorialOptimization(edE.-G.Talbi),JohnWiley&Sons,Inc.,Hoboken,NJ ,USA.doi:10.1002/9780470053928.ch1

[2]Y.Jiang,T.Weise,J.Lässig,R.ChiongandR.Athauda,"Comparingahybridbranchandbo undalgorithmwithevolutionarycomputationmethods,localsearchandtheirhybridsontheT SP,"ComputationalIntelligenceinProductionandLogisticsSystems(CIPLS),2014IEEESy mposiumon,Orlando,FL,2014,pp.148-155.

[3] T. Carneiro, A. E. Muritiba, M. Negreiros and G. A. Lima de Campos, "A New Parallel Schema for Branch-and-Bound Algorithms Using GPGPU," Computer Architecture and High Performance Computing (SBAC-PAD), 2011 23rd International Symposium on, Vitoria, Espirito Santo, 2011, pp. 41-47.