A Parallel implementation of Travelling Salesman Problem

CS 622 : Varun Jindal, Krishna Tayal, Sunil Pratap Singh

1. INTRODUCTION

The travelling salesman problem is a well studied NP-Hard problem. Many people look for an efficient implementation of the same because an optimum and correct output can mean huge reduction in costs involved.

1.1 Project Objectives

This project intends to study the behaviour and bottlenecks of the parallel algorithm for TSP in a distributed environment. The bottlenecks to be analyzed can be network bandwidth, computation overhead during communications over network, algorithm inefficiency. Ideally the number of processes involved should be equal to the speedup achieved.

2. EXPERIMENT ENVIRONMENT

For the purpose of this observational study 10 lab workstations were used. Each PC has a 4 core processor with all 10 connected over wired LAN network. A common user "mpiu" was created on each PC. SSH connection was configured to be password-less and a shared directory "/mirror" was created on the server using NFS-kernel-server. All the clients mounted this shared directory using NFS-common in read and write mode. Whatever any system writes in the common directory, is visible to other systems mounting the directory. **Message passing interface(MPI)** was used for communication over network. Particularly **MPICH2** was used as MPI implementation. MPI uses SSH to communicate to other PC over LAN network. Also the process assignment is completely controlled by MPI itself. So it can't be controlled that which PC runs how many processes.

3. TRAVELLING SALESMAN PROBLEM

This problem is about finding minimum weight Hamiltonian cycle in a given graph. Goal is to find a cycle such that all vertices of the graph are covered. For this problem it is irrelevant that which node is source node. So we can choose any arbitrary node as starting point. After choosing the starting node, while traversing, lets say we are at node **j** with path named as **P**, we can choose among any of the neighbour nodes of **j**. Say we choose **i**, we again can choose any neighbour of that node (i.e **node i**,if not already visited). All the possible paths can be easily seen to form a tree. Thus this problem can be visualized as finding minimum weight path from root of a tree to leaf (with path weight including weight of edge from leaf to root).

4. ALGORITHM IMPLEMENTED

A sequential brute force algorithm would be searching all the possible **(n-1)!** paths, where 'n' is number of vertices in graph. A good way to speedup the execution is to use branch and bound technique. In this method minimum weight encountered so far is regularly updated by processes. It helps reject any sub-tree where weight of path already traversed is greater that the min weight already found. We have implemented a parallel algorithm that has following properties:

- No use of locks and barrier at all.
- Number of tasks created are $O(n^2)$, considering the fact that number of processes are O(n). This way, most of the times, processes will be busy.
- Size of the task is not very small, otherwise overhead of message passing will exceed actual computation cost.
- Minimum use of message passing.
- Processes are not dependent on each other.

So we came up with the an algorithm that uses static partitioning of the load but distributes them dynamically among the processes. Algorithm follows branch and bound paradigm. Each process calculates the minimum cost a sub-tree assigned to them by the "master" process. Notation of task is a sub-tree rooted at second level from top of tree. For root **O**, it would mean a sub-tree rooted at **j** where path traversed so far would be **O -> i -> j**). This was done so that we have enough tasks to distribute **(i.e (n-1)(n-2) tasks)**. Algorithm is as follows:

- For a input graph of **n** vertices and **K** available processors(that is it contains n number of vertices), master process creates (n-1)(n-2) tasks. These many tasks are available for distribution among K-1 processes (master is for book-keeping)
- There are (n-1)(n-2) tasks because in the tree that needs to be traversed, traversal by each process starts from second level of tree where we have this many sub-trees.
- Master assigns tasks to all the processes and whenever one task is finished by a process, it requests master for another task.
- Upon receiving request for task assignment from process i, the master assigns next available task to the requesting processor.

- In the beginning of algorithm a valid weight value is initialized as **minCost** by each process using greedy approach.This is done hoping the initial value to be near the optimum value, so that more number of sub-trees can be rejected. It is ensured to be a valid path weight.

- Whenever a lower weight path is encountered the master is notified about the value. Every process asks for updates of **minCost** after fixed number of traversals so as to use a lower value to reject more number of sub-trees.

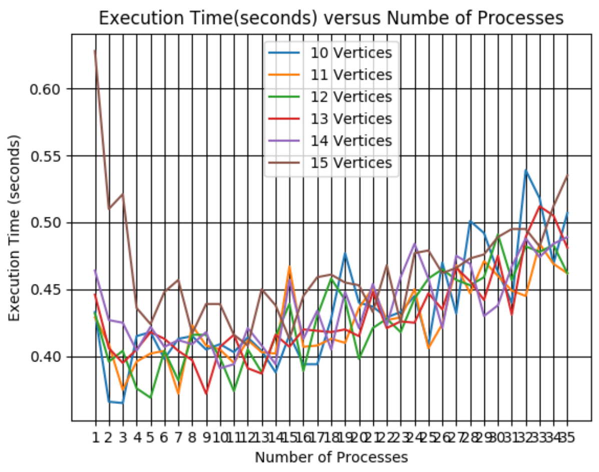
- When the process finishes it asks for another task, if tasks are available the master assigns the task to requesting process and communicates the **minCost** value found so far by earlier processes to the requesting process.

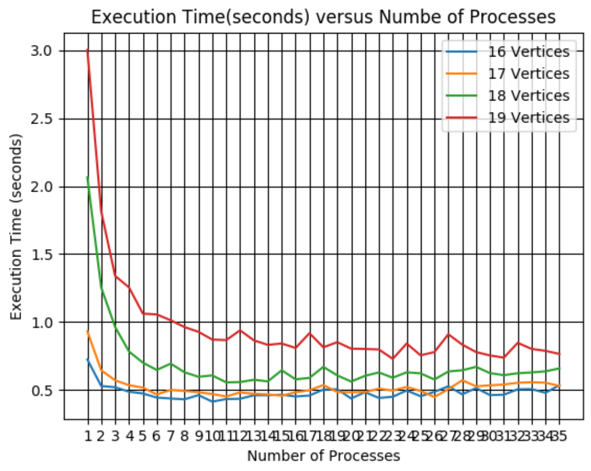
- When no tasks are remaining, processes start to exit. Whenever the last process exits, master finalizes the result and exits the program after printing the results to output.

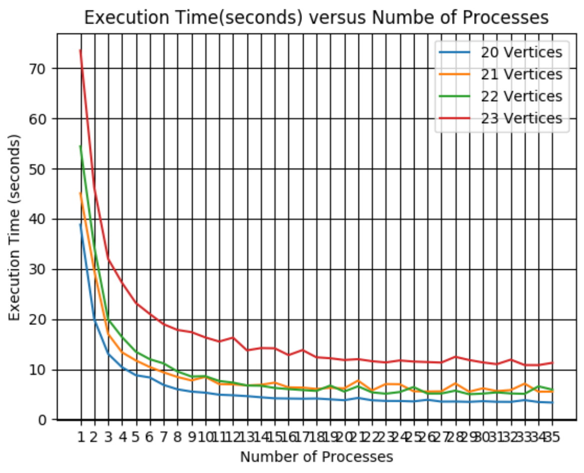
5. RESULTS

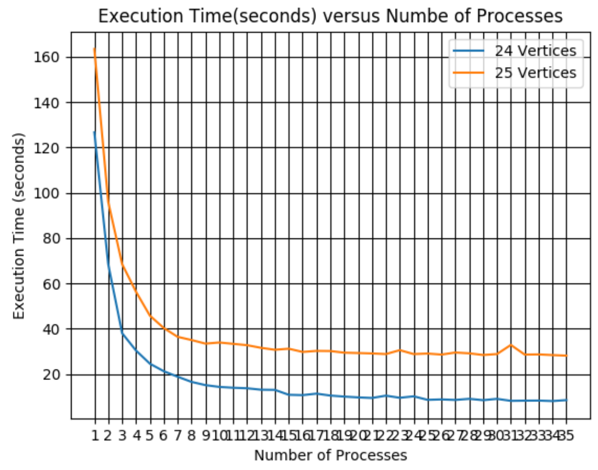
The following plots show the time spent in execution of the program. It was measured using the **time** utility of linux-kernal to measure program execution time. Following plots were obtained. As the master is always idle and doesn't traverse any path, we have used 9 PCs for actual traversal, thus we have gone up to 36 processes for each input graph. Input graphs were generated randomly.

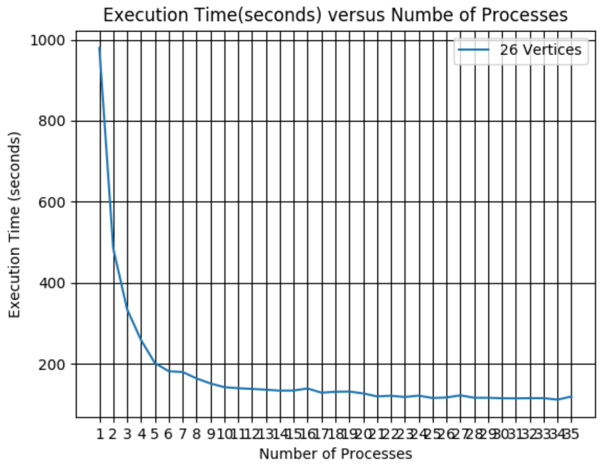
5.1 Time versus Number of processes for Complete graphs











In these figures, we can see that

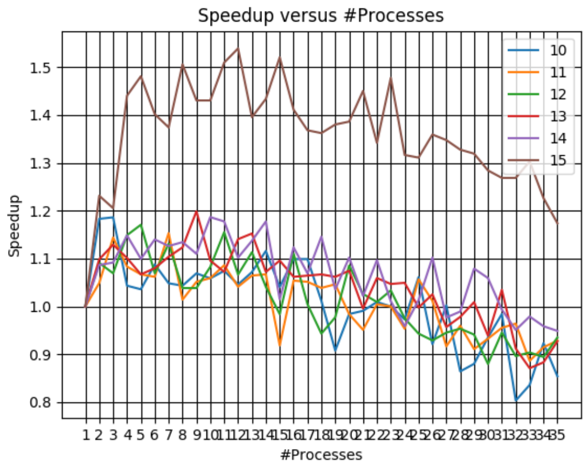
- For n <=15 time increases with number of processors after initial decrease, overall an increase after crossing certain number of processes is clearly visible.
- For n>15 up-to n=19, speedup is observed with minor fluctuations.
- For n>19, each task is big enough so constant decrease is observed.

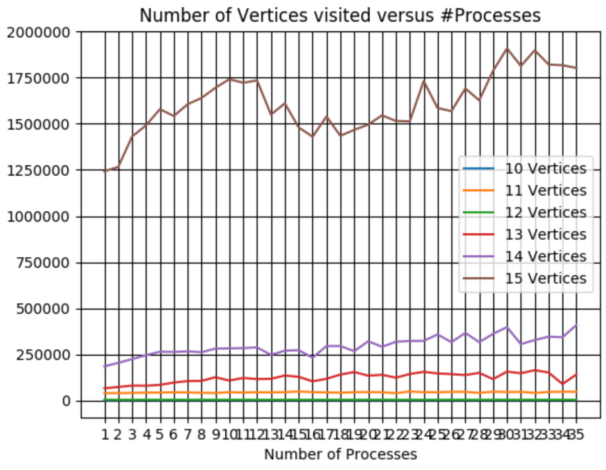
5.2 Speedup obtained and Nodes visited

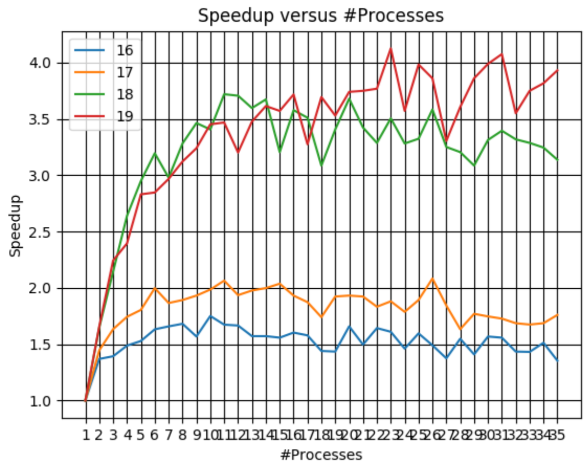
In this section, we are presenting two types of plots. One is for speedup obtained. Second one is for number of vertices visited by algorithm while calculating min cost. If number of vertices visited increases then execution time of algorithm also increases. Using these plots, we can prove the fact that ideal speedup cannot be achieved.

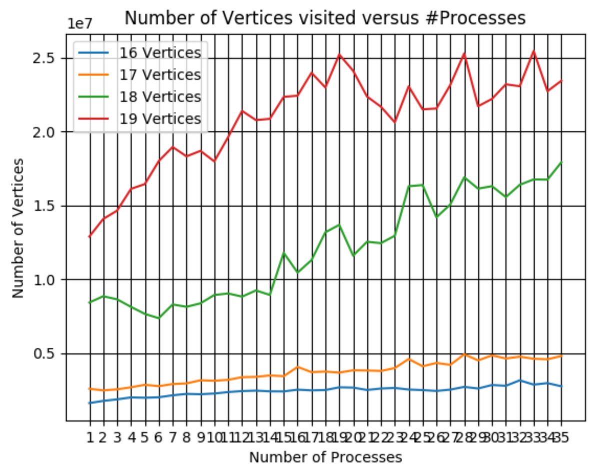
With increasing processes it can be observed that the number of total tree vertices visited is increasing, this is because of the fact that as the number of processes increase many parallel processes traverse along path that would have been rejected in sequential implementation. In parallel implementation by the time one process traverses a path with cost **W**, another process has already traversed a path with weight more than **W**.
As more work is performed speedup gets affected.

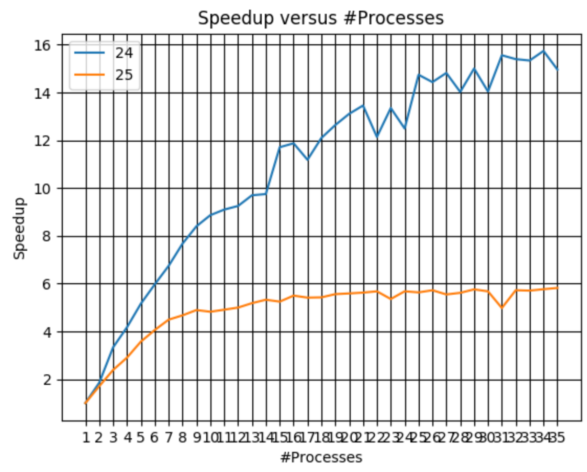
You will see following plots in alternate order. First one shows speedup and second one shows work done by algorithm for same graph used in first plot.

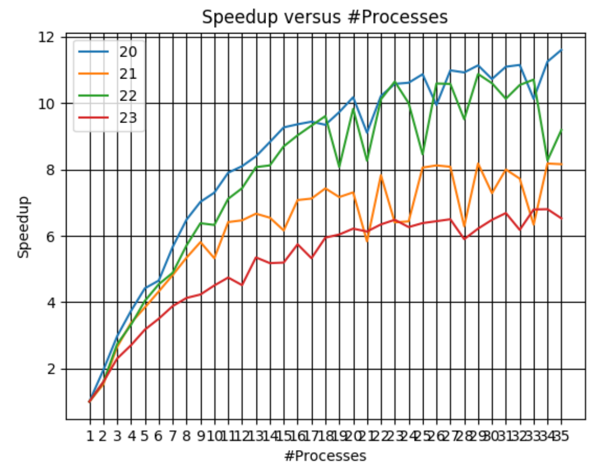


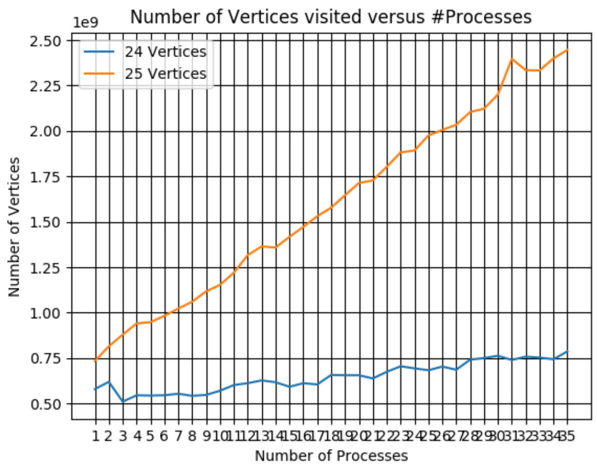


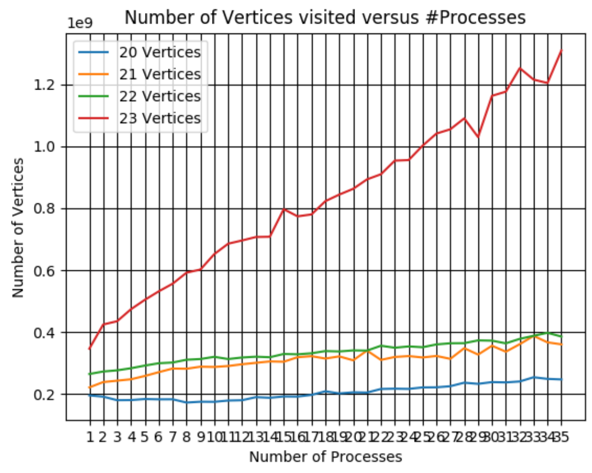


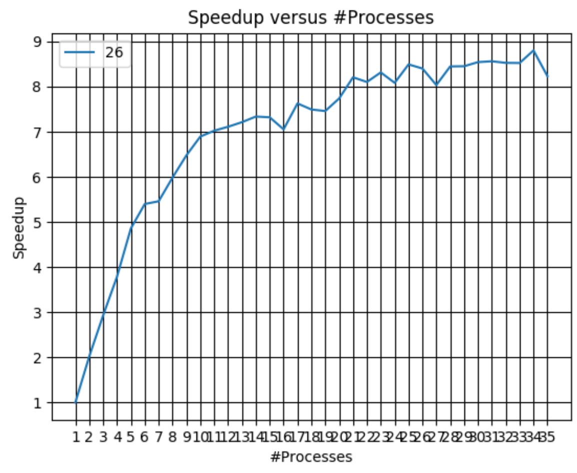

Number of Vertices visited versus #Processes
1e7
16 Vertices
17 Vertices
18 Vertices
19 Vertices
Number of Vertices
Number of Processes

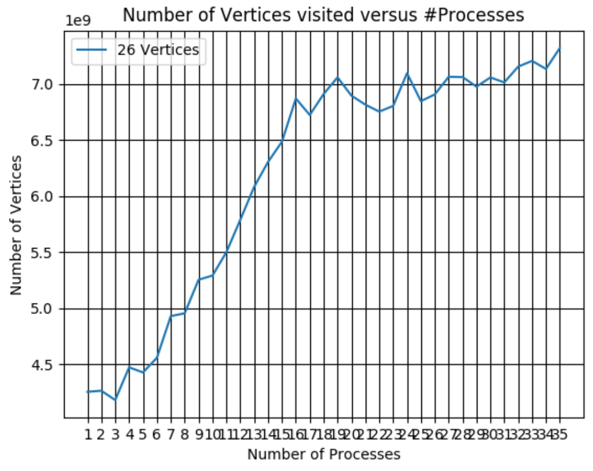

Speedup versus #Processes
24
25
Speedup
#Processes

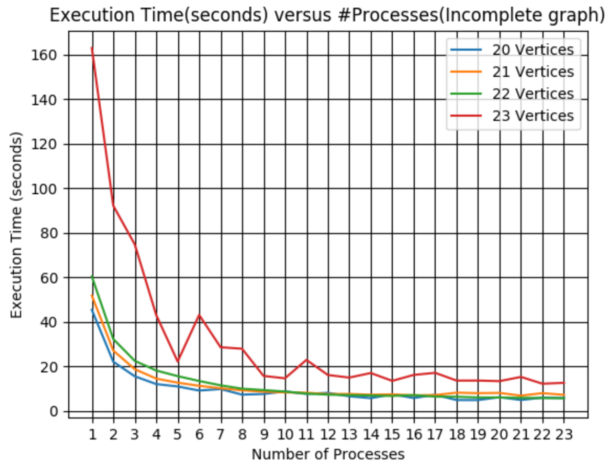

Speedup versus #Processes
20
21
22
23
Speedup
#Processes

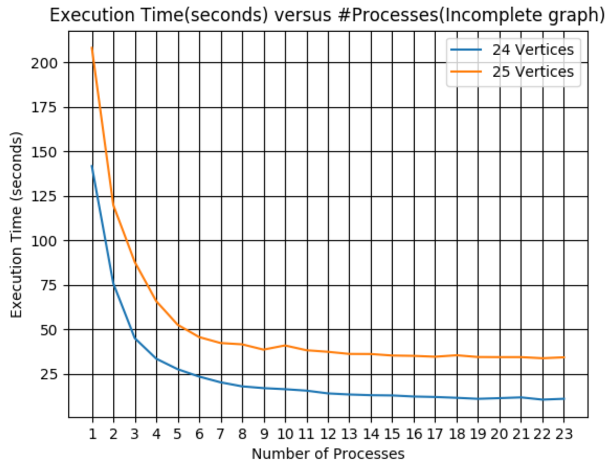

Number of Vertices visited versus #Processes
1e9
24 Vertices
25 Vertices
Number of Vertices
Number of Processes


Number of Vertices visited versus #Processes
1e9
20 Vertices
21 Vertices
22 Vertices
23 Vertices
Number of Vertices
Number of Processes

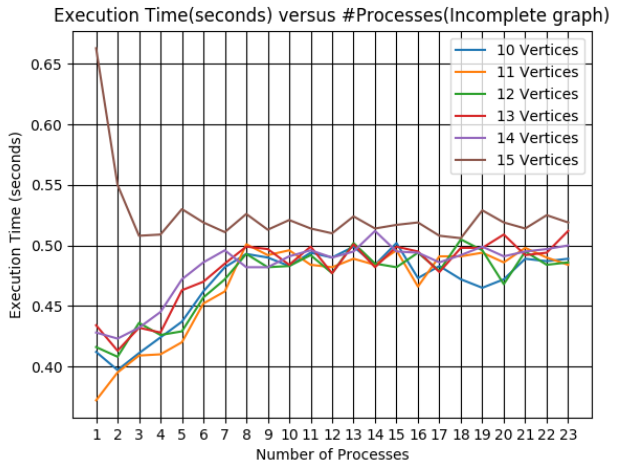

Speedup versus #Processes
26
Speedup
#Processes

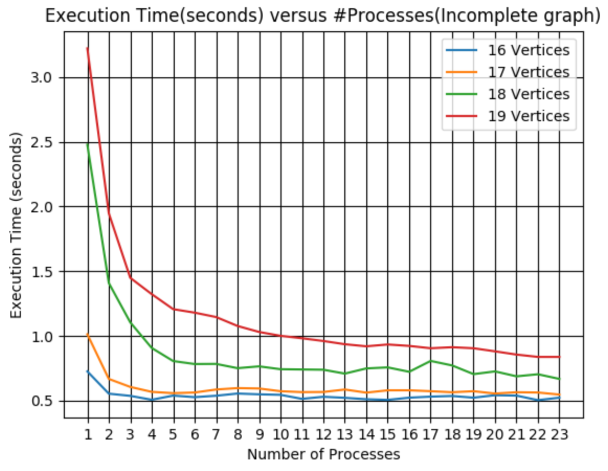


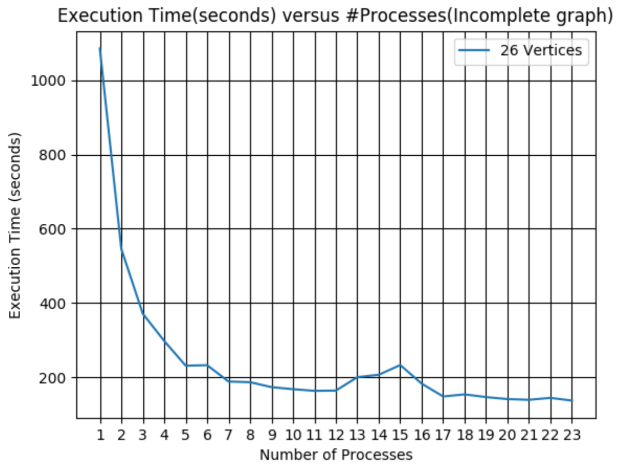




5.3 Time versus Number of processors for Incomplete Graphs



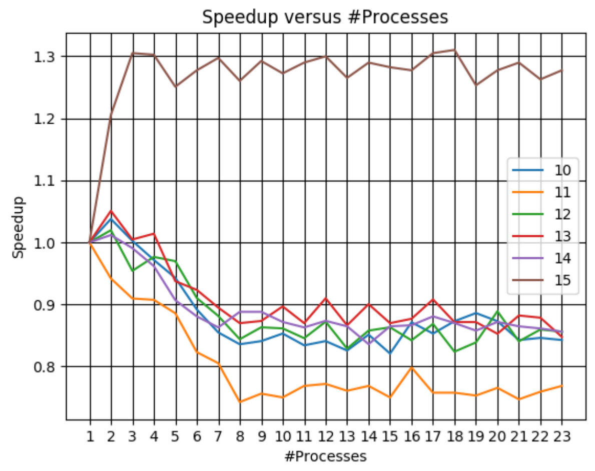


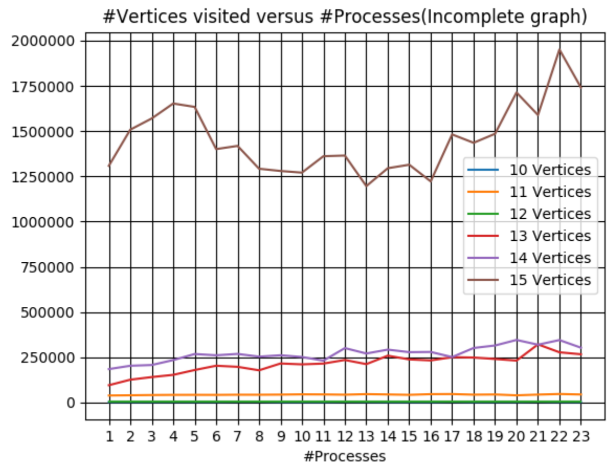


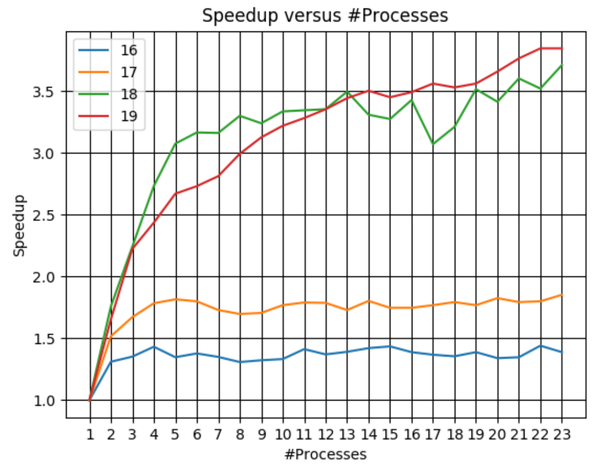
In these figures,we can see that for n <=15 time increases with number of processors after initial decrease, overall an increase after crossing certain number of processes is clearly visible.
For n>15 up-to n=19, speedup is observed with minor fluctuations.
For n>19, each task is big enough so constant decrease is observed.

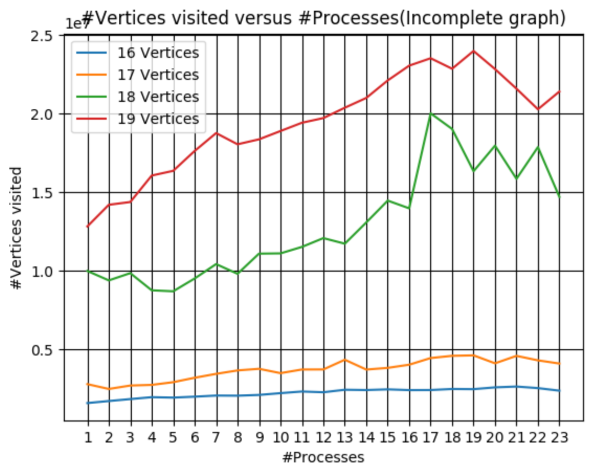
A constant decrease is visible thereafter.

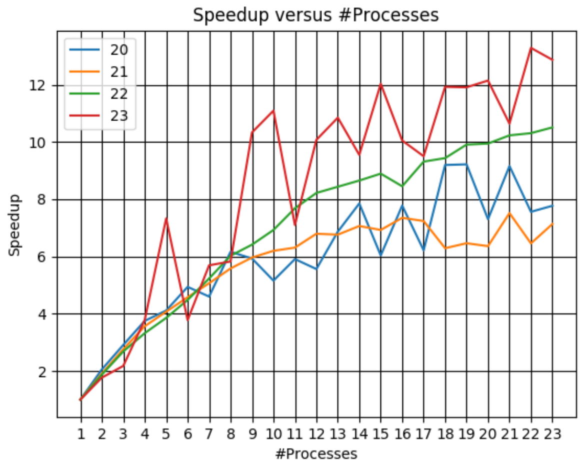
5.4 Speedup obtained and Nodes visited (Incomplete Graph)

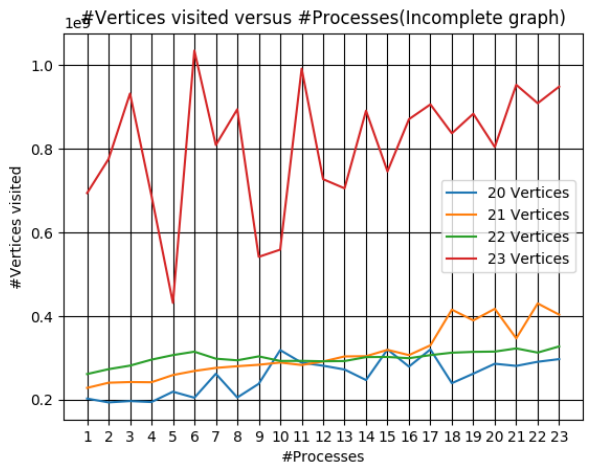


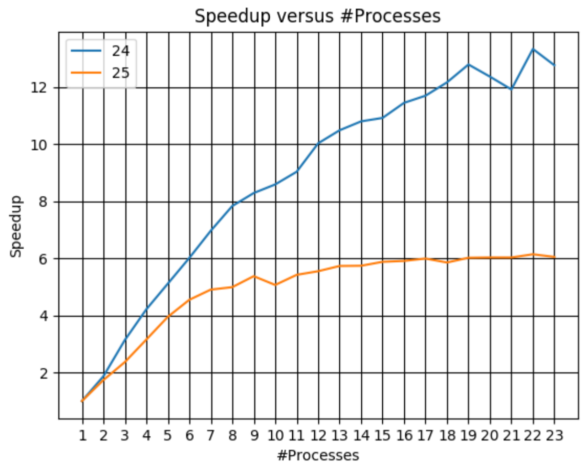


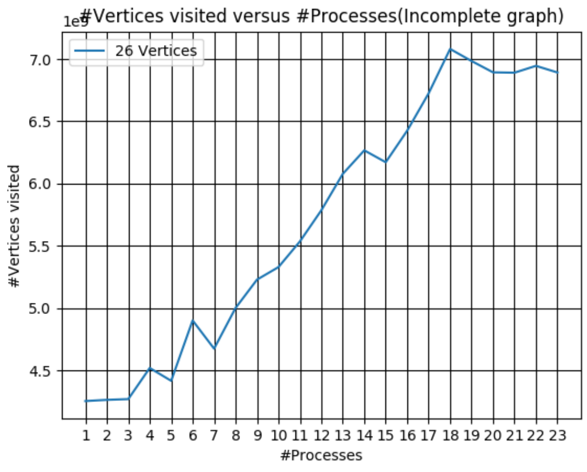


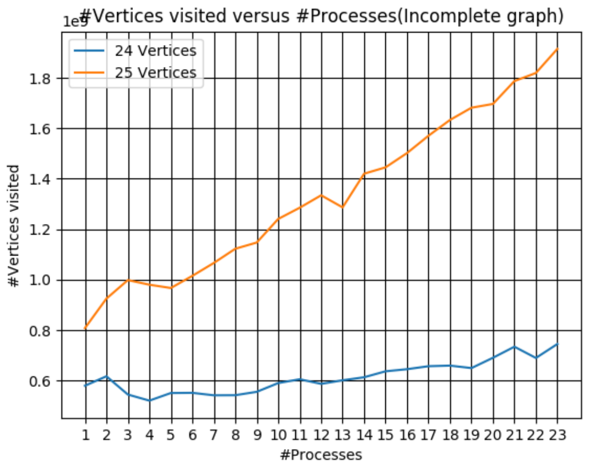


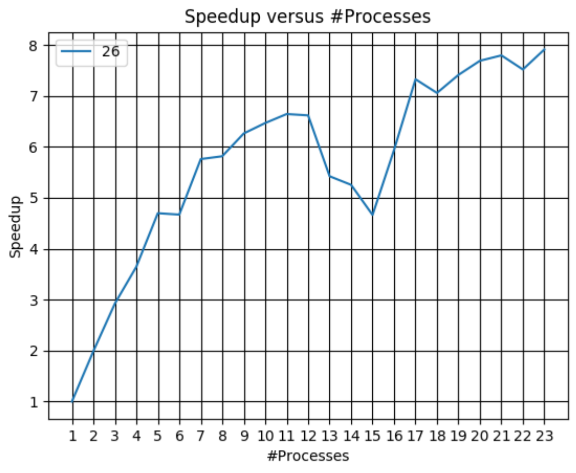






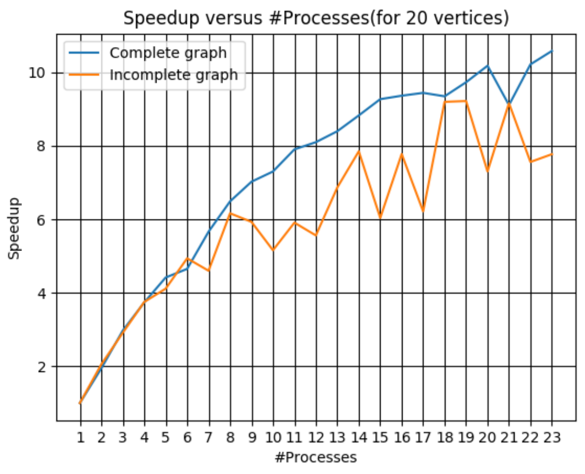


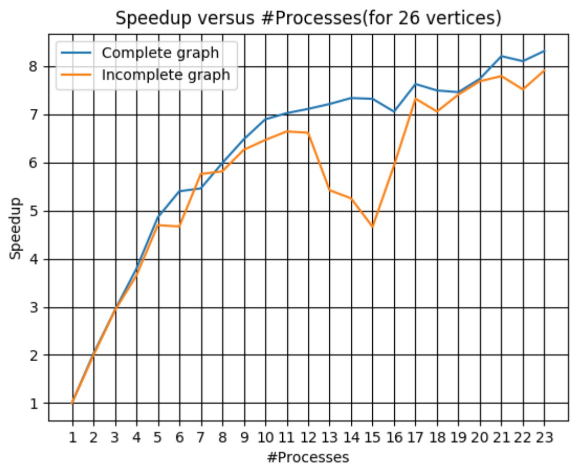


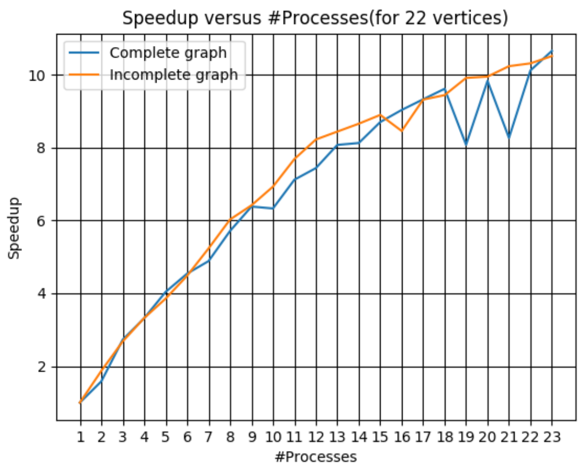


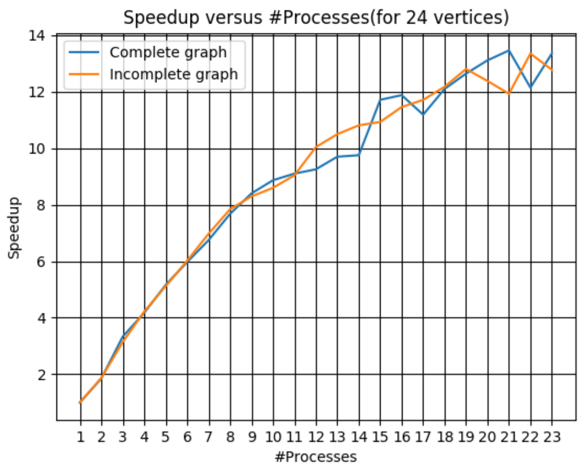
With increasing processes it can be observed that the number of total tree nodes visited is increasing, this is because of the fact that as the number of processes increase many processes traverse along path that would have been rejected in sequential implementation. In parallel implementation by the time one process traverses a path with cost **W**, another process has already traversed a path with weight more than **W**.
As more work is performed speedup gets affected.

5.5 Speedup Complete versus Incomplete Graphs



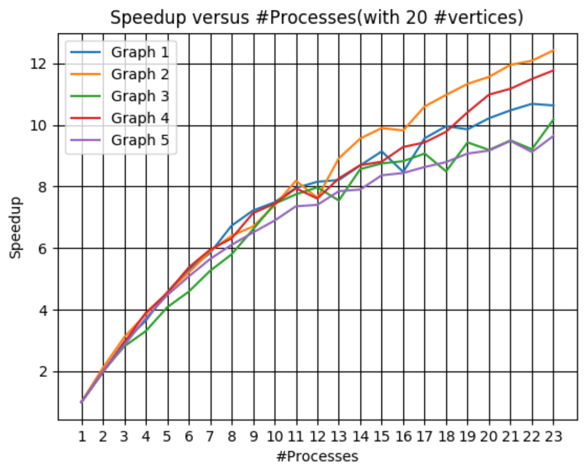






It can be seen that for particular algorithm there is no much difference whether the graph is complete or not.

5.6 Time data for different graphs of same Number of Vertices



The behaviour is not much dependent initially, but for larger number of processes speedup depends a little on what graph is input to the program

6. INFERENCES AND OBSERVATIONS

- It is observed that for smaller value of n (<17), each sub-task is very small and finishes within a second. That's why there is no point in observing the behaviour of these graphs.

- When n reaches 17-18 the behaviour starts to stabilize with as expected initial decrease with minor fluctuations.

- For n > 18 the behaviour is stable with monotonous decrease and only occasional speedup/slowdown. This occasional speedup/slowdown maybe due to certain pattern of values in the graph. Largely a monotonous decrease is observed.

- Speedup achieved in going from 1 process to 6-8 processes can be seen to be good, After that speedup is not that good.