PCP Team Project Report
Parallelised Single Source Shortest Path Algorithm

Introduction:
In our project we look to find the length of the shortest path from one node to every other node in a graph. It is a common problem which finds its application in diverse domains.
Dijkstra's Algorithm is the most famous algorithm which solves this problem. For large graphs, we need to improve the speed of Dijkstra's Algo. We do this by parallelising the algorithm but we very quickly realize that this task is not trivial.

In this project, we implement and compare 4 algorithms: Sequential Dijkstra's, Parallelised Dijkstra, Parallel Delta-Stepping and parSP2 Algorithm

Sequential Dijkstra's:
Dijkstra's algorithm iteratively explores(or relaxes) nodes in a graph, starting from the source node. It maintains an array of distances, initially assigning zero to the source node and infinity to all other nodes. In each iteration, the algorithm selects a node with the smallest distance, marking it as visited. Then, it relaxes the neighbors of the current node, recalculating their shortest distances by the distance to the current node and the weight of the edge. If the recalculated distance is shorter than the previously recorded distance, it is updated in the distance array. This process continues until the algorithm has visited all nodes or the destination node is reached. The final result provides the shortest distances from the source to every other node in the graph, uncovering the optimal paths.

Parallelised Dijkstra's:
Now comes the problem of parallelising the Dijkstra's. We take the approach of parallelising the relaxation of the node. Each neighbor node is relaxed by a different thread. To achieve this we make one key change to sequential dijkstra's: we use a priority queue instead of a queue.
The queue in sequential Dijkstra assumes that the elements are entered in the order of lower to higher weight. We achieve the same effect by entering the nodes in a priority queue. Now we don't have to care about the order in which the nodes are inserted in the queue and thus each node can be relaxed in parallel.
We use mutex locks to ensure thread safety while pushing nodes in the queue. Now the problem with this is that in real world setup relaxation of any node takes about the same time thus there will be high contention for the lock. Pushing the nodes in the queue is still effectively sequential. This can be overcome trivially so we look at better methods.

NOTE: there is a wait-free priority queue implementation provided by intel for x86 architecture but we are confining ourselves to the c++ standard library here.

Parallel Delta-Stepping Algorithm

DeltaSteppingSSSP-ALENEX07.pdf (psu.edu)

The Delta-Stepping algorithm is a single-source shortest path algorithm which seeks to increase performance of traditional SSSP methods in large graphs. The main concept is to create buckets in the graph according to the tentative distances from the source node. These buckets will only contain those vertices whose estimated distances lie in a range specified by a Delta value and its offsets,i.e, [0, Delta), [Delta+1, 2*Delta), etc . A bucket keeps track of vertices in a set, each of which is linked to Delta that indicates the tentative shortest distances that have been found so far.

Tentative shortest distances are stored in an array and are initialized to infinity while that of the source node is set to 0.The algorithm starts with the source, categorizes each of its edges as ‘heavy’ if distance to that node is greater than Delta or ‘light’ otherwise. It relaxes its light edges first, and modifies the distances of the impacted vertices. Heavy edges are relaxed once all the vertices of that bucket are processed. Until all the buckets are empty or all vertices have been analyzed, this procedure keeps repeating. Each iteration of the method can concentrate on a subset of vertices thanks to the delta parameter's threshold.

Delta Stepping is beneficial when the graph is too large to fit into memory, and parallelism can be exploited when the relaxations happen in parallel. It strikes a balance between reducing communication overhead and maximizing parallel processing, making it well-suited for distributed environments and graphs with irregular structures.

An illustrated example can be found here:
DELTA STEPPING (iupui.edu)

ParSP2 Algorithm

ParSP2 algorithm is an SSSP algorithm that is based off of Djikstra’s algorithm, but using a few observations of the Djikstra algorithm, is able to strengthen the mechanism by which vertices get fixed, that is, the distance of the shortest path of the vertex from the source is obtained.

The algorithm uses two observations:

1. If v is any non-fixed vertex, and all edges coming to v have been relaxed (that is, they have been checked whether using them could shorten the distance from source to vertex), then the vertex can be fixed, and the shortest distance calculated thus far can be taken as the shortest distance for the vertex.

2. If k is any non-fixed vertex, and v is a predecessor node to k that has been fixed, and if the shortest distance of k calculated thus far is less than the sum of the distance of v and the minimum edge length of the edges that are incoming to k, then k can be fixed.

Taking these observations into account, the algorithm is able to enhance the mechanism by which a vertex is fixed, as well as provide more points for parallelization of the algorithm.

For the algorithm, we maintain:

- D: an array [0 to n-1] of integers for maintaining the calculated shortest distance for each vertex, whose values are initially all set to infinity.
- H: a minimum heap of (j,d) where j and d are integers, j stands for vertex and d stands for current calculated distance of that vertex, initially empty. This heap only contains either non-fixed vertices, or fixed vertices whose edges have been explored. The minimum heap is ordered by the value of d.
- fixed: an array of Booleans which are all initially set to false. Maintains the fixed status of all vertices.
- Q, R : sets of vertices which are initially empty. R contains vertices which are fixed but whose edges have not been explored, while Q is used to temporarily store vertices which have not been fixed even after processing an edge, so that it can later be put back into the heap.
- pred: an array of integers which maintain the number of non-fixed predecessors each vertex has.

Initially, the algorithm pushes the source node into heap H, and then starts.

The algorithm checks whether R is empty. If R is empty, the algorithm takes the minimum node and puts it in R. If R is not empty, then a node is taken from R and its edges are iterated over.

In each iteration, an edge is distance and checking whether taking that edge will shorten the currently calculated shortest distance, and changing the distance if it does so. Then it checks whether vertex k can be fixed by either of the two properties above. If it satisfies either of the two properties above, then it is fixed, and pushed into R. If the vertex k cannot be fixed, but its calculated distance has been changed by relaxing the edge, then k is put into Q, for later insertion into H.

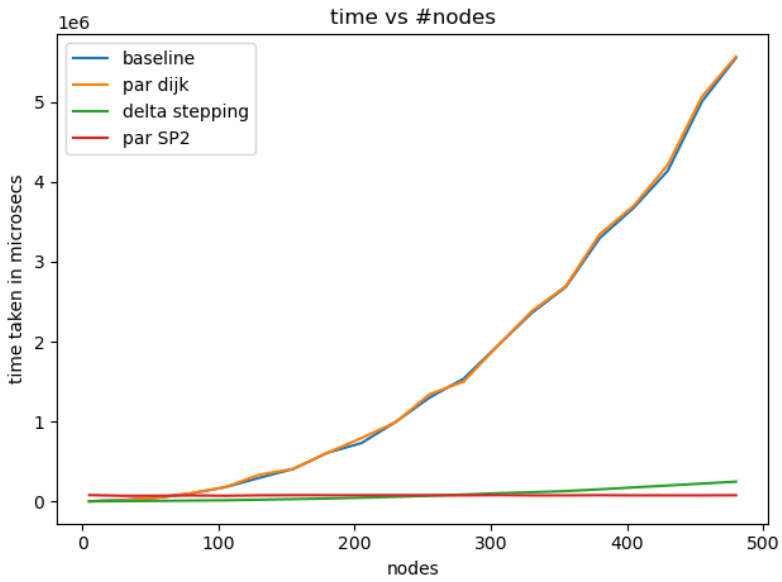
After iterating over all the edges, Q is checked for nodes, and if any are present, they are reinserted into H.

The above process repeats until both H and R are empty, at which point all the shortest distances have been calculated.

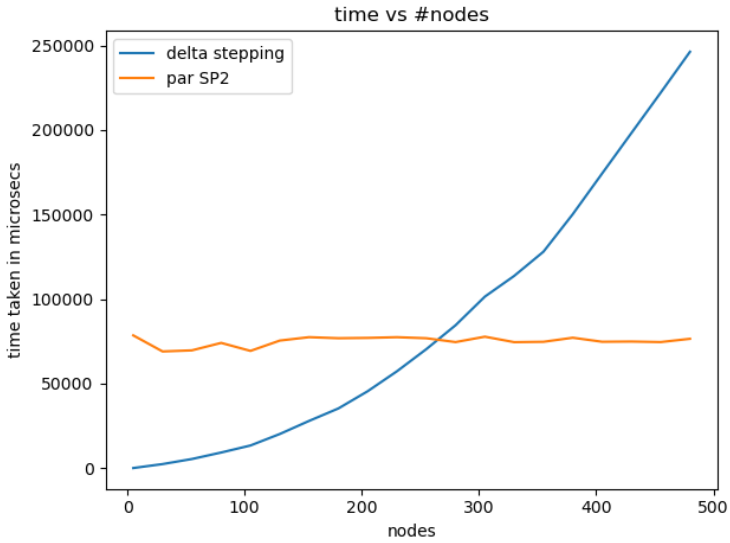
The main point of parallelization in this algorithm is the set of vertices R. The nodes in R can be taken and explored in parallel by many threads, that is each thread can take one node each from R and process them independently, as this algorithm does not need to explore the vertices in order of shortest distance, unlike Djikstra's. Hence it is more efficiently parallelizable. In the implementation, we use thread-safe lock-free queues for R and Q, while the minimum heap H is protected via mutex locks.

RESULTS:

First we compare the algorithms's performance with number of nodes.

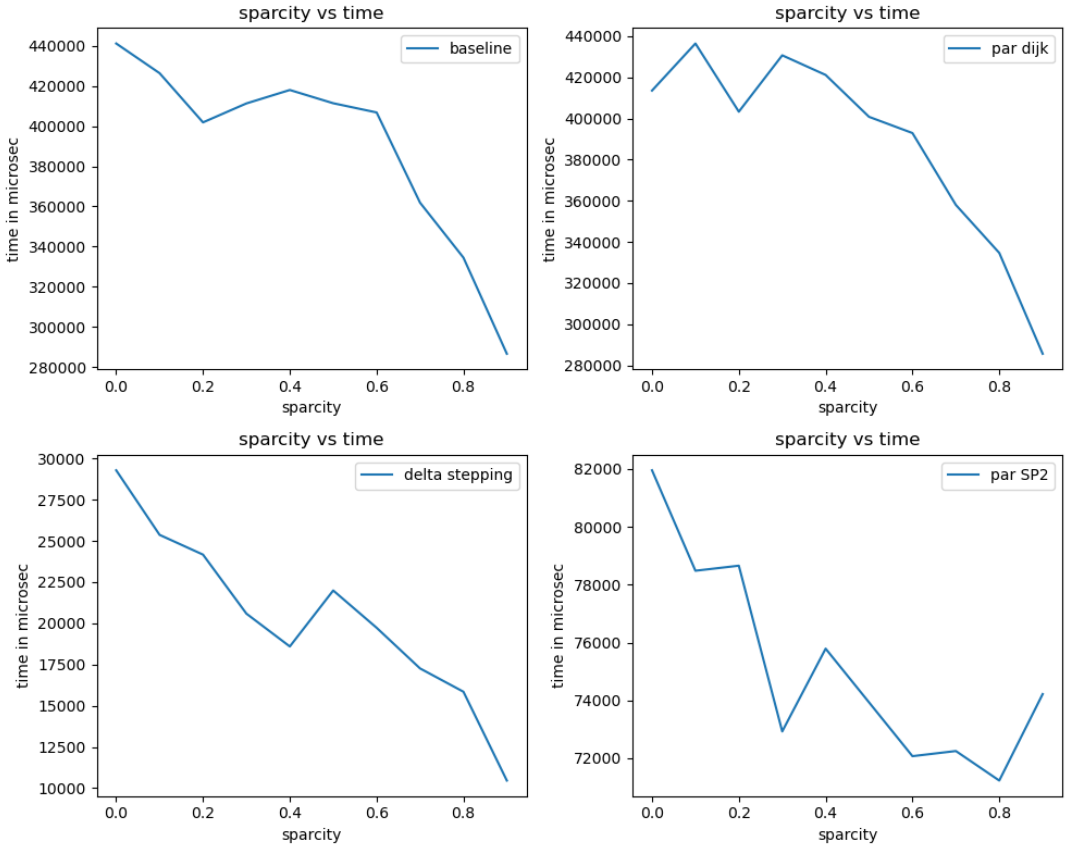


Observe how parallel Dijksta's performs very similarly to Sequential Dijkstra's (Reason discussed earlier).
Delta-Stepping and parSP2 perform much better. Now let us look at only delta-stepping and parSP2 for better comparison.



Delta-Stepping is faster on a smaller number of nodes but on 260< nodes parSP2 out performs by a big margin. And thus and as expected parSP2 is the most scalable algorithm.

We must also look at the effects of sparsity on the algorithms:



Here the trend we see is that all algorithms perform better as sparsity increases. This is because when the number of actual edges is smaller many threads can trivially complete their task.