A Study of Different Parallel Implementations of Single Source Shortest Path Algorithms

Dhirendra Pratap Singh
Department of Computer Science and Engineering
Maulana Azad National Institute of Technology,
Bhopal India

Nilay Khare
Department of Computer Science and Engineering
Maulana Azad National Institute of Technology,
Bhopal India

ABSTRACT
We present a study of parallel implementations of single source shortest path (SSSP) algorithms. In the last three decades number of parallel SSSP algorithms have been developed and implemented on the different type of machines. We have divided some of these implementations into two groups, first are those where parallelization is achieved in the internal operations of sequential SSSP algorithm and second are where an actual graph is divided into sub-graphs, and serial SSSP algorithm executes parallel on separate processing units for each sub-graph. These parallel implementations have used PRAM, CRAY super-computer, dynamically reconfigurable processor and Graphics processing unit as platform to run them.

Keywords
Parallel shortest path algorithm, Parallel algorithm, Graph algorithm, Dijkstra's algorithm.

1. INTRODUCTION
Single source shortest path problem is a classical optimization problem in graph theory, which is applicable in the wide range of applications like VLSI design, network routing, commodity flow, Advance traveler information system. Data of these applications can be represented as a graph having a collection of nodes and links between these nodes (i.e. edges) with some attributes related to them. In shortest path problem we try to find out a path between two nodes of a weighted graph such that sum of the weights of its constituent edges is minimum. The single-source shortest path problem computes shortest paths from single source node to all other nodes of the graph.

First, we introduce the basics of undirected weighted graph and some notations, which are used to define the shortest path algorithms. Graph is represented as an ordered pair G = (V, E) comprising a set V of nodes and a set E of edges which are 2-elemet subsets of V. Let n= |V| the number of nodes, m= |E| the number of edges and c a function assigning a non-negative weight to each edge of G. Weight of an edge (v, w) ϵE is presented by l(v, w). Let s is the source node then the objective of SSSP is to find the weight of a minimum-weight path from s to all other node v ϵV of the graph, which is denoted as d(v) for a node v. During the execution of shortest path algorithms, a node is called settled if its node weight is d(v). Most of the serial shortest path algorithms maintain tentative distance for each node [1]. Let δ(v) represents the tentative distance of node v, its value is always ∞ or the weight of some path from s to v. Graph algorithm's optimize the tentative distances by edge relaxation. Relaxing an edge (v, w) ϵ E means set δ (w) to a minimum of δ (w) and δ (w) + l(v, w).

We present the implementations of parallel SSSP algorithms under two sets. First set is having those algorithms where parallelization is achieved in internal operations of serial SSSP algorithm. Based on the approach used to update the tentative distance shortest path algorithms are divided into two types, label setting and label correcting. The label-setting algorithm assigns a permanent distance label to a node and relaxes the outgoing edges of that node, until all nodes not get their minimum weight. Under label setting we will talk about parallelization of Dijkstra's and Thorup's algorithms. Label correcting algorithms relax the edges of unsettled nodes and edges can be relaxed multiple times until the final step of the algorithm, under this we talk about parallel Bellman-Ford and parallelization in Δ-stepping algorithm. Second set is having those parallel SSSP algorithms where the actual graph is divided into sub-graphs, and parallelization is achieved by executing the serial SSSP program for each sub-graph on different processing unit. Under this we will talk about two implementations, first is graph portioning and iterative weight correcting method and second parallel SSSP on the multilevel graph.

2. PARALLELIZATION IN INTERNAL OPERATIONS OF SERIAL SSSP ALGORITHMS
Basic SSSP algorithm could be label-setting or label correcting. The most famous label setting algorithm is Dijkstra's SSSP algorithm [2]. It Divides V into three sets settled, queued and unreached nodes and for each node v ϵV maintains a tentative distance δ(v) [1]. For settled nodes δ(v) = d(v), for queued nodes δ(v) < ∞ and for unreached nodes δ(v) = ∞. If s is the source node, then initially s is queued, δ(s) = 0 and all other nodes are unreached. In each iteration, a node v with smallest tentative distance is selected from the queued nodes and all edges (v, w) ϵE are relaxed (i.e. δ(w) is set to min{ δ(w) , δ(v) + l(v, w)}) and if w was unreached put it in queued set. We know that δ(v) = d(v) if v is selected from the queue.

2.1 Label Setting Algorithms

2.1.1 A Parallelization of Dijkstra's Shortest Path Algorithm
In this implementation [1], they have divided the Dijkstra's SSSP algorithm into various phases and explain that how we can perform parallel operation with in a phase. The basic idea of this method is that in Dijkstra's algorithm, queue may contain multiple nodes, which are settled, so simultaneously remove such nodes from queue and relax their outgoing edges. However, the problem is to identify them. To identify such nodes they have given number of criteria's, like compute

a threshold defined via the weights of the outgoing edges. Let L= min {δ(u) + l(u, z): u is queued and (u, z) ƐE }, and remove all nodes v from the queue which satisfies δ(v) ≤ L.

They implemented it on CRCW PRAM for random graph and random edge weight. This implementation maintains of a global array for tentative distance of all nodes, and every processing unit is having two sequential priority queues, which deal with a subset of randomly assigned nodes. One queue store the tentative distance of it assign nodes and second stores the addition of tentative distance of node and minimum edge weight out of its all outgoing edges. Minimum edge weight for nodes are pre-computed during the initialization. Second queue of each processor is used to find nodes, which can be deleted in current phase. This implementation works similar to Dijkstra's algorithm. It starts work with source node, which is randomly assigned to a processor and its distance values are stored in processor local queues, and all other processor's queues will be empty. While any queue is nonempty algorithm execute a phase consisting following five steps.
Step1: Find the global minimum L of all elements in all queues in parallel.
Step2: Each Processing Unit (PU) removes the nodes with δ(v) ≤ L from local queue. Let R' denotes the union of all deleted nodes.
Step3: All PU Co-operate to generate a set Req = P {w, δ(v)+ l(v, w)}: v Ɛ R' and (v, w) Ɛ E.
Step4: Randomly distribute the nodes of Req. set between Processors.
Step5: Each processor checks its assign request (w, x) with x < δ(w), it update δ(w) to x and insert new nodes in local queue.

2.1.2 Parallelization of Dijkstra's Algorithm by Using the Parallel Priority Queue

Queue operations are one of the most important and time consuming part of Dijkstra's algorithm. There are two different ways by which we can add parallelism into a priority queue [3, 4]. First method tried to speed up the specific queue operation that handles a single element using a small number of processors. Second way is to support the simultaneous insertion and deletions of smallest elements. They have represented a parallel priority data structure that supports internal operations of the algorithm in O(1) time. Using this data structure, they [4] implemented Dijkstra's algorithm in O(n) time on a CREW PRAM. They used the adjacency list representation of a graph, which is sorted according to edge weight. In such a list, they have shown how perform the operations like determining a node of minimum weight distance and adding any number of new nodes or updating the distances of a node in constant time. The basic idea of this data structure is to use a pipeline structure; each processor takes the output of the processor before it, and does a constant time merge operation to select an element as its output to the next processor. Let S is the set of nodes whose shortest path has been found. They defined a set S' which is having all neighbors of the nodes in S excluding node in S.

In this implementation, each node is having a dedicated processor. Among the processors assigned to nodes in set S, one will be selected as master processor. They have defined four operations INIT, EJECT(S), EXTEND and EMPTY(S) supported by this data structure. INIT initializes the data structure. EJECT(S) delete a node from set S' which is having minimum node weight in set and assign this node and its weight to master processor. EXTEND to add a node to set S and assign a fixed weight label to it and processor assigned to this node become the new master processor. EMPTY(S) check the emptiness of S' for master processor. With the help of these operations, they define the Dijksta's algorithm as mentioned below.
Step1: Initialize the priority data structure.
Step2: Run operation EXTEND for source node.
Step3: Run the operation EJECT for current set S.
Step4: Run the operation EXTEND for node selected in previous step3.
Step5: Run the step3 and 4 while EMPTY(S) is false.

2.1.3 Parallel Implementation of Thorup's Algorithm

Unlike the Dijkstra's algorithm, Thorup's algorithm does not visit the nodes in order of increasing distance from the source node; instead of that it identifies the vertices that can be visited in any order [5]. To avoid the sorting bottleneck of Dijkstra's algorithm, they have used hierarchical bucketing structure for nodes on which internal operations are performed in constant time. These algorithm [5, 6] summaries the graph in a tree data-structure called the Component Hierarchy (CH). Each CH-node is called component, which represents a sub-graph of the graph G. Each component is identified by node v and a level i. Component (v, i) is the sub-graph of G having node v, the set of nodes reachable from v when traversing edges with weight < 2^i and all edges adjacent to {v} U S of weight less than 2^i. Algorithm use CH to identify the nodes that can be visited in arbitrary order. Figure1 show a graph which is divided into three components.

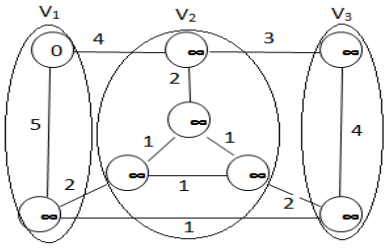

Fig 1: Initialized graph which is divided in three components V_1, V_2 and V_3

To identify the nodes which can be visited randomly, node set V is divided into disjoint subsets V_1, V_2, …, V_k, where all edges between subsets have weight at least Δ [5]. Let S be the set of settled nodes and for some i, v Ɛ V_i\S such that d(v) = min {d(u) | u Ɛ V_i\s} ≤ min { d(u) |u Ɛ V\S} + Δ, then d(v) = δ(v).

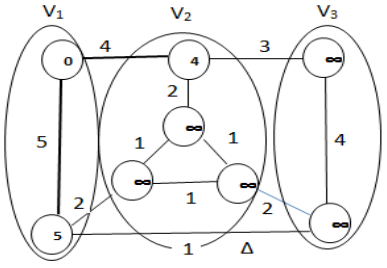

Fig 2: Graph after relaxation of the edges of a settled of node from component V_1

As per the lemma after the relaxation of the edges of a settled node from component V_1 in the next step we have two nodes one in V_1 and second in V_2 for which d(v) = δ(v). Let α= $\log_2\Delta$. Component V_i buckets its children according to min {d(u)|u Ɛ V_i\S} >> α. This algorithm maintains a current bucket for each component and all children in current bucket

are the list of children to be visited. Here it visits each children recursively starting from smallest bucket index and when it reached a leaf component which represents a node v, it is called visited and edges starting from v are relaxed. They used the Cray MAT-2 a massively multithreaded machine to implement this algorithm. MAT automatically executes the loop in parallel by reduction method. It can be summarized as follows:
Step1: Construct the component hierarchy for graph in parallel.
Step2: Initialize the weights of all nodes and set S.
Step3: Construct the unvisited data structure. Each component of component hierarchy bucket its child according to min $\{d(u)|u \in V_i \backslash S\}$.
Step4: Visit the component of hierarchy in parallel.

2.1.4 CUDA Solutions for the SSSP Problem

In this solution [7], they have shown the different ways for the parallel implementation of Dijkstra's algorithm. They used the CUDA interface for these implementations and graphics processing unit (GPU) to run the parallel threads. Basically, each step of these implementations removes all those queued nodes, which are having weight equal to current minimum weight value and relax their outgoing edges in parallel on threads running for them. In each iteration, they create n threads one for each node. They have also shown the parallelization in finding the minimum weight and creating the list of nodes having minimum node weight. This implementation can be represented in steps as shown below.
Step1: Parallel initialization of all node weights and flags in corresponding threads.
Step2: Define a minimum variable.
Step3: relax the edges of those queued nodes whose weight is equal to value of minimum variable.
Step4: Find new value of minimum variable, which is minimum weight out of all queued nodes weight.
Step5: select set of nodes whose outgoing edges will be relaxed in next iteration.
Step6: Repeat Step 3, 4 and 5 while there is a weight update for any node during step3.

2.2 Label Correcting Algorithms

2.2.1 Δ-Stepping: A Parallelizable Shortest Path Algorithm

They [8] proposed a parallel version of a label-correcting algorithm for SSSP finding. This algorithm maintains a list of eligible nodes with their tentative distances in an array of buckets B, each of which denotes a distance range of size Δ the "bucket width". During any iteration, this algorithm removes all nodes of the current nonempty bucket and relaxes their outgoing edges of weight at most Δ, while current bucket is non-empty. During the relaxation step new nodes are inserted into bucket. If any node v has been removed from the current non-empty bucket B[i] without its final distance value, then in some succeeding step of same iteration, v will surely be reinserted into B[i]. Edges having weight higher than Δ are relaxed only after their corresponding starting node is surly settled. So the edges of weight more than Δ originating from all nodes that have been removed from B[i] are relaxed once for all when B[i] finally becomes empty. Parallelism is obtained by simultaneously removing all nodes of the current non-empty bucket, relaxing their outgoing edges of weight at most Δ and finally relaxing edges having weight more than Δ [8, 9]. This algorithm can be represented in steps as shown below.

Step1: Divide the edges of graph in two sets heavy edges ($l(e) > \Delta$) and light edges ($l(e) \leq \Delta$) $e \in E$, and initialize all $d(v) = \infty$, $S = \Phi$.
Step2: Insert the source node s into bucket and $d(s) = 0$.
Step3: While the current bucket is non-empty, remove all its nodes, add them in a set S and relax all light edges adjacent to these nodes in parallel.
Step3: When the current bucket becomes empty, parallel relaxation of all heavy edges adjacent to nodes present in S.
Step4: Reset the set $S = \Phi$ and Repeat the step3 and 4 while bucket is not empty.
The performance of this approach is governed by the choice of Δ. The choice of Δ offers a trade-off between too many nodes re-considerations on the one hand and too many bucket traversals on the other hand.

2.2.2 Parallel Bellman-Ford Algorithm

Bellman-Ford algorithm is a well-known label correcting algorithm for SSSP [10, 11]. It is primarily used for graphs with negative edge weights. This algorithm relaxes all edges of a graph, $n - 1$ time. The repetitions allow minimum distances to accurately broadcast throughout the graph, since in the absence of negative cycles; the shortest path can only visit each node at most once. An important property of this algorithm is that during any iteration, we can relax the edges of a graph in random order, but the result will be same. In "Implementing parallel shortest path algorithms [12]" they used this property to implement the parallel version of the bellman-ford algorithm on CM-5 parallel supercomputer in two steps. At first step they divided the edge set of graph G in P (number of processors) different disjoint subsets. Each processor is assigned a subset of edges, and this assignment never changes during the execution of the program. In second step execute a program of modified Bellman-ford's algorithm in each processor.

This algorithm is divided into two parts a computation followed by a communication phase. In each iteration during the computation phase, each processor relaxes its assigned edges and updates its local label δ(v). After computation phase, their program performs a global communication, where for each node assigned to a processor set their local label δ(v) equal to the current minimum among all labels of δ(v) on different processors. After communication phase, each processor gets the best approximation of its all labels δ(v) that has been computed until now. After it all processors will start the next iteration. Parallel implementation can be represented as mentioned below.
Step1: Initialization all nodes of the graph as $\delta(s) = 0$; $\forall v \in (V \backslash s)$, $\delta(v) = \infty$.
Step2: Divide the edge set into p disjoint subsets and assign it to different processors.
Step3: Relax the edges of each subset assigned to a processor in parallel.
Step4: Each processor get the global minimum $\delta(v)$ $\forall v \in V$ for its assigned nodes.
Step5: Repeat the step 3 and 4 n-1 times.

3. GRAPH PARTITIONING BASED PARALLEL SSSP ALGORITHMS

3.1 Parallel Shortest Path Algorithm Based on Graph Partitioning and Iterative Correcting

In this implementation [13], they have represented a parallel shortest path algorithm based on graph partitioning and

iteratively correcting the nodes weight. This algorithm is divided into two phases, a graph partitioning phase followed by a weight correcting phase. In first phase, they used k-way portioning algorithm (k-way METIS) to divide the graph. Graph is portioned into disjoint sub-graphs, where each sub-graph has roughly the same number of nodes and the number of edges crossing sub-graphs is minimal. Phase two contains two steps; first step is a computation step, after graph partitioning each sub-graph is assigned to one processor, and each process find temporary shortest paths in its assigned sub-graph locally. In the first iteration only one processor actually compute temporary shortest path in assigned sub-graph, which is having source node information. Second step is a communication step where after computing the shortest path in local sub-graphs the boundary information's are exchanged between the adjacent sub-graphs. Now the number of processes having the information of the source node is increased. Algorithm continues iterating, and temporary shortest paths in each process keep correcting and updating with boundary information exchange. This algorithm continues until there is no message exchanged between the adjacent sub-graphs, and we will obtain the final shortest path. Figure 3 to figure 7 show the different phases of this implementation.

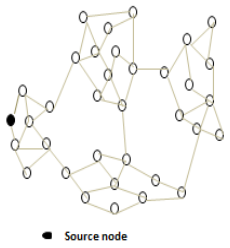

Fig 3: Source graph

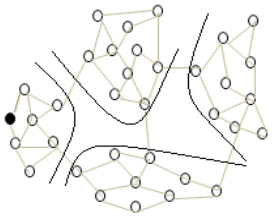

Fig 4: Graph Partitioning

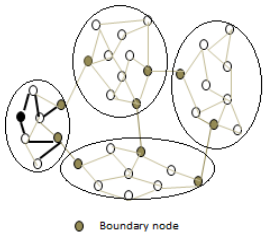

Fig 5: locally compute shortest path in first partition

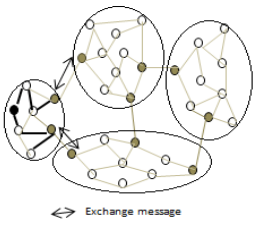

Fig 6: 1st time Exchange of message

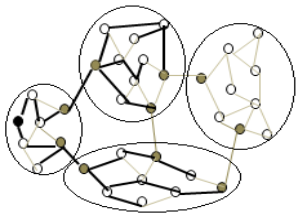

Fig 7: locally compute shortest path in 1st, 2nd and 3rd partition

3.2 A Hierarchical Shortest Path Algorithm

In hierarchical shortest path algorithms [14, 15], they have represented the graph in two different layers. Layer one will be represented by different disjoint sub-graphs of the graph, which are produced by partitioning of an actual graph. Layer two defines a boundary graph which is summarizing the sub-graphs. Figure 8 represents the boundary graph of partitioned graph in figure 9.

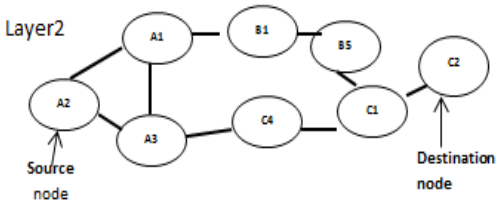

Fig 8: Boundary graph

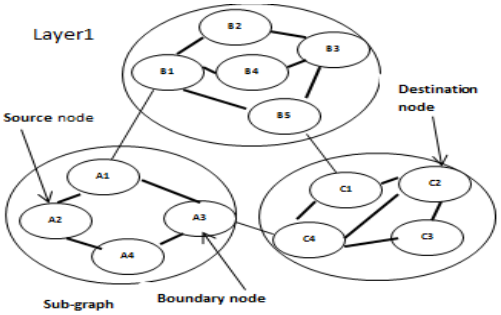

Fig 9: Partitioned graph

The hierarchical algorithm is having three steps. The first step in the finding of shortest path is to compute the boundary nodes. In second and most important step which we can execute in parallel is computing the shortest path inside the sub-graphs and here two different cases are possible. If a sub-graph contains the source/destination nodes of the problem, then find the shortest path between source/destination nodes to boundary nodes in corresponding sub-graphs. For other, sub-graphs find the shortest path between boundary nodes. Third step is to create the boundary graph and update its node weight using all previously calculated shortest paths inside the sub-graphs. Now finally it will calculate the shortest path between source and destination in the boundary graph.

Table 1. Comparative analysis

Algorithm	Platform for Parallel Implementation	Speedup/ Complexity	Graph Type/Size
A parallelization of Dijkstra's shortest path Algorithm	CRCW PRAM	$O(n^{1/3}\log n)$	Directed graph
Parallelization of Dijkstra's Algorithm by using the Parallel Priority Queue	CREW PRAM	O(n)	Directed graph
Parallel implementation of Thorup's algorithm	MAT-2, 40 Processors	12 times	$N= 2^{26}$ $M= 2^{28}$
CUDA solutions for the SSSP problem	Nvidia GTX280,256 cores	60 times	$N=2^{20,}$ M is not define
Δ-stepping: a parallelizable shortest path algorithm	MAT-2, 40 Processors	30 times	$N= 2^{28}$ M=4N
Parallel Bellman-Ford Algorithm	CM-5, 32 Processors	7.8 times	$N=2^{15,}$ $M=2^{22}$
A parallel shortest path algorithm based on graph partitioning and iterative correcting	IBM Cluster, 16 processors	15 times	N= 100000 M=280000
Hierarchical Shortest Path Algorithm	DAPDNA-2, 376 PE	99.6% less clock cycles	N=512,M is not define

4. COMPARATIVE ANALYSIS

In table 1 shows the comparative analysis of implementation of different parallel SSSP algorithms. Platform represents the type of machine used for that implementation, complexity or speedup of these algorithms are shown in compare of their serial algorithm and type or size of graph any implementation has used to get the mentioned performance. N denotes the number of nodes and M represents the number of edges in any graph.

5. CONCLUSIONS

After going through all these implementations, we found that label setting algorithms gave procedures by which it maximize the number of nodes selected from queued node set in each iteration and relax their outgoing edges parallel. Label correcting algorithms tried to minimize the number of relaxation for edges. Graph partitioning algorithms have tried to divide the graph in such a way so that it can calculate the maximum part of SSSP in sub-graphs. If we are able to find a step in serial SSSP algorithm, which can be parallels it always improve the performance of an algorithm. Partitioning of the graph data is going to add an extra overhead, but if we are going to use same graph for different SSSP request, then this can be avoided. PRAM and Cray supercomputer had provided good platform but now researchers are using CPU-GPU based hybrid machine to implement the parallel SSSP algorithms, as graphics cards are providing high speed and low cost parallel processing platform.

6. REFERENCES

[1] A. Crauser, K. Mehlhom, U. Meyer and P. Sanders, "A parallelization of dijkstra's shortest path algorithm", LNCS 1450, pp. 722-731, 1998.

[2] E. W. Dijkstra, "A note on two problems in connexion with graphs", Numerische Mathematik 1, 269–271, 1959.

[3] G. Brodal, J. Traff and C. D. Zarolingis, "A parallel priority queue with constant time operations", Journal of parallel and distributed computing 49, pp. 4-12, 1998.

[4] G. Brodal, J. Traff and C. D. Zarolingis, "A parallel priority data structure with applications", IEEE, pp. 689-693, 1997.

[5] J. R. Crobak, J. W. Berry, K. Madduri and D A. Bader, "Advanced shortest paths algorithms on a massively-multithreaded architecture", Parallel and Distributed Processing Symposium, 2007, IEEE, pp.1-8, 2007.

[6] M. Papaefthymiou and J. Rodrigue, "Implementing parallel shortest-paths algorithms", DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pp. 59-68, 1994.

[7] Pedro J. Martin, Roberto Torres and Antonio gavilanes, "CUDA Solutions for the SSSP problem", LNCS 5544, pp. 904-913, 2009.

[8] U. Meyer and P. Sanders, "Δ-stepping: a parallelizable shortest path algorithm", Journal of Algorithms 49, pp. 114-152, 2003.

[9] K. Madduri, D. Bader, J. Berry and J. Croba, " An experimental study of a parallel shortest path algorithm for solving large-scale graph instances", In workshop on Algorithm Engineering and Experiments (ALENEX), New Orleans, LA, January 2007.

[10] R. E. Bellman, "On a routing problem", Quarterly of Applied Mathematics, 16: 87-90, 1958.

[11] L. R. Ford Jr., and D. R. Fulkerson, "Flows in Network", Princeton University Press, 1962.

[12] M. Papaefthymiou and J. Rodrigue, "Implementing parallel shortest-paths algorithms", DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pp. 59-68, 1994.

[13] Y. Tang, Y. Zhang and H. Chen, "A parallel shortest path algorithm based on graph-partitioning and iterative correcting", IEEE, pp. 155-161, 2008.

[14] A. Fetterer and S. Shekhar, "A performance analysis of hierarchical shortest path algorithms", IEEE, pp. 84-93, 1997.

[15] H. Ishikawa, S. Shimizu, Y. Arakawa, N. Yamanaka and K. Shiba, "New parallel shortest path searching algorithm based on dynamically reconfigurable processor DAPDNA-2", IEEE, pp. 1997 – 2002, 2007.