Parallel Batch Dynamic Single Source Shortest Path Algorithm and Its Implementation on GPU based Machine

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

Abstract: *In this fast changing and uncertain world, to meet the user's requirements the computer applications based on real world data always try to give responses in the minimum possible time. Single Source Shortest Path (SSSP) calculation is a basic requirement of applications using graphs portraying real world data like social networks and road networks etc. to get useful information from them. Some of these real world data changes very frequently, so recalculation of the shortest path for all nodes of a graph depicting these real world data after small updates of graph structure is an expensive process. To minimize the cost of recalculation shortest path algorithms need to process only the affected part of a graph after any update, and to speed-up any process parallel implementation of algorithm is a frequently used technique. This paper proposes a new parallel batch dynamic SSSP calculation approach and shows its implementation on a CPU- Graphic Processing Unit (GPU) based hybrid machine. The proposed algorithm is defined for positive edge weighted graphs. It accepts multiple edge weight updates simultaneously. It uses parallel modified Bellman Ford algorithm for SSSP recalculation of all affected nodes. Nvidia's Tesla C2075 GPU is used to run the parallel implementation of the algorithm. The proposed parallel algorithm shows up to a twenty-fold speed increase as compared to best serial algorithm available in literature.*

Keywords: *Parallel algorithm, graph algorithm, dynamic shortest path algorithm, network algorithm.*

Received December 20, 2014; accepted October 30, 2016

1. Introduction

Graphs are the most common way to represent data in many scientific and engineering applications, such as routing in social networks [21], road network routing [11, 13], internet routing [18], robotics [1, 23] etc., Recently, researchers have also started developing software systems for graph algorithms to provide effective computational tools to support application prototyping, algorithm animation or further algorithmic research [29]. A directed graph with positive edge weights is defined as $G = (V, E)$, where $V = \{v_1, v_2, v_3 \ldots\ldots\}$ represents set of vertices or nodes and $E = \{e_1, e_2, e_3 \ldots\ldots\}$ represents set of edges. An edge $e \in E$ of graph is represented by an ordered pair of nodes (v_1, v_2), where v_1 is start node and v_2 is end node of the edge. One of the most studied problems in graphs is the shortest path problem. On the basis of nodes that participate in shortest path calculation, it can be categorized as: the single-pair shortest path problem, the single-source shortest path problem and the all-pair shortest path problem. The Single Source Shortest Path (SSSP) problem searches for shortest paths from node $s \in V$ to all other nodes of the graph G. These problems can be further classified in terms of static graph problems and dynamic graph problems. Many static and dynamic SSSP algorithms are defined to solve these problems. In static SSSP algorithms, when a graph is updated, the shortest path is recomputed from scratch, which is clearly inefficient as they do not use available information, while dynamic Single Source Shortest Path (SSSP) algorithms update graphs from some intermediate point using previously computed information. Two types of updates are possible in any graph. The first is the edge weight increase and second is the edge weight decrease. Edge insertion and edge deletion are considered as special cases of weight decrease and weight increase respectively. Algorithms having a provision for both edge weight increase and decrease are called fully dynamic, while if they support only one type of update at a time they are called semi-dynamic. The algorithm which supports both types of update simultaneously is called the batch dynamic algorithm.

Rest of the paper is organized in following manner; section 2 summarizes some previous works related to our research. Section 3 describes graph structure used by proposed implementation of algorithm. Section 4 formally introduces the problem and explains the proposed algorithm. Section 5 discusses about essentials of Graphic Processing Unit (GPU) programming using Compute Unified Device Architecture (CUDA). Section 6 shows parallel implementations of batch dynamic single source shortest path algorithm. The algorithms are tested on a range of standard datasets, their results and discussions are present in section 7. Concluding remarks are given in section 8.

2. Related Works

There are various serial dynamic algorithms in literature, which perform dynamic update on shortest path and handle multiple edge updates simultaneously. Reps and Ramalingam [32] have introduced the batch algorithm SWSF-FP that handles edge insertions and deletions iteratively. It uses the Dijkstra's [12] algorithm for SSSP calculation and considers the end node of an updated edge as affected node. The most important operation used by this algorithm is Con(v) for node v, in which it relaxes all edges where affected node v is the end node of the edge. Initially it calculates SSSP for all nodes by using the Dijkstra's algorithm then inserts the new weight of updated edges. Dynamic calculations of SSSP calculate the Con(v) value of all affected nodes and insert each affected node in queue if their weight gets changed. It removes the minimum weighted node from the queue and calculates the Con(v) for all end node v of the edges where the node removed from the queue is the start node of edge. If the node weight of v gets updated, put it in queue and repeat these last two operations until queue is not empty.

Narváez *et al*. [27] have proposed the Narváez-framework that allows implementing a variety of dynamic shortest path algorithms including the well-known Dijkstra, Bellman-Ford, D'Esopo–Pape algorithms. Frigioni *et al*. [15] have proposed an iterative algorithm that uses more complex auxiliary data and accounting function. Ramalingam *et al*. [31] have explored a different way to analyses the complexity of dynamic algorithms which measures cost in terms of sum of changes in input and output. Buriol *et al*. [7] presented a technique to reduce heap sizes used by several dynamic shortest path algorithms. King and Thorup [20] have proposed a technique that reduces space and work performed during computation of shortest path by maintaining a special shortest path tree. Misra and Ommen [26] have presented a learning automata based solution for dynamic SSSP calculation. They have shown that after using the learning automata if any edge weight change occurs, their solution does not probe all the edges of the graph.

Bauer and Wagner [2] have presented faster tuned variants of existing SWSF-FP [32]. It performs different operations after removing any node from queue according to its new weight. After removing the minimum weighted node v from the queue, if node's new weight is less than its old weight then for all end node w of the edges where v is the start node perform Con(w) and if any node weight is updated, it will be inserted in queue. If node's new weight is greater than its old weight then for all end node w of the edges which are the part of shortest path sub-graph and v is the start node of edge performs Con(w) and if any node's weight gets updated, it will be inserted in queue. It repeats these operations until queue is not empty.

When a large graph with, say one million vertices, is updated with small changes, an enormous amount of arithmetic computation has to perform in serial on mentioned algorithms, which is really very time consuming and tedious. To speed up the SSSP calculation many researchers have proposed the various parallel implementations [4, 8, 9, 36] of it for different type of machines. With the help of Nvidia's CUDA tool it is possible to explore the parallel and multithreaded environment of its today's GPU for general purpose computing. Today's GPU has provided a low cost and highly parallel platform for general purpose computing [6, 17, 33, 35], so many researchers have used GPUs for their parallel shortest path calculation. Harish and Narayanan [16] have presented first parallel implementations of SSSP on GPUs using CUDA. Their algorithm calculates the SSSP of a million vertices graph in 1-2 seconds. Katz and Kider [19] have presented an algorithm for the All Pair Shortest Path Problem on large graphs by using multiple GPUs. Martín *et al*. [24] have shown different parallel implementations for well-known Dijkstra's algorithm [12] on GPU. Dashora and Khare [10] have also presented parallel SSSP and other graph algorithms on GPU. Singh and Khare [34] have proposed two different parallel implementations of modified Dijkstra's algorithm [34] for GPU based machine. Many all pair shortest path [5, 25, 37] implementations also have been proposed for GPU based machines. In this paper first parallel batch dynamic SSSP algorithm have been proposed to enhance the performance and reducing the execution time of recalculation of affected nodes weight and its implementation is shown for GPU based machine.

3. Graph Representations

A graph G=(V, E) can be represented by adjacency list, adjacency matrix, hash tables and unordered edge sequences etc. Out of which hash tables are efficient on CPU, but GPU memory layout is optimised for rendering graphics and cannot support user-defined data structures efficiently [30]. In GPU computing most common graph representations are adjacency list with memory requirement of $O(|V|+|E|)$ as there is no wasted information, but more expensive lookup time. The adjacency matrix having more memory usage $O(|V|^2)$ but has an advantage of O(1) lookup time. The unordered edge sequence requires $O(|E|+|E|)$ memory with $O(|E|)$ navigation time.

Proposed algorithm and its implementations use two different graph representations. The first graph representation is adjacency list similar to the representation proposed by Harish and Narayanan [16]. This adjacency list data structure consists of three arrays, array of node (V) of size $|V| + 1$, array of edge (E) of size $|E|$ and array of edge weight (W) of size $|E|$.

Each index of array V represents a node number of graph and array E stores the end node number of all the edges in graph. The array E stores the end nodes number of the all outgoing edges of a node in sequential order. Array W stores the edge weight of all edges in the graph. Weight of an edge (i, j) is stored at the same index, where node j is stored in array E.

The second graph representation is unordered edge sequence; it uses three arrays; array Edge Start Node (ESN) which stores the start node of each edge, array Edge End Node (EEN) which stores the end node of each edge and array Edge Weight (EW), which stores the weight of each edge, all of size $|E|$.

4. Proposed Parallel Dynamic SSSP Algorithm

Let G= (V, E) be a directed graph having |V| nodes and |E| edges. A positive weight function assigns a weight to each edge of the graph. Given source node s∈ V, for a node v∈ V, NW[v] represents its distance from the source node in SSSP calculation. For any edge (u, v) ∈ E, edge relaxation operation updates the NW[v] out of minimum of NW[v] or NW[u] + W (u, v). For a node v∈ V, edge (u, v) ∈ E is responsible edge, if NW[v] = NW[u] + W (u, v) after SSSP calculation. Responsible edge of a node v∈ V is denoted by Resp[v]→(u, v) ∈ E. After SSSP calculation in a graph from given source node s, Shortest Path sub Tree (SPT) is a subset of graph having all the nodes reachable from s and their responsible edges. Node v is called an affected node if after the update of edge (u, v) weight NW[v] is also updated.

Dynamic single source shortest path algorithm re-computes the shortest path for all affected nodes in updated graph. When *G* is undergoing through batch update $U= \{U_1, U_2, U_3, ..., U_k\}$, where each $U_i \in U$ is in the form of triplet (u, v, W_{new}) consisting edge start node *u*, edge end node *v* and new edge weight W_{new}. A node $v \in V$ is possible affected node if $(u, v) \in U$. After the edge weight update in any graph possible affected nodes are defined as shown in Lemma.

- *Lemma*: All nodes of sub-graph reachable form the affected node are possible victim of graph update.
- *Proof*: After the calculation of the SSSP in the graph, sub-graph whose nodes are reachable from the affected will have two parts:

1. Nodes which are parts of the SPT with root node as affected node.
2. Nodes which are not parts of the SPT with root node as affected node.

Suppose *i* is an affected node and there are two edges (*i, j*) and (*j, k*) in SPT

$$\text{if } NW[j] = NW[i] + W[i,j] \text{ and } NW[k] = NW[j] + W[j,k] \quad (1)$$

$$\text{then } NW[k] = NW[i] + W[i,j] + W[j,k] \quad (2)$$

From Equations (1) and (2) it is clear that the weight of node *j* and *k* depends on node weight of node *i*, so node *j* and *k* are also affected nodes. With the help of affected nodes *j* and *k* other affected nodes will be discovered in SPT with root as node *i*.

Let an edge $(x, y) \in E$ and *y* be not the part of SPT and suppose after the recalculation of the node weight of the affected node *x*, if $NW[y] > NW[x] + W[x, y]$, the new weight of *y* should be $NW[y] = NW[x] + W[x, y]$, so *y* is also an affected node.

Algorithm 1 shows the proposed parallel batch dynamic SSSP algorithm for a given graph *G(V, E)*. Suppose SSSP has been calculated and node weight are stored in *NW[v]* for all nodes $v \in V$. Responsible edge for each node in SPT are calculated and store in array *Resp[v]* for all nodes $v \in V$. Define a *Flag* arrays of size |*V*| initialise it's all elements with zero. It used to maintain updated node list.

- *Step* 1: for all edges $(u, v) \in E$, in parallel compare the $W(u, v)$ with $W_{new}(u, v)$, if for any edge $(u, v) \in E$ the new edge weight is greater than the old edge weight and $(u, v) \in SPT$ then add the node *v* to discover node list and make the *NW[v]* to infinity. If for any edge the new edge weight is less than the old edge weight then relax this edge. After the edge relaxation if its end node weight is updated then add this node to updated node list.

Algorithm 1: Dynamic SSSP Algorithm (G, U, NW, Source node)

```
Input
G (V, E, W): A graph with |V| vertices |E| edges and edge length W.
U (u, v, W_new): A set of updates in edge (u, v) with W_new.
NW[v]: Permanent shortest distance from source to node v.
Rep[v]: Edge responsible for node v weight
Boolean variables: Loop and Loop1 initialised with 1and
Flag array of size |V|, initialised with zero.
begin
Step 1: for all (u, v) ∈ U do in parallel

    if W(u, v) < W_new (u, v) && Rep[v]= = E(u, v) then
    NW[v]=∞
    Flag[v] = 1
    end if
    if W(u, v) > W_new (u, v) then
    if  NW[v] < NW[u] +  W_new (u, v) then
    NW[v] = NW[u] +  W_new (u, v))
    Flag[v] =1
    end if
    end if
    end for
Step 2: while Loop > 0 do
    for all (u, v) ∈ E do in parallel
    Loop=0
    if NW[u]= = ∞ && Rep[v]= = E(u, v) then
    NW[v] = ∞
    Flag[v] = 1
    Loop = 1
    end if
    end for
    end while
Step 3: for all (u, v) ∈ E do in parallel
    if NW[v] = = ∞ then
    if NW[v] > NW[u] + W_new (u, v) then
    NW[v] = NW[u] + W_new (u, v)
    end if
    end if
    end for
Step 4: while Loop1 > 0 do
    for all v ∈ V do in parallel
    Loop1=0
    if Flag[v]=1 then
    for all (v, u) ∈ E do
    if NW[u] < NW[v] + W_new (v, u) then
    NW[u] = NW[v] + W_new (v, u)
    Flag[u] =1
    Loop1 = 1
    end if
    end for
    end if
    end for
    end while
end
```

- *Step* 2: parallel traverse the *SPT* for all those nodes whose node weight is infinity and make the all discovered node *v*, node weight infinity.For any edge *(u, v) ∈ E*, if *NW[u] = infinity* and *Resp[v]* is the edge number for edge *(u, v) ∈ E*.
- *Step* 3: parallel relax all the edges whose end weight are infinity and if due to the edge relaxation end node weight is updated then add it to the updated node list.
- *Step* 4: parallel check all the nodes, if any node is part of the updated node list, then relax all the outgoing edges of the node and remove this node from the updated node list. If after the edge relaxation end node weight is updated then add end node to the updated node list. Algorithm 1 repeats the step 4 until updated node list is not empty.

4.1. Complexity Analysis

The proposed parallel dynamic SSSP algorithm is a batch dynamic algorithm, but here the complexity analysis of this algorithm is defined for two different cases; first when the single edge weight is increased and second when the single edge weight is decreased. Complexity is defined in terms of how much time the algorithm takes to update the node weight of all affected nodes after any edge weight update.

Suppose for a graph *G (V, E)*, $|V|$ is the number of nodes, $|E|$ is the number of edges where $|E| > |V|$ and *d* is the average degree of *G*. After the initial SSSP calculation, the SPT also has degree *d*. We have $|E|$ processors and total numbers of affected nodes are represented as $|affected|$. Now, let us analyse the algorithm step by step to reveal its complexity.

4.1.1. Edge Weight Increase

After the single edge weight increase in Algorithm 1 step 1 takes a constant time to process with $|E|$ processors. Each iteration of step 2 takes a constant time to process with $|E|$ processors and it can iterate $O(\log_d |Affected|)$ times. Step 3 takes a constant time to relax all edges having end node weight infinity with $|E|$ processors. Each iteration of step 4 can work $O(d)$ jobs and it can also iterate $O(\log_d|Affected|)$ times. So the complexity of the algorithm in the case of edge weight increase is $O(d^* \log_d |Affected|)$.

4.1.2. Edge Weight Decrease

After the single edge weight decrease in Algorithm 1 Steps1, 2, and 3 take a constant time to process with $|E|$ processors. Each iteration of step 4 can work $O(d)$ jobs and it can also iterate $O(\log_d |Affected|)$ times. So the complexity of the algorithm in case of an edge weight decrease is $O(d^* \log_d |Affected|)$.

5. CUDA Programming Model

Compute Unified Device Architecture (CUDA) is a general purpose parallel programming interface which was introduced by Nvidia [30] for its GPU and it comes with a software environment which uses C as a high level programming language [28]. Using CUDA, the Nvidia's GPUs are available for general purpose parallel computations. The approach of solving general-purpose (i.e., not exclusively graphics) problems on GPUs is known as General Purpose Graphics Processing Unit (GPGPU). As can be seen from Figure 1-a GPU is collection of one or more Symmetric Multiprocessors (SM) and each SM has a set of processors, shared memory and instruction unit. Each processor of SM can access the shared memory. Each processor has its private register memory and can access the device memory implemented in external DRAM. This device memory has three parts: global, constant and texture memory. Constant and texture memories are read-only memory but the global memory can be used for both read and write purposes.

CUDA programming involves running code on two different platforms: a host system with one or more CPU cores and one or more CUDA-enabled NVIDIA GPUs. The CUDA program defines kernel, a set of instructions that will be executed in parallel on different data items. The data is copied from host memory to device memory, then the kernel is executed and data is copied back to the host memory. For the kernel execution CUDA program creates multiple threads execute the kernel function parallel on different data items.

A thread grid is assigned to the GPU for processing. The grid is a collection of thread blocks, which can be arranged in a one, two or three-dimensional way inside the grid. Blocks are a collection of threads, which can be arranged in a one, two or three-dimensional way inside the block. Each thread in a block and each block in a grid is assigned a unique index for each dimension of its logical arrangement. These unique indexes are used to generate the reference for the data items on which a thread has to work.

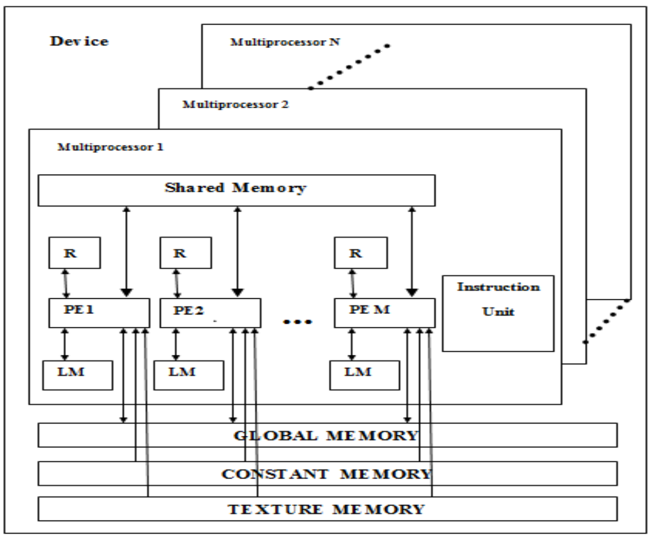

Figure1. Device understanding of CUDA.

Each block of a grid is assigned to a unique SM, and multiple blocks can be assigned to a SM. The SM divides the set of threads of its assigned blocks into a set of 32 threads called a warp. The threads present in any warp are executed concurrently and the threads of all blocks assigned to an SM are executed concurrently. Out of all warps present in an SM, warp scheduler randomly selects a warp for execution that has threads ready to execute its next instruction. The instruction unit presents inside the SM issues one instruction at a time which is executed by all the threads of the selected warp in parallel.

6. Parallel Implementation of Proposed Algorithm

This section explains the parallel implementation of the proposed algorithm for a GPU-based machine using CUDA. It uses an efficient and consistent variant of a GPU-based parallel SSSP algorithm proposed by Harish and Narayanan [16] for the initial node weight calculation for each node of the graph from a given source node. After the initial SSSP calculation it calculates the responsible edge for each node's minimum weight in the SPT.

Let a set of changes U, in the graph edge weight be applied. After these changes, to recalculate the SSSP the parallel batch dynamic algorithm is given in Algorithm 2. It uses an array NEW to store the updated edge weights and previous weights for unaffected edges. It uses four kernel functions for dynamic SSSP calculations, defined as EFFE_NODE, FIND_NODE, EDGE_RELAX and RELAX for CUDA implementation.

Algorithm 2: Dynamic SSSP (G, NW, Resp, U)

```
Create an affected node array, Aff and new edge weight array,
NEW of size |V|
NEW contains the copy of EW.
U (u, v, $len_{new}$): A set of updates in edge (u, v) with $len_{new}$
Create an array Flag of size |V|, two Boolean variables Loop
and Lock and a variable INFI contains very big number.
begin
[1] Update the new edge weight of affected edges in
    array NEW
[2] FFE_NODE( Flag, Resp, NW, Aff, INFI, EW, NEW,
    EEN, ESN) for each vertex e∈ E in parallel
[3] while Loop > 0 do
[4] Loop=0
[5] FIND_NODE(NW, ESN, EEN, Resp, Aff, Flag, INFI,
    Loop) for each edge e∈ E in parallel
[6] end while
[7] EDGE_RELAX( Aff, NW, ESN, EEN, EW) for each
    edge e∈ E in parallel
[8] Lock=1
[9] while Lock > 0 do
[10]    Lock=0
[11]    RELAX(N, ENN, NW, NEW, Flag, Lock) for each
        vertex v∈ V in parallel
[12]    end while
end
```

After storing the updated edge weights in array *NEW*, Algorithm 2 calls the Kernel 1 *EFFE_NODE* kernel to find the nodes which are affected by this change in the graph. It creates |*E*| threads, one for each edge of the graph to call this kernel. Each thread of this kernel checks if its assigned edge's new weight is greater than its old weight and if the edge is the responsible edge of its end node. Then the edge end node weight is set to infinity, node is marked affected and its flag value is set. If the new weight is less than the old weight then this edge is relaxed and if the edge end node weight is updated after the relax operation then the flag value corresponding to this node is set.

Kernel 1: EFFE_NODE (Flag, Resp, NW, Aff, INFI, EW, NEW, EEN, ESN)

```
begin
    [1]  id = getThreadID
    [2]  if EW[id]!= NEW[id] then
    [3]  if EW[id] < NEW[id] then
    [4]  if Resp[EEN[id]] = = id then
    [5]  NW [EEN [id]] =INFI
    [6]  Aff[EEN[id]] =1
    [7]  Flag [EEN [id]] =1
    [8]  Else
    [9]  if(NW[EEN[id]] > (NW[ESN[id]]+ EW[id]))
         then
    [10]  begin ATOMIC
    [11]  NW[EEN[id]]=(NW[ESN[id]]+ NEW[id]))
    [12]  end ATOMIC
    [13]  Flag[EEN[id]] =1
    [14]  end if
    [15]  end if
    [16]  end if
end
```

After finding the initial affected nodes, Algorithm 2 calls the Kernel 2 *FIND_NODE* to discover all possible affected nodes. It creates |*E*| threads to call the Kernel 2 one thread corresponding to each edge of the graph. Each thread checks that its assigned edge start node weight is infinity and it is the responsible edge for its end node and the edge end node is marked as not affected, then it sets the end node weight to infinity and marks it as affected and sets its flag value. Algorithm 2 will call Kernel 2 again, if it will find an affected node in kernel's current iteration.

After discovering all nodes possibly affected due to edge weight increase, Algorithm 2 calls the Kernel 3 EDGE_RELAX to calculate the temporary weights of the affected nodes. It creates |E| threads, one corresponding to each edge of the graph. Each thread relaxes its assigned edge if the edge end node is marked as affected. This edge relaxation operation is an atomic operation as multiple threads can try to update the weight of the same node.

Kernel 2: FIND_NODE (NW, ESN, EEN, Resp, Aff, Flag, INFI, Loop)

```
begin
    [1] id = getThreadID
    [2] if NW[ESN[id]]= =INFI  AND Resp[EEN[id]]
        = =id then
    [3] if Affected [EEN[id]] = = 0 then
    [4] NW [EEN [id]] = INFI
    [5] Aff [EEN [id]] = 1
    [6] Flag [EEN [id]] =1
    [7] Loop=1
    [8] end if
    [9] end if
end
```

Kernel 3: EDGE_RELAX (Aff, NW, ESN, EEN, EW)

```
begin
    [1] id = getThreadID
    [2] if Aff[EEN[id]] = = 1 then
    [3] begin ATOMIC
    [4] if(NW[EEN[id]]>(NW[ESN[id]] + NEW[id]))
        then
    [5] NW[EEN[id]]=(NW[ESN[id]] + NEW[id]))
    [6] end ATOMIC
    [7] end if
end
```

Lastly, Algorithm 2 calls Kernel 4 *RELAX* to find the final node weight of all affected nodes. It creates |*V*| threads to call the Kernel 4 one corresponding to each node of the graph. Each thread checks its assigned node's Flag value, and if it is set then all the outgoing edges of this node are relaxed. After any edge relaxation, if its end node weight is updated then the Flag value corresponding to the node is set. Algorithm 2 calls the Kernel 4 again, if there is any weight change in kernel's current iteration.

Kernel 4: RELAX (N, ENN, NW, NEW, Flag, Lock)

```
begin
    [1] id = getThreadID
    [2] if Flag [id] = =1 then
    [3] Flag [id] = 0
    [4] for all neighbours nid of id do
    [5]  if  NW[nid] > NW[id]+NEW[nid] then
    [6]  begin ATOMIC
    [7]  NW[nid]=(NW[id] + NEW[id]
    [8]  end ATOMIC
    [9]  Lock = 1
    [10] Flag [EEN[nid]] = 1
    [11] end if
    [12] end for
    [13] end if
end
```

7. Results and Analysis

This section discusses the experimental setup used for result evaluation, type of test graph and finally shows the results of the proposed parallel batch dynamic algorithm, its comparison with the best serial algorithm and analysis of results.

7.1. Experimental Setup

Results are evaluated on a system with following configurations:
CPU: Intel(R) Xeon(R) E5-2650 @ 2.00 GHz
RAM: 24 GB
OS: Windows 7 professional
GPU: Tesla C2075 (448 cores), compute capability 2.0
Language: CUDA 5
Programming Interface: Visual studios 2010

7.2. Test Graphs

To test the performance of proposed implementation different real world graphs available on the Stanford graph dataset [22] are used. The real world graphs that have been used are: Internet graphs in which nodes represent computers and edges represent communication; Web graphs in which nodes represent web pages and edges are hyperlinks; Social network graphs are online social networks where edges represent interconnections between people; and road network graph nodes represent the interconnections of roads and edges represent the roads connecting the interconnection. These graph instances have sizes up to 5.5 million edges, directed and are assigned positive edge weights ranging from 1 to 10.

7.3. Results

In this section, a comparison of the proposed parallel implementations of the dynamic SSSP algorithm with the serial dynamic SSSP algorithm is presented. The serial batch dynamic SSSP algorithm [2] shows best serial results is implemented in C language. Performance of proposed implementations are evaluated for three different cases: first, when the weight increase for those edges which are affecting approximately 10 % nodes weight in given graph;

second is when weight increase for those edges which are affecting approximately 5000 nodes weight in given graph; and last is when weight decrease for fifty random edges in any graph.

To find how many nodes have a minimum weight dependent on any edge in the graph, proposed algorithm first calculates the responsible edge and node for each node in SPT and then travers the SPT by using any node as a root node. Nodes present in the sub-tree after considering any node as a root node are affected nodes after increasing the weight of root node's responsible edge.

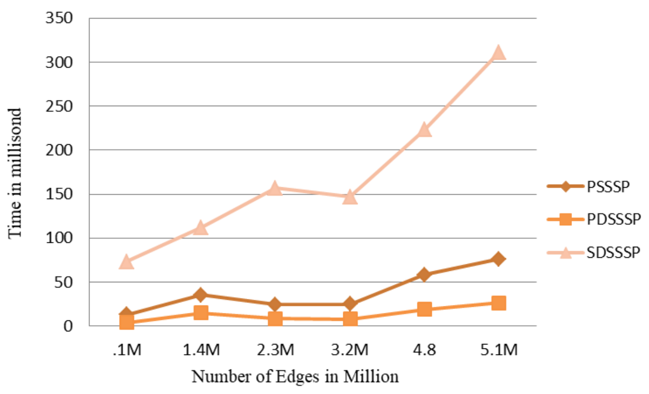

Figure 2. Results for 10% affected nodes.

Figure 2 shows the results of the proposed parallel dynamic SSSP algorithm (PDSSSP), the parallel SSSP algorithm [16] (PSSSP) and the serial dynamic SSSP algorithm [2] (SDSSSP) calculations after increasing the edge weight of those edges which affect the approximately 10% nodes weight in the SSSP of the corresponding graph. The proposed PDSSSP algorithm gives double the speed increase of the PSSSP algorithm and up to a 20-fold increase as compared to the SDSSS algorithm. The main reason of speedup is parallel recalculation of all affected node's weight using constrain based Bellman Ford algorithm [16]. This algorithm has added two conditions simple Bellman Ford algorithm [3, 14]; first condition relaxes only those edges in any iteration whose start node weight was modified in the last iteration, and second condition relaxes the edges until there is a weight change for at least one node in the last iteration.

Figure 3 shows the results for the PDSSSP and SDSSSP algorithm calculations after the edge weight increase of the edges which affect approximately 5000 nodes weight in the corresponding graph. The serial dynamic SSSP algorithm takes a similar time for re-computing the node weight of 5000 nodes in any graph, as it has to do approximately equal amount of work for all graphs. But the proposed parallel dynamic SSSP algorithm takes a different amount of time for different graphs to re-compute the node weights of 5000 nodes. This re-computing time depends on the number of nodes present in the graph because the number of threads created for any graph in the final computation depends on the number of nodes in the graph. If the number of threads is greater but they have to process the same number of nodes as fewer threads, then he former case will require more processing time because the system will take the same number of iterations but in each iteration it will take more time to manage the large number of threads.

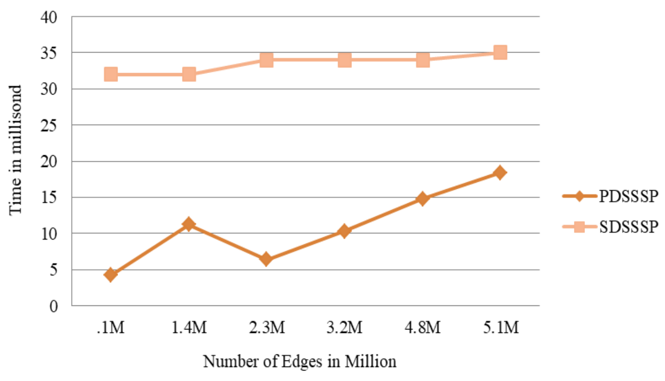

Figure 3. Results for 5000 affected nodes.

Table 1. Results edge weight decrease.

Graph Size No. of edges	Time in milliseconds		% of node weight affected
	PDSSSP	SDSSSP	
.94M	8.4	160	19%
1.4M	6.3	123	6%
2.3M	17.2	117	11%
3.2M	12.3	87	17%
4.8M	24.6	93	21%
5.1M	18.7	234	7%

Table 1 shows the results for the proposed parallel dynamic SSSP and serial dynamic SSSP algorithms after randomly reducing the edge weight of fifty edges of any graph. It also shows how many nodes weights are affected after this edge weight minimization. In the case of edge weight decrease it may be possible that dynamic SSSP processing time for small graphs will be greater than for any large graph because we have no idea how many nodes will be affected due to these edge weight decreases in any graph before re-calculating the SSSP.

Basically, speed-up depends on the structure and out-degree of a graph. As the number of vertices increases and the degree per vertex declines, then the level of a graph increases. As algorithms are dynamically updating an edge, when the level of a graph increases, more edges have to be updated, with the results that, for sequential implementation, more work has to be done while, for parallel implementation, all work is performed in a parallel manner. Hence, a greater speed increase can be achieved with parallel implementation.

8. Conclusions and Future Work

This paper has proposed first parallel batch dynamic SSSP algorithm for GPU based machine and shown its implementation using CUDA. Time Complexity analysis of proposed algorithm has shown with respect to possible affected nodes with |*E*| processors in parallel environment. Experimental results are shown for NVIDIA’S Telsa C2075 GPU and analysed for three different cases. Proposed algorithm has given up to 20 times speed up as compare to serial algorithm when the approximately 10% nodes weight are affected. When the fixed number of nodes has affected in different graph the parallel solution has taken time according to the size of the graph, so in future to solve this problem an implementation can be proposed to create the number of threads equal to the number of affected nodes in the graph.

References

[1] Barbehenn M. and Hutchinson S., “Efficient Search and Hierarchical Motion Planning by Dynamically Maintaining Single-Source Shortest Paths Tree,” *IEEE Transactions on Robotics and Automation*, vol. 11, no. 2, pp. 198-214, 1995.

[2] Bauer R. and Wagner D., “Batch Dynamic Single-Source Shortest Path Algorithms: An Experimental Study,” *in Proceedings of International Symposium on Experimental Algorithms*, Berlin, pp. 51-62, 2009.

[3] Bellman R., “On A Routing Problem,” *Quarterly of Applied Mathematics*, vol. 16, no. 1, pp. 87-90, 1958.

[4] Brodal G., Traff J., and Zarolingis C., “A parallel Priority Data Structure With Applications,” *in Proceedings of 11th International Parallel Processing Symposium*, Geneva, pp. 689-693, 1997.

[5] Buluc A., Gilberta J., and Budaka C., “Solving Path Problems on the GPU,” *Parallel Computing*, vol. 36, no. 5-6, pp. 241-253, 2010.

[6] Bura W. and Boryczka M., “The Parallel Ant Vehicle Navigation System with CUDA Technology,” *in Proceedings of International Conference on Computational Collective Intelligence*, Gdynia, pp. 505-514, 2011.

[7] Buriol L., Resende M., and Thorup M., “Speeding Up Dynamic Shortest-Path Algorithms,” *INFORMS Journal on Computing*, vol. 20, no. 2, pp. 191-204, 2008.

[8] Crauser A., Mehlhom K., Meyer U., and Sanders P., “A Parallelization of Dijkstra’s Shortest Path Algorithm,” *in Proceedings of 23rd International Symposium on Mathematical Foundations of Computer Science*, Brno, pp. 722-732, 1998.

[9] Crobak J., Berry J., Madduri K., and Bader D., “Advanced Shortest Paths Algorithms on a Massively-Multithreaded Architecture,” *in Proceedings of IEEE International Parallel and Distributed Processing Symposium*, Rome, pp. 1-8, 2007.

[10] Dashora S. and Khare N., “Implementation of Graph Algorithms over GPU: A Comparative Analysis,” *in Proceedings of IEEE Students' Conference on Electrical, Electronics and Computer Science*, Bhopal, pp. 1-8, 2012.

[11] Delling D. and Wagner D., “Landmark-Based Routing in Dynamic Graphs,” *International Workshop on Experimental and Efficient Algorithms*, Rome, pp. 52-65, 2007.

[12] Dijkstra E., “A Note on Two Problems in Connexion with Graphs,” *Numerische Mathematik*, vol. 1, no. 1, pp. 269-271, 1959.

[13] Eklund P., Kirkby S., and Pollitt S., “A Dynamic Multi-Source Dijkstra's Algorithm for Vehicle Routing,” *in Proceedings of Australian New Zealand Conference on Intelligent Information Systems*, Adelaide, pp. 329-333, 1996.

[14] Ford L. and Fulkerson D., *Flows in Network*, Princeton University Press, 2010.

[15] Frigioni D., Spaccamela A., and Nanni U., “Fully Dynamic Algorithms for Maintaining Shortest Paths Trees,” *Journal of Algorithms*, vol. 34, no. 2, pp. 251-281, 2000.

[16] Harish P. and Narayanan P., “Accelerating Large Graph Algorithms on the GPU Using CUDA,” *in Proceedings of International Conference on High-Performance Computing*, Goa, pp. 197-208, 2007.

[17] Jang H., Park A., and Jung K., “Neural Network Implementation Using CUDA and OpenMP,” *in Proceedings of Digital Image Computing: Techniques and Applications*, Canberra, pp. 155-161, 2008.

[18] Kadhar M., “A Deadlock-Free Dynamic Reconfiguration Protocol for Distributed Routing on Interconnection Networks,” *The International Arab Journal of Information Technology*, vol. 11, no. 6, pp. 616-622, 2014.

[19] Katz G. and Kider J., “All Pairs Shortest-Paths for Large Graphs on the GPU,” *in Proceedings of 23rd ACM SIGGRAPH/ EUROGRAPHICS Symposium Graphics Hardware*, Sarajevo, pp. 47-55, 2008.

[20] King V. and Thorup M., “A Space Saving Trick for Directed Dynamic Transitive Closure And Shortest Path Algorithms,” *in Proceedings of International Computing and Combinatorics Conference*, Guilin, pp. 268-277, 2001.

[21] Lattanzi S., Panconesi A., and Sivakumar D., “Milgram-routing in Social Networks,” *in Proceedings of 20th International Conference on World Wide Web*, Hyderabad, pp. 725-734, 2011.

[22] Leskovec J., “Stanford Large Network Dataset Collection,” Stanford University, http://snap.stanford.edu/data/, Last Visited, 2014.

[23] Li F., Klette R., and Morales S., “An Approximate Algorithm for Solving Shortest Path Problems for Mobile Robots or Driver Assistance,” *in Proceedings of Intelligent Vehicles Symposium*, Xi'an, pp. 42-47, 2009.

[24] Martín P., Torres R., and Gavilanes A., "CUDA Solutions for the SSSP Problem," *in Proceedings of International Conference on Computational Science*, Baton Rouge, pp. 904-913, 2009.

[25] Matsumoto K., Nakasato N., and Sedukhin S., "Blocked All-Pairs Shortest Paths Algorithm for Hybrid CPU-GPU System," *in Proceedings of 13th IEEE International Conference on High Performance Computing and Communications*, Banff, pp. 145-152, 2011.

[26] Misra S. and Oommen B., "Dynamic Algorithms for the Shortest Path Routing Problem: Learning Automata-Based Solutions," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 35, no. 6, pp. 1179-1192, 2005.

[27] Narva ́ez P., Siu K., and Tzeng H., "New Dynamic Algorithms for Shortest Path Tree Computation," *IEEE/ACM Transactions on Networking*, vol. 8, no. 6, pp. 734-746, 2000.

[28] Nickolls J. and Dally W., "The GPU Computing Era," *IEEE Micro*, vol. 30, no. 2, pp. 56-69, 2010.

[29] Nishizeki T., Tamassia R., and Wagner D., "Graph Algorithms and Applications," Dagstuhl-Seminar report, Schloss Dagstuhl, 1998.

[30] NVIDIA Corporation, "CUDA C programming guide (2013)," http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, Last Visited, 2014.

[31] Ramalingam G. and Reps T., "On the Computational Complexity of Dynamic Graph Problems," *Theoretical Computer Science*, vol. 158, no. 1-2, pp. 233-277, 1996.

[32] Reps T. and Ramalingam G., "An Incremental Algorithm for a Generalization of the Shortest-Path Problem," *Journal of Algorithms*, vol. 21, no. 2, pp. 267-305, 1996.

[33] Sancı S. and I ̧sler V., "A Parallel Algorithm for UAV Flight Route Planning on GPU," *International Journal of Parallel Programming*, vol. 39, no. 6, pp. 809-837, 2011.

[34] Singh D. and Khare N., "Parallel Implementation of the Single Source Shortest Path Algorithm on CPU-GPU Based Hybrid System," *International Journal of Computer Science and Information Security*, vol. 11, no. 9, pp. 74-80, 2013.

[35] Sintorn E. and Assarsson U., "Fast Parallel GPU-Sorting Using a Hybrid Algorithm," *Journal of Parallel and Distributed Computing*, vol. 68, no. 10, pp. 1381-1388, 2008.

[36] Tang Y., Zhang Y., and Chen H., "Parallel Shortest Path Algorithm Based On Graph-Partitioning and Iterative Correcting," *in Proceedings of 10th IEEE International Conference on High Performance Computing and Communications*, Dalian, pp. 155-161, 2008.

[37] Tran Q., "Designing Efficient Many-Core Parallel Algorithms for All-Pairs Shortest-Paths Using CUDA," *in Proceedings of 7th International Conference on Information Technology: New Generations*, Las Vegas, pp. 7-12, 2010.

Dhirendra Singh received his PhD degree in computer science and engineering from the Maulana Azad National Institute Technology, Bhopal, India in 2015. After his Post graduation, he has worked as Software Developer in NIIT Technologies Ltd., New Delhi, India. Currently he is working as Assistant Professor in the department of Computer Science and Engineering, Maulana Azad National Institute Technology, Bhopal, India.

Nilay Khare is Ex-HOD of the department of Computer Science and Engineering, Maulana Azad National Institute Technology, Bhopal, India. He is having more than 27 years of teaching and research experience. Currently he is working as Associate Professor in the department of Computer Science and Engineering, Maulana Azad National Institute Technology, Bhopal, India. His research interests include algorithms, theoretical computer science and VLSI design.