Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 Keller Hall
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 17-005

Investigation of a Transactional Model for Incremental Parallel Computing in Dynamic Graphs

Anand Tripathi, Rahul R. Sharma, Manu Khandelwal, Tanmay Mehta, Varun Pandey, Charandeep Parisineti

May 25, 2017

Revised

Investigation of a Transactional Model for Incremental Parallel Computing in Dynamic Graphs

Anand Tripathi, Rahul R. Sharma, Manu Khandelwal,
Tanmay Mehta, Varun Pandey, Charandeep Parisineti
Department of Computer Science & Engineering
University of Minnesota, Minneapolis MN 55455

***Abstract*—In many applications involving dynamic graph structures one may be interested in continuously observing certain properties of interest. We present here the result of our investigation of utilizing a transactional model of parallel programming for supporting continuous queries on dynamic and evolving graph structures. The goal of our work is to continuously monitor a graph data structure as it is updated to check for the properties of interest. One approach for continuous monitoring is to re-execute the graph analytics program on the entire graph structure after it is updated. However, this approach can lead to high computation cost and latency in case of large graphs. An alternate approach is to execute the analytics program only initially, and then perform incremental computations for supporting continuous queries as the graph data is modified. In our model, the graph updates are performed as transactions, which trigger execution of a set of transactional tasks to perform computations for a continuous query. In our testbed system, the graph data is stored in the RAM of cluster nodes, and continuous queries involve parallel execution of transactional tasks on cluster nodes. Using a set of graph problems we illustrate this approach and its performance benefits for supporting continuous queries in dynamic graph data structure.**

I. Introduction

In applications with dynamic and evolving graph structures, one may require continuous monitoring of certain properties of interest. For example, in a dynamically evolving graph, one may want to continuously determine the *shortest path* of any node to the source node, continuously find the *k-nearest neighbors* of one or more nodes in the graph, or continuously determine the maximal clique to which a given node belongs. Another example is to continuously monitor if two nodes belong to the same connected component in a graph. The primary focus of our work is on utilizing a transactional model of parallel computing for supporting continuous queries on dynamic graph structures. In dynamic graph structures, one approach for continuous monitoring is to re-execute the graph analytics program for the given property of interest on the entire graph structure after it is updated. For large-scale graph structures this approach can become expensive in terms of computation time and resource consumption. An alternate approach is to execute the analytics program only initially, and then perform incremental computations for supporting continuous queries as the graph data is modified.

We present here the result of our investigation of utilizing a transactional model of parallel programming for supporting continuous queries on large graphs using incremental computations. In this model, tasks for vertex-centric computations are executed in parallel as serializable transactions. The transactional model adopted here is supported by a parallel programming framework called Beehive [15]. Beehive executes on a cluster of computers, and the graph data is stored in the RAM of cluster nodes. Parallel computations on graph data are performed using transactional tasks. We utilize the Beehive system in our investigation for supporting continuous queries in dynamic graph structures through incremental computations facilitated by the transactional parallel computing model. In our approach, the updates to the graph structure are performed as transactions, which trigger execution of a set of transactional tasks to perform continuous query computations. Using a set of graph problems we illustrate the performance benefits of this approach. In our approach, graph updates can be applied concurrently with the execution of the incremental recomputation tasks for continuous query. Furthermore, in this approach the graph updates can also be applied in a batch mode, if necessary, before initiating recomputation tasks.

Several frameworks and systems are currently used for performing graph data analytics on computing clusters. Distributed GraphLab [12] uses a transactional model of computations with locking based concurrency control for task executions. Distributed GraphLab does not support problems with dynamic graph structures. Pregel [13] and Giraph [2] are specifically intended for graph data analysis. They are based on the Bulk Synchronous Parallel (BSP) model [16] of computing. The Galois system [14] supports speculative execution of computations for graph data analytics on shared memory multiprocessor systems.

The problems in supporting incremental computations in dynamic graph structures has been investigated in the past by many other researchers [8], [4], [6], [9], [1], [10]. Incremental computation problem for the PageRank [5] problem is presented in [8], and Incoop [4] addresses these for the MapReduce model. In [6], [17] this problem is addressed in the context of the BSP model of parallel programming. The approach in GraphInc [6] is for supporting incremental computations in the BSP model by saving the state of the preceding graph computation and applying incremental changes on the saved state. The updates and BSP-based incremental recomputations are performed in batch mode. Techniques for query processing on dynamic and time-evolving graphs are presented in [9], [10].

The primary focus and contributions of this paper are on the use of a transactional parallel computing model for supporting continuous queries using incremental computations in dynamic graph structures. We show here that this model provides simple programming abstractions for parallel programming and supports incremental computations in dynamic graph structures. In this model, incremental computations can progress concurrently with a continuous stream of graph updates. When a stream of updates is applied to the graph structure and when all of the recomputation tasks have finished execution, the Beehive system detects the quiescent state. After the quiescent state is reached, updated results for the continuous queries can then be returned as event notifications to the user-applications.

We use five graph problems to illustrate our approach. In our approach, for a given property of interest, one has to first develop a computation model to solve the problem using transactional tasks in a static graph structure. For supporting continuous queries, the next step is to identify the update transactions which modify the graph in a dynamic setting. For each type of the update transaction, one then needs to identify the conditions under which the computed results for the property of interest may get invalidated and what kind of recomputation tasks need to be triggered to obtain the query results for the modified graph structure. To illustrate this approach, we use here five graph problems: *single-source shortest path*, *k-nearest neighbors*, *graph coloring*, finding *maximal size cliques*, and finding connected components in a graph. We conducted our experiments using the most recent version of Beehive, which is more than 10 times faster than the system described in [15].

Section II presents an overview of the transactional model of parallel computing supported by the Beehive framework and utilized in this work for supporting continuous queries in dynamic graph structures. Section III describes the approach to solve the five graph problems using the transactional parallel programming model. Section IV presents our approach for performing parallel incremental computations for supporting continuous queries. We use the five graph problems noted above to illustrate our approach. Using these problems, in Section VI we present the results of our performance evaluation experiments to show the benefits of supporting continuous queries through incremental computations. Section VII summarizes the results of our investigation and presents the conclusions.

II. Transactional Model of Parallel Programming

A parallel program is composed of a set of vertex-centric computation tasks. A task performs computation on some specified node, and it can also read or update other nodes in the graph. A task may create new tasks on its completion. The central concept is to execute the computation tasks as serializable transactions [3], satisfying the properties of *atomicity* and *isolation*. We refer to these computations as *transactional tasks*. Multiple tasks can be executed in parallel. Each such task is required to be *well-formed* in the sense that its atomic computation step transforms the graph data from one consistent state to another. The successful completion of a transactional task execution may result in modifying the graph data and creation of new tasks.

The transactional programming model noted above is supported in the Beehive [15] framework. Beehive provides the abstraction of a global object storage system, which stores the graph data in the RAM of the cluster nodes. This global storage system provides key-value based data management with location-transparent access, eliminating any explicit message-passing. A parallel program can modify the graph structure at runtime to add/remove nodes and edges, or modify their attributes. Such updates can be performed as transactional tasks. The base *Node* class contains a core set of data items such as the node-id and information about the edges to its neighbor nodes. A node object is read or written using its node-id as the access key. An application can extend this class as needed.

A distributed task-pool [7] in the system maintains a set of tasks to be executed. On each Beehive cluster computer node, a pool of worker threads is maintained. A worker thread picks a task from the task-pool and performs its computation as a transaction. A task is removed from the task-pool upon its successful execution.

The model for transactional task execution is based on optimistic concurrency control techniques [11]. This is a lock-free model of execution, and a transaction reads only committed data. Under the optimistic execution model, tasks can be executed in parallel. The optimistic execution of a transactional task involves the following four phases: *read phase*, *compute phase*, *validation phase*, and *update phase*.

The task execution begins by first obtaining a *start timestamp* for the transaction. This is a logical timestamp, and it indicates the sequence number of the latest committed transaction such that the updates of all committed transactions with timestamps up to that value have been written to the global storage. A transactional task is committed in the validation phase only if it does not conflict with any other concurrently committed transactional tasks. The conflicts are defined based on the notion of *read-write* or *write-write* conflicts [3]. If two concurrent transactional tasks conflict, then one of them is guaranteed to complete execution while the other is aborted. The aborted task is re-executed again later. On successful validation, the transaction is assigned a commit timestamp. On committing, the transactional task writes the buffered updates to the global storage and any new tasks created by it are added to the task-pool.

Each cluster node runs a Workpool server, which implements a task-pool. The set of Workpool servers executing on the cluster nodes collectively support distribution and scheduling of tasks in the cluster. The *getTask* method of the local Workpool is called by a worker thread to fetch a task for execution. The programming framework provides the base *Task* class which contains some basic information about the task, such as its *taskId*, *nodeId* of the target node for which the task is intended, and an *affinity* specification to provide hints

for putting the task in the task-pool of a particular node in the cluster. A parallel program can extend this class to include any other data to be passed as a parameter to the task computation.

The Workpool server provides three operations to the application programs for adding new tasks to the task-pool. The method *addTask(T)* puts task T in the local task-pool or selects a location according to the specified affinity level. The *broadcast(T)* method creates a copy of the given task in the task-pool on each of the cluster nodes. This primitive is generally used for executing some initialization task at each of the cluster nodes. The third method for adding tasks to the task-pool is the *reportCompletion* method, which is called by a transactional task when it is committing. One of the parameters of this method is a set of new tasks created by the task computation. These new tasks are distributed across the cluster nodes according to the affinity level specified in each task and a configurable distribution policy.

The termination of a parallel program execution is detected by the Beehive run-time system when the following three conditions hold at all computing nodes in the cluster: (1) all worker threads are idle, (2) there are no pending tasks in all of the Workpools, and (3) there are no messages in the communication network. The framework detects the termination and communicates it to the application program. The application can then initiate a new computation phase, if needed.

III. Parallel Programming of Graph Problems

The Beehive framework provides the abstract *Worker* class, as shown in Figure 1. It fetches a task from the local task-pool, implemented by the Workpool server, by calling its *getTask* method. The worker then calls *beginTransaction* primitive to execute the task computation as a transaction. It passes the task to the *doTask* method for execution. This is an abstract method in the base Worker class, and its implementation should be provided by a problem-specific concrete worker class. This method performs the computation and updates the read/write sets of the transaction. It also returns a set of new tasks to be added to the task-pool.

After the execution of the *doTask* method, the worker thread performs transaction validation by invoking the *validate* method of the *validator* object, which represents the interface to the global validation service. The read-set and the write-set are passed as parameters to the validation function. On successful validation, the task is removed from the task-pool by invoking the *reportCompletion* method of the Workpool server. The set of updated objects (write-set) and the set of new tasks are passed as parameters to this method. If validation fails, the worker thread re-executes the task as a new transaction.

We now illustrate the transaction-based parallel programming model using the three graph problems: the *single-source shortest path* problem, the *k-nearest neighbors* for all vertices in a graph, *graph vertex coloring*, finding *maximal size cliques*, and finding *connected components* in a graph. Later we use these problems for developing incremental computation techniques to support continuous queries in dynamic graph structures.

```
public class Worker extends Thread {
 Set<Node> readSet, writeSet;
 Set<Task>  newTasks;
 public void run() {
  while(true) {
   Task task  = Workpool.getTask();
   finished = false;
   while (!finished) {
    tnxnId = beginTransaction();
    readSet = new Set(); //read  objects
    writeSet = new Set(); //updated objects
    newTasks = new Set<Task>(); //new tasks
    doTask(task);
    status = validator.validate(txnId, readSet,
                                writeSet);
    if (status == commit) {
     Workpool.reportCompletion(txnId, writeSet,
                               newTasks);
     finished = true;
    }
    else { abortTransaction( txnID );
           sleep( delayInterval );
    }
   }
  }
 }
 abstract void doTask(Task t) {
  // Application defined implementation
 }
}
```

Fig. 1. Base Class for Worker

A. Single-Source Shortest Path (SSSP) Problem

This program computes the shortest distance path from a given source node to each of the other nodes in an undirected graph. Each node maintains its currently known distance to the source node, and the id of the *predecessor* node on the shortest path to the source. When the currently known distance of a node to the source node decreases, new tasks are created for each of its neighbor nodes. Initially, a start-up task is executed at the source node to create a task for each of its neighbors. A task in this program contains the following additional fields: *senderId* containing the id of the node whose distance has changed, and *distance* indicating the new distance of this node from the source. Figure 2 shows the implementation of the *doTask* method in the worker class for this problem. We include one additional optimization to reduce generation of redundant tasks. We outline it below but omit these details from the code shown Figure 2. We do not generate a task for a node if we find that the new task execution at the target node will not reduce its distance to the source. For this purpose, each node also maintains information about the currently known distances of its neighbors to the source node. This information is updated when a task is received from a neighbor.

B. K-Nearest Neighbors (KNN) Computation

For a given undirected graph, the k-nearest neighbors program computes for each node the list of its k nearest nodes and their distances. Each node maintains a sorted list of its currently known k-nearest neighbors. This list is sorted

```
public class SPWorker extends Worker {
  public void doTask(Task task) {
         compute( task );
  }
  public void compute (SPTask task) {
    String nodeId  = task.nodeId;
    String senderId = task.senderId;
    Node  u = storage.getNode(nodeId);
    Edge  e = u.neighbors.get( senderId );
    if (u.distance > t.distance + e.length) {
       writeSet.add( u );
       u.distance = t.distance + e.length;
       u.predecessor = t.senderId;
       Set<String> nbrIds = u.neighbor.keys();
       foreach (String nbr in nbrIds) {
         SPTask t = new SPTask();
         t.nodeId = nbr;  // target node
         t.senderId = u.nodeId;
         t.distance = u.distance;
         newTasks.add( t );
       }
    }
  }
}
```

Fig. 2. Worker for the Single-Source Shortest Path Problem

in the ascending order of the neighbors' distances from the node. We refer to it as the *KSet* of that node. In the *KSet* of a node, for each member we maintain "via" information which points to the node's direct neighbor on the path to that member. Whenever, the *KSet* of a node gets modified, a new transactional computation task of type *KNNTask* is created for each of its direct neighbors to recompute their $KSets$. This task contains the id of the "sender" node whose *KSet* was modified and the new value of the sender's *KSet*. Figure 3 shows the structure of the KNN worker thread which executes the *KNNTasks*. Initially, one task is generated for each node, which computes its initial list of k-nearest nodes based on the distances to the direct neighbors and it then creates a *KNNTask* for each of its neighbors.

```
public class KNNWorker extends Worker {
  int K = k ; //paramemer for k-neighbors
  public void doTask(Task task) {
         compute( task );
  }
public boolean compute(KNNTask task){
   String nodeId  = task.nodeId;
   String src  = task.senderId;
   Node  u = storage.getNode(nodeId);
   int d = u.neighbors.get(senderId).length;

   boolean change = merge(u.KSet,task.KSet,d,src);
   if( change ){
       writeSet.add(u);
       Set<String> nbrIds = u.neighbor.keys();
       foreach (String nbr in nbrIds) {
         KNNTask t = new KNNTask();
         t.nodeId = nbr;  // target node
         t.senderId = u.nodeId;
         t.KSet = u.KSet;
         newTasks.add( t );
       }
   }
}
```

Fig. 3. Worker for the KNN Problem

The computation for a *KNNTask* involves updating the target

```
private boolean merge(KSet local, KSet remote,
                      int d, String sender) {
  boolean change = false;
  foreach (key in remote.keys() ) {
    if (key present in local) {
      int current  = local.get(key).distance;
      int distViaSender = remote.get(key).distance+d;
      if (current > distViaSender)
         local.get(key).distance = distViaSender;
         local.get(key).via = sender;
         change = true;
      }
      else if (local.size() < K) {
         local.addElement(key, distViaSender)
         local.get(key).via = sender;
         change = true;
      }
      else if (distViaSender<local.last().distance){
         local.removeLast();
         local.addElement(key, distViaSender)
         local.get(key).via = sender;
         change = true;
      }
  }
  return change;
}
```

Fig. 4. Updating KSets in the KNN Problem

node's *KSet* based on the *KSet* of the sender node contained in the task, and taking into consideration the distance between the sender node and the target node. This computation is performed by the *merge* method shown in Figure 4. In this method, *local* refers to the *KSet* of the target node and *remote* refers to the *KSet* of the 'sender' node. It updates the target node's *KSet*. It returns *false* if there is no change to the target's *KSet*, otherwise it returns *true*. If the execution of the *merge* results in updating the *KSet* of the target node, new tasks are created for its direct neighbors. Each task contains the target node's id as the 'sender' and the updated *KSet* of the target, and a list containing the ids and the distances of its k-nearest nodes.

C. Graph Coloring Problem

In the graph coloring problem, the color of a node is indicated by a positive integer, with 0 indicating that the node is not colored. The goal of this problem is to color each node of the graph with the smallest available color such that this color is different from the color of any of its neighbor nodes. In the beginning, none of the nodes are colored. A task is created for each graph node and added to the task-pool to color the node such that the color assigned to the node is different from those of its neighbors. Figure 5 shows the *GraphColorWorker* class defined for the graph coloring problem. It implements the *doTask* method, which reads the colors of the neighbor nodes, and assigns to the target node the smallest number color not used by any of its neighbors. The write-set contains the task's target node, and the read-set contains all the neighbors. In this problem, no new task is created by the transactional task as we create a coloring task for each node when the program execution is started.

```
public class GraphColorWorker extends Worker {
  public void doTask(Task task) {
        compute( task );
  }
  public void compute(Task task) {
    Node u = storage.getNode( task.nodeId );
    // u is target node to be colored
    // Read all neighbor nodes of u
    Set<Node>  Nbrs = getNeighbors( u );
    Vector<Integer> NbrColors = getNbrColors(NBrs);
    Collections.sort( NbrColors );
    int targetColor = 1;
    // Find smallest unassigned neighbor color
    foreach (Integer color in NbrColors) {
      if (color > targetColor ) {
          break;
      } else if (color == targetColor) {
                targetColor++;
      }
    }
    u.color = targetColor;
    writeSet.add(u);//add node u to write-set
    readSet.add(Nbrs);//add neighbors to read-set
    //No new task is created in this example
  }
}
```

Fig. 5. Worker for the Graph Coloring Problem

D. Maximal Clique Computation

In this problem, for each node in the graph we want to find the maximal size cliques to which that node belongs. In order to compute the maximal clique for a node, say u, we need to first compute for each of its neighbors, say v, all 3-size cliques (henceforth referred as *triples*) to which v belongs. The computation of maximal clique of node u then involves collecting the *triple* sets from all its neighbors. It then checks for the existence of all cliques of size 4 involving u, using the logic as noted below. Suppose that for node u we have computed its triple set as $\{(u,v,w), (u,v,x), (u,w,x)\}$, and if we find in any of its neighbors' triple-sets a member $\{(v,w,x)\}$, then we can infer that we have a 4-size clique involving (u,v,w,x). If any such clique is found, then after the computation has identified all cliques of size 4 to which u belongs, it proceeds to identify cliques of size 5 for u, following similar steps as noted above. This iterative step continues to find all cliques of size (*k+1*) if it succeeds in finding any clique of size *k*. This iterative step terminates if the computation fails to find any clique of the next higher size.

Figure 6 shows the Worker thread class for this problem. The maximal clique program execution is structured into two phases. In the first phase, called "ComputeTriple", for each node we execute a task to compute its set of triples. In the second phase, called "ComputeCliques", for each node maximal size cliques are computed. In the first phase, a task is created for each node in the graph to execute the *computeTriples* method for that node, as shown in Figure 6. The "phase" of each of these tasks is set to "ComputeTriple". It reads all the neighbor nodes of the target node N, using the *getNeighbors* method. Then for each node P in the target node's neighbor-list, it finds all common neighbors. If a target node N (whose node-id is n) and its neighbor P (whose node-id is p) have a common neighbor with node-id q, then the

```
public class CliqueWorker extends Worker {
  public void doTask(Task task) {
    String nodeId  = task.nodeId;
    Node  N = storage.getNode(nodeId);
    if (task.phase == "ComputeTriple") {
        computeTriples(N);
    }
    else if (task.phase == "ComputeCliques") {
         computeCliques(N);
   }
  }
  public computeTriples(Node N) {
    Vector<Node> nbrs = getNeighbors(N);
    Set nbrIds =  getNeighborIds(N);
    String n = N.nodeId;
    //for each neighbor, find common neighbors
    foreach ( P in nbrs ) {
      Set pNbrs = getNeighborIds(P);
      String p = P.nodeId;
      Set commonNbrs = pNbrs.retainAll(nbrIds);
      foreach ( q in commonNbrs ) {
        N.myTriples.add("n,p,q");
      }
    }
    writeSet.add(N);
    readSet.add(nbrs);
  }
  public void computeCliques (Node N){
    Vector<Node> nbrs = getNeighbors(N);
    foreach ( m in nbrs ) {
      tripleSet[m.nodeId] = m.myTriples;
    }
    int k = 4; //next clique size to check
    while (k != 0) {
      HashSet kCliques = findCliques(k);
      if (kCliques.size() != 0) {
        N.myCliques[k] = kCliques;
        k = k+1;
      } else  k = 0;
    }
    writeSet.add(N);
    readSet.add(nbrs);
  }
}
```

Fig. 6. Worker for the Maximal Clique Problem

triple *(n,p,q)* is recorded in node N's *myTriples* set. In the second phase, a task is created for each node to compute its maximal size cliques. The phase of this task is set to "ComputeCliques". This task computation is performed by the *computeCliques* method. This method first reads the *myTriples* of all the neighbors of the target node, and then it iteratively checks for the existence of cliques for size 4 and higher values. The detail of the *findCliques* method is omitted.

E. Connected Components (CC) Problem

This problem computes the sets of all nodes that are connected directly or indirectly to each other in the graph. Each such group of *connected nodes* form a connected component. Each node maintains the information about the *label* of the connected component to which it belongs. All the host machines maintain a set of *labelled* and *unlabeled* nodes present locally in their storage system at all times. Initially, each host selects a *seed node* from its set of unlabeled nodes, labels the seed node with its own node id and starts labelling all its neighbor nodes to the same label. In order to label the neighbor

```
public class CCWorker extends Worker {
  public void doTask(Task task) {
         if(task.subphase == "label-nodes")
           compute( task );
         else if(task.subphase == "merge-labels")
           relabel( task );
  }
  public void compute(Task task) {
    // u is target node to be labeled
    Node u = storage.getNode( task.nodeId );
    if(task.hasNoLabel())
      u.label=u.nodeId;
    else if(u.hasNoLabel())
      u.label=task.label;
    else if(u.label!=task.label)
      recordClash(task.label, u.label);
    //add node u to write-set
    writeSet.add(u);
    // Read all neighbor nodes of u
    Set<String> nbrIds = u.getNbrIds();
    foreach (String nbr in nbrIds) {
      Task t = new CCTask();
      t.subphase=1;
      t.nodeId = nbr;  // target node
      t.label = u.label;
      newTasks.add( t );
    }
  }
  public void relabel(Task task) {
    // u is target node to be relabeled
    Node u = storage.getNode( task.nodeId );
    u.label=task.label;
    //add node u to write-set
    writeSet.add(u);
  }
}
```

Fig. 7. Worker for the Connected Components Problem

nodes, it creates transactional tasks (containing its own label) for each of its neighbors. On receiving a transactional task, the target node labels itself with the label present in the task and creates tasks for all of its neighbors to label them with its own label. In this way, all the nodes that are reachable from a seed node get labeled with the label of that seed node. For each connected component, its label corresponds to the node id of one of its member nodes. This property may be essential in various applications that will make use of the results of connected components computation.

Since any connected component can include nodes that are spread across several host machines, it is possible that seed nodes selected at different hosts may correspond to the same connected component. In this case, the tasks created to label the neighbor nodes may try to label a neighbor which was already labeled by a task originating from a different seed node. This essentially means that both the labels correspond to the same connected component. All such label-clashes corresponding to the same connected component form an *equivalence set*, and all the nodes in this connected component can be labeled with the smallest label from this label-clash equivalence set. Therefore, whenever a task tries to label an already labeled node with a different label, we store this information as a label-clash in the storage system globally and the task terminates without changing the label of the target node.

After all the above mentioned transactional tasks have completed their execution, the master thread reads the global

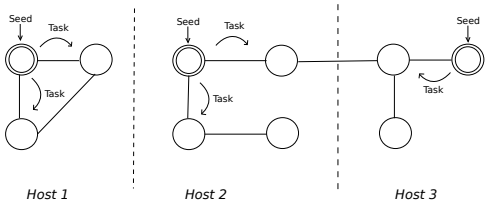

Fig. 8. Selection of seed and the label-nodes sub-phase

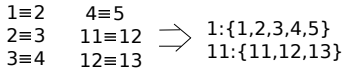

Fig. 9. Merging of equivalence sets

label-clashes registered. The master forms the equivalence sets of all the labels that clash with each other. A relabel task is created for each of these nodes to relabel them to the smallest label from their label-clash set. In this way, all the nodes corresponding to the same connected component gets labeled with the same label. The information of all the locally present labeled nodes maintained by the host is used to create local relabel transactional tasks to relabel the required nodes present in the storage system of each host.

In this way, the process of labeling any connected component consists of two sub-phases (a) *label nodes* phase, and (b) *merge-labels* phase. In the *merge-label* phase equivalence sets are derived from all label-clash sets and all nodes in a connected component are assigned a single label. Figure 8 and Figure 9 show examples of these two phases. Figure 7 shows the *CCWorker* class defined for the connected component problem. It implements the *doTask* method, that contains the logic for the two sub-phases discussed above. These two phases are repeatedly executed until all nodes are labeled. To speed up the entire process, multiple seed nodes are selected per host in every cycle, instead of only one.

IV. Continuous Queries in Dynamic Graph Structures

We now illustrate how the transactional programming model of Beehive can be used for supporting continuous queries through incremental computation in dynamic graph structures. The goal is to eliminate the need of executing an analytics program again to get the new results for a query when only a small number of updates are made to the graph structure. Re-executing the program again after a small number of updates can be potentially inefficient and expensive for large graphs. Our approach is to reduce this computation cost by applying the changes in the graph directly on the graph data state resulting from the initial execution of the analytics program and perform only the necessary incremental computations. Our work is driven by the requirement of supporting concurrent execution of a stream of graph updates with the incremental recomputation operations for a continuous query. The graph updates can also be applied in a batch mode, if necessary, before initiating recomputation tasks.

A. Model for Incremental Computations

In our approach, the graph data structure is updated using the same transactional task execution model which we use for performing parallel computations for the initial analytics program for the problem. In this model, the commitment of a transactional task can result in creation of new transactional tasks, called *cascaded tasks*, in addition to modifying the graph structure. For a given problem, we refer to the transactional tasks executed in the initial computation for the problem as the *initial computation tasks*. The tasks that are created to perform incremental re-computations are called *incremental computation tasks*.

When the initial query program for a given graph problem terminates, the system reaches a quiescent state and there are no more tasks in the system. The resulting state of the graph data satisfies certain properties related to the requirements of the problem solution. In designing the logic for incremental computations, one need to analyze how a given update transaction on the graph affects the properties of the result state for the problem. An update can perturb the quiescent state by invalidating these properties for some of the nodes in the graph. In such cases, updating the graph data would require some incremental computation tasks to be executed to preserve the required properties, and these tasks may be different from the initial computation tasks. We need to determine whether the nature of the tasks required for incremental computations are identical to the basic computation tasks. It is generally easier to identify the logic for incremental computation when the properties to be maintained for a given problem are *local* to each of the nodes, i.e. the property is defined for a node based on its own state and that of its neighbors. We illustrate this below using five graph problems: finding shortest path from any node to the source node, computing k-nearest neighbors for a node, graph vertex coloring, finding the maximal size cliques a node belongs to, and finding the connected components of a graph. We consider here three types of update operations on graph structure: addition of new edges, deletion of existing edges, and updating properties of some existing edges in the graph. Addition and deletion of nodes can be emulated using multiple edge addition transactions and multiple edge deletion transactions respectively. So, we omit further discussion of addition and deletion of nodes in the graph for the five graph problems noted above.

B. Continuous Queries for Single Source Shortest Path

We now develop a solution for incremental computations of continuous queries for the SSSP problem when graph updates are applied after the initial execution of the program for this problem. After executing the SSSP program using the task computation logic shown in Figure 2, we obtain for each node in the graph its shortest *distance* from the source and the *id* of its predecessor node on path to the source. We consider the following types of updates to a graph after the SSSP execution: adding edges, deleting edges, and updating edges. For supporting incremental computations we define two new task classes: *UpdateGraphTask* and *RecomputeSPTask*. The task class *UpdateGraphTask* performs the update graph transaction that will modify the structure of the graph and create *RecomputeSPTask*. *RecomputeSPTask* will perform incremental computations at a node, and it may create other *RecomputeSPTasks*.

1) Addition of Edges : In the update transaction for adding a new edge between a pair of nodes, we update the neighbor list of the two nodes and check (for both the nodes) if the shortest path for the node has decreased further by addition of this new edge. This is determined by using equation 1 for each of the neighbor node *'nbr'* of the given node *'n'* with edge length *'l'*:

$$dist(nbr) + l < dist(n) \tag{1}$$

The *performAddEdge* operation is shown in Figure 10 below. Addition of a new edge requires no further computations if the new edge creates a path that has a greater length than the existing shortest path of the node. So, we need to perform only one task in this scenario. If the shortest path for the node decreases, then we send tasks to all of its neighbor nodes, except the predecessor, informing them that the shortest path for this node has decreased further. This new task for the neighbor nodes is exactly the same as the tasks in the basic computations (shown in Figure 2) for the SSSP problem.

```
public void performAddEdge(SPNode tgtNode,
UpdateGraph t){
  String nbrId = getNeighbor(t);
  int edgelength = getEdgeLength(t);
  SPNode nbrNode = storage.getNode(nbrId);
  addEdge(tgtNode,nbrNode,edgelength);
  addEdge(nbrNode,tgtNode,edgelength);
  writeSet.add(tgtNode);
  writeSet.add(nbrNodes);
  if (nbrNode.distance+edgelength
      <tgtNode.distance ){
    SPTask newTask = new SPTask(tgtNode);
    tasksCreated.add(newTask);
  }
  else if (tgtNode.distance+edgelength
           <nbrNode.distance) ){
    SPTask newTask = new SPTask(nbrNode);
    tasksCreated.add(newTask);
  }
}
```

Fig. 10. Transaction to add a new edge

2) Updating of Edge Lengths : When we update the length of an existing edge between a pair of nodes, we update the neighbor list of the two nodes and check (for both the nodes) if the edge is a predecessor edge or not, and if the edge length has increased or decreased. If the updated edge corresponds to a non-predecessor edge and the edge length has increased, then the updated edge cannot improve the shortest path distance of any node. So, we do not require any further computations in such a case. If the updated edge corresponds to a decrease in edge length or the edge is a predecessor edge for a node, then we have the following two possible cases to consider.

Case 1: The shortest path distance of one of the two nodes has decreased by updating the edge. This case can

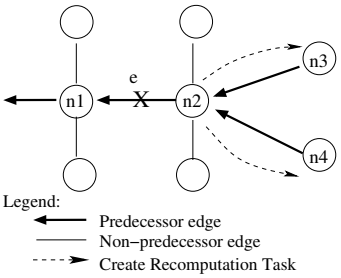

Fig. 11. Recomputation tasks when a predecessor edge is deleted or its length is increased

happen when the *length* of an edge is decreased. The update transaction determines the new distance and predecessor for the affected node. In this case, a task is sent to all of its neighbor nodes, except for the new predecessor, informing them that the shortest path for this node has decreased further. This new task for the neighbor nodes is identical in nature to the basic SSSP task.

Case 2: Updated edge is a predecessor edge and its length has increased. In this case, the shortest path distance will increase for the node for which this edge is the predecessor edge. We refer to this node as the target node. This scenario is illustrated in Figure 11. The update transaction reads the shortest path distances of all its non-predecessor neighbor nodes. It then computes the target node's shortest path distance and the new predecessor, using equation 1. After computing the shortest path distance for the target node, the update transaction creates *RecomputeSPTasks* for all its neighbor nodes that have the target node as their predecessor, so that they can recompute their own shortest path distance. Note that we do not need to create *RecomputeSPTasks* for the neighbors that do not have the target node as their predecessor because they have a "shortest path" to the source through some other node.

3) Deletion of Edges : The transaction for deleting an existing edge between a pair of nodes removes the edge from the neighbor lists of the two nodes and checks if the edge was a predecessor edge for any of them. If the updated edge corresponds to a non-predecessor edge, then we do not require any recomputations, as deleting a non-predecessor edge cannot affect the shortest path distance for any of the nodes.

This scenario is illustrated in Figure 11. If the deleted edge is a predecessor edge for any of the two nodes, then the scenario is similar to Case 2 in the *Update Edge* transaction, with the only change that the edge is deleted instead of being updated. In this case, the shortest path of the target node will increase from its previously known value. The update transaction reads the distance values of all the neighbor nodes of the target node that do not have the target node as their predecessor. It then computes, for the target node, the shortest path distance and a new predecessor using equation 1. If there are no such nodes, then the distance of this node from source node is set to infinity and its predecessor is set to *unknown*. The update transaction then creates a *RecomputeSPTask* for each of its neighbor nodes that have the target node as the predecessor, so that they can recompute their shortest path distance.

C. Continuous Queries for K-Nearest Neighbors

We now consider the problem of supporting continuous queries for the KNN problem using the incremental computation approach. We develop a solution for continuous queries when graph updates are applied after an initial execution of the basic algorithm for this problem, presented in Section III-B. After executing the KNN program on a given input graph data using the task computation logic shown in Figure 3 and Figure 4, we obtain for each node in the graph a set of k-nearest neighbors. We consider the following types of updates to a graph after the execution of the initial program for the KNN computation: adding edges, deleting edges, and updating edges in the graph.

```
private void addEdge(KNNNode x, KNNNode y, int w){
  //add an edge of length w between nodes x and y
  insertEdge(x,y,w);
  insertEdge(y,x,w);
  writeSet.add(x);
  writeSet.add(y);

  //do steps for x
  flag = false;
  if(x.KSet.contains(y)){
    if(x.KSet.get(y).distance > w){
      x.KSet.replaceElement(y,w);
      x.KSet.get(y).via = y;
      flag = true;
    }
  }
  else if(x.KSet.last().distance > w){
    x.KSet.addElement(y,w);
    x.KSet.get(y).via = y;
    flag = true;
  }

  if(flag == true){ //KSet has changed
    for(KNNNode nbr : x.neighbors){
      KNNTask task = new KNNTask(nbr,x,x.KSet);
      newTasks.add(task);
    }
  }

  //do the same steps for y as done for x

}
```

Fig. 12. KKN Computation for Add Edge Transaction

1) Addition of Edges : The update transaction for adding a new edge between a pair of nodes, say *m* and *n* is executed as a transactional task. Figure 12 shows the operations performed by an *Add Edge* transaction. The task performs the following actions. For node *m*, if *n* is in the *KSet* and the distance for *n* in the *KSet* is greater than the length of the new edge, then we update the distance of node *n* as well as *'via'* to point to *n*. Otherwise, if the size of the *KSet* is less than *k*, then we add *n* to it. Else, if the new edge has its length smaller than the highest distance value among the k-nearest neighbors then we update the *KSet* by adding *n* to it. In case the *KSet* has changed, we propagate the change by creating a *KNNTask* for each neighbor of *m*. The above computations are similarly

```
public void deleteEdge(m, n){
  //delete the edge between nodes m and n
  deleteEdge(m,n);
  deleteEdge(n,m);
  writeSet.add(m);
  writeSet.add(n);

  if(m.KSet contains n){
    Step 1: Remove from KSet all entries s.t.(via=n)
            Add these node-ids to m.deleteSet

    Step 2: Add all m's neighbors in m.KSet to fill
            the vacancies.

    Step 3:
      3.1) Get KSets from all neighbours of m
           (except the neighbor who sent this task)
      3.2) From the KSets of all its neighbors remove
           all entries having via = m
      3.3) Merge each nbr's KSet obtained from 3.2
           into m.KSet

    Step 4: If m.KSet is modified, create
            KNNRecomputeTask for all its neighbors,
            passing m.KSet and m.deleteSet as
            parameters to it.
  }

  if(n.KSet contains m){
    //do the same steps as for m
  }
}
```

Fig. 13. KNN computation for Delete Edge Transaction

```
public void recompute(nbr, m.KSet, m.deleteSet){
 nbr.deleteSet = {}
 Step 1: From nbr's KSet remove all the elements
         that have (via=m) and are also
         present in m.deleteSet
         add these elements to nbr.deleteSet;

 Step 2: Add all nbr's neighbors in nbr.KSet to
         fill the vacancies.

 Step 3: foreach of  nbr's neighbor w, except m:
           merge(nbr.KSet, w.KSet);

 Step 4: Create recompute tasks for all neighbors
     if(nbr.KSet is modified){
       for(u  in nbr.neighbors){
         if (nbr.deleteSet.size()==0) {
           KNNTask newTask = KNNTask(u, nbr.KSet);
         } else {
            KNNRecomputeTask newTask =
            new KNNRecomputeTask(neighbor, nbr.KSet,
                                 nbr.deleteSet);
         }
         tasksCreated.add(newTask);
       }
     }
}
```

Fig. 14. KNN Recomputation Task

executed for the node *n*, in the same transaction. In this case the tasks executed for incremental computations are of the same type as executed during the initial computation.

2) Deletion of Edges : In the update transaction to delete an existing edge between a pair of nodes, we execute the steps shown in Figure 13 on both the nodes as a transactional task. We use the graph shown in Figure 15 as an example to

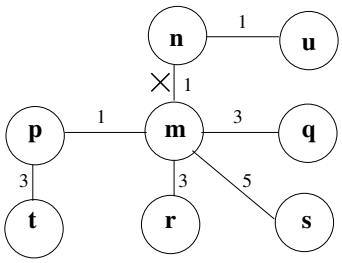

Fig. 15. An example of Delete Edge operation

illustrate the steps involved in this transaction and the resulting incremental computations. In this illustration the value of k used is 4. Suppose an edge between *m* and *n* is deleted as shown in Figure 15. The initial states of the *KSet*s of *m* and one of it's neighbor *p* are shown in Figure 16. The *KSet* of *m* needs to be updated because of the deletion of its edge to *n*. We delete all entries from *m*'s *KSet* which are going through *n* (as shown in step 1 in Figure 17), since these entries are no longer valid. Hence, the entries of *n* and *u* will be removed from *m's KSet*. We record all deleted entries from *m's KSet* in the *deleteSet*. Then, to fill in the vacancies in the *KSet*, we first add the neighbors of *m* to its *KSet*, as shown in step 2 of Figure 17. Then we get the *KSet* from all the neighbors of *m*, and delete from it all the entries going through *m*. We then merge it with *m*'s *KSet*. Figure 18 illustrates this for merging of neighbor *p*'s *KSet*. After this step all the entries in *m*'s *KSet* are correct and no further computations are needed for *m*. But now we need to propagate *m*'s new *KSet* to all its neighbors to update their *KSet*. For this we create a *KNNRecomputeTask* for each of *m*'s neighbors. Each of these tasks contain *m*'s *KSet* and its *deleteSet*.

The steps executed by a *KNNRecomputeTask* are shown in Figure 14. In Figure 19, we illustrate the execution of the *KNNRecomputeTask* at one of *m*'s neighbor, say node *p*. In the recompute method, we first delete all entries in *p*'s *KSet* which are present in the *deleteSet* passed from *m* and if *p* has a route to those nodes through *m*. These entries are now invalid. In a new *deleteSet*, we keep track of the nodes deleted from *p*'s *KSet*. On node *p* we now execute computations similar to those done on node *m*. We first add the neighbors of *p* in its *KSet* to fill any vacancies created due to deletions of elements in the above step. Then we fetch the *KSet* from all of *p*'s neighbors and merge them with *p*'s *KSet*.

If node *p*'s *KSet* gets modified, then we need to initiate incremental recomputation tasks on all its neighbors other than *m*, following the steps shown in Figure 14. The nature of the tasks to be created depends on whether the *deleteSet* of *p* is empty or not. If the *deleteSet* is empty, then we need to create tasks of type *KNNTask* for the neighbors, as is done in the initial computation logic. Otherwise, tasks of type *KNNRecomputeTask* are created, which contain the *KSet* and *deleteSet* of node *p*

3) Update of Edges : If the update operation on an edge decreases its length, the recomputation operation is similar to that for adding a new edge between the two nodes. On the

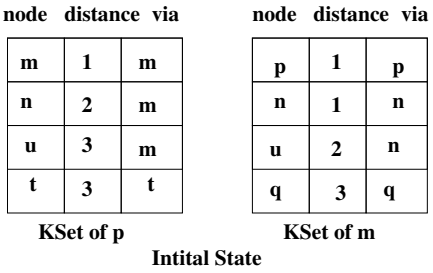

Fig. 16. KNN Edge Delete Example: initial state

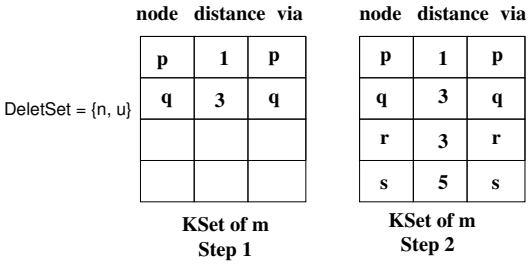

Fig. 17. KNN Edge Delete Example: step 1

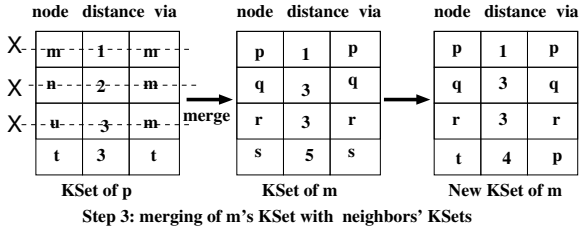

Fig. 18. KNN Edge Delete Example: step 3

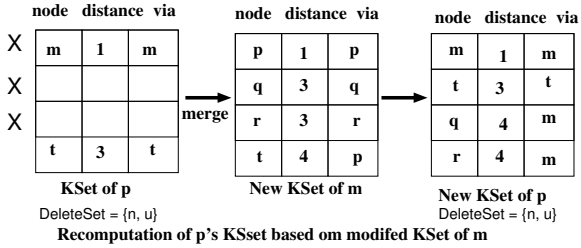

Fig. 19. KNN Edge Delete Example: step 4

other hand, if the update operation on an edge increases its length, then the operation performed is similar to the delete edge operation discussed earlier to recompute the k-nearest neighbors.

D. Incremental Computation for Graph Coloring Problem

We now consider how to design incremental computation tasks for the graph coloring problem. We develop a solution for incremental computations when graph updates are applied after an initial execution of the basic algorithm for this problem, shown in Figure 5. After executing the graph coloring program on a given input graph, we obtain, for each node in the graph, its unique smallest color such that no two neighbor nodes have the same color. We consider the following types of updates to a graph after the initial execution of the graph coloring program: adding edges, deleting edges, adding nodes and deleting nodes in the graph. Updating the edges or nodes have no significance for the graph coloring problem. Here, we consider a modified version of graph coloring problem where only the constraint of "distinct color" for neighboring nodes in the graph is considered. The initial computation preserves the constraint of smallest unused color for each node, but the incremental computations disregard this constraint for delete edge and delete node operations in order to simplify the incremental computation task and provide insights into some of the scenarios where incremental computations may not provide high gain.

1) Addition of Edges : In the update transaction for adding a new edge between a pair of nodes, we update the neighbor list of the two nodes and check if both the nodes have the same color. If the two nodes have different color, then we do not need to perform any further computation. But if the two nodes have the same color, we need to assign a new color to one of the two nodes such that no other neighbor has the same color as this node. We consider only one of the two nodes, called the "target node", and obtain the color of all of its neighbors. We arrange the colors fetched in increasing order and from the list of all available colors, we select the smallest unused color for the target node.

2) Deletion of Edges : In the update transaction of deleting an existing edge between a pair of nodes, we only remove the edge from the neighbor lists of the two nodes. We do not perform any other computation since the two nodes already have distinct colors, and we have relaxed the requirement of using the smallest unused color from the set of neighbors. This scenario may be favorable in applications where only the "distinct color" property is of significance. Also, note that this is one such situation where we do not require any cascaded sequence of recomputation tasks.

3) Addition of Nodes : In the *Add Node* update transaction, we update the *neighbor list* of all the neighbor nodes for the newly added node and compute for this node the smallest color not used by any of the neighbor nodes.

4) Deletion of Nodes: For the *Delete Node* update transaction, we update the *neighbor list* of all the neighbor nodes to indicate that the node to be deleted is no longer a neighbor. We also clear its color and then remove this node from the graph. Similar to the delete edge scenario, here too we only consider the constraint of distinct color for neighbor nodes. Since all the nodes already had distinct color, deleting a neighbor node will not affect their distinct color property. So, no further computation is required.

E. Continuous Queries for Maximal Cliques

We now consider the problem of designing incremental computation tasks for supporting continuous queries for the maximal clique problem in dynamic graphs. We consider the following types of updates to the graph after the initial maximal clique computation: adding edges and deleting edges. Updating of edge lengths has no impact on the maximal clique computation. We develop a solution for continuous queries

when graph updates are applied after an initial execution of the basic algorithm. After performing the incremental changes in the graph, we once again compute the maximal clique for the nodes using computeCliques() procedure discussed earlier.

Addition of an edge can result in creation of new triples in the graph, and similarly the deletion of an edge can cause some previously existing triples to be removed. We maintain a consistent state of the *myTriples* set for each of the nodes affected by an update transaction. The maximal clique of a node can change only if the *myTriples* set of that node is modified. This property forms the basis for performing incremental computations for a continuous query to detect any changes in the maximal size cliques of a node. An application may want to continuously *watch* a set of nodes for changes in their maximal clique sets. For each such node, it would set a flag in the node to indicate if continuous query computations should be initiated for it when its *myTriple* set gets modified. The incremental computation for a continuous query for such a node is triggered by creating a new task to recompute its new maximal size cliques.

1) Addition of Edges : The procedure to perform the *Add Edge* transaction is shown in Figure 20. The update transaction for adding a new edge between a pair of nodes, say *u* and *v*, is executed as a transactional task which performs the following operations. For node *u*, we add an edge to *v* in *u's* neighbors set. Similarly, for node *v*, we add an edge to *u* in *v's* neighbors set. After the two edges are added, we find the common neighbors of node *u* and *v*. If the two nodes share a common neighbor, say *q*, then they will form a new triple in the graph, and this new triple will be added to the *myTriple* set of nodes *u*, *v* and *q*. We add *u*, *v* and all their common neighbors to the write-set as part of this transactional task. We create a new task for each such node whose *myTriple* set gets modified. The phase of these tasks is set to "ComputeCliques".

2) Deletion of Edges : In the update transaction for deleting an existing edge between a pair of nodes, say *u* and *v*, we execute a transactional task that performs the following operation. For node *u*, we delete the edge to *v* from *u's* neighbors set. Similarly, for node *v*, we delete the edge to *u* from *v's* neighbors set. After the two edges are deleted, we find the common neighbors of node *u* and *v*. If the two nodes shared a common neighbor, say *q*, then they earlier had formed a triple in the graph, and this old triple should be deleted from the *myTriples* sets of nodes *u*, *v* and *q*. We add *u*, *v* and all their common neighbors to the write-set as a part of this transactional task. Similar to the case of addition of edges, we perform the maximal clique computation only for those nodes for which *myTriples* set gets modified as a result of deletion of some edges from the graph. A new task is created for each such node to recompute its maximal size cliques. The phase of these tasks is set to "ComputeCliques". The *deleteEdge* method is very similar to the *addEdge* method shown in Figure 20, except that instead of adding triples to the *myTriple* sets of the affected nodes, it removes triples for those sets.

```
public addEdge(Node U, Node V) {
  Set uNbrIds = getNeighborIds(U);
  Set vNbrIds = getNeighborIds(V);
  String u = U.nodeId;
  String v = V.nodeId;
  U.neighbors.put( new Edge(v) );
  V.neighbors.put( new Edge(u) );
  writeSet.add(U);
  writeSet.add(V);
  Set commonNbrs = uNbrIds.retainAll(vNbrIds);
  Set taskTargets = new Set();
  foreach (q in commonNbrs) {
     Node Q = storage.getNode(q);
     U.myTriples.add("u,v,q");
     V.myTriples.add("u,v,q");
     Q.myTriples.add("u,v,q");
     if (U.watchFlag) {taskTargets.add(u);}
     if (V.watchFlag) {taskTargets.add(v);}
     if (Q.watchFlag) {taskTargets.add(q);}
     writeSet.add(Q);
  }
  foreach (p in taskTargets) {
     Task t = new CliqueTask(p);
     t.phase = "ComputeClique";
     newTasks.add(t);
  }
}
```

Fig. 20. Maximal Clique Computation for Add Edge Task

F. Incremental Computation for the Connected Components Problem

We now consider the problem of designing incremental computation tasks for the connected components problem. We develop a solution for incremental computations when graph updates are applied after an initial execution of the basic algorithm for this problem, presented in Section II. After executing the connected components program on a given input graph data using the transactional task computation shown in Figure 7, we obtain for each node in the graph the label of the connected component to which it belongs. We consider the following types of updates to a graph after the initial execution: adding edges and deleting edges. Here, updating edge properties or updating other attributes of a node does not affect the connected component to which it belongs. Moreover, the operations of adding (deleting) nodes from the graph can be simulated by multiple add (delete) edge operations. Therefore, these operations are omitted from the below discussion.

After the initial execution of the connected component program, updates to the graph are applied in phases. In an update phase any number of add or delete edge operations can be performed in parallel. After the update phase, a *merge-label* phase is executed, as in the initial computation.

For supporting incremental computations we define two new task classes. The task class *RecomputeTask* is defined for performing incremental computations at a node. The task type *UpdateGraph* is used for performing the update transactions for the above operations, by calling the *update* method shown in Figure 22. The *doTask* method of the Worker thread executes the appropriate functions for these tasks as shown below in Figure 21.

```
public class CCWorker extends Worker {
  public doTask(Task task) {
    String taskClass = getTaskClass(task);
    switch (taskClass) {
      case "UpdateGraph":
        update(task); break;
      case "RecomputeTask":
        recompute(task); break;
      case "CCTask":
        if(task.subphase == "label-nodes")
           compute( task );
         else if(task.subphase == "merge-labels")
           relabel( task );
        break;
    }
  }
}
```

Fig. 21. doTask method of CCWorker for incremental computations

```
public void update(Task task){
  Node tgtNode = storage.getNode(t.nodeId);
  UpdateGraph t = (UpdateGraph)task;
  switch(t.opname){
    case CCTask.addEdge:
      performAddEdge(tgtNode,t);
      break;
    case CCTask.deleteEdge:
      performDeleteEdge(tgtNode,t);
      break;
  }
}
```

Fig. 22. Update method of CCWorker for incremental computations

1) Addition of Edges : In the transaction for adding a new edge between a pair of graph nodes, if the labels of these nodes differ then a label-clash is registered in the global storage. If the labels of the two nodes considered above are the same, then no further relabeling is required as the connected components in the graph do not change. In the next sub-phase, the master will merge the label-clash equivalence sets and relabel the required nodes. The merging of label-clash equivalence sets and the execution of the relabel tasks are exactly the same as in the initial computation. In this way, finally all the nodes belonging to the same component will have the same label. This preserves the invariant property that all nodes of the same connected component have the same label, and the system returns back to a quiescent state. In an update phase, multiple add edge transactions can run in parallel. The add edge operation is shown in Figure 24 and the logic for the add edge operation is illustrated in Figure 23. The add edge operation shown in Figure 24 will register a label-clash (10,70) in the global storage.

2) Deletion of Edges : The transaction for deleting an existing edge between a pair of nodes removes the edge from the neighbor-list of the two nodes and may break the connected components into two parts if this is the only edge that connects the two components together. Consider the example in Figure 26 where the edge between nodes 9 and 10 is deleted first. It initiates recomputation tasks at these nodes. These tasks, represented by T(9) and T(10), are initiated at nodes 9 and 10, respectively. Here, T(10) would relabel all nodes from 10 through 13 with label 10. Each task specifies the new

```
public void performAddEdge(Node tgtNode,
UpdateGraph t){
  Node nbrNode = storage.getNode(t.nbrId);
  addEdge(tgtNode,nbrNode);
  addEdge(nbrNode,tgtNode);
  recordClash(tgtNode.label, nbrNode.label);
  writeSet.add(tgtNode);
  writeSet.add(nbrNodes);
}
```

Fig. 23. CC Transaction for edge addition

label and the expected original label at the target node. If it reaches a node whose label differs from the expected label, a label-clash is registered.

For each of the two nodes, if the node id of the node is different from its existing label, then its node id is taken as its new label, and *RecomputeTasks* are created for all its neighbors to relabel them. Otherwise, no relabeling is performed on that node. If the delete edge operation does not disconnect the component, then the *RecomputeTasks* starting from the two nodes of the deleted edge will lead to a label-clash. The label-clash is registered in the global storage, and the master would then merge the equivalence sets and relabel the nodes in the *merge-label* phase, as in the initial computation. The delete edge operation is shown in Figure 26.

The operation of deletion of a single edge at a time is simple. Several challenges arise when multiple delete edge operations run concurrently. When two delete edge operations delete the edges of the same component, both initiate *RecomputeTask* on the two nodes of the deleted edge. It is possible that a *RecomputeTask* from a later edge-delete operation is executed at a node earlier than the *RecomputeTask* initiated by a preceding delete-edge operation. When the *RecomputeTask* of the later edge-delete operation executes before a preceding *RecomputeTask* (called *straggler*) that has still not executed, that straggler would declare an incorrect label clash. Consider the example of Figure 26 further with the deletion of edge between nodes 10 and 11. This is illustrated in Figure 27. This results in creation of recomputation task T(11) at node 11, and propagating to node 12. Suppose that task T(11) is executed at node 12 before the arrival of task T(10). It relabels node 12 to label 11. Task T(10) would find the label of node 12 as 11 instead of expected label 6, and it would declare an incorrect label-clash (10, 11), and propagation of the relabeling task would terminate. This false label-clash would lead to incorrect

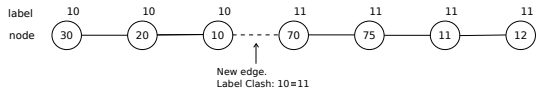

Fig. 24. Addition of Edge in Connected Components problem

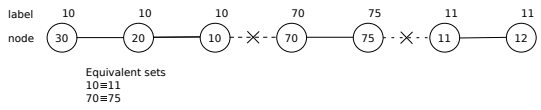

Fig. 25. Concurrent Addition and Deletion of Edges

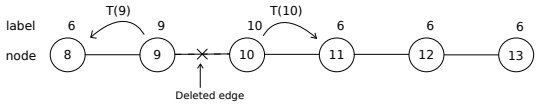

Fig. 26. Deletion of Edge in Connected Components problem

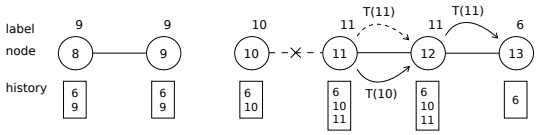

Fig. 27. Concurrent Deletion of Multiple Edges

declaration of the two disconnected components as connected.

```
public void performDeleteEdge(Node tgtNode,
UpdateGraph t){
  Node nbrNode = storage.getNode(t.nbrid);
  removeEdge(tgtNode,nbrNode);
  removeEdge(nbrNode,tgtNode);
  writeSet.add(tgtNode);
  writeSet.add(nbrNodes);

  if(nbrNode.hasReportedLabelClash() ||
  tgtNode.hasReportedLabelClash()){
    clearGlobalLabelClash(tgtNode,nbrNode);
    nbrNode.deleteLabelClashWithNbr(tgtNode);
    nbrNode.manageInvalidSet(tgtNode);
    tgtNode.deleteLabelClashWithNbr(nbrNode);
    tgtNode.manageInvalidSet(nbrNode);
  }
  if(tgtNode.nodeId != tgtNode.label){
    tgtNode.updateHistory();
    Set<String> nbrIds = tgtNode.getNbrIds();
    foreach (String nbr in nbrIds) {
      Task tnew = new RecomputeTask(nbr);
      tnew.label = tgtNode.label;
      tnew.history = tgtNode.history;
      newTasks.add( tnew );
    }
  }
  if(nbrNode.nodeId != nbrNode.label){
    nbrNode.updateHistory(t);
    Set<String> nbrIds = nbrNode.getNbrIds();
    foreach (String nbr in nbrIds) {
      Task tnew = new RecomputeTask(nbr,
      tgtNode, nbrNode);
      tnew.label = nbrNode.label;
      tnew.history = nbrNode.history;
      newTasks.add( tnew );
    }
  }
}
```

Fig. 28. CC Transaction for edge deletion

There are still other problems that arise when multiple add edge and delete edge operations are executed concurrently. Consider the example in Figure 24 where the edge between nodes 10 and 70 is added. This results in a label-clash (10,70). Now consider that in the same update phase, the same edge between 10 and 70 is deleted along with another edge between node 75 and 11, concurrently (refer to Figure 25). *RecomputeTasks* will start at nodes 70 and 75. Node 70 is relabeled to label 70 and node 75 is relabeled to 75. The *RecomputeTasks* will register a label-clash (70,75). Now in the *merge-label* phase, the master will merge the labels 10 and 11 and it would result in the nodes 30, 20 and 10 being incorrectly declared as a part of the same component as nodes 11 and 12.

```
public void recompute(Task tsk){
  Node u = storage.getNode( task.nodeId );
  if(sender is in InvalidNeighbors set OR
     u's label is same as task's label){
    //do nothing
    //task is from a node which is no longer
    //a neighbor or there is no label change
  }
  else if(task's history doesn't have u's label){
      //label not present in task history
      //it is a label-clash
      1. record the clash (tsk.label, u.label)
      2. add entry (tsk.label, u.label) in
         u's clash-set for the sender node.
      3. Add u to write set.
  }
  else if(u's history doesn't have task's label){
    // not a straggler task
    1. update u's label to the task's label
    2. merge task's history into u's history.
    3. add u to write set.
    4. for each neighbor create a new recompute
       task and pass u's label and u's history with
       the new task.
  }
  else{
    //it's a straggler. Do nothing.
  }
}
```

Fig. 29. CC Transaction for recompute

To solve all such problems, some extra information needs to be stored at the nodes of the graph to support concurrent delete and add edge edge operations. We maintain three data items with each node: (1) a set called *InvalidNeighbors* contains the nodes that are no longer neighbors because of edge deletions, (2) a set called *clash-set* records label-clashes detected at this node, and this set is maintained per neighbor basis, and (3) a set named *history* consisting of all labels assigned to the node in the current update phase. If a node is in the *InvalidNeighbor* set, then there cannot be a *clash-set* present for it. When a neighbor is added to the *InvalidNeighbor* set on an edge deletion, the entries for it in the *clash-set* are cleared and they are also removed from the global label-clash registry. When any task originating from a node in the *InvalidNeighbor* set is received, it is simply discarded. Additionally, each recomputation task created by a node for its neighbors contains its *history*. When such a task updates a node's label, the node's history is updated to include the history contained in the task. A node's history can be used to discard a straggler task. If the current label is not present in the history contained in the task, we declare a label-clash (as opposed to declaring a label clash when the node's label was equal to the expected label for deleting a single edge). Otherwise, if the label contained in the task is present in the node's history, then it is a straggler task and it is discarded. Otherwise the label and the history is updated by task execution and new recomputation tasks are propagated to the neighbors. This is the crux of our approach which allows execution of multiple edge-add and edge-delete operations to run in parallel.

In the example shown in Figure 27, when node 11 sends task T(11) to node 12, it will include its history containing {*6, 10, 11*} in the task. Execution of this task at 12, will update the node's history to the task's history. When the straggler task T(10) arrives at 12, it finds that the its label 10 is present in the node's history, and therefore that task would be discarded. This avoids incorrect declaration (10,11) as label-clash by the straggler task.

In reference to the problem discussed in example shown in Figure 25, when the edge between the nodes 10 and 70 is deleted, then at node 10, 70 is added to its *InvalidNeighbor* set and 70's clash record (10,11) is removed from the *clash-set* of node 10 as well as from the global label-clash registry. Similar computations would take place at node 70. Hence, now the master will not merge the labels (10,11) in the *merge-label* phase. This prevents incorrect label-clash declaration. Figure 28 shows the logic for deleting the edges and initiating the recomputation tasks. Figure 29 shows the logic for executing the recomputation tasks (*RecomputeTask*).

The overhead of storing extra information at the graph nodes is small as this information needs to be retained only up to the next sub-phase when all the tasks created due to the current seed node(s) have completed, and the add and delete edge operations, if any, have also completed. During the *merge-label* phase, this information is cleared from all the nodes. For better performance, the host machines locally store all this information for the nodes present in their storage system and clear this information when the current sub-phase completes.

V. Performance Evaluations

We conducted experiments using the five graph problems described above to evaluate the performance benefits of supporting continuous queries through incremental computations using the transactional programming model of Beehive. Our goal was to determine the relative latencies in performing continuous queries when the input graph data was updated after the execution of the initial analytics program for these problems. A set of update transactions were applied to the graph data after the initial program execution. We measured the time for the system to reach the quiescent state from the point when these update transactions were initiated. The evaluation measure used was the incremental computation time expressed as a percentage of the initial execution time. We measured this for different types of update operations independently, and for different load of the update transactions.

We conducted these experiments on a cluster computer where each node had 8 CPU cores of 2.8 GHz and 22 GB main memory, connected with 40-gigabit network. We conducted evaluations using graphs with number of nodes 100K, 200K, 400K, 800K, 1million, and 2 million. In our experiments we used graphs which had random connectivity, with average node degree of 67. In these graphs the edge lengths were uniformly distributed between 1 and 100.

For the SSSP problem, we used five types of update transactions: *Add Edge*, *Update Edge*, *Delete Edge*, *Add Node*, and *Delete Node*, where the nodes and edges were randomly selected. We measured the time for the system to reach the quiescent state from the point when these update transactions were initiated. The time required for incremental computations were separately measured for each of these types. For each of these update types, we injected transactions to change close to 1% of the graph structure. For example, in the experiment for the *Add Edge*, *Delete Edge*, and *Update Edge* updates, the number of edges added, deleted, or updated was 1% of the number of edges in the input graph. Similarly the number of nodes added or deleted, for the *Add Node* and *Delete Node* updates, were 1% of the nodes in the input graph.

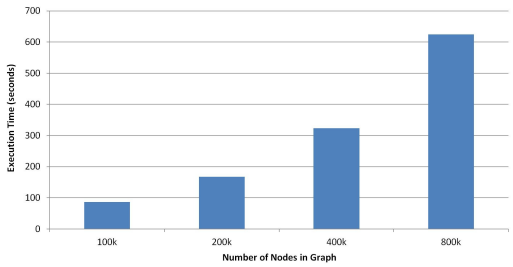

Fig. 36. Initial execution times of the maximal clique program for different graphs

Figure 30 shows the execution time for the SSSP problem, for the four graphs of different sizes noted above. Figure 31 shows the incremental execution time for different types of update operations, expressed as a percentage of the initial execution time for these four graphs. From these experiments we observe that the incremental execution cost for the operations to add and update edges is less than 1% of the initial execution times, and for the delete edge operations it is around 2%. Similarly we observe that the incremental execution cost for the add node operations is in the range of 5-8%, and for the delete node operations it is around 4-5%. The node operations are slightly more expensive as we have several add/delete edge operations associated with them. For this problem we find that there are clear benefits of using the incremental computation approach.

For the KNN problem, we used three types of update transactions: *Add Edge*, *Update Edge*, and *Delete Edge*. For each of these update types, we conducted experiments with the number of update transactions set close to 0.1%, 0.5%, 1%, and 5% of the number of nodes in the graph. For example, in the experiments for 1% update transactions, the number of edges added, deleted, or updated was 1% of the number of nodes in the input graph. The edges in these update transactions were randomly selected.

Figure 32 shows the initial execution time for the KNN problem, for graphs with number of nodes 100K, 200K, and 400K. Figure 33 shows for each of these three graphs the latencies in performing continuous queries using incremental computations for the *Add Edge* operations with different number of update transactions. In this graph, latency is expressed as a percentage of the initial execution time. Similarly Figure 34 and Figure 35 show latencies for the

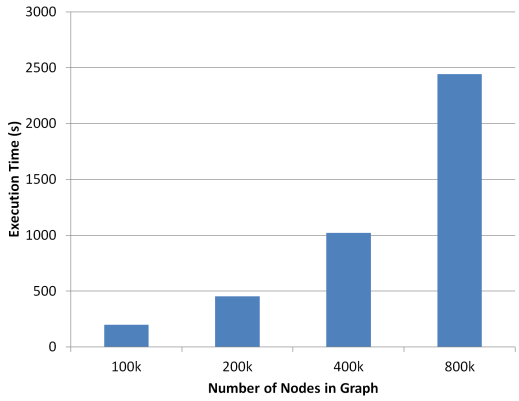

Fig. 30. Initial execution times of the SSSP program for different graphs

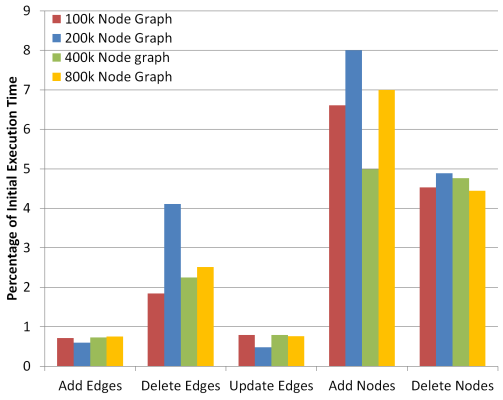

Fig. 31. Incremental computation times as percentage of initial execution times for different update transactions for SSSP

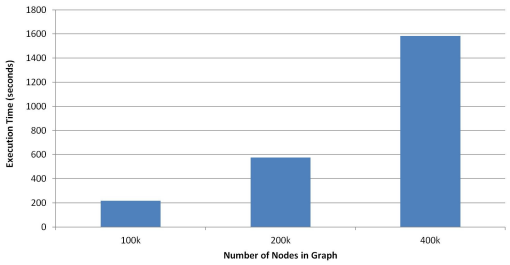

Fig. 32. Initial execution times of the KNN program for different graphs

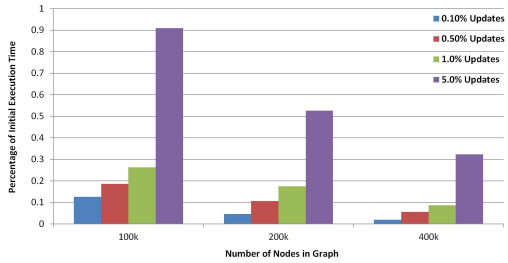

Fig. 33. Query latencies as percentage of initial execution times for Add Edge transactions for KNN

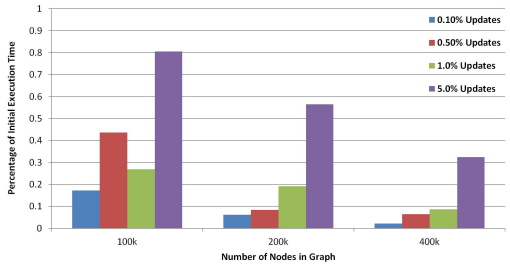

Fig. 34. Query latencies as percentage of initial execution times for Delete Edge transactions for KNN

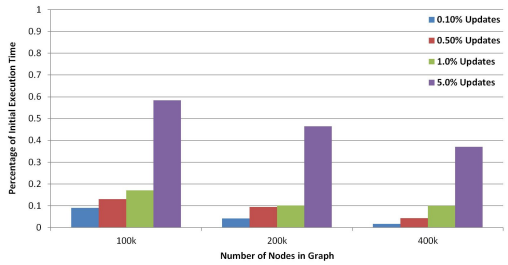

Fig. 35. Query latencies as percentage of initial execution times for Update Edge transactions for KNN

Delete Edge and *Update Edge* operations, respectively. From these experiments we observe that the incremental execution cost for the operations to add, delete and update edges is less than 1% of the initial execution times, even after making 5% updates in the graph. Thus, processing of continuous query is less expensive than executing the entire analytics program again. We see that as the percentage of updates in the graph increases, the continuous query cost as a percentage of initial execution time also increases. As the graph gets larger, the continuous query cost as a percentage of initial execution time decreases. Hence, we see that there are clear benefits of using the incremental computation approach in supporting continuous queries in large graphs.

For the maximal clique problem, we conducted experiments using graphs of 100K, 200K, 400K, and 800K nodes. We measured the time for incremental recomputations for *Add Edge* and *Delete Edge* update transactions. In our experiments, the *watch-list* contained all nodes present in the graph. The experiments for these two update types were conducted separately. For each of these type, we measured the latencies for different volumes of updates, which were set to 1%, 5%, and 10% of the size of the graph in terms of the number of nodes. For this problem we observe that the incremental computation cost tends to be high. For example, for 1% update volume the incremental execution cost was around 5% of the initial execution cost. It should be noted here that for this problem the initial execution cost itself is relatively smaller as compared to the other problems.

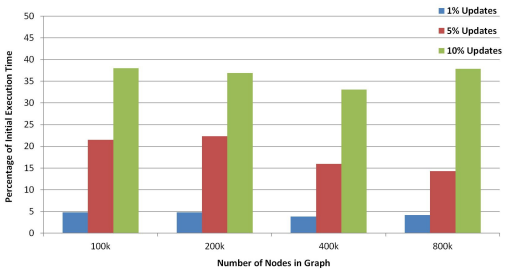

Fig. 37. Maximal clique query latencies as percentage of initial execution times for Add Edge transactions

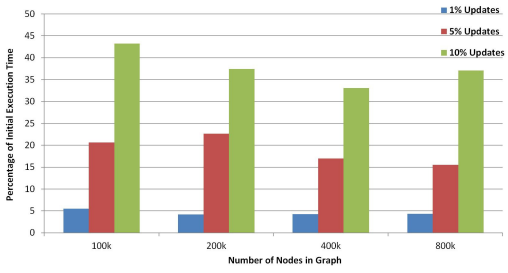

Fig. 38. Maximal clique query latencies as percentage of initial execution times for Delete Edge transactions

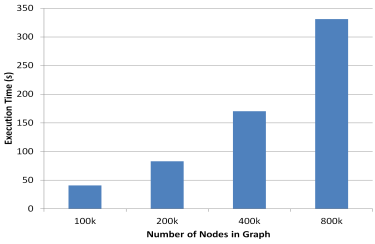

Fig. 39. Initial execution times of Graph Coloring program for different graphs

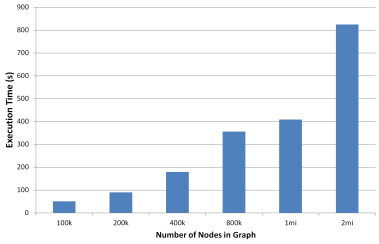

Fig. 40. Initial execution times of the Connected Components program for different graphs

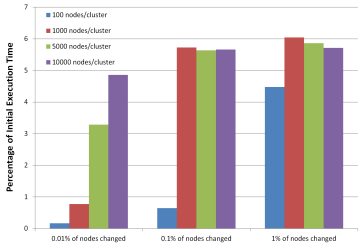

Fig. 41. Change in incremental execution time with increase in percentage of nodes changed for Addition of Edges in Connected Components problem for 1 million node graph

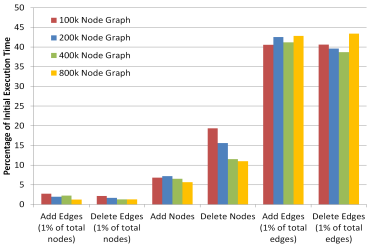

Fig. 42. Incremental computation times as percentage of initial execution times for different update transactions for Graph Coloring

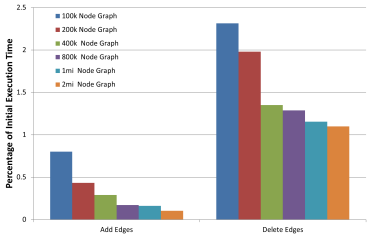

Fig. 43. Incremental computation times as percentage of initial execution times for different update transactions for Connected Components

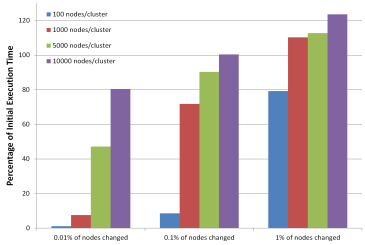

Fig. 44. Change in incremental execution time with increase in percentage of nodes changed for Deletion of Edges in Connected Components for 1 million node graph

VI. Performance Evaluations

Using the Beehive framework we conducted experiments to evaluate the performance benefits of the incremental computation techniques using the two problems presented in Section IV. Our goal was to determine the utility of performing incremental computations when the input graph data is updated after an initial execution of the parallel program for a given problem. We measured the execution time required by incremental computations when a set of updates were performed. The evaluation measure used was the incremental computation time expressed as a percentage of the initial execution time. We measured this for different types of update operations, to determine which kinds of update operations can benefit from incremental computations. Our evaluations show that the benefit of using incremental computations depends on the nature of the graph problem, cost of the initial execution of the parallel program for this problem, type of the update operations, and the number of update operations.

We conducted these experiments on a cluster of 16 computers where each node had 8 CPU cores of 2.8 GHz and 22 GB main memory, connected with 40-gigabit network. We conducted evaluations using several graphs with the number of nodes 100K, 200K, 400K, 800K, 1 million, 2 million and 4 million. In our experiments we used graphs which had random connectivity, with average node degree of 67.

For the graph coloring problem, we used four types of update transactions: *Add Edge*, *Delete Edge*, *Add Node*, and *Delete Node*, where the nodes and edges were randomly selected. We measured the time for the system to reach the quiescent state from the point when these update transactions were initiated. The time required for incremental computations were separately measured for each of these types. Figure 39 shows the execution time for the graph coloring problem for the graphs with 100K, 200K, 400K and 800K nodes.

For this problem, we consider only the add/delete edge and add/delete node operations because the update edge operations do not affect the result state of the initial computation. In this problem, the number of tasks executed in the initial computation is exactly equal to the number of nodes in the graph, as only one task is created for each node to determine its color, as shown in Figure 5. In contrast, in the connected components problem, tasks are dynamically created and the number of tasks that get executed and committed tends to be about an order of magnitude larger than the number of nodes in the graph.

For each of these update types, we injected transactions to update the graph structure for two scenarios: (a) number of update operations equal to 1% of the total number of nodes in the graph, and (b) number of update operations equal to 1% of the total number of edges in the graph. It should be noted that (b) is an extreme case of incremental changes as each node has an average degree of 67 due to which the number of nodes updated is extremely high in (b) in comparison to (a). We evaluated the relative performance of these two scenarios.

Figure 42 shows, for the graph coloring problem, the incremental execution time for different types of update operations, expressed as a percentage of the initial execution time. In case of the 100K node graph, scenario (a) involved addition (deletion) of 1000 edges, whereas for scenario (b) this number was 33621. Since the initial computation had only 100K task computations, addition (deletion) of 1% of edges led to almost 33% of the initial computation load. This is reflected in the data presented in Figure 42 where we observe that for scenario (a), the incremental computation costs are around 1-2% of the initial computation time, whereas for the scenario (b) this cost is around 40-45%. The incremental computation cost for the add node operations is about 5-7% of the initial computation, and for the delete node operations it is 11-18%. The values obtained above for the add/delete edge operations in scenario (a) and (b) shows that the execution time for incremental computations is very small for small changes in the graph, but the execution time becomes larger for large changes.

For the connected components problem, we used two types of update transactions: *Add Edge* and *Delete Edge*. For each of these update types, we added transactions to update 1% of the total number of connected components in the graph. Figure 40 shows the initial execution time for the connected components problem for graphs of sizes ranging from 100K to 2 million nodes. In these graphs, each connected component contains exactly 100 nodes. Therefore, the number of connected components present in the graph increases with increase in the number of nodes. Figure 43 shows the execution time for incremental computations for different types of update operations, expressed as a percentage of the initial execution time. From these experiments we observe that the incremental execution cost for the operations to add edges is less than 1% of the initial execution times, and for the delete edge operations it is between 1% and 3%.

In another experiment for the connected components problem, we performed a fixed number of changes (0.01% of the number of nodes) in a 1 million node graph and observed the effect of increasing the cluster size of the connected components in this graph from 100 nodes/cluster to 1000, 5000 and 10000 nodes/cluster. Then, we increased the fixed number of changes from 0.01% of the number of nodes to 0.1% and 1% of the number of nodes in the graph and performed the same experiment. When the number of add/delete edge operations are constant for a given size graph, the graph with large number of smaller clusters will involve changes in less number of nodes compared to the graph with small number of large clusters. For example, if we perform add/delete edge operations on 100 random nodes in the graph, then in the graph with 100 nodes/cluster, the number of nodes affected will be 10000 (1% of all nodes); whereas in the graph with 5000 nodes/cluster, the number of nodes affected will be 500000 (50% of all nodes), much higher than the previous case. Therefore, the same number of operations become much more expensive when the graph with the same number of nodes has small number of very large clusters. Figure 41 and Figure 44 show the execution time for incremental computations expressed as a percentage of the initial execution time for add-edge and delete-edge operations, respectively. As the number of nodes affected by the add/delete edge operation increases and becomes comparable to (or almost equal to) the number of nodes in the graph, the execution time for incremental computations become as high as the initial computations, and re-executing the graph analytics problem would be a better alternative. But, if these changes affect only a very small number of connected components, then incremental computations can greatly save execution time.

Based on our experiments with the above two graph problems,we make the following observations. The benefits of performing incremental computations depend on several factors, including: (1) the cost and complexity of the initial computation; (2) cost and complexity of performing the incremental computations for a specific type of update operation, which may depend on the structure of the input graph; and, (3) ratio of the above two measures, (1) and (2), in conjunction with the number of different types of update operations. In cases where the cost of performing incremental computations becomes comparable to the cost of the initial computations, it could be preferable to re-execute the analytics program. On the other hand, one can benefit from the use of incremental computation approach in most of the other situations.

Based on the experiments described above we make the following observations. The benefits of supporting continuous queries using incremental computations depend on several factors. First is the cost and complexity of the baseline analytics program for the property of interest. Second is the cost and complexity of performing the incremental computations for a query for a specific type of update operation. Third is the volume of update operations. In cases where the cost of performing incremental computation becomes comparable to the cost of the initial computations, it could be preferable to re-execute the analytics program to get the query results. In cases when the volume of update operations is low, continuous

queries can be effectively supported through the incremental computation approach.

VII. Conclusion

We have shown here how the transactional model for parallel computing can be used for supporting continuous queries in dynamic graph structures by utilizing incremental computations. This model allows such incremental computations to execute concurrently with graph updates, thus making it suitable for performing continuous queries on dynamic and evolving graph structures. The transactional approach for incremental computing presented here is simple to use. Using five graph problems we have illustrated here the approach for developing transactional tasks to perform incremental computations for continuous queries under different types of updates in dynamic graph structures. Using the transactional computing model supported by the Beehive framework, we have demonstrated the feasibility and benefits of utilizing this approach for eliminating the need of re-executing an analytics program in dynamic graph structures when the number of updates and the associated incremental computation costs are low. The benefits of incremental computations depend on several factors which include how the cost of executing the incremental computations compares with the execution of the basic analytics program based on the nature and the number of update operations. For small number of update in a graph structure, the incremental computation approach provides high benefits.

Acknowledgements: This work was supported by NSF Award 1319333 and computing resources were provided by the Minnesota Supercomputing Institute.

References

[1] Umut A. Acar and Yan Chen. Streaming big data with self-adjusting computation. In *Proceedings of the 2013 Workshop on Data Driven Functional Programming*, pages 15–18, New York, NY, USA, 2013. ACM.

[2] Apache. Giraph. Available at http://giraph.apache.org/.

[3] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[4] Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A. Acar, and Rafael Pasquin. Incoop: Mapreduce for incremental computations. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, New York, NY, USA, 2011. ACM.

[5] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proc. of WWW'98*, 1998.

[6] Zhuhua Cai, Dionysios Logothetis, and Georgos Siganos. Facilitating real-time graph mining. In *Proceedings of the Fourth International Workshop on Cloud Data Management*, New York, NY, USA, 2012. ACM.

[7] Nicholas Carriero and David Gelernter. How to write parallel programs: a guide to the perplexed. *ACM Comput. Surv.*, 21:323–357, September 1989.

[8] Prasanna Desikan, Nishith Pathak, Jaideep Srivastava, and Vipin Kumar. Incremental page rank computation on evolving graphs. In *Special Interest Tracks and Posters of the 14th International Conference on World Wide Web*, WWW '05, New York, NY, USA, 2005. ACM.

[9] A. Fard, A. Abdolrashidi, L. Ramaswamy, and J. A. Miller. Towards efficient query processing on massive time-evolving graphs. In *Proceedings of the 2012 8th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom 2012)*, COLLABORATECOM '12, Washington, DC, USA, 2012. IEEE Computer Society.

[10] Anand Padmanabha Iyer, Li Erran Li, Tathagata Das, and Ion Stoica. Time-evolving graph processing at scale. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, GRADES '16, New York, NY, USA, 2016. ACM.

[11] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6:213–226, June 1981.

[12] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new framework for parallel machine learning. In *Proceedings of Conference on Uncertainty in Artifical Intelligence (UAI)*, 2010.

[13] G. Malewicz, M. H. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proc. of ACM SIGMOD '10*, pages 135–146, 2010.

[14] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 456–471, New York, NY, USA, 2013. ACM.

[15] Anand Tripathi, Vinit Padhye, and Tara Sasank Sunkara. Beehive: A Framework for Graph Data Analytics on Cloud Computing Platform . In *Seventh International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2), held in conjunction with ICPP'2014*, 2014.

[16] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[17] Charith Wickramaarachchi, Charalampos Chelmis, and Viktor K. Prasanna. Empowering fast incremental computation over large scale dynamic graphs. In *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, IPDPSW '15, Washington, DC, USA, 2015. IEEE Computer Society.