A Transactional Model for Parallel Programming of Graph Applications on Computing Clusters

Anand Tripathi, Vinit Padhye, Tara Sasank Sunkara, Jeremy Tucker
BhagavathiDhass Thirunavukarasu, Varun Pandey, Rahul R. Sharma
Department of Computer Science & Engineering
University of Minnesota, Minneapolis MN 55455

***Abstract*—We present here the results of our investigation of a transactional model of parallel programming on cluster computing systems. This model is specifically targeted for graph applications with the goal of harnessing unstructured parallelism inherently present in many such problems. In this model, tasks for vertex-centric computations are executed optimistically in parallel as serializable transactions. A key-value based globally shared object store is implemented in the main memory of the cluster nodes for storing the graph data. Task computations read and modify data in the distributed global store, without any explicitly programmed message-passing in the application code. Based on this model we developed a framework for parallel programming of graph applications on computing clusters. We present here the programming abstractions provided by this framework and its architecture. Using several graph problems we illustrate the simplicity of the abstractions provided by this model. These problems include graph coloring, k-nearest neighbors, and single-source shortest path computation. We also illustrate how incremental computations can be supported by this programming model. Using these problems we evaluate the transactional programming model and the mechanisms provided by this framework.**

I. Introduction

Parallelism in many graph problems tends to be fine-grained with irregular structure [14], [8], which makes it difficult to partition data for parallel computing using commonly used frameworks such as MapReduce [5]. To harness the irregular parallelism inherent in such problems, also called *amorphous parallelism* [14], [8], techniques based on optimistic parallel execution of concurrent tasks have been investigated in the past [8]. Speculative execution techniques for extracting parallelism have also been widely investigated in the context of multi-core and multi-threaded architectures.Such techniques are based on the notion of *software transactional memory* [6], [10] to guarantee the atomicity and isolation properties of concurrently executed tasks.

Our work has been driven by the goal of developing a transaction-based model of parallel programming on cluster computers, utilizing optimistic execution techniques for harnessing amorphous parallelism in graph problems. We implemented this model in a parallel programming framework called Beehive [18]. It is intended for cluster computing of large problems for which the RAM of a single computer may not be sufficient. Its design has been driven by the goals of providing simple programming abstractions and support for fault-tolerance and recovery. We present here the transaction-based programming abstractions and the system architecture of this framework. The primary focus of this paper is on the results of our investigation and our experience in programming with the transactional model of parallel computing on clusters.

The computation model supported by our programming framework is vertex-centric, supporting parallel execution of computation tasks on different vertices of a graph. Each task is executed as a serializable transaction [3], implemented using optimistic concurrency control techniques [9]. A global object storage is implemented in the memory of the cluster nodes for providing location-transparent access to graph data. This relieves the programmer from the burden of programming explicitly with message-passing primitives. The motivations for adopting this approach is to provide a simple model for parallel programming of graph algorithms as compared to the message-passing models.

In comparison to our previous work in [18], the Beehive system described here provides significantly better performance by supporting object caching, fine-grain operations on remote objects, and multi-threaded implementation of the transaction validation service. This system also supports fault-tolerance using checkpointing and recovery mechanisms. In this paper we also present an evaluation of the optimistic execution model using several graph problems, which include graph coloring, single-source shortest path, and k-nearest neighbors. The programs for these problems illustrate the simplicity of the Beehive programming abstractions and also show that one can easily develop a parallel program for a problem through simple adaptation of a sequential algorithm. We also illustrate here an important advantage of the transactional programming model in its ability to support incremental computations when a graph structure is updated after the execution of a parallel program.

The next section presents the related work. Section III presents an overview of the transactional model of parallel computing provided by Beehive. Section IV presents the programming abstractions, Section V presents an overview of the system architecture. Section VI outlines the significant refinements of the initial design presented in [18]. In Section VII we illustrate parallel programming in this framework using a set of graph problems. Using these example programs, in Section VIII we present the results of our performance evaluation experiments. Section IX discusses and summarizes the results of our investigation, and presents the conclusions.

2159-6190/17 $31.00 © 2017 IEEE
DOI 10.1109/CLOUD.2017.62

IEEE computer society

II. Related Work

Our approach is conceptually close to the work presented in [8] and the Galois system [13], which provides an infrastructure to support speculative execution of computations for graph data analytics on shared memory multiprocessor systems. The Ligra [17] system supports vertex-centric parallel computing on shared memory systems. In contrast, our work is aimed for execution on cluster computing platforms. In the recent years several frameworks and systems have been developed specifically for performing graph data analytics on computing clusters. Distributed GraphLab [11] uses locking based concurrency control for task executions. The approach investigated in our work is based on the optimistic concurrency control model [9]. Pregel [12] and its open-source counterpart Giraph [2] are specifically intended for graph data analysis. They are based on the Bulk Synchronous Parallel (BSP) model [20] of computing. This requires programmers to adapt their algorithms to utilize the BSP model. Moreover, the BSP model may not be suitable for all types of graph processing problems. Utilizing the Pregel model also requires suitable partitioning of the graph. The Dryad [7] system for graph data analytics is based on data-flow based parallel computing.

The Piccolo [15] system provides a programming model based on the abstraction of a shared data store. However, it does not support a general model of transactional task execution; it provides atomic operations and concurrent combining operations limited to a single key-based data item in the storage. Trinity [16] is specifically designed for parallel computing of graph problems utilizing RAM of cluster computing nodes for storing graph data using key-value abstractions. Its synchronization primitives are based on spin-locks; it lacks higher level synchronization mechanisms such as transactions. Our focus is on leveraging a global object storage abstraction for transactional parallel computing.

In contrast to the systems noted above, Beehive provides several unique capabilities. It provides a richer model for representing graph data. Nodes and edges are defined as Java objects, which can be extended to contain different kinds of application-specific properties. This also facilitates parallel programming of applications that involve hypergraphs, where multiple edges representing different types of relationships among nodes are needed. Beehive model also supports dynamic graph structures, i.e. in a computation edges and nodes can be added or removed from the graph. The transactional model of computation also facilitates incremental recomputations when the graph structure is modified. Moreover, Beehive also supports concurrent execution of different types of tasks on multiple nodes, unlike many of the above systems where one specific computation is executed on all nodes.

III. Transactional Parallel Computing Model

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. It is possible for a successfully completing task to create new tasks for further computation. Each task is executed as a serializable transaction [3] satisfying the properties of *atomicity* and *isolation*. We refer to these computations as *transactional tasks*. 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. This ensures that the final computation state is one that would result from a sequential execution of the tasks. It should be noted that the serialization order of the transactional tasks is non-deterministic. Therefore, the final state resulting from a parallel program execution in this model may not be a state that would result from a sequential algorithm's execution. For example, different runs of the single-source shortest path problem in this model may give different shortest paths to a node but they all will be of the same length, or different runs of a maxflow problem may use different edges for flow, but all runs would give the same maxflow value.

The transaction model for task execution is based on optimistic concurrency control [9]. This is a lock-free model of execution, and a transaction reads only committed data. A transactional task is committed only if it does not conflict with any other concurrently committed task. The conflicts are defined based on the notion of *read-write* or *write-write* conflicts, as in database systems [3].

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 reads the required data items from the global storage into its local memory buffers. In the compute phase, all updates are written to the private buffer memory. After the compute phase, a validation is performed to check for conflicts. The validation procedure ensures that no concurrently committed transaction with commit timestamp larger than the *start timestamp* of the transaction being validated has modified any items in its *read-set* or *write-set*. Otherwise, we have a conflicting concurrent transaction already committed, and therefore this transaction must be aborted and re-executed. On successful validation, the transaction is assigned a commit timestamp. Commit timestamps are monotonically increasing, without any gaps. On committing, the transaction writes the buffered updates to the global storage and any new tasks created by the transactional task are added to the set of tasks to be executed.

The motivation in adopting the optimistic execution approach is to exploit latent parallelism present in many graph problems by performing parallel execution of tasks on different vertices of the graph, assuming that the probability of conflicts among concurrently executed tasks would be low. We show here through experimental evaluations that this assumption holds for several graph problems.

IV. Programming Abstractions

The programming primitives provided in Beehive are based on three abstractions: First, it provides a distributed global object storage system maintained in the main memory of the computing cluster nodes for storing the input graph data for the program. The global object storage system provides key-value based location-transparent access, eliminating any explicit use of message-passing primitives in the program. Second, it uses the task-pool paradigm [4]. A distributed pool of tasks is maintained in the system. Third, it provides a pool of worker threads on each cluster node. A worker thread picks a task from the task-pool for execution using the transactional computing model.

```
public class Worker extends Thread {
 Set<Node> readSet, writeSet;
 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 = doTask(task);
    status = validator.validate(txnId, readSet,
                                writeSet);
    if (status == commit) {
     Workpool.reportCompletion(txnId, writeSet,
                               newTasks);
     finished = true;
    }
    else { abortTransaction( txnID );
    }
   }
  }
 }
 abstract TaskSet doTask(Task t) {
  // Application defined implementation
 }
}
```

Fig. 1. Base Class for Worker

We now illustrate the primitives provided for transaction-based parallel programming. For this purpose we use the graph coloring problem. The color of a node is indicated by a positive integer, with 0 indicating that the node is not colored. 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 it such that the color assigned to the node is different from those of its neighbors. The framework provides the abstract *Worker* thread 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. Figure 2 shows the *GraphColorWorker* class defined for the graph coloring problem. It implements the *doTask* method. 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. For coloring

```
public class GraphColorWorker extends Worker {
  public TaskSet doTask(Task task) {
    TaskSet newTasks = new TaskSet();
    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
    return null;
  }
}
```

Fig. 2. Example of Worker for Graph Coloring Problem

a node, it 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.

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. The new tasks are distributed to different cluster nodes according to the specified load distribution policies. If validation fails, the worker thread re-executes the task as a new transaction.

The termination of a parallel program execution is detected by the 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 the local task-pool, and (3) there are no messages in the communication network. The framework detects the termination and communicates it to the application program.

V. System-Level Abstractions

Each node of the computing cluster runs a multithreaded Java process called Computation Engine (CE). The computation engines running on the cluster nodes collectively implement the core abstractions and the runtime environment for executing a parallel program. One of the cluster nodes executes the Transaction Validation Service (TVS), which is used to check for conflicts among concurrent transactions. Figure 3 shows a conceptual view of this system architecture. A Computation Engine contains the following four components: an object storage server, a Workpool server, a local validator, and a pool of worker threads. The storage server in a Computation

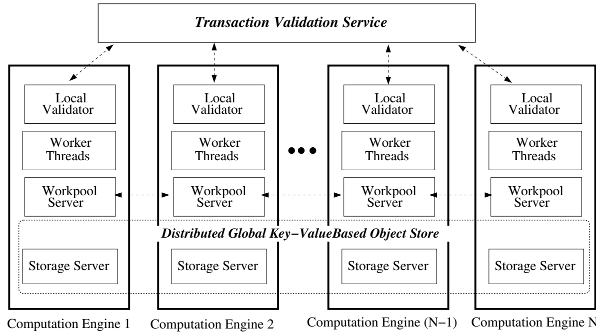

Fig. 3. System Architecture

Engine is RMI based, and all such servers running in the cluster together implement the distributed key-value based object storage service. The Workpool server implements a local task-pool and facilitates distribution of tasks to the other cluster nodes. The Computation Engine also contains a local validator for hierarchical validation of transactions. We present below the programming abstractions together with a brief overview of the framework architecture.

A. Global Object Storage Service

The framework provides the abstraction of a key-value based object storage service, supporting location-transparent access to graph data, which is stored as node objects. 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 accessed using the node-id as the access key. In addition to providing primitives for reading or writing a node object, the storage service also provides mechanisms for fine-grain remote operations on a node object to either read/update its member data or remotely invoke execution of one of its methods on its storage server. A parallel program can modify the graph structure at runtime to add/remove nodes and edges. The storage system supports mechanisms for data caching and dynamic relocation of data items, which can be utilized to improve data locality for tasks. The unit of data relocation is a node object. The object relocation mechanism can be used by an application for clustering of graph data.

The distributed object storage system is implemented by the set of storage servers contained in the Computation Engines. The storage server in a Computation Engine stores objects for a subset of the keys in the global key-space. The location of an object is determined using a hash-based scheme, which identifies its default location, referred to as its *home site*. The objects can be relocated at runtime from their default location to any other Computation Engine. The *home site* of an object maintains its current location information.

At each Computation Engine, an object called *StorageSystem* implements the global storage system abstraction, providing location-transparent data access. When accessing an object, the *StorageSystem* first contacts the currently known storage server responsible for that object. If that server no longer holds the object, it responds with the address of its new location. If the new location is not known, the home site of the object is contacted, which always records its current location. The new location is cached for future references.

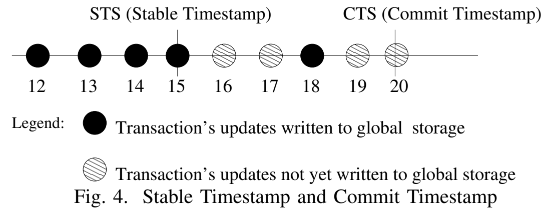

Fig. 4. Stable Timestamp and Commit Timestamp

B. Transaction Validation Service

The validation of transactions is performed in a hierarchical manner to make the validation procedure efficient. In our experiments we found that for some problems 70% of abort decisions could be made locally. A transactional task is first validated by the local validator in its Computation Engine. A transaction invokes the *validate* method of the local validator, specifying its start-timestamp and its read/write sets. The local validator first checks for conflicts with any of the local transactions that committed after the given start-timestamp. If no conflict is found, then the validation request is forwarded to the global Transaction Validation Service (TVS); otherwise the transaction is aborted. TVS performs a similar checking for conflicts. If no conflicts are found, the transaction is assigned a commit timestamp by TVS using the monotonically increasing CTS (Commit Timestamp) counter, and a commit response is sent to the task. The task then writes the modified data to the shared storage and reports completion to the Workpool, which removes the task from the pool and communicates its transaction's completion to TVS.

TVS maintains a table to record, for each data item, the commit-timestamp of the latest transaction that modified it. To validate a transaction, for each item in its read-set and the write-set, the last update timestamp value for that item in the table is compared with the transaction's start-timestamp. If the start-timestamp is less than the update timestamp value of any of the items, then the transaction is aborted. In case of no conflicts, the transaction is assigned a commit-timestamp, which is is recorded in the table for each item in the write-set.

TVS keeps track of the completed transactions and maintains a counter called STS (Stable Timestamp), which reflects the largest commit timestamp value up to which all committed transactions are known to have completed and written their updates to the global storage. The STS value is used for assigning the start-timestamp for new transactions. Figure 4 shows an example where the most recent commit timestamp assigned to a transaction is 20, and all transaction with commit timestamp value up to 15 have written their updates to the global storage. The transaction with commit timestamp value of 18 has completed by writing its updates to the global storage but those with timestamps 16 and 17 have not yet reported completion. STS would be advanced to 18 only after these two transactions report completion to the validation service.

Each new transaction requires reading the STS value from the validation service for its start-timestamp. To reduce the load due to such read requests, the validation service continuously piggybacks the current STS value in its validation response messages. Each Computation Engine maintains a cached value of STS, which is used for assigning start-timestamps to local transactions. This scheme reduces the load on TVS without any impact on the abort rate.

C. Task-Pool Service

Each Computation Engine 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 executing the task on a particular cluster node.

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 of each of the Computation Engines in the cluster. This primitive is generally used for executing some initialization task at each of the Computation Engines. For example, in the graph coloring program, we execute a start-up task at each Computation Engine to create a coloring task for each of the graph nodes stored in its local storage server. 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 Computation Engines in the cluster according to the affinity level in each task and a configurable distribution policy.

VI. Design Refinements

A. Remote Data Access Mechanisms

Our initial design [18] supported reading and writing of an entire node object by a task computation. This imposed performance overheads due to the serialization cost when accessing a remote object. We realized that in most cases only a small subset of a node's data items are needed to be read or written in a computation task. This motivated us to develop support for fine-grain operations in the distributed storage system. For this purpose we introduced programming primitives for a task to *get* or *put* the value for a member in the remote node object. We also provided the *exec* primitive for remote invocation of a node's methods at its storage server.

One of the main requirements in designing the above APIs was to keep in mind that the Beehive framework is agnostic of the application-level data. Application program can extend the *Node* class, but the storage framework is unaware and independent of this application-defined node structure. Hence remote *get/put* operations and *exec* invocations are carried out by runtime inspection of the code to retrieve or alter the node properties. We use the Java Reflection functionality for this purpose. While runtime inspection of code may incur a slight overhead, this is compensated by the reduction in the serialization costs when the entire node is read or written. Additionally, we also implemented two mechanisms to reduce the cost of remote data access. One is to support caching of remote data objects, and the second mechanism aggregates multiple remote data access operations into one RMI call to amortize the remote communication overheads.

B. TVS Design Refinements

Our experiments with large graph problems indicated high memory consumption by TVS due to the increasing size of the table used for storing update timestamps of modified objects. This motivated us to develop a scheme to periodically truncate the table by removing some entries. For this purpose, TVS periodically sets a timestamp value called *Truncation Timestamp (TTS)* to a value less than STS by some fixed window size. If an item in the table is not being accessed by any current request and its update timestamp is smaller than TTS, then it is removed from the table. Any validation request with start-timestamp less than TTS is aborted because information about transactions committed with timestamp below TTS has been discarded. In our experiments we found that setting difference between STS and TTS to 1000 resulted in reducing the table size by 75% to 80% without causing any observable impact on the abort rate. The truncation scheme used here has some conceptual similarity to the scheme used in [1].

C. Fault Tolerance

Fault-tolerance in Beehive is based on coordinated checkpointing. One worker thread is responsible for coordination of system checkpointing. It first brings the system to a quiescent state in which no tasks are being executed by any worker thread and there are no messages in communication. Each Computation Engine then saves its local checkpoint state. Task execution resumes only after all Computation Engines have checkpointed their state. The checkpoint state of a Computation Engine includes the state of its storage server component and the state of the local task-pool. All worker threads are inactive in this state. Therefore, there is no need to checkpoint the state of the worker threads. The checkpoint state of TVS includes the values of the CTS and STS counters, which have the same value in the quiescent state. No other state of TVS is needed to be saved in the quiescent state.

Failure detection is based on a heartbeat protocol. A monitor thread exists in each Computation Engine to periodically send heartbeat messages to a failure detector process. If no heartbeat is received within a specified interval the Computation Engine is considered failed. The failure detector then shuts down the system and restarts it in the recovery mode. When restarted in the recovery mode, each Computation Engine reads its checkpoint file and loads the checkpointed state. All Computation

Engines are then in the *quiescent* state. At this point, the master instructs them to resume task processing.

VII. Programming with Beehive

In this section we present examples of parallel programs for a set of graph problems to illustrate the abstractions provided by our framework for the transactional model of parallel programming. We present here the program structures for computing graph coloring, single-source shortest paths (SSSP), and k-nearest nodes (KNN). Using the SSSP problem, we illustrate how incremental computations can be supported by the Beehive model when the graph structure is modified after executing an analytics program. We also use these problems for performance evaluation experiments.

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 5 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 in Figure 5. 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.

```
public class SPWorker extends Worker {
  public TaskSet doTask(Task task) {
    TaskSet newTasks = new TaskSet();
    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.weight) {
       writeSet.add( u );
       u.distance = t.distance + e.weight;
       u.predecessor = t.senderId;
       Set<String> nbrIds = u.neighbor.keys();
       foreach (String nbr in nbrIds) {
         Task t = new Task();
         t.nodeId = nbr;  // target node
         t.senderId = u.nodeId;
         t.distance = u.distance;
         newTasks.add( t );
       }
    }
    return newTasks;
  }
}
```

Fig. 5. Worker for Single-Source Shortest Path

B. K-Nearest Nodes (KNN) Problem

For a given undirected graph, this program computes for each node the list of its k nearest nodes and their distances. The structure of this program is very similar to the program for the single-source shortest path problem. Therefore, we omit the presentation of its code. Each node maintains a list of its currently known k-nearest nodes. The task computation in this program involves updating this list. If execution of a task results in updating this list for the target node because of adding new nodes to the list or by decreasing the distance value for a node already in the list, new tasks are created for the direct neighbors of the target node. Each task contains the sender node's id, and a list containing the ids and the distances of its k-nearest nodes. Based on this information, the execution of this task updates the target node's list of k-nearest nodes. This can then result in creation of new tasks. 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 task for each of its direct neighbors.

C. Supporting Incremental Computations

We have utilized the transactional programming model of Beehive for supporting incremental computations in dynamically evolving graph data structures [19]. This eliminates the need of re-executing the program for a large graph when only some small number of updates are made to the graph structure. For example, after executing the SSSP program on a graph, we consider the following kinds of updates to the graph: changing the distance value of an edge, and adding or removing an edge. Increasing the distance value of an edge which is not a "predecessor edge" requires no re-computation, but other changes require that each of the two nodes joined by that edge should "send" an SSSP task to the other node, similar to the task creation in Figure 5. Similarly, adding a new edge between two nodes requires very similar computation tasks to be initiated. In the case of an edge deletion, there are two cases to consider. If none of its two nodes is the predecessor of the other, then no re-computation needs to be initiated. Otherwise, for the node that had this deleted edge pointing to its predecessor, a transactional task is executed to determine which of its neighbors should become its new predecessor and what is its new distance to the source. This task would then initiate new such tasks for its neighbor nodes. If such a task results in changing the distance of a node, then further creation of new such tasks is needed for that node's neighbors.

VIII. Performance Evaluations

We conducted several experiments to evaluate the performance of the transactional programming model and the mechanisms provide by the Beehive framework. We used the problems described in Section VII for our evaluations. Our experiments were driven by the following goals. Specifically we wanted to evaluate the performance of the optimistic execution model for different graph problems in terms of the abort rates. We also wanted to study how the execution times for a problem depend on the graph size, the number

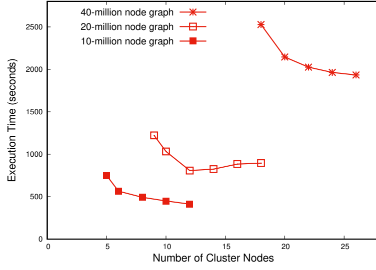

Fig. 6. Graph coloring execution for different graph and clusters nodes

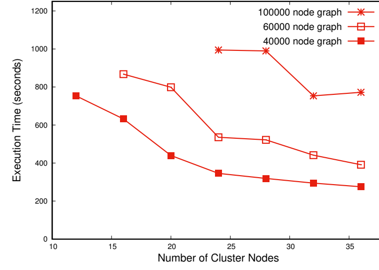

Fig. 7. K-Nearest-Nodes execution for different graphs and cluster nodes (k=40)

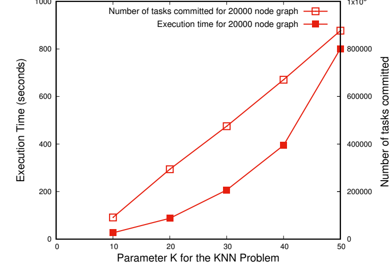

Fig. 8. Impact of parameter k on K-Nearest-Nodes problem on 10-node cluster

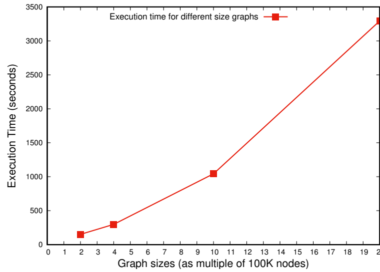

Fig. 9. Single-Source Shortest Path execution for different graphs on 12-node cluster

of tasks created and committed, and the number of cluster nodes. We also wanted to evaluate the performance benefits of the framework-level mechanisms such as object caching, reflection-based remote operations, and hierarchical validation.

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. For all problems, we generated random graphs with average node degree of 67.

1) Performance of Optimistic Execution and Hierarchical Validation: We used the graph coloring, SSSP, and KNN problems to evaluate the performance of the optimistic execution model. For the graph coloring problem we evaluated the benefits of the hierarchical validation scheme.

The data shown in Table I for the graph coloring shows that the optimistic approach performs well for this problem. Table I presents the number of transactions committed and the number of transactions aborted. For the aborted transactions, the numbers are given separately for those aborted in local validation and those by the TVS. The number of transactions committed represents the number of tasks that were needed to be executed to solve the given problem. We make the following observations based on these experiments. In this program the number of tasks executed and committed is the same as the number of nodes in the graph. For this problem the probability of transactions committing ranged from 95% to 99%, indicating low contention among tasks. The number of aborts is significantly lower for this problem, indicating a high degree of parallelism. The fraction of local aborts is low here, but its impact is insignificant as the total number of aborts tends to be quite low.

	Graph size (number of nodes)		
	200K	400K	1000K
Local Aborts	1192	1223	906
TVS Aborts	9196	8382	7136
Total Commits	200K	400K	1000K
Fraction of local abort	11.4%	12.7%	11.2%
Prob. of Task commit	0.95	0.97	0.99

TABLE I
ABORT/COMMIT STATISTICS FOR THE GRAPH COLORING PROBLEMS

Similarly we found that optimistic approach performs well for the SSSP and KNN problems. For these problems we found that the abort rates tend to in 1-3% range, indicating that these problems perform well under the optimistic execution model.

2) Scaling-out on Cluster with Graph Sizes: We executed on different number of cluster nodes the parallel programs for graph coloring, and KNN using different size input graphs to assess the scale-out performance. Figures 6 and 7 show the execution times for these programs for different graphs and cluster sizes. For all these problems, as the number of cluster nodes is increased for a given problem, initially we see a significant and almost linear decrease in the execution time, but it then gradually levels off for larger cluster sizes. This starts happening when the remote data access costs start dominating. Figure 9 shows the execution times of the SSSP program on a 12-node cluster for graphs with number of nodes 200K, 400K, 1-million, and 2-million. These programs have different characteristics in regard to the memory demands and computation load imposed by the tasks and the task-pool. In the case of the graph coloring problem, the number of tasks created is equal to the number of graph nodes. No new tasks are created dynamically by a task computation.

We also noticed that some problems could not complete when executed on a small number of cluster nodes. We found that two factors influenced this. One is the memory requirement of the storage server in a Computation Engine for storing the graph data. The other is the storage requirement for its Workpool server for storing tasks.

3) Performance of Dynamic Task Creation Model: For the SSSP and KNN problems we found that a large number of tasks were created, and many of them did not result in any useful computation. Such tasks are effectively futile and waste computing resources. For the SSSP problem we included an

optimization, as discussed earlier, to reduce the number of such redundant tasks. On a 10-node cluster, the SSSP program for a 1-million node graph executed close to 500 million tasks, and committed close to 12.5 million of them. The number of total completed tasks was about 40 times the committed tasks. This indicates a large number of redundant tasks.

For the KNN problem, the number of tasks committed depends on the value of parameter k along with the graph size. Figure 8 shows for a 20K node graph the execution times and the number of tasks committed for different values of k by this program's execution on a 10-node cluster. The number of tasks committed increase linearly with the value of k. However, the execution time tends to increase in a super-linear manner due to the overheads incurred when the number of tasks in the system is high.

IX. Discussion and Conclusion

The transactional model of parallel programming presented here provides a conceptually simple approach for harnessing amorphous parallelism in graph problems. Our experiments show that this approach performed well for graph coloring, SSSP, and KNN problems. Our experience in parallel programming with the Beehive framework presented here validates the simplicity of its transaction-based programming model.

Our experiments indicate that for a given problem, scaling-out beyond certain cluster size has marginal performance benefits. Typically, this occurs because the remote data access latencies start dominating the execution times. Also, executing a program for a large graph on a small number of cluster nodes can lead to poor performance due to the overheads imposed by the framework level mechanisms. These overheads stem from transaction management functions and synchronization of access to the data structures in the taskpool and the storage system. In contrast to message-passing based programming models, the approach presented here does not require significant conceptual redesign of the algorithm. We have shown here that a parallel program for a problem can be developed through simple adaptation of a sequential algorithm. This is facilitated by the transactional computing model coupled with the abstraction of a global object store implemented in the RAM of cluster nodes. However, the implementation of the algorithm needs to be driven towards amortizing or reducing remote data access cost. To reduce the cost of remote data access we provide mechanisms for fine-grain remote operations, data caching, and aggregation of remote calls.

The Beehive model allows graph structure to be modified by the transactional tasks. This makes Beehive suitable for dynamic and evolving graph structures. We have also described here the ability of this model in supporting incremental computations for dynamic graph structures. Moreover, Beehive model allows analysis of complex and rich graph structures, such as hypergraphs, with different kinds of edges between vertices, and vertices can represent entities of different types.

Acknowledgements: This work was supported by NSF Award 1319333 and computing resources were provided by NSF award 1512877 and the Minnesota Supercomputing Institute.

References

[1] Atul Adya and Barbara Liskov. Lazy consistency using loosely synchronized clocks. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC '97)*, pages 73–82, Santa Barbara, CA, August 1997.

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

[5] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. In *Proc. of OSDI'04*, pages 137–150, Berkeley, CA, USA, 2004. USENIX Association.

[6] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[7] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72. ACM, 2007.

[8] M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Casçaval. How much parallelism is there in irregular applications? In *ACM Symp. on Principles and Practice of Parallel Programming*, pages 3–14, 2009.

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

[10] James Larus and Christos Kozyrakis. Transactional memory. *Communications of the ACM*, 51(7):80–88, July 2008.

[11] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *Proc. VLDB Endowment*, 5(8):716–727, April 2012.

[12] 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.

[13] 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.

[14] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. The Tao of Parallelism in Algorithms. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 12–25, New York, NY, USA, 2011. ACM.

[15] Russell Power and Jinyang Li. Piccolo: building fast, distributed programs with partitioned tables. In *Proc. OSDI'10*, pages 1–14, Berkeley, CA, USA, 2010. USENIX Association.

[16] Bin Shao, Haixun Wang, and Yatao Li. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 505–516, New York, NY, USA, 2013. ACM.

[17] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 135–146, New York, NY, USA, 2013. ACM.

[18] 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.

[19] Anand Tripathi, Rahul R. Sharma, Manu Khandelwal, Tanmay Mehta, and Varun Pandey. Incremental Parallel Computing using Transactional Model in Large-scale Dynamic Graph Structures. In *International Workshop on Big Graph Processing (BGP 2017), held in conjunction with ICDCS'2017*, 2017.

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