Beehive: A Framework for Graph Data Analytics on Cloud Computing Platforms

Anand Tripathi, Vinit Padhye and Tara Sasank Sunkara
Department of Computer Science & Engineering
University of Minnesota, Minneapolis MN 55455
Email: {tripathi, padhye, sunkara}@cs.umn.edu

Abstract—Beehive is a parallel programming framework designed for cluster-based computing environments in cloud data centers. It is specifically targeted for graph data analysis problems. The Beehive framework provides the abstraction of key-value based global object storage, which is maintained in memory of the cluster nodes. Its computation model is based on optimistic concurrency control in executing concurrent tasks as atomic transactions for harnessing amorphous parallelism in graph analysis problems. We describe here the architecture and the programming abstractions provided by this framework, and present the performance of the Beehive framework for several graph problems such as maximum flow, minimum weight spanning tree, graph coloring, and the PageRank algorithm.

I. INTRODUCTION

Recent years have seen emergence of cloud computing platforms which can be used as utility infrastructures for performing large-scale data analytics. Many data analytics applications require processing of large-scale graph data. Analysis of such large-scale data requires parallel processing utilizing a large number of computing resources. This requires suitable parallel programming frameworks that can efficiently utilize a large pool of computing resources in a reliable and scalable manner. Parallel processing of graph computation raises several issues due to the unique nature of graph problems. In many graph problems, parallelism generally tends to have irregular structure [16], and such problems do not exhibit coarse-grain parallelism. In such graph problems it is difficult to extract parallelism through data partitioning. Such issues and challenges in parallel processing of graph problems are discussed in [21]. To harness the irregular parallelism inherent in such problems, also called *amorphous parallelism*[16], techniques based on optimistic parallel execution of concurrent tasks have been considered more attractive [16], [27].

For parallel processing of large data sets on cluster-based computing environments, the MapReduce [9] programming model is widely used in many applications. However, this model cannot be easily utilized in graph data analytics problems where it is difficult to partition the data for parallel processing. Moreover, this model does not expose intermediate data of computations which is often needed in many graph algorithms [26]. In recent years, several other frameworks [22], [26], [20] have been developed specifically for performing large-scale graph data analytics using a cluster of distributed computing resources. Dryad [15] is targeted for data-flow

based parallel computing. The Pregel [22] framework is specifically intended for graph data analysis. The computing model of Pregel is based on the message-passing paradigm and utilizes the Bulk-Synchronous Parallel (BSP) model [30]. 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. It is noted in [20] that it is difficult in the Pregel model to express certain dynamic parallel iterative computations, such as the dynamic PageRank computation. Distributed GraphLab [20] is another recent system which is specifically aimed at supporting parallel computing of graph problems, utilizing the notion of transactional task execution, but it does not support dynamic graph structures.

We present here our ongoing work on the design and development of the Beehive system for parallel programming of graph problems on cluster-based computing environments. The design of the Beehive framework is based on speculative computing approach, using optimistic concurrency control techniques, for harnessing amorphous parallelism in graph problems. The design of this framework is being driven by the following goals: (1) a programming model which simplifies programming of graph algorithms; (2) a computation model which can efficiently harness amorphous parallelism in graph problems; (3) efficient models and mechanisms for fault-tolerance and recovery. The computation model of Beehive is based on two central elements. The first provides a distributed globally shared object storage system for maintaining data structures in the memory of the cluster nodes. This global storage provides key-value based primitives for data storage and access of information based on vertices and edges in graph structures. This relieves the programmer from the burden of explicitly using message-passing primitives. The second element supports transactional execution of computation tasks to speculatively harness amorphous parallelism in graph problems using the optimistic concurrency control techniques. The motivations for adopting this approach is to provide a simple model for programming graph processing algorithms as compared to the message-passing model.

The next section presents a discussion of the related work on parallel programming frameworks for graph problems. Section III presents an overview of the Beehive computation model. Section IV gives an overview of the architecture of the

Beehive framework. In Section V we describe our experience in designing the Beehive framework and programming several graph applications with it. The last section presents our conclusions and future directions of this project.

II. RELATED WORK

The Beehive programming model is based on optimistic execution of concurrent tasks. The speculative execution techniques for extracting parallelism in an application have been widely investigated in the context of multi-core and multi-threaded architectures [4], [1], [27], [23]. Such techniques have utilized the notion of *software transactional memory* [14], [18], [28] to guarantee the atomicity and isolation properties of concurrently executed threads to ensure that the result of the speculatively executed parallel computations is equivalent to a serial execution of the threads. In contrast, our work on the Beehive framework investigates transaction-based speculative models of parallel computing in cluster-based computing environments. Distributed GraphLab [20] is another recent system which is specifically aimed at supporting parallel computing of graph problems, utilizing the notion of transaction-based task execution. The approach underlying the Beehive framework differs from that in Distributed GraphLab in several distinct ways. It is based on optimistic concurrency control model [17] for task execution as opposed to the locking based approach. Moreover, Distributed GraphLab does not support dynamic graph structures.

In developing the transaction management model for speculative execution in Beehive, we utilized the transaction management techniques we developed for key-value stores [25]. Alternatively, instead of using the speculative task execution model, one may perform conflict-free task scheduling. For example, one may avoid scheduling two or more tasks which are likely to modify common data. This scheduling scheme eliminates the need of conflict-checking, however, this approach can potentially limit parallelism because it is conservative and requires considering all potential conflicts. Furthermore, this approach requires having prior knowledge of the data items that would be accessed by a task, which may not be possible in some graph problems.

The Beehive model of parallel programming is based on a key-value based global data store and transaction-based task execution. The storage system model provides higher level of abstractions than a distributed shared memory (DSM) systems [19] and tuple-spaces[7]. A parallel program using a distributed shared memory system needs to incorporate suitable higher level synchronization mechanisms beyond the page-level memory coherency provided by such systems. Moreover, with DSM it is difficult to optimize for performance because of issues related to false-sharing and data locality. The Piccolo [26] system has provided a programming model based on the abstraction of a shared data store. However, it does not provide transactional task execution model, limiting the atomic operations and concurrent combining operations to a single key-based data item in the storage. This system is also based on the BSP model [30], requiring the parallel computations to be structured in phases and synchronizing them using the barrier primitive. This disallows purely asynchronous task execution models of parallel computing.

Parallel BGL [13] is a C++ library for parallel programming of graph algorithms on distributed memory multiprocessor systems. The steps involved in parallelizing a sequential graph algorithm for distributed memory systems are described in [13]. This library provides primitives for accessing and updating remote data, and provides synchronization mechanisms for distributed processes to execute in BSP-like phases. A parallel programming model for graph problems based on the notion of *active messages* is presented in [10]. The delivery of an active message at its destination results in execution of a designated message handler. This facilitates parallel programs with asynchronous message passing. In contrast to these approaches, the Beehive model is based on providing a distributed global storage for graph data, which is accessed using the Java RMI mechanism.

In our initial work on Beehive we investigated use of the Hadoop/HBase [2] system for key-value based global storage system. We found the HBase system unsuitable for use in the Beehive framework. First, the HBase system was found to be quite slow to provide any acceptable performance. Second, the HBase system does not provide easy control over dynamic placement of data items on HBase Region-Servers [2], [12]. Such control over data locality is desired from performance considerations in many graph algorithms. This lack of control in HBase becomes another major obstacle to achieve any acceptable performance. We also considered using the Memcached system [24], but it does not provide support for transactions, data persistence and recovery, data locality management, and multi-version data management. This motivated us to custom design and build a distributed in-memory key-value storage service in the Beehive framework.

III. COMPUTATION MODEL OF BEEHIVE

The computation model of Beehive is based on three core abstractions: First, it provides a distributed global object storage system maintained in memory of the nodes of the computing cluster. The graph data structures required for computation are stored in this global storage system and are managed using a key-value based access model. Second, it uses the task-pool paradigm [6]. A distributed pool of tasks is maintained in the system. Each task represents some computation to be performed on the data in the shared object storage system. Third, it provides a pool of worker threads. These threads execute tasks in parallel on different nodes in the cluster computing environment. A worker thread picks a task from the task-pool and executes it as an atomic transaction, in isolation with other concurrently executed tasks, Any worker thread can pick and execute any task from the pool. The execution and commitment of a task as a transaction may result in modifying some data in the global storage and creation of new tasks, which are then added to the task-pool. The task specification includes the necessary information for task execution, such as the computation to be performed and the

parameters for the computation. A parallel program for an application can define concurrent tasks performing different kinds of computations. In this sense, the Beehive model is more general than the existing programming models [20], [22], [26], where a single kernel function is executed on all vertices.

To harness irregular parallelism, the Beehive computation model is based on the approach of speculative parallel execution of tasks. The parallel execution of tasks is performed by the worker threads. Each task is executed as a transaction, and the transaction model is based on optimistic concurrency control [17]. However, in the speculative execution approach it is possible that two or more tasks running concurrently may access or modify some common data items. This requires concurrent tasks to be executed as transactions satisfying the properties of *atomicity* and *isolation* to ensure that the final state of the computation is equivalent to the one produced by some sequential execution of the tasks. Therefore, a transactional task is committed only if it does not conflict with any of the concurrently committed transactions. The conflicts are defined based on the notion of read-write or write-write conflicts, as in traditional database systems [3]. The rationale in adopting the optimistic execution approach is to exploit latent parallelism present in many graph problems by performing parallel execution of tasks on different parts of the input data assuming that the probability of conflicts among concurrently executed tasks would be low.

Following the optimistic execution model [17], the transactional task execution by a worker thread involves the following four phases: read phase, compute phase, validation, and writing to the global storage on transaction commitment. The task execution begins by first obtaining a timestamp for the transaction. This reflects the timestamp of the latest committed transaction such that the updates of all committed transactions with timestamps up to this value have been written to the global storage. The worker thread then reads the required data items from the global storage in its local memory buffers. In the compute phase, all updates are written to the buffered data. After the compute phase, the transaction is assigned a commit timestamp and a validation is performed to ensure that none of the items in its read-set or write-set have been modified by any concurrently committed transaction. On successful validation, the buffered updates are written to the global storage and any new task created by the computation are added to the task-pool.

In the Beehive programming model, a parallel program for a graph problem is composed of a set of functions that can be executed in parallel on different parts of the input graph data. We illustrate this parallel programming model using the *preflow-push* algorithm for the max-flow problem [8]. In this problem, a parallel program involves execution of two functions, *push*, and *lift*, on a vertex. When a vertex has some *excess* flow, i.e. its in-flow exceeds out-flow, the *push* operation is executed on that vertex to push the excess flow on out-going edges that have available capacity and are directed to vertices at lower *levels*. This push operation can cause some other vertices to have excess flow, making them candidates for the

```
public class Worker extends Thread {
 public void run() {
   while(true) {
    Task task  = getTask();
    BeginTransaction();
       newTaskSet = doTask(task);
       reportCompletion(task, newTaskSet);
    EndTransaction();
  }
 }
 abstract public TaskSet doTask(Task t) {
   // Application defined method
 }
}
```

Base Worker Thread Structure

```
public class MaxFlow extends Worker {
  public TaskSet doTask(Task task) {
    TaskSet newTasks = new TaskSet();
    vrtxId  = task.getVertexId();
    Vertex u = storage.getVertex(vrtxId);
    while (u.hasExcessFlow()) {
       boolean pushed;
       pushed = pushFlow(u, newTasks);
       if (not pushed)
          lift(u);
    }
    return newTasks;
  }
}
```

Task Definition for the Max-Flow Problem

Fig. 1.   An illustration of programming the max-flow problem using Beehive

application of the *push* operation. If the *push* operation cannot be performed on a vertex because all edges with available capacity are directed to vertices at the same or higher level, then the *lift* operation is performed on that vertex to increase its level. A task for this problem represents the execution of these functions on a vertex that has some excess flow. As noted above, the execution of such a task can result in creation of some other vertices with excess flow, thus resulting in creation of new tasks. A task is removed from the task-pool upon its successful execution by a worker thread, and a task execution may result in creation of new tasks, which are then added to the task-pool.

Figure 1 shows the example code for programming this problem in the Beehive model. The Beehive framework provides the abstract *Worker* thread class which fetches a task from the local task-pool by calling the *getTask* method. It then executes, as a part of a transaction, the *doTask* method to execute that task. After task execution, the worker contacts the Transaction Validation Service to validate the transaction. On commitment, the task is removed from the task-pool and any newly created tasks are distributed among different Beehive nodes according to the load distribution policies. With the optimistic execution model for tasks, it is possible for such a transaction to abort due to conflicts with some other concurrent tasks. In such a case, the worker thread re-executes the task as a new transaction.

The Beehive model does not enforce the use of the transaction semantics for task execution. In a parallel program, if
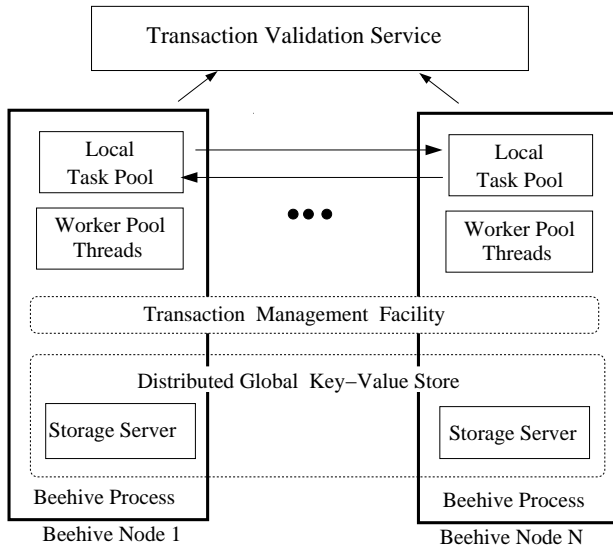
Fig. 2.   Beehive Architecture

concurrently executed tasks are known to be non-conflicting, then such tasks can be programmed to be executed without using the transaction semantics. Furthermore, Beehive also provides higher level synchronization mechanisms for barrier-based phase execution model. For example, problems such as PageRank can be implemented using barrier-based model to execute tasks which do not use transactional semantics. We illustrate this model in Section V. It also does not enforce that the entire computation of a task to be performed as one single transaction. A task may be executed as a sequence of multiple transactions.

IV. Beehive Architecture

We describe here the various components and mechanisms of the Beehive architecture. Figure 2 shows the architecture of the Beehive framework. The basic building-block for parallel computing in this framework is the Beehive process. This process implements all the three abstractions noted above. A collection of Beehive processes form the execution environment for parallel execution of a graph data analytics application. We refer to such processes as the Beehive nodes of the application execution environment. A Beehive node represents a logical entity. A global transaction validation service is used to check for update conflicts among concurrent transactions, following the optimistic concurrency control approach.

A Beehive process implements a storage server, which contains the data items for a subset of the keys in the global key-space. A Beehive process also contains a local pool of pending tasks, and a pool of worker threads. A task is placed in the local task pool of a Beehive process based on the locality considerations. A task defines certain computations to be performed on a set of vertices in the graph. In many graph problems, typically a task performs computations on a given vertex and possibly on its neighbor vertices.

The Beehive storage system abstraction provides a simple model for managing graph data. The data related to a vertex

and its edges to other vertices are stored as a single storage-level item accessed by the vertex id as the access key. The framework supports graph analytics problems where the graph structure may be modified dynamically by the computation. For this purpose, the storage model supports dynamic addition and deletion of vertices and edges in the graph structure. The storage model provides location-transparent data access to the parallel programs. However, a process may notice difference in latency while accessing a remote data item. In this regard, the architectural design of the Beehive node is driven by data locality considerations. The local storage server component along with Worker threads is embedded within the Beehive process implementing a Beehive node. The Beehive storage system supports mechanisms for dynamic relocation of data items, which can be utilized to improve data locality for tasks. The unit of data relocation is all the data related to a vertex.

A. Distributed Data Store

Beehive framework provides the abstraction of a shared storage implemented as a distributed key-value based object store. The storage system interface is extensible which goes beyond simple put/get interface and provides interfaces for executing methods on remote objects. The distributed store is implemented by the set of storage servers embedded in the Beehive nodes. The data items are distributed across the storage servers. The location of a data item is determined using a hash-based scheme, which identifies the default location for an item, referred to as the *home site* of the item. The data items can be relocated at runtime from their default location to any other node. Dynamic relocation raises the issue of finding the current location of a data item when a worker thread wants to access the item. For this purpose, we use a simple forwarding scheme as described below. When accessing an item, a Beehive process first contacts the current known storage server responsible for that item. If that server no longer holds the item, it responds with the address of the new location for the item. If the new location is not known, the home site is contacted, which always records the current location of the item. The Beehive process caches the location information for future references. This mechanism can be used by an application for clustering of graph data.

B. Transaction Model

We use timestamp-based optimistic concurrency control [17] approach. In this approach, a transaction optimistically performs its read/write operations without acquiring locks, and the writes are buffered in the local memory of the transaction until the commit point. At the commit point, a validation is performed to determine if the transaction conflicts with any concurrently committed transaction. A transaction is allowed to commit only if it does not conflict with any other committed concurrent transaction. This validation is performed by the Transaction Validation Service shown in Figure 2. The validation service can be scaled using appropriate replication schemes, as we have shown in [25]. We have been able to

achieve validation throughput of close to 24000 requests per second with 8 service replicas.

When a transaction starts, it obtains a *start timestamp* from Transaction Validation Service which indicates the commit timestamp value such that the updates of all transactions with commit timestamp up to this value are present in the shared storage. At the time of validation the Transaction Validation Service checks for read-write conflicts with any of the concurrently committed transactions. If no conflicts are found, the transaction is assigned a commit timestamp using a monotonically increasing counter and commit response is sent to the task. The task then writes the modified data to the shared storage and reports its *completion* to the validation service. The validation service 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 have completed. The STS value is used for assigning the start timestamp for new transactions. For efficient validation we use a hierarchical scheme in which a task is first validated by a local validator at a node to check for conflicts with any local tasks, and a validation request is sent to the global validator only if no local conflicts are found. In our experiments using the max-flow problem we found that this scheme reduced the load on the global validator by more than 60%.

C. Task Distribution and Placement

The execution of a task may result in creation of new tasks, and such tasks are added to the task-pool when the task is completed. We describe below the task distribution and placement schemes supported in the Beehive framework. The first aspect is the distribution of tasks among Beehive nodes for proper balancing of load and the second aspect is the placement of tasks on a node on the basis of the locality of data. For load distribution we implemented different policies based on *sender-initiated* and *receiver-initiated* load balancing approaches [29]. In Beehive, with the sender-initiated load balancing scheme, newly created tasks are distributed to k nodes selected using the following policies: random selection, k least loaded nodes, and round robin selection. In our experiments, we observed that the random selection and load-aware selection policies perform equally well.

Another important aspect is the placement of a task on a node based on its affinity. The affinity is determined typically based on the data locality considerations. It is desirable to schedule a task on a node which stores the data required for task computation. We define three task affinity levels based on data locality considerations: (1) *strong affinity*, which indicates that the task must always be executed at its desired node, (2) *weak affinity*, which indicates that it is preferred to execute the task on its desired node, however it can be executed at any other nodes, (3) *no affinity*, which indicates that the task can be executed at any node. Apart from data locality considerations, the affinity can also be specified if certain special tasks are required to be executed on specific nodes, for example certain initialization tasks to be executed at a

node. Moreover, different tasks may have different affinity levels. The affinity levels are considered together with the load distribution policies discussed above while distributing tasks among Beehive nodes. The affinity level of a task takes priority while distributing that task to a node. Thus, if a task has strong affinity level, it is executed on its desired node irrespective of load balancing considerations. In our experiments using the max-flow problem we found that setting the strong affinity level for all tasks does not perform well, and therefore setting weak affinity or no affinity is desirable. Another option is to set strong affinity level only for the tasks that have all or most of their data located at a single node.

V. Programming with Beehive

In this section we describe our initial experience in programming with the Beehive framework and the insights that we gained through these experiments. During the course of the framework development, we experimented with several graph problems. The insights gained through this helped us in understanding and improving the performance of different components of the framework. This initial experience also helped us in understanding how to write parallel programs efficiently using this framework. We describe here the results of our experiments in programming the following four graph problems.

We programmed maximum-flow analysis problem using *preflow-push* algorithm using the programming model described above. We also developed parallel programs for *minimum spanning tree* and the *graph coloring* problem. For the spanning tree problem, we use the method described in [11]. We implemented the PageRank algorithm to experiment with the barrier-based synchronization model of Beehive. Our program for PageRank utilizes the non-transactional task execution mode.

We conducted experiments using the cluster provided by the Minnesota Supercomputing Institute (MSI). Each cluster node has 8 CPU cores with 2.8 GHz capacity and 22 GB main memory, connected with 40-gigabit network.

A. Maximum Flow

The maximum-flow problem is a classical example of problems in which parallelism tends to be amorphous, making it difficult to explicitly parallelize the computation. Moreover, in this problem the degree of parallelism can vary quite significantly across different graphs depending on their structure and link capacities. It is also difficult to assess the degree of parallelism through static analysis of the graph. We used this problem as a case study during the initial phase of the design and development of the Beehive framework. Programming of this problem demonstrated how Beehive model simplifies the task of writing a parallel program. In case of the max-flow problem, we transcribed the push-lift operations of the sequential preflow-push program as transactional tasks in the Beehive model. Programming this problem also gave us useful insights such as the impact of the task granularity on performance. The task granularity is the amount of computation performed by

Vertices	Edges	Beehive Nodes	Time (secs)
1600	4760	10	336
2500	7450	10	622
5000	14900	10	2254
10000	29800	20	5878

TABLE I
EXECUTION TIMES FOR MAX-FLOW

Vertices	Edges	Beehive nodes	Time(in secs)
1000	16827	10	16
2000	65971	10	24
5000	84905	10	32
10000	337,482	10	96
20000	672,725	10	728
50000	1,682,659	10	7138

TABLE II
EXECUTION TIMES FOR MST

Vertices	Edges	Beehive nodes	Time(in secs)
100,000	3,373,321	10	110
200,000	6,724,266	10	208
300,000	10,089,422	10	308
400,000	13,459,419	10	425
500,000	16,818,073	10	541
1,000,000	33,642,660	10	1253
2,000,000	67,265,322	10	4304
2,000,000	67,265,322	20	1462
2,000,000	67,265,322	30	1406

TABLE III
EXECUTION TIMES FOR GRAPH COLORING

a task. We also used this problem to evaluate the benefit of hierarchical validation scheme.

We generated the test graphs using Washington-Graph-Generator[1]. We generated *random level graphs* of different sizes and capacities for our experiments. For these graphs, we performed clustering of vertices using the object relocation mechanisms of Beehive, prior to executing the max-flow computation. In our initial implementation of the preflow algorithm, the task involved performing push and lift operations on a single vertex. We then experimented with increasing the granularity of a task to perform push/lift operations on a vertex as well as on all of its neighbors with excess flow. We found that increasing the granularity provides better performance. For example, for a 1600-vertices graph, the computation time reduced from 471 secs to 379 secs with increase in granularity. We also investigated the impact of setting different affinity levels for task placement. We found that the use of weak affinity performed better than strong affinity. For the 1600-vertices graph, setting strong affinity level led to execution time of 964 seconds whereas with weak or no affinity it reduced to 471 seconds on our research lab cluster. In Table I, we show the execution times for the max-flow problem for graphs of various sizes, using the MSI cluster. In these experiments, we used weak affinity level and higher task granularity as described above.

B. Minimum Spanning Tree

We implemented the minimum spanning tree (MST) algorithm [11] in Beehive. In our implementation of the MST algorithm initially a task is created for each vertex. The task computation involves finding the nearest neighbor of the task's vertex and merging them to form a cluster. The task vertex that forms the cluster becomes the *cluster head*. Each cluster formation operation is executed as a transaction. When two clusters are merged the information for the two cluster head vertices is modified. The transaction semantics ensure that in case of concurrent merge operations on a cluster, only one would succeed. The task then iteratively performs merging operations to form a larger cluster until it does not have any neighbors or the task vertex is no longer a cluster head because it has been merged into another cluster. This algorithm also detects connected component in the graph, finding spanning tree for each connected component. The computation time for randomly generated graphs of different sizes are presented in

Table II. For graphs with 1000 and 5000 vertices, the number of neighbors for a vertex was randomly selected between 1 to 50. For all other graphs the number of neighbors was randomly selected between 1 to 100. The program was run with 10 Beehive processes located at different computing nodes of the cluster. Each Beehive process had 10 worker threads.

C. Graph Coloring

In this problem, we create coloring tasks for each vertex. Initially all the vertices are uncolored. A coloring task for a vertex involves selecting the smallest unused color among its colored neighbors and assigning that color to the vertex. A coloring task for a vertex is executed as a transaction. The neighbors of the vertex form the read-set and the task vertex forms the write-set of the transaction. A task is aborted in case of any read-write conflict with any concurrent tasks and the coloring operation is re-executed.

We generated random graphs of different sizes and measured the computation time for executing the coloring algorithm. The number of neighbors for a vertex was uniformly distributed between 1 to 100. We also measured the speedup achieved by increasing the number of Beehive nodes. We observe that for 2 million vertices graph increasing the cluster size from 10 nodes to 20, resulted in performance improvement by a factor of more than 2. However, increasing it further to 30 nodes provided marginal improvement.

D. PageRank

We implemented PageRank [5] algorithm to experiment with barrier-synchronization based phased execution model. In this problem, transaction semantics in task execution is not required. PageRank algorithm involves multiple iterations

Vertices	Edges	Beehive Nodes	Time(in secs)
100,000	3,373,321	10	93
200,000	6,724,266	10	181
1,000,000	33,642,660	10	1746
1,000,000	33,642,660	20	721
1,000,000	33,642,660	30	635
2,000,000	67,265,322	20	2072

TABLE IV
EXECUTION TIMES FOR PAGERANK

Problem	Completed Tasks	Aborts	Time (secs)
Max-Flow	9716609	71685675	5878
Graph Coloring	10003	299	19.6

TABLE V
ABORT STATISTICS

of page-rank computation and, in each iteration, page rank of a web-page (vertex) is calculated based on the previous ranks of the web pages that have links to this page. In our implementation, each barrier phase corresponds to a single iteration of the page-rank algorithm. For each vertex/web-page we maintained two rank values corresponding to the current and previous iteration. Each iteration involved creating tasks for each vertex to calculate page-rank for that web-page for the given iteration. A task for a vertex requires reading ranks of all the vertices that have links pointing to it, and updating the rank value for the vertex in the global storage. To synchronize phases we designated one of the processes as *coordinator* which was responsible for initiating tasks for a phase and detecting termination of a phase across all Beehive nodes. We evaluated PageRank algorithm in Beehive using random graphs of various sizes. Table IV shows the execution times for PageRank on various graphs for performing 50 iterations of the algorithm.

VI. DISCUSSION AND FUTURE WORK

Our experience in parallel programming with the Beehive framework validates the simplicity of its programming model. For example, in case of the max-flow and the graph coloring problem, we essentially transcribed the computation performed in the sequential algorithms for these two problems into transaction tasks. Thus, in contrast to the message-passing model, the Beehive model does not require significant conceptual redesign of the algorithm. However, the implementation of the algorithm needs be driven towards amortizing or reducing remote data access cost. For example, in case of the max-flow problem, we observed that increasing the task granularity provides significant performance improvements. In this problem, larger granularity meant performing flow balancing for more than one vertex in a transactional task. Similarly, in case of the MST problem, we had to focus on reducing the data access cost in identifying cluster-head nodes.

Based on the results of our experiments described above we make the following observations. The parallelism in a graph problem can vary significantly based on the nature of the problem and the structure of the graph. The number of aborts of speculatively executed tasks is one indicator of the amount of parallelism in the given problem. A high abort rate for tasks indicates low degree of parallelism. We illustrate this by observing the number of aborts for the max-flow and graph coloring problems for a graph of 10000 vertices. This data is

shown in Table V. The ratio of abort to commit for the max-flow problem is significantly higher (close to 7.3) than that for the graph coloring problem (close to 0.03). The high abort rate in the max-flow problem for this graph is indicative of low amount of parallelism.

Our experiments also 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. This indicates that for a given problem there is typically an optimal cluster size for the best performance. Another critical aspect is the time required for initial loading of data. For large problems, we had to devise efficient methods for parallel loading of data.

There are several aspects that need further investigation. One area that needs further investigation is data clustering for improving data locality. Another important area for future work is the investigation of fault tolerance and recovery mechanisms. Currently all data is maintained in the memory of the cluster nodes. Appropriate checkpointing and recovery mechanisms need to be developed in the Beehive framework to deal with node crashes. Unlike MapReduce applications, recovery mechanisms in graph problems are even more complex as it is not simple data partition but a state of graph that needs to be restored to avoid a complete rerun of the algorithm. That too in a optimistic-task execution model, a complete snapshot of the graph has to be maintained and recovery always involves additional computation.

Another area of future work is to investigate adaptive methods for scheduling tasks based on the observed abort rate. Adaptive methods can be used to control the degree of optimistic execution, i.e. the number of tasks executed optimistically in parallel. Another direction to explore is the use of a hybrid of optimistic and conflict-free scheduling methods, where the framework would dynamically shift from optimistic to conflict-free scheduling when the parallelism drops below a certain threshold. The amount of parallelism can be measured by the commit rate.

VII. CONCLUSION

The work presented here describes our initial experience with the Beehive framework. We have presented here the optimistic execution model and the system architecture of the Beehive framework. Our experiments and experience in programming with the Beehive framework show that the model of transaction-based optimistic execution of tasks can be effectively utilized for harnessing amorphous parallelism

in graph problems. This model also provides a conceptually simple model for the programmers to develop parallel programs because it relieves the programmer from the burden of explicit message-passing and synchronization operations. Our future work will focus on the integration of checkpointing and recovery mechanisms in this framework to deal with node crashes. We also plan to further investigate different scheduling strategies which can adapt the degree of optimistic execution based on the observed abort rates.

REFERENCES

[1] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman, "Compiler and runtime support for efficient software transactional memory," in *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '06, 2006, pp. 26–37.

[2] Apache, "Hbase, http://hbase.apache.org/." [Online]. Available: http://hbase.apache.org/

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

[4] A. Bhowmik and M. Franklin, "A general compiler framework for speculative multithreading," in *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '02, 2002, pp. 99–108.

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

[6] N. Carriero and D. Gelernter, "How to write parallel programs: a guide to the perplexed," *ACM Comput. Surv.*, vol. 21, pp. 323–357, September 1989.

[7] ——, "Linda in context," *Communcations of the ACM*, vol. 32, no. 4, pp. 444–458, Apr. 1989.

[8] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*. The MIT Press, 1999.

[9] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," in *Proc. of OSDI'04*. Berkeley, CA, USA: USENIX Association, 2004, pp. 137–150.

[10] N. Edmonds, J. Willcock, and A. Lumsdaine, "Expressing graph algorithms using generalized active messages," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS '13, 2013, pp. 283–292.

[11] R. G. Gallager, P. A. Humblet, and P. M. Spira, "A Distributed Algorithm for Minimum-Weight Spanning Trees," *ACM Trans. Program. Lang. Syst.*, vol. 5, pp. 66–77, January 1983.

[12] L. George, *HBase - the Definitive Guide*. O'Reilly, 2011.

[13] D. Gregor and A. Lumsdaine, "Lifting sequential graph algorithms for distributed-memory parallel computation," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '05, 2005, pp. 423–437.

[14] M. Herlihy and J. E. B. Moss, "Transactional memory: architectural support for lock-free data structures," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ser. ISCA '93. New York, NY, USA: ACM, 1993, pp. 289–300.

[15] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, ser. EuroSys '07. ACM, 2007, pp. 59–72.

[16] 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*, 2009, pp. 3–14.

[17] H. T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Trans. Database Syst.*, vol. 6, pp. 213–226, June 1981.

[18] J. Larus and C. Kozyrakis, "Transactional memory," *Communications of the ACM*, vol. 51, no. 7, pp. 80–88, Jul. 2008.

[19] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Trans. Comput. Syst.*, vol. 7, no. 4, pp. 321–359, Nov. 1989.

[20] 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*, vol. 5, no. 8, pp. 716–727, Apr. 2012.

[21] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, "Challenges in parallel graph processing," *Parallel Processing Letters*, vol. 17, no. 1, pp. 5–20, January 2007.

[22] 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*, 2010, pp. 135–146.

[23] M. Mehrara, J. Hao, P.-C. Hsu, and S. Mahlke, "Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory," in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09, 2009, pp. 166–176.

[24] Memcached, "Memcached system, http://www.memcached.org."

[25] V. Padhye and A. Tripathi, "Scalable Transaction Management with Snapshot Isolation NoSQL Data Storage Systems," in *IEEE Transactions on Services Computing*, 2014 (to appear).

[26] R. Power and J. Li, "Piccolo: building fast, distributed programs with partitioned tables," in *Proc. OSDI'10*. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–14.

[27] A. Raman, H. Kim, T. R. Mason, T. B. Jablin, and D. I. August, "Speculative parallelization using software multi-threaded transactions," *SIGARCH Computer Architecture News*, vol. 38, no. 1, pp. 65–76, March 2010.

[28] N. Shavit and D. Touitou, "Software transactional memory," in *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '95, 1995, pp. 204–213.

[29] N. Shivaratri, P. Krueger, and M. Singhal, "Load distributing for locally distributed systems," *IEEE Computer*, pp. 33–44, December 1992.

[30] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990.