Technical Report

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

TR 13-022

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

Anand Tripathi, Vinit Padhye, and Tara Sasank Sunkara

July 24, 2013

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

ABSTRACT

We describe here our current work on the development of a programming framework called *Beehive* for graph data analysis on cloud computing environments. This framework is based on the speculative computing approach using transactional task executions for harnessing amorphous parallelism in graph data analysis problems. We describe here the architecture and the programming abstractions provided by this framework. We present here the results of programming several graph problems using this framework.

1. 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 such 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 [7], and such problems do not exhibit coarse-grain parallelism. In such graph problems it is difficult to extract parallelism through data partitioning. To harness the irregular parallelism inherent in such problems, also called *amorphous parallelism* [7], approaches based on the speculative parallel execution model have been considered more attractive [7, 13].

For parallel processing of large data sets, MapReduce 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 needed in many graph algorithms [12]. In recent years, several other frameworks [4, 9, 10, 12] have been developed specifically for performing large-scale graph data analytics using a cluster of distributed computing resources. The Pregel [10] 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 [14]. It is noted in [9] that it is difficult in the Pregel model to express certain dynamic parallel iterative computations, such as the dynamic PageRank computation. Distributed GraphLab [9] 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.

In this paper, we present a framework called *Beehive* for parallel programming of graph problems. In developing this framework, our goal is to provide a simple programming model based on speculative computing approach for parallel computing of graph problems on cloud computing platforms. The design of this framework is being driven by the following goals: (1) a programming model which simplifies implementation 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. One is to provide a distributed shared storage for storing and manipulating graph data. This global shared storage provides key-value based primitives for data storage and access of information based on vertices and edges in the graph. The second element of the computation model is transactional execution of computation tasks to speculatively harness amorphous parallelism in graph problems. 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 investigation of the speculative approach has mostly been conducted on multi-core architectures. In contrast, our work is investigating this approach for cluster-based parallel computing.

The next section presents the overview of the *Beehive* computation model. Section 3 presents the design of the *Beehive* framework. In Section 4, we describe our initial 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.

2. COMPUTATION MODEL OF BEEHIVE

The *Beehive* computation model is based on the task-pool

Figure 1: Beehive Architecture

paradigm [2], where a dynamic pool of tasks is maintained along with a set of worker processes executing on different processors of a cluster or cloud computing environment. The parallel execution of tasks is performed by the worker processes. All data items pertaining to the given graph are stored in the shared data store. Therefore, any worker process can pick and execute any task from the pool. To harness parallelism in a given problem, tasks are executed speculatively by the workers processes. To ensure consistency in case of any update conflicts among concurrent tasks, each task execution is performed as a transaction with ACID [6] (*atomicity, consistency, isolation, and durability*) properties. The transaction model is based on optimistic concurrency control [8]. The execution and commitment of a task may result in creation of new tasks. Basically there are three main abstractions provided by this framework: a shared key-value based data store, a pool of tasks where each task represents a computation that is performed as an atomic action and in isolation with other concurrent tasks, and a set of worker processes which execute tasks from the pool.

Figure 1 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. A parallel program for an application can define many different kinds of task computations. In this sense, the Beehive model is more general than the existing programming models [9, 10, 12], where a single kernel function is executed on all 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.

In the Beehive programming model, a parallel program for a graph problem is composed of a set of functions that could be executed in parallel on the different parts of the input graph data. We illustrate this parallel programming model using the *preflow-push* algorithm for the max-flow problem [3]. 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 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 2 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 trans-

action. 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.

```
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();
        vtxId = task.getVertexId();
        Vertex u = storage.getVertex(vtxId);
        while (u.hasExcessFlow()) {
            boolean pushed;
            pushed = pushFlow(u, newTasks);
            if (not pushed)
                lift(u);
        }
        return newTasks;
    }
}
```

Task Definition for the Max-Flow Problem

Figure 2: An illustration of programming the max-flow problem using Beehive

The Beehive model does not enforce the use of the transaction semantics for task execution. In a parallel program, if 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 4. 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.

3. BEEHIVE ARCHITECTURE

We describe here the various components and mechanisms of the Beehive architecture.

3.1 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.

3.2 Transaction Model

We use timestamp-based optimistic concurrency control [8] 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 Transaction Validation Service shown in Figure 1. We have demonstrated the applicability of such an approach of using a dedicated service for conflict detection in our work [11] on scalable transaction management for key-value store. With appropriate replication scheme, we have been able to achieve validation throughput of close to 24000 requests per second with 8 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 only sent to the global validator 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%.

3.3 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. In sender-initiated load balancing scheme new tasks created 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.

4. 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 experiences. 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* problem based on the method described in [5], and the *graph coloring* problem. 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.

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

We generated the test graphs using Washington-Graph-Generator¹. We generated *random level graphs* of different sizes and capacities for our experiments. For these graphs, we performed clustering of vertices using the 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 used this problem to evaluate the impact of setting affinity levels for tasks. We found that for

¹<http://www.informatik.uni-trier.de/~iaehner/Professur/research/generators/maxflow/wash/>

# of vertices	Beehive Nodes	Time (secs)
1600	10	336
2500	10	622
5000	10	2254
10000	20	5878

Table 1: Execution times for Max-Flow

# of vertices	Beehive nodes	Time(in secs)
1000	10	27
2000	10	55
5000	10	470
10000	10	1653

Table 2: Execution times for MST

the same 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. However, in case of other graph problems described below strong affinity was used. In Table 1, 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.

4.2 Minimum Spanning Tree

We implemented the minimum spanning tree (MST) algorithm [5] 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 components. The computation time for randomly generated graphs of different sizes are presented in Table 2. The number of neighbors for a vertex was randomly selected between 1 to 50. Each Beehive node had 10 worker threads.

4.3 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

# of vertices	Beehive nodes	Time(in secs)
100,000	10	110
200,000	10	208
300,000	10	308
400,000	10	425
500,000	10	541
1,000,000	10	1253
2,000,000	10	4304
2,000,000	20	1462
2,000,000	30	1406

Table 3: Execution times for Graph Coloring

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

Table 4: Abort Statistics

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.

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. 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 4. The ratio of abort to commit for the max-flow problem is significantly higher (close to 7.3) than that is 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.

4.4 PageRank

We implemented PageRank [1] algorithm to experiment with barrier-synchronization based phased execution model. In this problem, transaction semantics were not required and therefore there was no need of using the Transaction Validation Service. PageRank algorithm involves multiple iterations of page-rank computation and in each iteration page rank of a web-page (vertex) is calculated based on previous ranks of the web pages which 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

# of vertices	Beehive Nodes	Time(in secs)
100,000	10	93
200,000	10	181
1,000,000	10	1746
1,000,000	20	721
1,000,000	30	635
2,000,000	20	2072

Table 5: Execution times for PageRank

web-page for the given iteration. A task for a vertex requires reading ranks of all its upstream vertices and updating the vertex in the 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 5 shows the execution times for PageRank on various graphs for performing 50 iterations of the algorithm.

5. CONCLUSION

The work presented here describes our initial experience with the Beehive framework which is still under development. Our experience with the Beehive validates the simplicity of its parallel 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 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. Similarly, in case of the MST problem, we had to focus on reducing the data access cost in identifying cluster-head nodes.

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 of the aspects is modulation of the speculative execution model to include conflict-free scheduling and throttling of task execution on aborts to reduce the abort rates. Our future plans include investigation of fault tolerance and recovery mechanisms. Another area that needs further investigation is data clustering for improving data locality.

Acknowledgement: This work was carried out in part using computing resources at the University of Minnesota Super-computing Institute

6. REFERENCES

- [1] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proc. of WWW'98*, 1998.
- [2] Nicholas Carriero and David Gelernter. How to write parallel programs: a guide to the perplexed. *ACM Comput. Surv.*, 21:323–357, September 1989.
- [3] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. The MIT Press, 1999.
- [4] 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.
- [5] R. G. Gallager, P. A. Humblet, and P. M. Spira. A Distributed Algorithm for Minimum-Weight Spanning Trees. *ACM Trans. Program. Lang. Syst.*, 5:66–77, January 1983.
- [6] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, December 1983.
- [7] M. Kulkarni, M. Burtcher, 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.
- [8] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6:213–226, June 1981.
- [9] 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.
- [10] 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.
- [11] Vinit Padhye and Anand Tripathi. Scalable Transaction Management with Snapshot Isolation on Cloud Data Management Systems. In *Proc. of IEEE 5th Intl. Conference on Cloud Computing*, 2012.
- [12] 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.
- [13] Arun Raman, Hanjun Kim, Thomas R. Mason, Thomas B. Jablin, and David I. August. Speculative parallelization using software multi-threaded transactions. *SIGARCH Computer Architecture News*, 38(1):65–76, March 2010.
- [14] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.