Maestro: A cloud computing framework with automated locking

(Invited Paper)

Murat Demirbas
University at Buffalo, SUNY
demirbas@buffalo.edu

Serafettin Tasci
University at Buffalo, SUNY
serafett@buffalo.edu

Sandeep Kulkarni
Michigan State University
sandeep@cse.msu.edu

I. Abstract

Concurrent execution is a big challenge for distributed systems programming and cloud computing. Using locks is the most common technique for developing distributed applications that require tight synchronization. Unfortunately, locking is manual, error-prone, and unscalable. To address this issue, we propose a scalable automated locking framework called Maestro.

Maestro consists of a master and several workers, which can be dynamically instantiated on demand. Maestro examines the program actions of the workers before deployment and automatically decides which worker actions can be executed locally (without contacting the master) and which actions require synchronization through the master.

Maestro has applications in graph processing, real-time enterprise analysis, and web-services domains. By enabling the developers to write code at a higher-level of abstraction (shared-memory), Maestro improves productivity and lowers the cost of entry to cloud computing backend development.

***Keywords*-data center computing; tightly synchronized applications; automated locking; fault-tolerance**

II. Introduction

Concurrent execution is a big challenge for distributed systems programming and cloud computing. Concurrency needs to be boosted to improve the system performance and availability. For embarrassingly parallel data processing applications, concurrency is boosted and scalability is achieved by adding more workers [1]. However, for data processing applications that require tight synchronization, such as large-scale graph processing systems, real-time enterprise analysis systems, banking and stock market applications, uncontrolled concurrency wreaks safety violations. In these kind of applications, concurrent execution needs to be controlled to prevent latent race conditions and synchronization bugs.

Using locking primitives is the most common technique to deal with tight synchronization requirements [2], [3]. Unfortunately, locking is manual and hence is error-prone. If the developer misses to place a lock where required, then safety is violated. (And these kind of bugs may be latent heisenbugs and hard to debug.) In contrast if the developer inserts unneeded locks, then the performance of the system suffers due to the unnecessary synchronizations.

Here we suggest that it is possible achieve automated concurrency control as well as a scalable system. We propose an automated locking service architecture with a master and several workers. Our framework, Maestro, accepts program actions for the workers in a guarded command structure (in the event-driven programming terminology, a guard corresponds to the event detection condition and the command corresponds to the event-handling code). By examining the guards of the actions, Maestro automatically decides which worker actions can be executed locally (without contacting the master), and which actions require synchronization/locks through the master, as we describe in more detail in the next section.

The advantages of Maestro are many. Maestro enables the developers to write programs in a high-level language without having to worry about the distribution and concurrency challenges involved in datacenter computing. Maestro automates the insertion of locks without involving the developer. This lowers the cost of entry to cloud-computing backend application development. While there has been a lot of work on parallel batch processing systems based on MapReduce [1], Maestro fills a void in real-time tightly-synchronized/consistency-critical cloud applications, such as graph processing, financial applications, and real-time enterprise analysis applications. By doing the compilation from high-level shared-memory model to the message-passing datacenter computing model, Maestro introduces many opportunities to optimize the performance of the cloud applications (such as partitioning data/tasks to workers appropriately, pre-assigning leases to appropriate workers). Finally, Maestro is scalable and lightweight, and provides masking fault-tolerance to worker/master failures.

A. Maestro goals

Our project has the following goals:

Automatic insertion of locks/synchronization without involving the developer. By analyzing the high-level program input by the developer, Maestro automatically inserts locks where they are needed and deploys the corresponding distributed message-passing program with the needed supporting services in the datacenter. Safety (correctness) of the high-level input program is preserved by this transformation,

while fast performance and scalability is unhampered by unnecessary/avoidable synchronization.

Improving the effectiveness and efficiency of the developer. Maestro enables the developer to write code at the shared memory level without having to worry about distribution and concurrent execution challenges. By highlighting the async and sync actions in the input program, Maestro empowers the developer to improve performance even further.

High-scalability and elastic-scalability. Maestro aims to beat transactional and manual-locking solutions via the use of async actions and input-program-specific optimizations. In Maestro, locks are assigned subject to a predicate evaluation of the guards of sync actions, and this reduces the wasted concurrency in traditional transactions and manual locking approaches due to *busy polling*. [1]

Masking fault-tolerance against worker & master failures. Maestro aims to make workers fully soft-state and easily restartable, so that a master can instantiate and start a new worker in place of a failed or slow worker quickly. Maestro also aims to make the master hot-swappable to mask master failures as well.

III. Maestro framework

Maestro accepts as input high-level (shared memory) program actions for workers in a guarded command structure (i.e., guard → command). In the event-driven programming terminology, a guard corresponds to an event detection predicate and the command corresponds to the corresponding event-handling code. By examining the guards of the worker actions, Maestro automatically decides which worker actions can be executed locally (without contacting the master), and which actions require synchronization through the master, as follows. If the guard of an action is local, it means there is no need to communicate with other nodes before evaluating the guard since other nodes cannot affect the truth value of the guard. If the guard contains global variables but no other node can change the truth value of the guard until this worker performs another action, again there is no need to obtain the most up-to-date information from other workers to perform this operation. On the other hand, if the truth of the guard can be changed by other workers after the guard is evaluated to true, the worker needs to check in with other workers to obtain the latest values of its variables before performing this operation.

We name the first two types of guards as asynchronous (async) guards, because unlike the third case, they do not require synchronization with other nodes for guard evaluation. The advantage of async guards is that they do not require locking of the variables. Other workers do not have to wait for the completion of this operation and they can continue their operations without affecting the truth value of this guard.

[1]Busy polling is the unnecessary locking of variables by some workers continuously just to evaluate a condition only to find it false and abort the remainder of computation.

A. Maestro architecture

The master is responsible for keeping the durable copies of all program variables, tables, and databases. (Master can use dedicated storage nodes for keeping the databases, but what we mean is master always has the authoritative copy and controls the access to the databases and permanent copies of variables.) The workers keep a soft state and synchronize it with the master when necessary. The workers do the computation using the variables and data they cache from the master according to the rules we describe below.

While executing an async action, the worker does not contact the master and evaluates the guard locally (using local and cached variables). In the case that the guard evaluates to true, the worker executes the statement body (command part) of the action. If the statement body updates some global variables, these updated values must be pushed back to the master. However, the pushing of these values to the master can be delayed until the updated variables are read by a sync action at that worker. Since execution of a sync action involves contacting the master anyways, the pushing of the updated variables to the master is piggybacked at the beginning of that action execution.

While executing a sync action, the worker needs to contact the master in order to retrieve the most up-to-date values of the global variables the guard includes. When contacted by a worker for a sync action, the master evaluates the guard. If the guard holds, the master grants a lock to that worker for execution of the action and sends the most up-to-date values of the variables needed for the statement execution. This lock ensures that the sync action is executed atomically with respect to other workers. Other workers cannot modify these locked variables during that action execution. The worker writes back the results to the master right after execution and this automatically releases the lock at the master. Figure 1 below illustrates the interaction between the master and workers.

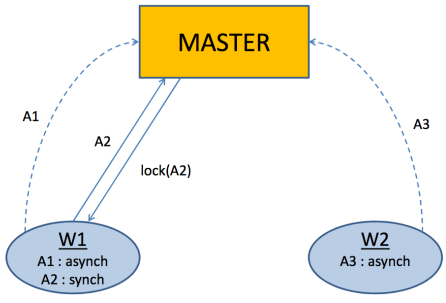

Figure 1. Interaction between master and workers

Thus, in Maestro architecture, the master serves as a lock server for the workers similar to traditional datacenter com-

puting platforms [4] in many aspects. However, the Maestro framework introduces two significant improvements. Firstly, in Maestro, the locks are assigned subject to a predicate evaluation of the guards of sync actions, and so we call these locks as "predicated locks". The predicated locks reduce the wasted concurrency in traditional transactions and manual locking approaches due to busy polling.

The second improvement Maestro introduces is to automate the locking process. To this end, Maestro first determines async and sync actions by using syntactic analysis, analysis of individual program steps or more complex techniques such as model checking. The automated locking prevents race condition bugs (which may occur due to missed locks in manual locking) and performance penalties (which may occur due to over-application of locks in manual locking).

B. Use of formal methods for determining Sync and Async actions

Maestro utilizes three approaches to determine if an action is an async action. All three approaches are sound: if either declares that an action is an async action then it indeed is. However, the approaches differ in terms of their simplicity and their success in determining that a given action is an async action. To explain these three approaches, let p be a program and let ac be an action of p of the form $g \longrightarrow st$. Note that a stable action permits us to check the guard, g, at some point in the computation and then execute the statement, st, at some later point. To permit this, it suffices to ensure that g is still true when the statement st is executed. Towards this end, we first construct program p' that includes all actions of p except action ac. Now, we need to check that if p' executes in any state where g is true then g continues to be true in the subsequent state. This can be expressed in terms of the Hoare triple $\{g \wedge reachable\} ac' \{g\}$, where ac' is any action of p' and reachable captures the predicate that requires that the state is reachable from some initial state of p.

Approach 1: Syntactic Analysis. For several actions, the verification of the above Hoare triple can be achieved with syntactic checking, i.e., by checking that ac' does not update variables involved in g.

Approach 2: Analysis of One Program Step. In some cases, the Hoare triple can be verified by checking the compatibility of executing action ac' in a state where g is true. Specifically, if the conjunction of g and the guard of ac' is false then it implies that ac' cannot be executed in any state where g is true. Hence, the condition in the above Hoare triple can be easily satisfied. Moreover, to verify the above Hoare triple, it also suffices to verify that $\{g\} ac' \{g\}$, i.e., execution of ac' in any state where g is true results in a state where g is true. This condition can be easily converted to a Boolean formula that can be checked with SAT solvers (e.g., [5]) or theorem provers (e.g., [6]). Observe that this approach partitions the tasks involved in verifying that ac is a locally stable action into several small subtasks that only involve one action at a time. Hence, we expect that the verification for the same should be achieved quickly.

Approach 3: Analysis of several consecutive program steps. As mentioned above, the above approaches are sound but not necessarily complete. In other words, execution of ac' does not prevent ac from being executed at a later point, then that conclusion is valid. However, since we omitted the condition, it is possible that ac' does not conflict with ac although we fail to verify it. In such a case, the designer has two options. Either accept that ac' is a conflicting action and, hence, ac needs to be executed as a synchronous action. Alternatively, the designer can utilize a model checker [7] to verify the above condition. Specifically, here, the input program will be p', i.e., the program without action ac. And, the condition to be verified can be expressed trivially in temporal logics such as LTL or CTL.

C. Cache management

Async actions are very amenable to caching, and can always be served from the worker's cache. While executing an async action, the worker does not need to fetch the most recent copy of the global variables from the master, as they are either stable or locally stable. This enables the workers to execute async actions without contacting the master to lock any variables. However, an async action can still modify some global variables in the statement body, and this may have some consequences for serialization/consistency. For example, if an async action at worker w updates a global variable c, then a sync action at w will have access to the updated version of c whereas another sync action at another worker will use c's old value, leading to a serialization/consistency problem.

Thus after an async action execution, the new values assigned for any global variables need to be synchronized back to the master. When should worker w inform the master about the update of a global variable c by an async action? As long as w does not read c, w does not need to synchronize with the master. When w reads c in a sync action w will get the latest value of c. So it needs to inform the master with the new value of c at this point. In this case we can piggyback the act of informing master about the new value of c to that of contacting the master for the execution of the current sync action (guard evaluation).

D. Optimizations

The master can use several optimizations to improve the scalability. One significant optimization is for the master to partition the data to workers so that workers have more local variables/actions. We give an example of this optimization in our case study in Section IV. Master can also partition tasks (actions) to workers to improve locality of execution hence the performance and scalability of the system. We anticipate

that in some cases instantiating workers with specialized actions is more advantageous to instantiating all workers with identical actions. We plan to investigate these patterns in future work. Other possible optimizations also exist for master-side. The master can give leases (tokens) to transform some sync actions to async actions. For example if a sync action checks whether sufficient resources are available, the master can allocate/lease sufficient resources to that worker in advance, so that the worker does not need to ask anymore and can execute this action as an async action.

A powerful optimization for the worker-side is to use "Hinted evaluation of the guards locally". Since the evaluation of the guard has no side effects, the workers can unilaterally evaluate the guards of even sync actions using cached values as hints. Only when the guard evaluates to true, the worker synchronizes with the master to obtain the latest update for the variables to evaluate the guard. If the guard continues to evaluate to true, the worker begins execution of its statement, else if the guard fails to evaluate to true then no statement is executed. This hinted evaluation of the guard further reduces the query traffic from the workers to the master. This optimization suggests a tradeoff between updating worker with more recent information and evaluation of guards, which we will explore in future work.

To optimize concurrent execution efficiency further, we allow the developer to annotate some variables to provide hints to Maestro. One such tag is "Compatible". A variable tagged as "Compatible" is not locked/synced by the master even if that variable is (read-write) shared between the master and workers.

IV. Case Study: Graph subcoloring

Large-scale graph processing is essential for many services, including social networking sites (Facebook has 500 million users, Twitter 100 million users), search engines (where nodes correspond to URLs and vertices can represent different relationships), semantic web data processing, and even software model-checking with large state. An example of graph processing which needs a high degree of coordination between vertices is distributed graph subcoloring. In graph subcoloring, the aim is to assign colors to a graph's vertices such that each color class induces a vertex disjoint union of cliques. Assigning different colors to each vertex solves the problem vacuously but the challenge is to find a solution using minimal number of colors. This problem is NP-complete and it is hard to find an optimal solution.

To maintain a minimal graph subcoloring, no 2-hop neighbors which do not have a direct connection can have the same color. Figure 2 shows a valid but not minimal graph subcoloring example. Instead of using 3 colors, it is possible to make all the three nodes in the middle red and set the leftmost and rightmost nodes to the same color (either blue or green). Figure 3 shows an invalid graph subcoloring due to the existence of 2-hop neighbors that have the same color.

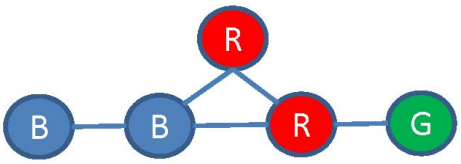

Figure 2. A valid but not minimal graph subcoloring

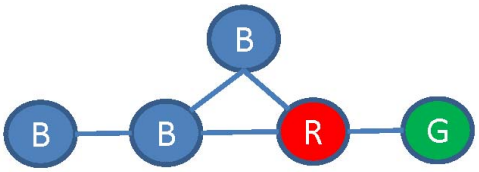

Figure 3. An invalid graph subcoloring

We are given the below three actions in guarded command format as the input program to Maestro. Initially all vertices are assigned a different color. For simplicity, the input program omits a lot of optimizations and employs periodically executing actions.) Action 1 denotes that any vertex i periodically caches the state of any of its neighbor j, and learns the color of j as well as the colors of neighbors of j. Action 2 is also periodically executed. It provides i hints about which color is best for minimizing the number of colors for subcoloring. For each possible color c, if a 2-hop neighbor which is not a neighbor of the initiator vertex also has color c, then this color is not a good candidate and Action 2 marks it as such, else Action 2 increments the safety score of c for i. Since Action 2 is about providing hints in a best-effort manner, it works on cached copies of neighbors' state, and does not attempt to collect the most up-to-date state. Action 3 is the action to finally update the color of i. Action 3 selects the color with the highest score and before adopting this color for vertex i, it checks to make sure that no vertex in the 2-hop neighborhood that is not an immediate neighbor of i has the same color.

$j \in nbr.i \quad \longrightarrow \quad i.copy.j := j.copy.j$

true $\longrightarrow$ for each color c that i knows of:
if $\exists j : j \in nbr.i \wedge i.copy.j.color = c \wedge$
$\exists k : k \in i.copy.j.nbr \wedge k \notin nbr.i \wedge i.copy.j.color.k = c$
then safe.c = -∞
else safe.c ++

$c = max(\forall c : safe.c) \wedge$
$\neg(\exists k : k \in nbr.nbr.i \wedge k \notin nbr.i \wedge color.k = c)$
$\longrightarrow \quad color.i := c$

The Maestro rules classify actions 1 and 2 as async easily as their guards are stable predicates. Action 3 is a sync action

because its predicate is not locally-stable in the most general sense. However, we explain below how Maestro can make Action 3 into an async action by partitioning the graph data to the workers.

In the Maestro setup, after the master partitions the data to the workers, each worker w_i becomes responsible for a neighborhood of vertices. Each neighborhood contains two types of vertices: *middle* vertices have all 2-hop neighbors belonging to the same worker, while *border* vertices may contain 2-hop neighbors outside the current worker boundary. Figure 4 shows an example in which there are two workers and each worker is responsible for a 3-by-3 neighborhood of nodes. Here only the leftmost and rightmost vertices are middle vertices.

While Action 3 is operating on a border vertex b_i of worker w_i, w_i needs to contact the master to learn the latest colors of the 2-hop neighbors of b_i that belong to another worker. In addition, it needs to execute the color update atomically after learning the neighbor's colors. After w_i completes the color update of b_i, it has to inform the master about the new color. As an optimization, w_i may delay this update until another worker contacts with master to learn the color of w_i.

On the other hand, Action 3 operating on a middle vertex m_i of worker w_i does not need to contact the master, because it already has all the color information about the neighbors. In addition, it does not need to execute the color update atomically because it knows that until a color update occurs within this worker, the state (i.e. color) of the neighbors will be preserved.

To contrast, in a basic distributed implementation in which Maestro is not used, the programmer would need to enforce executing the three program actions (or at least Action 3) atomically for all vertices to ensure correct execution. This involves locking the 2-hop neighborhood every time Action 3 is executed, and is very detrimental to the performance of the system.

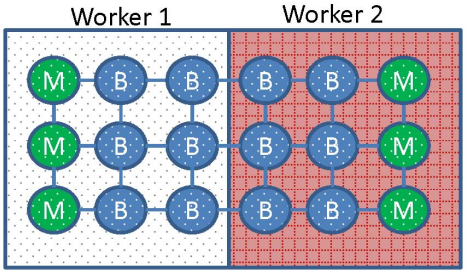

Figure 4. Middle vertices denoted with M and border vertices with B

In the above mesh graph structure, as the number of vertices per worker increases, the ratio of middle vertices also increases. Since Maestro can execute color update on middle vertices asynchronously, we expect the performance gain provided by Maestro to increase as well. (Of course increasing this too much is also a problem because we lose parallelism from having many workers processing the graph.) Next, in the performance evaluation section we present experiments with different graph sizes and graph types (mesh, social, random) to illustrate this effect.

V. System Evaluation

To provide a preliminary measure on the effectiveness of the Maestro framework, we present experiment results for our case study, the graph subcoloring problem.

A. Setup

We ran experiments on Amazon EC2 using a master node and a number of worker nodes. The master node is a medium EC2 instance which consists of 5 EC2 compute units and 1.7 GB memory. The workers are standard micro instances consisting of 2 EC2 compute units and 613 MB memory. In our workloads we did not observe the master ever becoming a bottleneck in a workflow, mainly because the master does not perform heavy operations in the Maestro framework. It just answers the nonlocal color update requests of workers and update its graph coloring map at each color update message received from the workers.

B. Scenarios

We compare Maestro with two other master-worker type distributed computing solutions. These alternative scenarios also employ workers for all subcoloring tasks and use master as a coordinator during the subcoloring process. The main difference lies in the way vertices in the input graph are partitioned between master and workers. This partitioning affects the usage of locks on vertices and the amount of messaging between master and workers.

In the *all-centralized* framework, only the master has access to the graph data. Workers do not hold any graph-related data, instead they send requests to the master for acquiring leases to start the color-update program mentioned in Section IV on a specific vertex. Master checks the availability of the vertex and all of its 2-hop neighbors and gives the lease for that vertex to the worker if all neighborhood is available at that moment. If the lease of any vertex in the neighborhood has already been given to another worker, then the lease request is rejected by master and the worker tries to get a lease for another vertex. If the worker manages to get the lease for the vertex, then it also gets the colors of all the vertices in the 2-hop neighborhood of this vertex. After calculation of the new color for that vertex based on the neighborhood's color map, the worker submits the new color to the master and releases the lease. Workers perform this procedure until colors for all nodes stabilize. This solution avoids the need for 2-hop messages but requires workers to communicate with the master for every vertex.

In the *all-distributed* framework, master keeps no color or neighborhood data about the graph. Instead each worker has a separate portion of the graph and they have direct access to their vertex set in this partition. Master just keeps the mapping of vertices to workers. In this scenario, a worker can perform the color-update program only on its vertex set. If the current vertex is a middle vertex, i.e. all vertices in the 2-hop neighborhood belong to the same worker, the worker does not need to contact the master throughout the procedure. On the other hand, if the vertex is a border vertex, then the worker sends a color request to the master indicating which vertices need to be contacted. Then the master sends a color request message to the workers that have access to the data about these vertices and sends the color replies to the requester worker. Unlike the all-centralized solution, this solution does not have to contact master for middle vertices. On the other hand, when a worker needs data about nonlocal vertices, the retrieval operation is more costly compared to the all-centralized framework.

Maestro combines the advantages of both solutions. In Maestro each worker is responsible for a neighborhood of vertices, so a middle vertex does not need to contact with master since it already has all the color information about the neighbors. In addition, since the master keeps color copies for all vertices in the graph and synchronizes it with the workers as vertex colors are updated, a color request for a nonlocal vertex can be answered directly by the master without further forwarding this request to the worker which has direct access to that vertex.

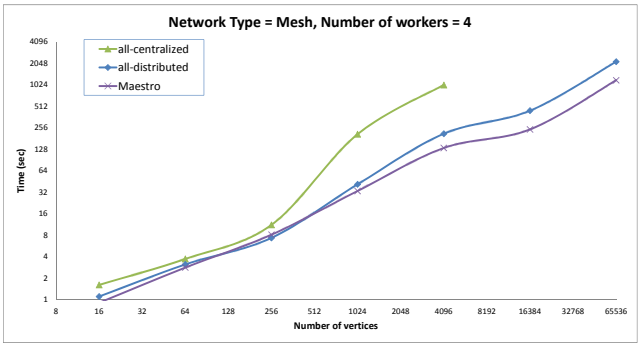

Figure 5. Comparison of frameworks in a mesh graph

C. Results and Evaluation

We compare a basic Maestro implementation with all-distributed and all-centralized solutions over three graph types. Mesh graph is a regular graph in which vertices are organized as in 4. Random graph is formed by incrementally adding a new vertex to the graph at each step and connecting it to two random neighbors. Finally social graph is generated by using the Barabasi-Albert graph model with the parameters given in http://current.cs.ucsb.edu/socialmodels/ to closely follow social network data from Facebook. Assignment of vertices to the workers is done randomly in random and social graphs due to the difficulty of determining disjoint sets of vertices. On the other hand, in mesh network, we partition the graph based on the spatial proximity of the vertices on the mesh. The completion time of the subcoloring algorithm (for all-centralized, all-distributed, and Maestro frameworks alike) is taken as the last vertex color update time in the graph. This is detected by using a very generous timeout for monitoring any color update activity.

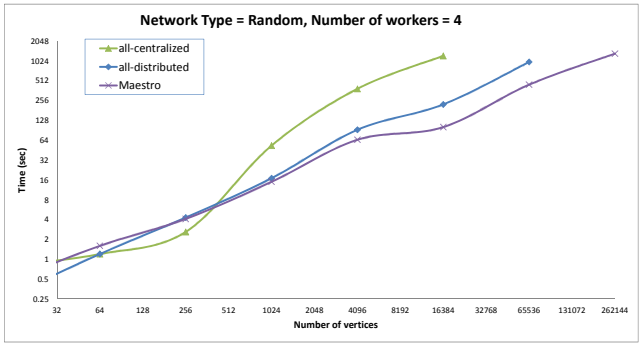

Figure 6. Comparison of frameworks in a random graph

Figure 5 shows the completion time of the subcoloring algorithm for mesh graph for the three different frameworks as the graph size increases. We see that all-centralized framework always exhibits the worst performance. This is caused by the inefficient utilization of workers due to excessive locking of nodes. Despite the small margin, Maestro consistently outperforms the all-distributed solution. This may be because Maestro eliminates extra messages between the master and the workers for nonlocal color requests that are present in the all-distributed solution.

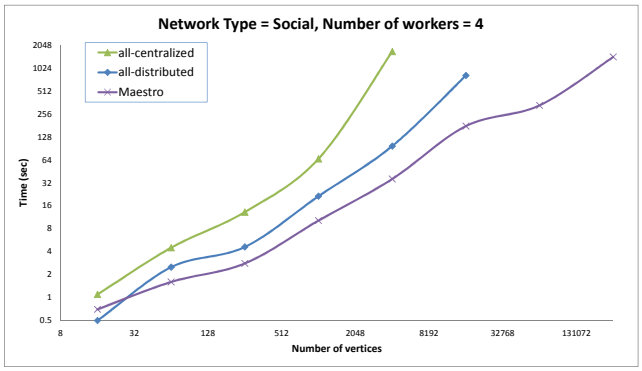

Figure 7. Comparison of frameworks in a social graph

The difference between the frameworks is less obvious in a random graph since the graph structure and assignment of vertices to workers is not identical between different runs

of the subcoloring program. However we still observe that the gap between each method increases as the graph size increases (Figure 6). Maestro maintains the linear relationship between runtime and graph size as the number of vertices increase. (Both x-axis and y-axis are given in log-scale.) This indicates that Maestro is more scalable compared to the other approaches. We also see this linear behavior of Maestro in other graph types (Figure 8).

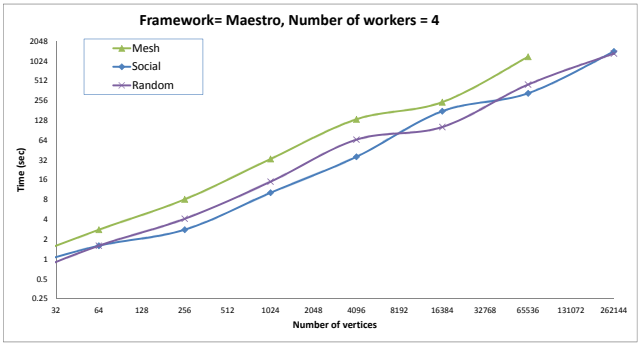

Figure 8. Maestro framework in different graph types

Finally, in Figure 7 we see the performance of Maestro in a social graph. Maestro outperforms other methods, except for the very small graph sizes (which can be explained by the delay due to the initial copying of vertex colors to the master from workers).

VI. Related work

A popular approach for dealing with synchronization requirements is to use transactions. Transactions free developers from worrying about manually controlling concurrent execution, and are therefore desirable [8]. However, since transactional systems require a coordinator to process/moderate all transactions, they are inherently limited in their scalability. In order to guarantee safety and correctness, everything used in a transaction must be locked before evaluating & processing of that transaction, hence transactions are susceptible to the busy polling phenomena we described in the introduction. This significantly decreases concurrency, preventing nodes to access shared variables simultaneously. The scalability of distributed transactions are also limited due to flooding of requests among multiple coordinators, and the control overhead due to the coordinators reaching an agreement.

Distributed lock managers (DLMs) provide concurrent access to shared resources via synchronizing the accesses to these resources by means of locking. A very popular example is the Chubby locking service by Google [3]. Chubby is designed primarily for loosely-coupled distributed systems. To keep the load lighter on the system, Chubby provides coarse-grained locks instead of finer-grained locks. This locking scheme may not always be appropriate for systems that require tight synchronization. Chubby depends on manual locking from the developers and is prone to the disadvantages of locking approaches. Zookeeper [2] is a related centralized coordination service for distributed applications. It provides locking mechanisms similar to Chubby as well as common services such as naming, configuration management, synchronization, and group services for distributed systems. Most Zookeeper applications are loosely synchronized, and does not need tight synchronization. In contrast, Maestro aims to provide high-performance distributed processing for tightly synchronized systems. Moreover, Zookeper requires the developer to insert and manage locks manually, whereas Maestro inserts and manages locks automatically without involving the developer.

MapReduce [1] employs a master-worker architecture that is applicable mostly for embarrassingly parallel applications. Workers do not need to interact with each other during individual job executions and therefore MapReduce does not provide tight synchronization or locking of shared variables.

MapReduce is inefficient for processing graph data because it is a functional language and requires passing the entire state of the graph from one stage to the next. As a result, several graph processing frameworks have been developed recently to more efficiently process large graphs. Pregel framework from Google is a well-known example [9]. Pregel programming model asks developers to write code from the point-of-view of a vertex in the graph. Pregel employs the Bulk Synchronous Parallelism (BSP) approach [10] and hence requires system-wide synchronization steps. Pregel inspired several open source projects, including Golden Orb, Apache Giraph, and Apache Hama. These projects also use the BSP model, and their basic architectures and programming model is similar to Pregel. GraphLab and Signal-Collect projects also develop graph processing frames for processing large scale graphs. Maestro does not need system-wide synchronization as in the BSP approach. Thus, Maestro's concurrency is not hampered, and Maestro's execution model is more flexible.

VII. Discussion & Future work

One risk with Maestro maybe that of developers showing reluctance to adopt Maestro's guarded-command input language. To prevent or alleviate this risk, we plan to develop GUI-based Integrated Development Environment (IDE) and relate our input language to event-driven programming which the developers are more familiar with.

Another risk can be the master becoming a communication/computation performance bottleneck. In the Maestro framework, there is a single master, but the master is not contacted for every decision and acts only as a lightweight lock manager for sync actions. Experience with similar cloud computing frameworks, such as the Google File System (GFS) [4] and Zookeeper [2], shows that in

practice the single master does not constitute a communication/computation bottleneck if it is not contacted for every operation/heavyweight-operations. To alleviate this risk, Maestro deployments will ensure that sufficient network bandwidth and computational resources is available at the node where the master is hosted. Furthermore, Maestro will employ Paxos [11] replication of the master to mask master failures.

We will build applications on top of Maestro to showcase Maestro's benefits for cloud computing application developers as well as benchmarking Maestro's performance and comparing to related tools. In future work we will build social network graph processing and real-time enterprise analytics applications using the Maestro framework and compare Maestro's performance against transactional and manual-locking solutions.

Since Maestro performs compilation of high-level code to the datacenter computing level, we have a high-degree of flexibility in this compilation and there are many opportunities to optimize and customize Maestro. As we mentioned before, developing smart data/task partitioning at the master to the workers is a very promising venue for performance optimization. Another opportunity is to develop smart leasing (resource allocation) at the master to the workers. We will investigate ways to customize Maestro for target application environments as our project progresses.

REFERENCES

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *OSDI*, p. 13, 2004.

[2] P. Hunt, M. Konar, F. Junqueira, and B. Reed, "Zookeeper: wait-free coordination for internet-scale systems," *USENIX technical conference*, 2010.

[3] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *OSDI*, 2006, pp. 335–350.

[4] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," *SIGOPS Oper. Syst. Rev.*, vol. 37, pp. 29–43, 2003.

[5] B. Dutertre and L. de Moura, "The Yices SMT solver," SRI International, 2006.

[6] S. Owre, J. Rushby, N. Shankar, and F. von Henke, "Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS," *IEEE Transactions on Software Engineering*, vol. 21, no. 2, pp. 107–125, Feb. 1995.

[7] G. Holzmann, *The Spin Model Checker, Primer and Reference Manual*. Reading, Massachusetts: Addison-Wesley, 2003.

[8] J. Gray and A. Reuter, *Transaction Processing : Concepts and Techniques*. Morgan Kaufmann Publishers, 1993.

[9] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," *SIGMOD*, pp. 135–146, 2010.

[10] L. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[11] L. Lamport, "Paxos made simple," *ACM SIGACT News*, vol. 32, no. 4, pp. 18–25, 2001.