SYNC or ASYNC: Time to Fuse for Distributed Graph-Parallel Computation

Chenning Xie† Rong Chen† Haibing Guan§ Binyu Zang† Haibo Chen†

Shanghai Key Laboratory of Scalable Computing and Systems
†Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
§Department of Computer Science, Shanghai Jiao Tong University
{xiechenny, rongchen, hbguan, byzang, haibochen}@sjtu.edu.cn

Abstract

Large-scale graph-structured computation usually exhibits iterative and convergence-oriented computing nature, where input data is computed iteratively until a convergence condition is reached. Such features have led to the development of two different computation modes for graph-structured programs, namely synchronous (*Sync*) and asynchronous (*Async*) modes. Unfortunately, there is currently no in-depth study on their execution properties and thus programmers have to manually choose a mode, either requiring a deep understanding of underlying graph engines, or suffering from suboptimal performance.

This paper makes the first comprehensive characterization on the performance of the two modes on a set of typical graph-parallel applications. Our study shows that the performance of the two modes varies significantly with different graph algorithms, partitioning methods, execution stages, input graphs and cluster scales, and no single mode consistently outperforms the other. To this end, this paper proposes *Hsync*, a hybrid graph computation mode that adaptively switches a graph-parallel program between the two modes for optimal performance. *Hsync* constantly collects execution statistics on-the-fly and leverages a set of heuristics to predict future performance and determine when a mode switch could be profitable. We have built online sampling and offline profiling approaches combined with a set of heuristics to accurately predicting future performance in the two modes. A prototype called PowerSwitch has been built based on PowerGraph, a state-of-the-art distributed graph-parallel system, to support adaptive execution of graph algorithms. On a 48-node EC2-like cluster, PowerSwitch consistently outperforms the best of both modes, with a speedup ranging from 9% to 73% due to timely switch between two modes.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Distributed programming

Keywords Distributed Graph-parallel Computation; Computation Modes

PPoPP'15, February 7–11, 2015, San Francisco, CA, USA.
Copyright © 2015 ACM 978-1-4503-3205-7/15/02…$15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

1. Introduction

Large-scale graph-structured computation has emerged to address a number of machine learning and data mining (MLDM) problems in a wide range of areas, including social computation, web search, natural language processing and recommendation systems [7, 19, 29, 34, 40, 44, 47]. As the concept of "Big Data" gains increasing momentum, it is now common to run parallel and distributed MLDM algorithms on a cluster of machines to accommodate the increasing data size and problem complexity. This has driven the design and development of many graph-structured computation systems, including Pregel [31] and its open-source alternatives [2, 3, 37], GraphLab [18, 30], Cyclops [9] and GraphX [20].

Many graph computation systems usually take the "think as a vertex" philosophy [31] by coding graph computation as vertex-centric programs to process vertices in parallel and communicate along edges. Typically, many MLDM problems usually exhibit iterative computation nature by iteratively computing (e.g., refining) input data until a convergence condition has been reached. Such iterative and convergence-oriented computation have driven the development of two execution modes: 1) synchronous (*Sync*), which synchronously computes over a set of active vertices in each iteration (i.e., super-step) and propagates the updates to other vertices at the end of each iteration in a batch. A vertex can only see updates from its neighboring vertices after the end of the iteration; 2) asynchronous (*Async*), which provides no explicit synchronization points but allows the state of a vertex to be visible to its neighboring vertices as soon as possible.

Prior theoretical analysis on graph algorithms has shown that many graph algorithms can be executed both synchronously and asynchronously [6, 41]. Hence, many state-of-the-art graph-parallel systems, such as PowerGraph [18], Trinity [38], GRACE [42] and PowerLyra [10] have been built with support for both execution modes by separating computation logic and scheduling orders. Hence, the same graph algorithm can run in both *Sync* and *Async* modes. This flexibility, however, also forces programmers to either blindly select a mode for execution or experience a long learning curve to understand the internals of underlying graph engines. Further, for most graph algorithms, different stages in a single execution demand different modes to achieve optimal performance. This leads to either unwanted complexity to users, or suboptimal performance, or both. Worse even, choosing a wrong mode may cause a graph algorithm to run infinitely without being converged.

This paper presents the first comprehensive study on the execution properties of the two modes, by using a state-of-the-art graph-parallel system (i.e., PowerGraph [18]) supporting both modes. Unlike conventional wisdom that *Async* mode generally has superior performance [6, 30] than *Sync* mode, our study shows that the two modes exhibit different properties such as communication, convergence and resource utilization, which lead to different performance not only *across* various graph algorithms but also *within* different execution stages of the same graph algorithm. For example, *Sync* mode can group messages together to reduce communication cost and favors I/O-bound algorithms, while *Async* mode may converge faster and favors CPU-bound algorithms. As a concrete example, PageRank [7] performs much better in *Sync* mode, while LBP (Loopy Belief Propagation) [19] performs notably better in *Async* mode and Graph Coloring [17] cannot even converge under synchronous execution. More interestingly, *Async* mode performs very good in the beginning and the end of SSSP (Single-Source Shortest Path) [5], but *Sync* mode has superior performance during the middle of execution, due to the effect of execution behavior on convergence speed, computation and communication load in different execution stages. Finally, configurations like input data size, scale of clusters and graph partitioning approaches all impact the efficiency of the two modes. Based on our study, this paper provides a general guideline on the properties of the two modes[1].

To gain optimal performance while insulating programmers from tedious low-level details, this paper further proposes *Hsync*, a hybrid graph execution mode that adaptively switches execution between *Sync* and *Async* modes. PowerSwitch constantly collects execution statistics like throughput, active vertices and convergence speed, and leverages online sampling, offline profiling and a set of heuristics to accurately predict optimal mode switch points.

PowerSwitch is built with an efficient approach to allowing fast and seamless mode switches, while preserving the consistency of graph states. The key of PowerSwitch is allowing an efficient sharing of states between the two modes, enforcing a safe point for consistent switch states and providing automatic mode switch at the appropriate time.

We have implemented a full-fledged version of *Hsync* based on the state-of-the-art PowerGraph [18] framework. Our system, called PowerSwitch[2], is implemented as a separate engine and thus fully retains compatibility with existing applications in PowerGraph. A comprehensive performance evaluation using four typical graph algorithms, namely PageRank, SSSP, LBP and Graph Coloring, shows that PowerSwitch can accurately predict future execution performance and make a near-optimal decision of mode switches. The efficient mode switches and highly accurate prediction not only makes PowerSwitch enjoy the best of the two execution modes, but also leads to notable performance boost over the best performance of *Sync* and *Async* modes, ranging from 9% to 73% (from 9% to 123% over *Sync* and from 30% to 183% over *Async*). The mode switch time is also small (around 0.1s and 0.6s from *Sync* to *Async* and vice versa) even for a graph with several millions of vertices and billions of edges.

This paper makes the following contributions:

- The first comprehensive study on the performance characteristics of *Sync* and *Async* modes on different graph algorithms, partitioning methods, execution stages, graph and machine scales (Section 2).
- The *Hsync* graph computation mode that enjoys the best of both worlds of the two modes, by adaptive switching between *Sync* and *Async* modes (Section 3 and 4).
- An algorithm to determine which mode is efficient combined with optimized online sampling, offline profiling and a set of heuristics (Section 5).
- A full-fledged implementation based on PowerGraph and a thorough evaluation that demonstrates the performance gain of PowerSwitch (Section 7).

[1] We choose PowerGraph because it is a best-known distributed graph-parallel system that has already been commercialized.

[2] The source code and a brief instruction of how to use PowerSwitch are at `http://ipads.se.sjtu.edu.cn/projects/powerswitch.html`

Table 1: *A comparison between Sync and Async mode*

	Sync	*Async*
Properties		
Communication	Regular	Irregular
Convergence	Slow	Fast
Favorites		
Algorithm	I/O-intensive	CPU-intensive
Execution Stage	High Workload	Low Workload
Scalability	Graph Size	Cluster Size

2. Performance Characterization of Different Execution Modes

Many graph-parallel systems like Pregel [31], GraphLab [30] and PowerGraph [18] follow the "think as a vertex" philosophy and model a graph algorithm as a vertex-centric program. Specifically, a vertex program $\mathbb{P}$ runs on a(n) (un)directed graph $G = \{\mathbb{V}, \mathbb{E}, \mathbb{D}\}$ and computes in parallel on each vertex $v \in \mathbb{V}$. Users can associate arbitrary vertex data D_v where $v \in \mathbb{V}$, and edge data $D_{s,t}$ where $(s, t) \in \mathbb{E}$. The graph execution engine controls the scheduling of computation on vertices, either synchronously (*Sync*) or asynchronously (*Async*), through making updates of D_v or $D_{s,t}$ visible to other vertices at different times and activating the vertices to do the computation.

This section briefly introduces the two execution modes combining with their different emphases in implementation, and then characterizes the execution properties resulting from these emphases using a state-of-the-art graph engine (i.e., PowerGraph [18]). As the graph algorithm as well as the source code are the same for both modes, it is possible to perform an "apple-to-apple" comparative study. Base on the study, this section summarizes the impact of different modes on different graph algorithms, machine scales, graph properties, execution stages, and convergence speed.

2.1 Execution Modes of Graph-parallel Systems

State-of-the-art graph-parallel systems usually separate computation logic from scheduling order. Hence, the major difference between *Sync* and *Async* modes is the scheduling order of vertex computation, which provides different visibility timing of updated values for subsequent vertex computation. These differences in visibility and dependency result in various technical choices for the efficient implementation of the scheduling. Hence, different execution modes may have different execution properties, as summarized in Table 1 according to our study.

Algorithm 1: Synchronous Mode

```
Input: Data Graph $G = (\mathbb{V}, \mathbb{E}, \mathbb{D})$
Input: Initial active vertex set $\mathbb{V}_a$

while $iteration \leq max\_iteration$ do
    if $\mathbb{V}_a == \emptyset$ then break

    $\mathbb{V}'_a \leftarrow \emptyset$
    foreach $v \in \mathbb{V}_a$ do
        $\mathbb{A} \leftarrow$ compute($v$)
        $\mathbb{V}'_a \leftarrow \mathbb{V}'_a \cup \mathbb{A}$

    $\mathbb{V}_a \leftarrow \mathbb{V}'_a$
    $iteration$ ++
```

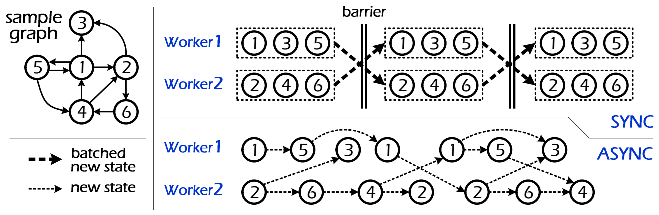


Figure 1: *A comparison of execution flows between different modes.*

Synchronous mode (***Sync***), presented in Algorithm 1, abstracts a graph algorithm as a sequence of iterations (i.e., super-step), in which all active vertices in $\mathbb{V}_a$ execute vertex programs (`compute`) in parallel using the values of neighboring vertices updated in the previous iteration. Activated vertices ($\mathbb{A}$) are saved for the computation in the next iteration. The communication for updating vertex data between workers employs batched message passing, which is similar to the *Bulk Synchronous Parallel* (BSP) [41] model.

The upper part of Figure 1 illustrates the execution flow of *Sync* mode. All vertices in the sample graph are executed in a fixed order within each iteration. A global barrier between consecutive iterations ensures that all vertex updates in current iteration are simultaneously visible in the next iteration for all workers. This mode prefers to handle larger graphs under limited computing resources. The delay of visibility and synchronized stages make it possible for batched data update and well-optimized network message dispatching with high resource utilization.

Table 1 illustrates the major properties and favorite scenarios of *Sync* mode. First, since the batched communication messages make network bandwidth better utilized, *Sync* mode favors I/O-intensive algorithms (e.g., *PageRank* [7]) in which the computation on each vertex is lightweight. Second, iterative computation converges asymmetrically in many graph algorithms, which implies that a large number of vertices will rapidly converge in a few iterations, while the remaining vertices will converge slowly over many iterations [15]. Furthermore, the number of active vertices may vary during execution. For example, the number of active vertices in *Single-Source Shortest Path* (SSSP) [5] increases and then decreases along its execution. *Sync* mode prefers execution stages at which there are a large number of active vertices and provides better scalability with the increase of graph size. This is because the overhead in each iteration caused by the global barrier can be largely amortized. Finally, *Sync* mode is not suitable for graph algorithms requiring coordination of adjacent vertices. For example, *Graph Coloring* [17] aims at assigning different colors to adjacent vertices using a minimal number of colors. In a greedy implementation, all vertices simultaneously pick minimum colors not used by any of their adjacent vertices. The greedy algorithm for graph coloring may not converge in *Sync* mode, since adjacent vertices with the same color will simultaneously pick the same colors back and forth according to the same previous color.

Algorithm 2: Asynchronous Mode

```
Input: Data Graph $G = (\mathbb{V}, \mathbb{E}, \mathbb{D})$
Input: Initial active vertex set $\mathbb{V}_a$
while $\mathbb{V}_a \mathrel{!=} \emptyset$ do
    $v \leftarrow$ dequeue($\mathbb{V}_a$)
    $\mathbb{A} \leftarrow$ compute($v$)
    $\mathbb{V}_a \leftarrow \mathbb{V}_a \cup \mathbb{A}$
```

In **asynchronous** mode (***Async***), the computation on a vertex is scheduled on the fly, and uses the new state of neighboring vertices immediately without a global barrier. Algorithm 2 shows the semantics of *Async* mode engine, which dequeues an active vertex v from scheduling queue $\mathbb{V}_a$ and runs vertex computation on it. Newly activated vertices $\mathbb{A}$ are enqueued to $\mathbb{V}_a$.

The lower part of Figure 1 illustrates the execution flow of *Async* mode. Compared to *Sync*, there is no global barrier to synchronize vertex execution on workers, and the update on vertex is visible to neighboring vertices as soon as possible. *Async* mode is designed for timely visibility of update, and emphasizes the fast convergence speed under sufficient hardware resources. In addition, *Async* mode could employ pipeline of vertex processing to hide the network latency. However, since the message communication in *Async* mode happens at any time between different machines, it is difficult to batch enough messages to amortize network cost and any delay of batched messages will hurt the timely visibility of updates. Worse even, the mixed read and write for vertex data also require to maintain the atomic vertex data update, which causes significant scheduling overhead.

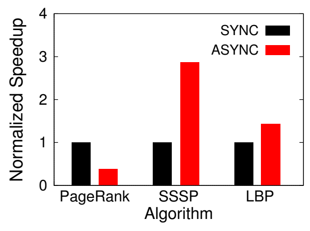
Table 1 also illustrates the major properties and favorite scenarios of *Async* mode. First, *Async* mode can accelerate the convergence of program [16]. It prefers CPU-intensive algorithms (e.g., *Loopy Belief Propagation* (LBP) [19]), in which the piped vertex computation can fully hide communication cost in a lack of message batching. Second, overhead of execution in *Async* systems is mainly from the lock contention on a vertex during vertex computation, which depends on the number and degree of active vertices. The increase of active vertices with their edges on multiple machines also results in a heavy contention of network resources. Since the communication in *Async* mode could happen at any time, it is difficult to make a full utilization of network resources. Therefore, *Async* mode has better performance on the stage of execution with less amount and fewer degree of active vertices than those of *Sync* mode, and provides better scalability with the increase of machines. Finally, some graph algorithms like Graph Coloring and Clustering based on Gibbs Sampling [30], may only converge in *Async* mode.

2.2 Performance of the Two Modes

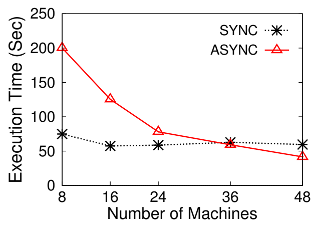
Since *Sync* and *Async* modes have different favorite scenarios, using a single mode can hardly achieve optimal performance for different scenarios. In the following, we will use PowerGraph [18], a well-known distributed graph-parallel framework that provides both *Sync* and *Async* modes, to illustrate performance variation of the two modes with typical algorithms and configurations.

Graph Algorithms: Since *Sync* mode has more efficient communication but slower convergence than that of *Async* mode, the performance of various **algorithms** on different modes is hard to predict. In Figure 2(a), we evaluate three graph algorithms (PageRank, SSSP and LBP) on 48 machines using *Sync* and *Async* modes. All performance results are normalized to the *Sync* mode. *Sync* mode outperforms *Async* mode by 2.60X for PageRank with Twitter Follower graph [26]. In contrast, *Async* outperforms *Sync* by 2.86X and 1.43X for SSSP with RoadCA [28] and LBP with 3-million pixels [30] respectively. Hence, using an inappropriate execution mode may result in a significant performance loss.

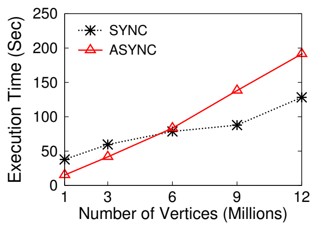
Graph and Machine Scales: Even for the same graph algorithm, different **configurations** can also lead to different choices of execution modes. As shown in Figure 2(b), the execution time of *Async* mode for LBP algorithm on a 3-million vertex graph rapidly decreases with the increase of machines. The convergence point is on 36 machines, and *Async* outperforms *Sync* by 1.43X on 48 machines. In contrast, the increase of execution time using *Sync* mode for LBP on 48 machines is obviously slower than *Async* mode with the increase of graph size, and the inflection point is on the 6-million pixels graph (Figure 2(c)). In Figure 2(d), we evaluate the performance of the LBP algorithm on a 3-million pixels graph us-



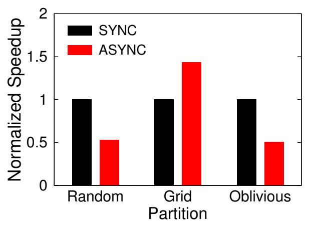
(a) Algorithms



(b) Number of Machines (LBP)

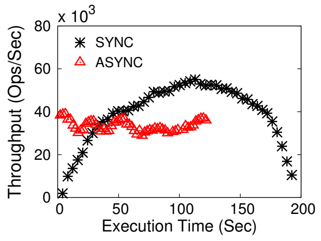


(c) Size of Graph (LBP)

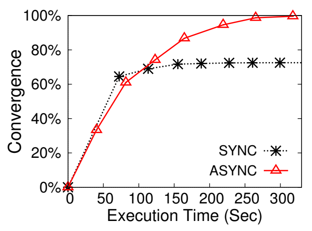


(d) Partitioning Alogrithms (LBP)

Figure 2: *A comparison of performance between Sync and Async mode. (a) with different algorithms. (b) with the increase of machines. (c) with the increase of graph size. (d) with various graph partitioning algorithms.*



(a) SSSP



(b) Graph Coloring

Figure 3: *(a) Throughput of Sync and Async modes for SSSP on 12 machines. (b) The percent of active vertices for Graph Coloring on 48 machines using Sync and Async modes.*

ing various graph partitioning algorithms. The replication factor is 4.79, 2.18, and 4.04 using Random, Grid [25] and Oblivious [18] partitioning accordingly. All performance results are normalized to the *Sync* mode. *Sync* outperforms *Async* by 1.89X and 1.96X using Random and Oblivious partitioning respectively. In contrast, *Async* outperforms *Sync* by 1.43X using Grid partitioning. Hence, for same algorithm, such as the CPU-intensive algorithm LBP, it is still difficult for the user to judge whether the algorithm of vertex computation is complicated enough under certain configuration to make *Async* mode outperform *Sync* mode.

Execution Stages: Even with the same algorithm and configuration, using only a single-mode may only achieve suboptimal performance, because different **execution stages** of a single algorithm also require different modes. Figure 3(a) shows the normalized throughput of *Sync* and *Async* for SSSP using the RoadCA graph on 12 machines[3]. In SSSP, only the source nodes are active initially, and the number of active vertices rapidly increases with the propagation of shortest distance. When most vertices have the shortest distance, the number of active vertices would decrease to none. Note that the vertex computation throughput is normalized with the speed of convergence of *Async* and *Sync* modes for SSSP. *Async* mode has relatively higher throughput in the initial and final stages, while *Sync* mode is with superior performance in the middle stage.

Convergence Speed: Some algorithms hardly converge in *Sync* mode due to conflicts in vertex computation and can only converge in *Async* mode (e.g., Graph Coloring). However, *Async* mode may result in sub-optimal performance due to slow computation. In contrast, *Sync* mode can accelerate convergence in certain stages but cannot finally converge afterward. Such a conflict between efficiency and convergence makes it hard or impossible to embrace both under a single mode. Figure 3(b) illustrates a comparison of the progress of *Sync* and *Async* modes for Graph Coloring algorithm [17] with the Twitter Follow graph [26].

In short, under a combined effect of varying degrees from various factors (e.g. algorithm design, configuration, properties of different execution stages), the execution mode can be improved or it might not even be known without executing the program.

[3] We only use 12 machines as SSSP is essentially not scalable with the increasing number of machines.

Hence, our system, PowerSwitch, attempts to gain optimal performance by employing right mode of different execution stages, with the following challenges:

- **Correctness in Semantics:** When combining *Sync* mode with *Async*, the dependency of data update changes between different scheduling. The consistency level should be illustrated to define supported algorithms and ensure the correctness.
- **Efficient Switch:** The switch between different modes involves different data structures in the scheduling implementation and the maintenance of data consistency. Besides, the efficient state conversion with low overhead is necessary to get enough benefit from a better mode.
- **Accurate Switch:** The uncertainty of the better mode without actual execution has been presented above. With the sampling and statistics during program execution, some quantified strategy should be defined to compare the efficiency of the two modes.

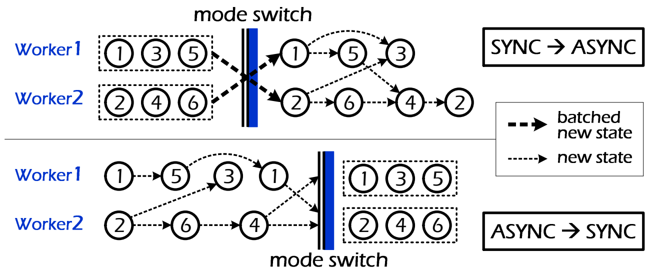


Figure 4: *An example of execution flows on Hsync mode.*

3. Hybrid-synchronous Execution

This section introduces *Hsync*, a hybrid-synchronous execution mode, which embraces the best of both worlds in *Sync* and *Async* modes by adaptively switching graph computation between the two modes. Hence, *Hsync* insulates users from being concerned with the underlying execution engines.

3.1 Graph-parallel Abstraction and Mode

Hybrid-synchronous (*Hsync*) mode shares the same graph abstraction as *Sync* and *Async* modes of graph-parallel systems, including graph organization and programming interface, as well as graph computation logic. The major difference lies in how *Hsync* schedules vertex computation and when the updated vertex and edge data will be visible for the following computation. Like *Sync* and *Async* modes, the data graph is partitioned to multiple nodes, and vertices are replicated to provide local cache for shared memory access in computation. *Hsync* uses the "think as a vertex" philosophy to provide vertex-centric interfaces and runs a user-defined vertex-program P on a sequence of vertices in loop, which updates the

Algorithm 3: Hybrid-synchronous Mode

```
Input: data graph $G = (\mathbb{V}, \mathbb{E}, \mathbb{D})$
Input: initial active vertex set $\mathbb{V}_a$
Input: parameters for predicting $O$

mode ← init_mode($G$, $\mathbb{V}$, $O$)
while $\mathbb{V}_a$ != $\emptyset$ do
    if is_sync(mode) then
        sync($\mathbb{V}_a$)
        mode ← eval(mode)
        if is_async(mode) then
            mode_switch($\mathbb{V}_a$, ASYNC)
    else
        async($\mathbb{V}_a$)
        mode ← eval(mode)
        if is_sync(mode) then
            mode_switch($\mathbb{V}_a$, SYNC)
```

state of vertices through interaction with neighboring vertices. Unlike *Sync* and *Async* modes, *Hsync* splits a sequence of executions into multiple time intervals (epochs) and periodically evaluates the potential benefit from a mode switch in the next epoch. The speed of the mode in use is monitored, while that of future modes is predicted. Within each epoch, *Hsync* adopts *Sync* or *Async* mode to schedule and update vertices.

Figure 4 illustrates two sample execution flows on *Hsync* mode. The upper part starts from *Sync* mode and then switches to *Async* mode, and the lower part starts from *Async* mode and then switches back to *Sync* mode. It is important to note that the mode switch is transparent to a graph algorithm and may happen multiple times during a single execution.

3.2 Execution of *Hsync* Mode

The execution of *Hsync* mode follows the semantics in Algorithm 3. According to some initial information, *Hsync* engine first predicts a suitable initial mode to start vertex computation (line 1). The initial information contains input graph properties, initial state of graph algorithm and parameters for predicting. While there are active vertices remaining in $\mathbb{V}_a$, *Hsync* executes an epoch in synchronous (line 4) or asynchronous (line 9) semantics. The length of an epoch is irregular and depends on the condition to switch of the two modes. *Hsync* forms an epoch from a few iterations in *Sync* mode, and several seconds in *Async* mode. The epoch-based synchronous and asynchronous execution is similar to the original semantics in Algorithm 1 and Algorithm 2, except the loop condition. *Hsync* re-evaluates the mode for the next epoch at the end of current epoch, and initiates a mode switch if necessary.

3.3 Semantics of *Hsync*

Even though *Hsync* switches execution modes between *Sync* and *Async*, it still maintains the same consistency level with the vertex consistency [18, 30] in asynchronous graph-parallel execution, which supports vertex-level atomic update. Though there are also other more strict consistency modes like edge consistency and full consistency, vertex consistency is the most generally used consistency level of *Async* mode with a faster convergence speed. For example, though GraphLab [30] raises the notion of edge consistency and full consistency, the full consistency is never supported and edge consistency is rarely used; almost all graph algorithms use vertex consistency in GraphLab.

Here, we informally show that *Hsync* retains the vertex consistency by mapping the synchronous scheduling order to a special case in asynchronous scheduling. First, we assume there is a partition that divides all adjacent vertices of the input graph onto different machines. This is possible if there are more machines than neighboring vertices. At the beginning, there will be a few vertices that have been activated. In *Async* mode with vertex consistency, if the updates of current active vertices are broadcasted with delay longer than the total computation time of all the local active vertices, no latest vertex data for neighbor vertices will be observed. In this case, the local computation will continue execution with a consistent but stale data of its neighbors, which is equivalent to an iteration in *Sync* mode. Hence, *Sync* mode can be regarded as a special case of *Async* mode in vertex consistency, and the hybrid scheduling of the two modes could ensure at least the same semantic as *Async* mode.

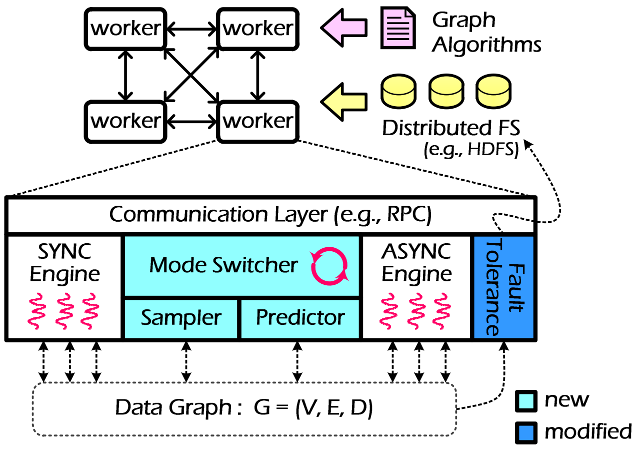


Figure 5: *The system architecture of PowerSwitch*

4. Supporting Mode Switch

The key to *Hsync*'s performance is supporting fast and consistent switches between two execution modes. This section describes the graph execution engine to accommodate the two modes simultaneously and to support efficient mode switches.

4.1 Architecture

Figure 5 illustrates a high-level overview of PowerSwitch. PowerSwitch runs two execution modes in a single graph engine and allows adaptive switches between the two modes. Before running a graph algorithm, the performance *sampler* collects some execution statistics of a new graph algorithm on PowerSwitch by running an algorithm with very small input for several seconds. Such statistics will be stored into a knowledge base in PowerSwitch for later use by the performance *predictor* (Section 5). During graph execution, the predictor estimates the key performance metrics of the current mode, and predicts the performance of algorithms running on *Sync* and *Async* modes periodically (Section 5). When a mode switch is predicted to be profitable, the *mode switcher* will perform dynamic switches between *Sync* and *Async* modes.

4.2 Mode Switch

The key in switching execution modes is ensuring consistency of graph states. There are two execution modes coexisting in a single graph engine and different execution modes have their own states, including graph structures, scheduling queues and intermediate messages. A naive approach would be maintaining two separate copies of state for different execution modes and performing a state transformation between the two copies during a mode switch. However, such an approach will be extremely slow and complex, due to the amount of state to be transformed and the difficulty in writing the state transforming functions.

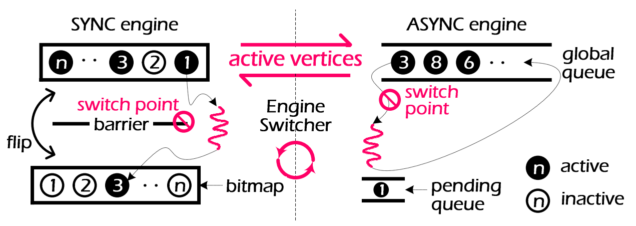


Figure 6: *The conversion between Sync and Async engines*

PowerSwitch takes a hybrid design by sharing as much state as possible between the two modes and only transforming the portion of the state that are different. Specifically, the graph structure state, including vertices, edges and their values, is shared between two execution modes. PowerSwitch only needs to ensure that such state is consistent after a mode switch such that a new mode can directly continue execution upon the previous execution mode's graph state. PowerSwitch could also have used a shared schedule queue for two execution modes to accelerate mode switch speed. However, scheduling for *Sync* mode is bulk-synchronous and that for *Async* requires discriminated priorities, and thus the essential differences of scheduling in two modes impede efficiently sharing of the schedule queue.

As shown in Figure 6, to allow consistent transformation of schedule queues, PowerSwitch uses two bitmaps in *Sync* mode to identify active vertices in current and next iterations accordingly. The bitmaps are updated when a message activates a vertex. After an iteration, all messages have been processed and two bitmaps in each worker are flipped. For *Async* mode, a global priority (e.g., FIFO) queue is used to schedule active vertices. Each worker thread has a local pending queue to save the stalled active vertices, which may wait for the response of messages from neighboring vertices.

To guarantee consistency during a mode switch, *Hsync* ensures that the graph state is consistent and no pending vertex computation and messages are in progress. During a switch from *Sync* to *Async* mode, all active vertices in the bitmap for the next iteration are imported to a global queue of *Async* mode. During a switch from *Async* to *Sync* mode, PowerSwitch enforces a safe point by prohibiting all worker threads from fetching new vertices from the global queue and then starts importing all active vertices in the global queue to the bitmap of current iteration in *Sync* engine until all computation on pending active vertices has been done.

4.3 Fault Tolerance

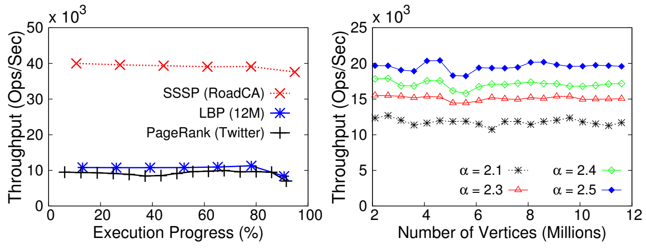
Both *Sync* and *Async* modes have their own fault tolerance support using distributed checkpointing. However, *Sync* mode generates *synchronous* snapshots during global barriers, but *Async* mode generates incremental *asynchronous* snapshots based on the Chandy-Lamport snapshot algorithm [8] at fixed intervals. *Hsync* can follow the same checkpointing and snapshot mechanisms to support fault tolerance in the two modes. To ensure the integrity of asynchronous snapshot, *Hsync* can generate an extra synchronous snapshot during a mode switch from *Async* to *Sync*.

5. Switch Timing

The key to gain optimal performance on PowerSwitch is deciding the right *timing* of a mode switch. This section first describes the ultimate metrics to characterize the performance in different modes, and then illustrates how to predict such metrics in both *Sync* and *Async* modes. Finally, the accuracy of prediction is validated by comparing the predicted and optimal switch points.

5.1 Performance Metrics

There are a number of factors that may impact the performance of *Sync* and *Async* modes, including the amount of messages, number of active vertices, changing rate of active vertices, application properties, hardware environments and the specific options such as the policy of scheduler used in *Async* mode. It is not easy to balance these factors to derive an optimal model to characterize the performance. Fortunately, we observe that all these factors can be reflected as the **Throughput** ($Thro$) of graph processing, which is the amount of vertices processed per unit time:



(a) Algorithms (Async) *(b) Training Graphs (Async)*

Figure 7: *Throughput of Async mode on 48 machines for (a) various algorithms, (b) PageRank with various training graphs. α means power-law constant.*

$$Thro = \frac{|V_{proc}|}{T}$$

However, *Sync* and *Async* modes have different convergence speed, which results in different amount of vertex computation to reach convergence for the same graph algorithm. Consequently, it is not viable to use raw throughput to directly compare the performance of *Sync* and *Async* modes. The convergence ratio μ, which is the total amount of vertex computation in *Async* mode to that of *Sync* mode, should be considered to normalize the throughput of different modes. Note that the ratio is mainly determined by graph algorithms and the properties of input graphs, but with only slight effect from the size of input graphs and hardware configurations. Therefore, it can be either set by experience or offline profiling results on application with a small input. For example, the convergence ratios μ for PageRank, SSSP and LBP with typical input graphs are 0.8, 0.5 and 0.8 respectively, which are used to normalize the throughput of *Sync* mode for all experiments.

5.2 Predicting *Throughput* of Current Mode

***Sync* Mode**: there is a constant overhead from the global barrier in *Sync* scheduling, which should be amortized over all active vertices. PowerSwitch uses Equation 1 to model the throughput of next iteration in current mode:

$$\begin{aligned} Thro_{next} &= \frac{|V_{next}|}{T_{comp} + T_{barrier}} \\ &= \frac{1}{T_{vert_{next}} + T_{barrier}/|V_{next}|} \end{aligned} \tag{1}$$

where $T_{barrier}$ denotes the constant overhead, and can be measured online. $|V_{next}|$ denotes the number of active vertices in next iteration, which is collected at the end of current iteration. T_{vert} denotes the average time to process a single vertex, which is estimated as weighted average of current and history (Equation 2).

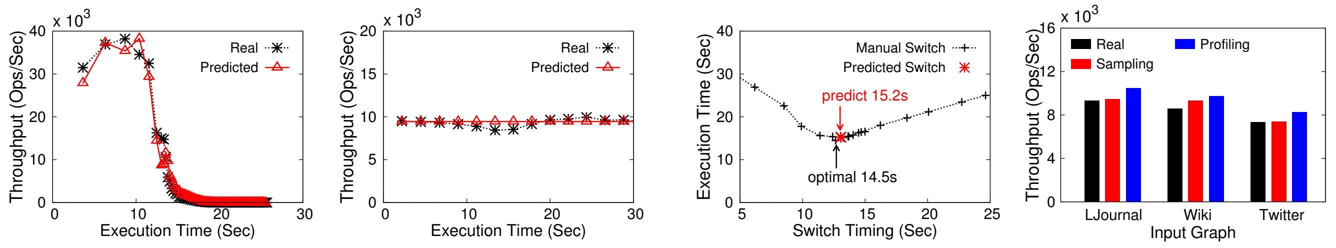
$$T_{vert_{next}} = \alpha \cdot T_{vert_{current}} + (1-\alpha) \cdot T_{vert_{history}} \tag{2}$$

where α is constant weighting factor ($0 \le \alpha < 1$), and we fix $\alpha = \frac{2}{3}$ in our experiments.

***Async* Mode**: the model in *Async* mode is simplified to Equation 3 without the constant overhead in *Sync* mode.

$$Thro_{next} = \frac{1}{T_{vert_{next}}} \tag{3}$$

where $T_{vert_{next}}$ is calculated the same as that in *Sync* mode (Equation 2).



(a) Throughput of Sync *(b) Throughput of Async* *(c) Switch Timing* *(d) Sampling vs. Profiling*

Figure 8: *A comparison between real and predicted throughput for (c) Sync and (b) Async modes respectively. (c) The execution time for PageRank with different switch points. (d) A comparison of performance between online sampling and NN-based offline profiling.*

5.3 Predicting *Throughput* of Alternate Mode

Compared with predicting throughput of the current mode, it is harder to exactly predict the throughput of the other mode. According to our study on typical algorithms with various graphs, PowerSwitch combines online sampling, offline profiling and some heuristics to provide a reasonable prediction.

Async Mode: since software pipelining is widely used in *Async* scheduling, the throughput of *Async* mode is mostly stable as long as there are enough active vertices to be processed. Figure 7(a) illustrate the real throughput in *Async* mode for three typical algorithms (SSSP, LBP and PageRank). As expected, the throughput of *Async* mode is fairly stable during the whole execution.

Based on the above observation, PowerSwitch provides either an **online sampling** or an **offline profiling** to predict the throughput of *Async* mode in *Sync* execution.

For online sampling, PowerSwitch initiates graph computation in *Async* mode, and then switches to the original initial mode. Throughput sampling of *Async* mode is executed with current input graph in a short time (e.g., 500ms). There is a little overhead for graph algorithms inherently started in *Sync* mode (e.g., PageRank).

For offline profiling, we use a set of training graphs to build a neural network model (NN), which predicts the throughput of *Async* mode for current input graphs. The NN-based offline profiling is relatively friendly to the scenario in which one algorithm is repeatedly used for various input graphs, since the training overhead can be amortized and there are abundant input graphs for more accurate training.

For Gather-Apply-Scatter (GAS) model [18] in PowerGraph, the processing time spent on a single vertex consists of three parts: 1) the unit computation time in the Gather (T_G) and Scatter (T_G) phase, which are proportional to the degree of vertex ($|D|$); 2) the unit computation time in the Apply (T_A) phase; 3) the unit communication time (T_M), which is proportional to the replication factor ($|R|$). Equation 4 illustrates our model for the throughput of *Async* mode in GAS model.

$$Thro_{async} = \frac{1}{(T_G + T_S) \cdot |D| + T_M \cdot |R| + T_A} \tag{4}$$

where $|D|$ denotes the average degree of vertex (i.e., $|E|/|V|$), and $|R|$ denotes the average number of replicas. Therefore, our neural network model adopts two dimensions (i.e., $|D|$ and $|R|$) of input vectors to model the throughput of *Async* mode, which present the main properties of input graph.

To evaluate the flexibility of our neural network model, we train the neural network with a large number of synthetic graphs [18] with various graph sizes. Figure 7(b) validates that the throughput of *Async* mode is sensitive to the properties of input graph (e.g. power-law constants) rather than the size of training graph.

Sync Mode: the throughput of *Sync* mode highly depends on the number of active vertices, which is costly to collect in *Async* mode since there are a large number of in-flight messages and the scheduling queue in each machine is frequently updated.

Fortunately, we observe that the throughput of *Sync* mode is better than that of *Async* mode when there are sufficient active vertices, thanks to the message batching and lower scheduling cost. Therefore, PowerSwitch employs a simple heuristic rule to indirectly make the decision. PowerSwitch first samples the increment speed of active vertices in the scheduling queue, and then compares it with the predicted throughput of *Async* mode. When the number of *newly* generated active vertices during a time interval (e.g., 500ms) exceeds the throughput of *Async* mode, it implies that *Async* mode is overloaded and it is time to switch mode.

5.4 Heuristic Rules

Besides the normalized throughput that is the major factor to guide the switch timing, PowerSwitch also combines several heuristics to improve the accuracy of decision.

To balance the trade-off between performance boost and switching overhead and avoid thrashing between two modes, PowerSwitch employs a pessimistic strategy that prefers staying in the current mode, to minimize overhead due to mode switches. The watermark value is set to 10% above the throughput of current mode, which guarantees the benefit from a mode switch is enough to overcome the switching cost.

Further, in our experience, *Async* mode does not work well under heavy workload due to potentially heavy lock contention. Hence, PowerSwitch not only samples the number of active vertices to estimate throughput, but also computes the changing rate of active vertices as an indication of workload prediction. PowerSwitch will switch to *Sync* mode during the increase of active vertices and back to *Async* mode during the decrease of active vertices.

Finally, if the algorithm only converges in a specific mode (e.g. *Async* mode for Graph Coloring), a user-defined progress function will be used to calculate the current convergence. For example, the percentage of converged vertices is used to measure the progress for Graph Coloring. When no progress is made, PowerSwitch will prior switch the current mode to the user-defined convergence mode.

5.5 Accuracy of Prediction

To evaluate the accuracy of prediction, we first compare the throughput of *Sync* and *Async* modes *predicted* by history in current mode with the real throughput online estimated, and then present an integrated analysis of switch timing for PageRank under the sampled convergence ratio $\mu = 0.8$ (see Section 5.1). Finally, we compare the throughput of *Async* mode predicted by online sampling and offline profiling with the real one.

Figure 8(a) and (b) compare the real and predicted throughput in *Sync* and *Async* mode accordingly. The dashed lines present the real throughput online estimated, while the solid lines present the predicted throughput by history using equations in Section 5.2. As shown in the figures, the throughput of both *Sync* and *Async* modes can be accurately predicted.

We further study the prediction mechanisms by collecting the switch points predicted by PowerSwitch. To find an optimal switch point, we repeatedly run PageRank algorithm for the LJournal

Table 2: *A collection of real-world and synthetic graphs.*

Algorithm	Graph	$\|V\|$	$\|E\|$
PageRank [7]	LJournal [13]	5.4M	79M
	Wiki [22]	5.7M	130M
	Twitter [26]	42M	1.47B
LBP [19]	SYN-ImageData [30]	1-12M	2-24M
SSSP [5]	RoadCA [28]	1.9M	5.5M
Coloring [17]	Twitter [26]	42M	1.47B

graph [13], and manually switch modes from *Sync* to *Async* at different points during the execution. As shown in Figure 8(c), with the switch point being postponed, the overall execution time would decrease efficiently and then gradually increases; the best switching point is at time 12.6 seconds, with the execution time of 14.5 seconds. The predicted switch point by PowerSwitch is at time 13.1 seconds, with the execution time of 15.2 seconds, which is only slower by 4.6% compared to the optimal one. Due to the additional online sampling overhead, the total execution time of *Hsync* mode is 15.9 seconds. As the sampling cost is mostly constant, the cost could be further amortized for processing larger graphs.

Figure 8(d) shows the *Async* throughputs predicted by both online sampling and NN-based offline profiling are close to the real throughput. The error of predicted throughput using online sampling and offline profiling are from 0.5% to 7.9% and from 10.8% to 11.5% respectively. Since online sampling provides more accurate throughput while causes a few runtime overhead, users can on-demand select either one to predict accurate *Async* throughput.

6. Case Studies

In this section, we analyze the behavior of *Hsync* using four typical graph algorithms: *PageRank* [7], *Loopy Belief Propagation* (LBP) [19], *Single-Source Shortest Path* (SSSP) [5] and *Graph Coloring* [17]. The detailed performance of the four algorithms under *Hsync* mode will be presented in next section.

PageRank is widely used to evaluate the relative importance of webpages, which models webpages and their relationships as an unweighted graph and updates the rank of webpages based on interaction between neighbors. In PageRank, all vertices will be initially activated and converge asymmetrically. Hence, PowerSwitch initiates PageRank in *Sync* mode for batching computation tasks and communication messages at the beginning stage. With the decreasing of active vertices, constant overhead from global barrier gradually becomes the major cost, which decreases the throughput of *Sync* mode. When the throughput of *Sync* mode is lower than that of *Async* mode, PowerSwitch switches to *Async* mode for more efficient convergence.

Loopy Belief Propagation (LBP) is an approximate inference algorithm, which used to estimate the marginal distributions by iteratively re-computing parameters associated with each edge until convergence. Due to similar behavior to PageRank, PowerSwitch also initiates LBP in *Sync* mode and switches to *Async* mode when the throughput of *Sync* mode is lower than that of *Async* mode. However, PageRank is sensitive to the size of graph while LBP is more sensitive to the hardware configuration and partitioning.

Single-Source Shortest Path (SSSP) provided by PowerGraph applies data-driven execution of the Bellman-Ford algorithm [5], which starts from a single source and propagates shortest path to neighbors until convergence. In SSSP, the number of active vertices first increases and then decreases, which gives the throughput of *Sync* mode a parabolic shape (like in Figure 2(e)). In addition, the peak throughput of *Sync* mode is notably higher than that of *Async* mode. Hence, PowerSwitch initiates SSSP in *Async* mode due to lower initial workload, and switches execution mode between *Async* and *Sync* according to the change of workload.

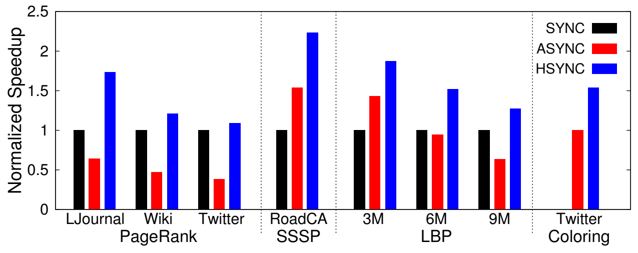


Figure 9: *A performance comparison between PowerSwitch and GraphLab using different modes for various algorithms and datasets.*

Graph Coloring assigns a color to each vertex of input graph such that no adjacent vertices share the same color. In *Sync* mode, all vertices concurrently adjust their colors relying on stale colors of neighbors. Even if *Sync* mode has good throughput thanks to high parallelism, the stale color causes slow or even failed convergence due to frequent conflicts. In *Async* mode, the latest colors of neighbors will be used to update vertex color, which may result in fast convergence. However, the throughput of *Async* mode is restricted by the network performance, which is much lower than that of *Sync* mode, especially for large graphs (e.g., Twitter Follow graph). Hence, PowerSwitch initiates Graph Coloring in *Sync* mode, and then switches to *Async* mode.

7. Evaluation

We have implemented PowerSwitch based on GraphLab 2.2 (released in July 2013), which runs the PowerGraph [18] engine. PowerSwitch is implemented as a separate engine in GraphLab and thus can seamlessly run all existing graph algorithms for GraphLab.

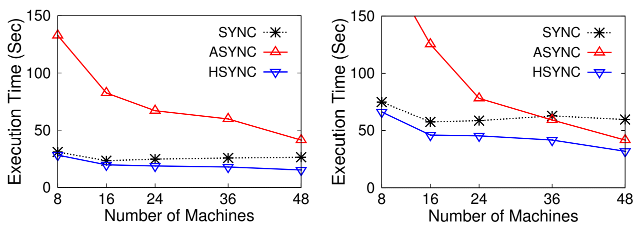
In this section, we first present the overall performance improvement of PowerSwitch using four typical graph algorithms (i.e., PageRank, SSSP, LBP and Graph Coloring) from the GraphLab toolkits. We then analyze performance using two graph algorithms under different graph configurations, including machine scales, input size scales and graph partitioning approaches. We also analyze the computation stages changes of SSSP and Graph Coloring to study the benefit of fine-grained mode switches.

Table 2 lists a collection of large graphs used in our experiments. Most of them are from Stanford Large Network Dataset Collection [35] and The Laboratory for Web Algorithmics [1]. The Wiki dataset is from [22]. The dataset for the LBP algorithm is synthetically generated by tools provided from that used in the Gonzalez et al. [18]. The SSSP algorithm requires the input graph to be directed and weighted. Since the RoadCA graph [28] is not originally weighted, we synthetically assign a weight value to each edge, where the weight is generated based on a log-normal distribution ($\mu = 0.4, \sigma = 1.2$) from the Facebook user interaction graph [43].

The correctness of all algorithms is validated by comparing with the result of the original *Sync* and *Async* modes alone. As many MLDM algorithms are non-deterministic in essence, different runs may cause results with a very small deviation. We set the same threshold of convergence for the evaluated algorithms and compare the final results to ensure they are within the tolerable range of differences.

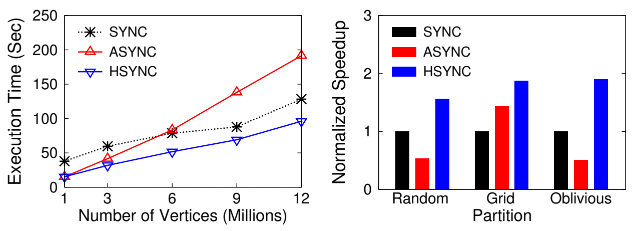
All experiments are performed on a 48-node EC2-like cluster. Each node has four AMD Opteron cores, 12GB of RAM, and connected via a 1 GigE network. Unless specified, all experiments were performed using online sampling for PowerSwitch, and the sampling overhead has been included in results. The reported throughputs[4] of *Sync* mode is also normalized using the convergence ratio in Section 5.1.

[4] All of reported throughputs are measured on a single machine of cluster.



(a) PageRank (b) LBP

Figure 10: *A comparison between PowerSwitch and GraphLab with the increase of machines for (a) PageRank and (b) LBP.*



(a) Size of Graph (b) Partitioning Algorithms

Figure 11: *A comparison between PowerSwitch and GraphLab for LBP with (a) the increase of graph size and (b) various partitioning algorithms.*

7.1 Overall Performance Improvement

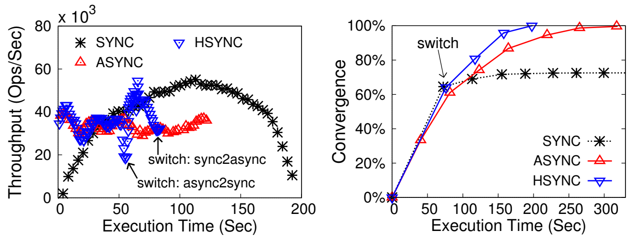
Figure 9 shows the overall speedup of PowerSwitch for the four algorithms with different datasets on 48 machines. PowerSwitch outperforms both *Sync* and *Async* engines of GraphLab on all four algorithms by up to 2.23X (from 1.09X) and 2.71X (from 1.31X) accordingly, which mainly stem from the accurate prediction and timely mode switches. As PowerSwitch can always choose a best mode for execution, it can accelerate the convergence speed and vertex processing throughput. For SSSP, since it essentially does not scale with machine scale, we run the test dataset with only 12 machines, though the performance trend in larger setting is similar.

We evaluate the performance scalability with the increasing number of machines on PageRank with LJournal and LBP with 3-million pixels, which confirms the properties and favorites scenarios of *Sync* and *Async* modes as analyzed before. As shown in Figure 10(a) and (b), *Async* mode does scale with machines, and *Sync* mode actually gets performance degradation when the machine number is larger than 36, since the synchronization cost is also increased with machine scale. However, with large dataset such as in PageRank, *Sync* mode has significant performance advantages. For applications with long computation time and low communication traffic (e.g., LBP), *Async* mode has superior performance. In such cases, *Sync* mode still has advantages with a small number of machines. This may be because when the number of machines is small, each machine will be assigned with a very heavy workload at beginning of the execution but the communication is small, a configuration in which *Sync* mode can process vertices more efficiently.

In Figure 10(a) and (b), the *Hsync* mode outperforms all the two modes alone by up to 1.87X (from 1.04X) and 5.43X (from 1.31X) accordingly, since PowerSwitch essentially embraces the best of both modes. Even in the worst case, performance of PowerSwitch is still comparable with the faster one of *Sync* and *Async* modes.

7.2 Evaluation of Different Configurations

As mentioned in Section 2, different configurations such as various input size and partition methods may result in different choices of execution modes. Hence, we compare *Hsync* mode with *Sync* and *Async* modes in various configurations, to show *Hsync* mode could get superior performance in all cases.



(a) SSSP (b) Graph Coloring

Figure 12: *A comparison between PowerSwitch and GraphLab for (a) SSSSP and (b) Graph Coloring.*

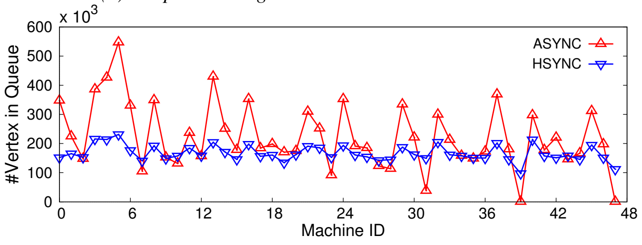


Figure 13: *A comparison of load balance for Async and Hsync modes.*

Input Sizes: Figure 11(a) evaluates the performance using different modes for LBP with increasing graph size. *Async* mode outperforms *Sync* mode with smaller inputs due to convergence acceleration without synchronization delay, while *Sync* mode has better performance with large inputs due to batching computation and network communication. *Hsync* mode outperforms both *Sync* and *Async* modes alone by up to 2.00X and 2.47X accordingly.

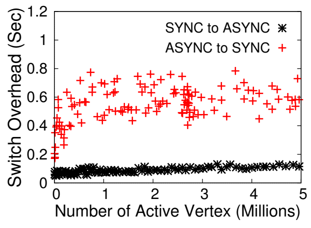
Graph Partitioning: Figure 11(b) shows the speedup of LBP with 3-million pixels using different graph partitioning algorithms, which have different trade-offs in graph ingress, runtime and load balance. Random simplifies graph ingress phase, Oblivious achieves better partition results (i.e., lower replication factor), and Grid provides a compromise. As shown in Figure 11(b), execution modes are affected by partition algorithm in varying degrees, but *Hsync* mode still provides better performance compared to both *Sync* and *Async* modes.

7.3 Execution Stage Analysis

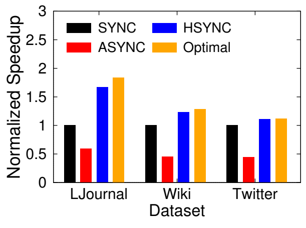
In this part, we present the benefit of our system by a performance comparison of the entire execution with SSSP and Graph Coloring algorithms.

Figure 12(a) compares the normalized throughput in different stages for SSSP on 12 machines. The throughput of *Hsync* always stays at a high level, and only drops at the switch point due to state transfer, especially from *Async* to *Sync*, which requires waiting for pending vertex computations to be completed. Since *Hsync* mode embraces the best throughput of both *Async* (at the duration of 0s to 54s, and 78s to 84s) and *Sync* (at the duration of 55s to 77s) modes, *Hsync* can outperform *Async* and *Sync* mode by 1.45X and 2.29X respectively (84s vs. 192s and 122s).

Figure 12(b) compares the progress of vertex convergence in Graph Coloring algorithm, which is evaluated in different execution modes on 48 machines. We can find that *Sync* mode does not converge, while the convergence speed of *Async* mode will gradually decrease. *Hsync* has the best performance by combining the benefits of both *Sync* and *Async* modes. *Hsync* accelerates the convergence of the latter half of execution, and obtains notable performance improvement compared to *Async*. Figure 13 further breaks down the execution status after executing about 100 seconds for *Async* and *Hsync* modes, and compares the distributions of remaining active vertices on each machine. In *Async* mode, when resource contention becomes a bottleneck of some machines, the skewed delay of vertex computation causes the imbalanced distribution of



(a) Switch Overhead



(b) Sampling Overhead

Figure 14: *(a) The switch overhead between the two modes with the increase of active vertices. (b) Normalized speedup using various modes.*

remaining jobs over the entire cluster. However, in *Hsync* mode, this phenomenon is avoided by early synchronous scheduling, and the balanced job distribution contributes to a fast convergence in asynchronous scheduling of the later execution.

7.4 Overhead

We evaluate the mode switch cost of PowerSwitch using a micro-benchmark, as shown in Figure 14(a). Since graph structures are shared by both modes in PowerSwitch, PowerSwitch only needs to transform the activation of vertices between the two modes. The mode switch cost from *Sync* to *Async* mode and vice versa is only around 0.1 and 0.6 second respectively. The switch cost from *Async* to *Sync* mode is relatively high and unstable, as PowerSwitch needs to wait until all pending vertex computation and flying messages have been done and delivered.

To evaluate the overhead of online sampling, we compare the total execution time for PageRank using different modes. We also estimate the optimal execution time by manually adjusting the switch point with full information of the execution procedure. As shown in Figure 14(b), *Hsync* outperforms both *Sync* and *Async* modes, and only incurs less than 10% overhead compared to optimal results even with online sampling.

7.5 Limitation Discussion

The performance difference of the two modes is mainly due to their design strategies. *Sync* mode mainly aims to reduce the message communication overhead and improve the hardware utilization, while *Async* mode aims to accelerate convergence speed under sufficient hardware support. Even if both of these two modes can be well optimized, PowerSwitch could still leverage the performance difference to get an extra speedup. However, the performance speedup has limitations in some cases. For instance, when the execution procedure of specific application is relatively stable, e.g. without workload change or communication request burst, PowerSwitch may only help to choose a better mode to execute the application with no need to switch.

8. Related Work

There have been a consider number of work extending MapReduce [14] to support iterative graph processing. To accommodate the unique characteristics of graph processing, Pregel [31] and its open-source alternatives [2, 3, 37], Cyclops [9] and GraphX [20] adopt the bulk-synchronous parallel (BSP) processing. All these frameworks essentially use synchronous scheduling. GraphLab [30] pioneers in supporting asynchronous scheduling.

Both PowerGraph [18], PowerLyra [10] and Trinity [38] provides a unified graph abstraction for a program to run in either *Sync* or *Async* engine. GRACE [42] also at unifying the graph interface through allowing the same graph algorithm to run either *Sync* or *Async* modes. However, none of them supports adaptive mode switching.

PowerSwitch departs from existing work in separating graph abstraction and computation logic from scheduling, by additionally allowing accurate prediction to adaptively choose an optimal mode for execution. The optimization in existing work could also be used to improve the scheduling implementation in PowerSwitch.

In addition, PowerSwitch also shares the similarity in finding an optimal configuration for MapReduce applications [23, 24], but targeted at finding optimal switch points. There are also several efforts [4, 45] aiming at dynamically and automatically switching traversal strategies for breadth first search algorithm (BFS).

The importance of graph processing has also popularized it to a number of usage scenarios and generated a number optimization efforts. Much of work has attempted to optimize graph processing on multicore [27, 33, 36, 39, 46] and GPU [48, 49]. There are also a few systems considering streaming processing [12, 32] or the graph properties [11, 21]. For graph algorithms with different performance characteristics, it could also be beneficial to apply the adaptive mode switching approach in PowerSwitch to boost their performance.

9. Conclusion

This paper made a comprehensive study on execution characteristics of two typical execution modes of a state-of-the-art graph-parallel system. The study revealed that performance between the two modes varies significantly for different graph algorithms, partitioning algorithms, execution stages, graph and cluster scales, and thus no single mode consistently outperforms the other. Based on this observation, this paper proposed *Hsync*, a hybrid-synchronous execution mode that allows dynamic switching between synchronous and asynchronous modes to gain optimal performance. The resulting system, called PowerSwitch, periodically collects execution statistics and combines online sampling or offline profiling with domain-specific knowledge to predict a best execution mode for next stages. Evaluation shows that PowerSwitch consistently outperforms the best of each mode alone.

10. Acknowledgments

We thank the anonymous reviewers for their insightful comments. This work is supported in part by the Doctoral Fund of Ministry of Education of China (No. 20130073120040), the Program for New Century Excellent Talents in University, Ministry of Education of China (No. ZXZY037003), a foundation for the Author of National Excellent Doctoral Dissertation of PR China (No. TS0220103006), the National Natural Science Foundation of China (No. 61303011 and 61402284), the Open Project Program of the State Key Laboratory of Mathematical Engineering and Advanced Computing (No. 2014A05), the Shanghai Science and Technology Development Fund for high-tech achievement translation (No. 14511100902), the IBM CRL university joint research grant, a research grant from Intel and the Singapore NRF (CREATE E2S2).

References

[1] The laboratory for web algorithmics. http://law.dsi.unimi.it/datasets.php.

[2] Apache. The Apache Giraph Project. http://giraph.apache.org/.

[3] Apache. The Apache Hama Project. http://hama.apache.org/.

[4] Beamer, S., Asanović, K., and Patterson, D. Direction-optimizing breadth-first search. *Scientific Programming 21*, 3 (2013), 137–148.

[5] Bertsekas, D. P., Guerriero, F., and Musmanno, R. Parallel asynchronous label-correcting methods for shortest paths. *Journal of Optimization Theory and Applications 88*, 2 (1996), 297–320.

[6] Bertsekas, D. P., and Tsitsiklis, J. N. *Parallel and distributed computation: numerical methods*. Prentice-Hall, Inc., 1989.

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

[8] CHANDY, K., AND LAMPORT, L. Distributed snapshots: determining global states of distributed systems. *ACM TOCS 3*, 1 (1985), 63–75.

[9] CHEN, R., DING, X., WANG, P., CHEN, H., ZANG, B., AND GUAN, H. Computation and communication efficient graph processing with distributed immutable view. In *Proc. HPDC* (2014).

[10] CHEN, R., SHI, J., CHEN, Y., GUAN, H., AND CHEN, H. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. Tech. Rep. 2013-11-001, Shanghai Jiao Tong University, 2013.

[11] CHEN, R., SHI, J., ZANG, B., AND GUAN, H. Bipartite-oriented distributed graph partitioning for big learning. In *Proc. APSys* (2014).

[12] CHENG, R., HONG, J., KYROLA, A., MIAO, Y., WENG, X., WU, M., YANG, F., ZHOU, L., ZHAO, F., AND CHEN, E. Kineograph: taking the pulse of a fast-changing and connected world. In *Proc. EuroSys* (2012).

[13] CHIERICHETTI, F., KUMAR, R., LATTANZI, S., MITZENMACHER, M., PANCONESI, A., AND RAGHAVAN, P. On compressing social networks. In *Proc. SIGKDD* (2009).

[14] DEAN, J., AND GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. *CACM 51*, 1 (2008), 107–113.

[15] EFRON, B., HASTIE, T., JOHNSTONE, I., AND TIBSHIRANI, R. Least angle regression. *The Annals of statistics 32*, 2 (2004), 407–499.

[16] ELIDAN, G., MCGRAW, I., AND KOLLER, D. Residual belief propagation: Informed scheduling for asynchronous message passing. In *Proc. UAI* (2006).

[17] GONZALEZ, J., LOW, Y., GRETTON, A., AND GUESTRIN, C. Parallel gibbs sampling: From colored fields to thin junction trees. In *Proc. ICAIS* (2011).

[18] GONZALEZ, J., LOW, Y., GU, H., BICKSON, D., AND GUESTRIN, C. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proc. OSDI* (2012).

[19] GONZALEZ, J. E., LOW, Y., GUESTRIN, C., AND O'HALLARON, D. Distributed parallel inference on large factor graphs. In *Proc. UAI* (2009).

[20] GONZALEZ, J. E., XIN, R. S., DAVE, A., CRANKSHAW, D., FRANKLIN, M. J., AND STOICA, I. Graphx: Graph processing in a distributed dataflow framework. In *Proc. OSDI* (2014).

[21] HAN, W., MIAO, Y., LI, K., WU, M., YANG, F., ZHOU, L., PRABHAKARAN, V., CHEN, W., AND CHEN, E. Chronos: a graph engine for temporal graph analysis. In *Proc. EuroSys* (2014).

[22] HASELGROVE, H. Wikipedia page-to-page link database. http://haselgrove.id.au/wikipedia.htm, 2010.

[23] HERODOTOU, H., AND BABU, S. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *Proceedings of the VLDB Endowment* (2011).

[24] HERODOTOU, H., DONG, F., AND BABU, S. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In *Proc. SoCC* (2011).

[25] JAIN, N., LIAO, G., AND WILLKE, T. L. Graphbuilder: scalable graph etl framework. In *Proc. GDM* (2013).

[26] KWAK, H., LEE, C., PARK, H., AND MOON, S. What is twitter, a social network or a news media? In *Proc. WWW* (2010).

[27] KYROLA, A., BLELLOCH, G., AND GUESTRIN, C. GraphChi: Large-scale graph computation on just a PC. In *Proc. OSDI* (2012).

[28] LESKOVEC, J., LANG, K., DASGUPTA, A., AND MAHONEY, M. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics 6*, 1 (2009), 29–123.

[29] LIU, Y., WU, B., WANG, H., AND MA, P. Bpgm: A big graph mining tool. *Tsinghua Science and Technology 19*, 1 (2014), 33–38.

[30] LOW, Y., BICKSON, D., GONZALEZ, J., GUESTRIN, C., KYROLA, A., AND HELLERSTEIN, J. M. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *Proc. VLDB* (2012).

[31] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: a system for large-scale graph processing. In *Proc. SIGMOD* (2010).

[32] MURRAY, D. G., MCSHERRY, F., ISAACS, R., ISARD, M., BARHAM, P., AND ABADI, M. Naiad: a timely dataflow system. In *Proc. SOSP* (2013).

[33] NGUYEN, D., LENHARTH, A., AND PINGALI, K. A lightweight infrastructure for graph analytics. In *Proc. SOSP* (2013).

[34] PANDA, B., HERBACH, J., BASU, S., AND BAYARDO, R. PLANET: massively parallel learning of tree ensembles with MapReduce. *Proc. VLDB* (2009).

[35] PROJECT, S. N. A. Stanford large network dataset collection. http://snap.stanford.edu/data/.

[36] ROY, A., MIHAILOVIC, I., AND ZWAENEPOEL, W. X-stream: Edge-centric graph processing using streaming partitions. In *Proc. SOSP* (2013).

[37] SALIHOGLU, S., AND WIDOM, J. GPS: A Graph Processing System. http://infolab.stanford.edu/gps/, 2012.

[38] SHAO, B., WANG, H., AND LI, Y. Trinity: A distributed graph engine on a memory cloud. In *Proc. SIGMOD* (2013).

[39] SHUN, J., AND BLELLOCH, G. E. Ligra: a lightweight graph processing framework for shared memory. In *Proc. PPoPP* (2013).

[40] SMOLA, A., AND NARAYANAMURTHY, S. An architecture for parallel topic models. *Proc. VLDB* (2010).

[41] VALIANT, L. G. A bridging model for parallel computation. *CACM 33*, 8 (1990), 103–111.

[42] WANG, G., XIE, W., DEMERS, A. J., AND GEHRKE, J. Asynchronous large-scale graph processing made easy. In *Proc. CIDR* (2013).

[43] WILSON, C., BOE, B., SALA, A., PUTTASWAMY, K. P., AND ZHAO, B. Y. User interactions in social networks and their implications. In *Proc. EuroSys* (2009).

[44] YE, J., CHOW, J., CHEN, J., AND ZHENG, Z. Stochastic gradient boosted distributed decision trees. In *Proc. CIKM* (2009).

[45] YOU, Y., SONG, S. L., AND KERBYSON, D. An adaptive cross-architecture combination method for graph traversal. In *Proc. SC* (2014).

[46] ZHANG, K., CHEN, R., AND CHEN, H. Numa-aware graph-structured analytics. In *Proc. PPoPP* (2015).

[47] ZHAO, X., CHANG, A., SARMA, A. D., ZHENG, H., AND ZHAO, B. Y. On the embeddability of random walk distances. *Proc. VLDB* (2013).

[48] ZHONG, J., AND HE, B. Medusa: Simplified graph processing on gpus. *IEEE TPDS* (2013).

[49] ZHONG, J., AND HE, B. Parallel graph processing on graphics processors made easy. *Proc. VLDB* (2013).