Graph Analytics Through Fine-Grained Parallelism

Zechao Shang†, Feifei Li‡, Jeffrey Xu Yu†, Zhiwei Zhang§, Hong Cheng†
†The Chinese University of Hong Kong, ‡University of Utah, §Hong Kong Baptist University
{zcshang,yu,hcheng}@se.cuhk.edu.hk, lifeifei@cs.utah.edu,
cszwzhang@comp.hkbu.edu.hk

ABSTRACT

Large graphs are getting increasingly popular and even indispensable in many applications, for example, in social media data, large networks, and knowledge bases. Efficient graph analytics thus becomes an important subject of study. To increase efficiency and scalability, in-memory computation and parallelism have been explored extensively to speed up various graph analytical workloads. In many graph analytical engines (e.g., Pregel, Neo4j, GraphLab), parallelism is achieved via one of the three concurrency control models, namely, bulk synchronization processing (BSP), asynchronous processing, and synchronous processing. Among them, synchronous processing has the potential to achieve the best performance due to fine-grained parallelism, *while ensuring the correctness and the convergence of the computation*, *if* an effective concurrency control scheme is used. This paper explores the topological properties of the underlying graph to design and implement a highly effective concurrency control scheme for efficient synchronous processing in an in-memory graph analytical engine. Our design uses a novel hybrid approach that combines 2PL (two-phase locking) with OCC (optimistic concurrency control), for high degree and low degree vertices in a graph respectively. Our results show that the proposed hybrid synchronous scheduler has significantly outperformed other synchronous schedulers in existing graph analytical engines, as well as BSP and asynchronous schedulers.

1. INTRODUCTION

Graph is a perfect model to represent complex relationships among objects, in particular, data interconnectivity and topology information. As a result, the graph data model has been used extensively to store, query, and analyze data from a number of important application domains ranging from social media data to large computer or transportation networks to knowledge bases (semantic web) and to biological structures [1, 17]. Large amounts of graph data and many applications running on top drive the needs for efficient and scalable graph analytics.

To increase both efficiency and scalability, *in-memory computation* and *parallelism* have been explored extensively as two very useful techniques to speed up analytics over large graphs. Inspired by the design and success of early systems like Pregel [35], most graph systems support the *vertex-centric* data model and programming paradigm, in which a graph is represented by a set of vertices and their edge lists. An edge list for a vertex v simply stores all neighboring vertices of v in the graph. A typical analytical workload over a vertex-centric graph model can often be represented as follows (suppose the graph has N vertices and UDF is some user defined function):

```
for iteration i = 1...n      // in iteration
   for j = 1...N             // all vertices
         UDF(v_j, edgeList_j)
```

For example, the classic PageRank algorithm is an instantiation of the above computation framework. We often speed up this computation through parallelism, and existing graph systems have used different consistency models for such parallelism. The most popular choices are the following three consistency models, namely, *the bulk synchronous parallel (*BSP*), asynchronous, and synchronous parallel processing*. A major difference among these models is their message passing mechanism and the staleness of messages.

BSP proceeds in iterations (or super-steps), and message passing takes place at the end of an iteration, therefore the message staleness is exactly one iteration. AP may introduce a delay (of uncertain length) in delivering messages. In contrast, in SP processing, a worker (thread) receives messages without any degree of staleness. In other words, the implication is that workers in the BSP model reach synchronized states at the end of an iteration, but they will proceed with asynchronized states in the asynchronous model, and they have to wait for synchronization and only proceed when a synchronized state is obtained in the synchronous model. In the context of vertex-centric analytics, data consistency model decides whether vertices are able to access up-to-date and consistent data values from other vertices during the computation.

BSP is the most popular model in practice due to its simplicity; it is adopted by the general parallel computing frameworks such as MapReduce [8], Spark [66], and graph analytical systems such as GraphX [64] that is based on Spark. Systems and programming frameworks using the asynchronous model, including PowerGraph, Galois, Grace [15, 45, 59], were also proposed. The synchronous model was explored by a few existing systems (e.g., in GraphLab [33]), however, the overhead in ensuring synchronization needs to be carefully addressed to avoid introducing significant negative impacts to system efficiency.

Ideally, no message staleness is the most desirable design since it leads to synchronized states for all workers in a parallel computation framework. However, ensuring this in a synchronous model is not trivial and waiting for synchronization may introduce significant overheads in a system. In practice, the BSP and asynchronous models address this issue by exploring the trade-off between mes-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA
© 2016 ACM. ISBN 978-1-4503-3531-7/16/06...$15.00
DOI: http://dx.doi.org/10.1145/2882903.2915238

sage staleness and efficiency. In particular, the BSP model limits the massage staleness by introducing a barrier in its computation, i.e., workers need to wait for the completion of the last thread from the current iteration, which ensures that the states of different workers are synchronized by the end of each iteration. However, it suffers from the infamous *straggler problem.*

The asynchronous model, on the other hand, removes the restriction of synchronization altogether, hence, can be very efficient. However, since asynchronous message might be significantly delayed in reaching its target, which could lead to incorrect or unstable computations. For many graph analytical workloads, the algorithms can recover the correct, synchronized states in next iteration. However, this is not the case for all analytics, and there are algorithms that do not have an implementation that will eventually converge using the asynchronous model. Even algorithms on convex problems which always leads to the correct answer may fall into local minimum caused by the chaos of asynchrony, e.g., SGD (stochastic gradient descent) may fail to converge [69, 42].

Our contributions. In light of these limitations, we focus on improving the efficiency and scalability of parallel graph analytics using *synchronous parallel processing*. We assume an in-memory computing engine, to focus on the effects of the concurrency control model and eliminate the impacts of disk I/Os. We encapsulate the computations initiated by each vertex in a given round as a transaction. Using this view, the synchronous computation is made of transactions. Each transaction is an atomic unit of operations. By ensuring conflict serializability among these transactions, the results of the concurrent execution of these transactions are equivalent to the results from a sequential execution of these transactions. The synchronous model eliminates unbounded latency in communication and relieves developers from reasoning and debugging a non-deterministic system as that in the asynchronous model.

That said, we can use a standard concurrency control scheme, such as strict 2PL (two phase locking) or OCC (optimistic concurrency control) to ensure the concurrent execution of these transactions. We define the degree of a vertex v as the number of edges connected to v. An important observation is that the distribution for degrees of vertices in a graph has a strong influence on the likelihood of having a transaction conflict. This observation explains why traditional schedulers such as 2PL or OCC lead to bad performance for graph analytics using the synchronous model. In particular, degrees of vertices in a graph often follow a very skewed distribution. This means that the locking-based approaches such as 2PL will often lead to high locking overhead, while the non-locking based scheduler such as OCC will suffer from high abort rates at the time of transaction commit.

Inspired by this observation, we propose *HSync*, a *hybrid scheduler* that combines both locking-based and non-locking based schedulers, to achieve high performance graph analytics through synchronous parallel processing. HSync adapts to a scale-free graph where the degrees of vertices may vary: it uses *a locking-based scheduler for high degree vertices*, and *a non-locking-based scheduler for low degree vertices*. This design ensures that transactions for high degree vertices will never need to abort due to low degree ones (which is important since it is expensive to abort these transactions). But transactions for low degree vertices may need to abort, however, they are not expensive, and doing so ensures that almost no deadlocks will ever occur as transactions for low degree vertices will never block the execution and commit of a high degree vertex's transaction. Extensive experimental results confirmed the superior performance of our hybrid scheduler over existing schedulers for the synchronous model. They also show that by using our hybrid scheduler, the synchronous model will outperform the asynchronous and BSP models for in-memory parallel graph analytics.

Organization: In Section 2, we formalize the *vertex-centric parallel programming and computing* framework, and compare different task schedulers and data consistency models in this framework. In Section 3, we formalize and describe synchronous parallel processing for fine-grained parallelism in graph analytics. In Section 4, we discuss the impact of large scale-free graphs on synchronous parallel processing for graph analytics. In particular, we identify that the skew distribution for degrees of vertices in a graph is the root cause for the poor performance of existing schedulers. In Section 5, we propose our hybrid scheduler HSync that uses the locking-based approach and the non-locking based scheme for high-degree and low-degree graph vertices respectively. We show our experimental results in Section 6 and discuss the related work in Section 7. The paper is concluded in Section 8.

2. BACKGROUND AND FORMULATION

As introduced in Section 1, we focus on *in-memory graph analytics*. We assume that the graph data is owned and used by *a single user* for each analytical workload, which is fairly common in in-memory analytical systems, e.g, in Spark [66], GraphX [64], GraphLab [33], and many other graph analytical systems. A simple way to generalize to multiple users is to make a copy of the underlying graph for the execution of each analytical workload. We also assume that the graph data is static during the execution of an analytical workload, i.e., there are no updates to the graph during the execution of a graph analytics job. A user submits an analytical job and expects the exact results at the end of task completion. Reducing the wall-clock processing time is the main objective.

We also assume that the data resides in memory during the entire execution. We make this assumption for two reasons. First, in-memory computing is increasingly common [67], and many graph data does fit in memory. For example, as reported in [52], the median size of data used in Yahoo's and Microsoft's analytics jobs is 14GB. On the other hand, memory capacity increases quickly and it is not uncommon for many desktop machines nowadays to have memory in tens of GB, and commercial multi-core servers can support up to 6TB of memory.[1] Moreover, high speed network and RDMA (remote direct memory access) significantly reduce the latency of remote memory accessing and the speed of accessing memory in a remote node is close to the speed of accessing local memory. Therefore, with the help of middleware like FARM and RAMCloud [11, 46], even really large data can still reside entirely in memory over a cluster. Secondly, the focus of our work is to analyze the impacts of concurrency control schedulers for synchronous parallel processing in graph analytics. Thus, we would like to eliminate the impacts by other factors such as disk I/Os and highlight the bottlenecks introduced by the concurrency model.

2.1 Vertex-Centric Parallel Computing

Most graph analytical systems, like Pregel [35], GraphLab [33], and GraphX [64], are built based on the vertex centric (VC) model. In this model, user implements a user-defined function (UDF), which is executed on each vertex iteratively as shown in Section 1.

The vertex-centric computing model can be made to run in parallel, which leads to vertex centric parallel programming and computing. We illustrate a simplified framework of vertex centric parallel programming and computing in Fig. 1. The system consists of multiple workers (computer nodes and/or cores). Each worker executes the UDF on a vertex through the coordination of the task scheduler.

[1] For example, Dell R920 by November 2015

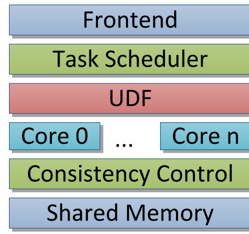


Figure 1: Vertex-centric parallel programming and computing.

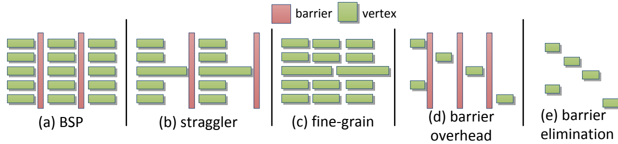


Figure 2: Coarse-grained vs. fine-grained scheduling

The UDF may be executed multiple times on the same vertex until a convergence condition is met. The concurrency control model dictates the consistency states among different vertices.

Although this simple framework is being used on literally all graph analytics systems, differences in two key components often lead to very different instantiation and performance in practice. The first design choice is *how to pick the next vertex v in task scheduler*. Jobs associated with different vertices may execute simultaneously, or in a random order, or by a user-defined priority assignment. The other design choice is *the data consistency model*: when more than one workers are executing the UDF over multiple vertices concurrently, it is entirely possible that they will interfere with each other, for example, UDF (v_1) updates the states of some vertices that are also used by UDF (v_2) when UDF (v_1) and UDF (v_2) are running concurrently. Systems may choose a strong data consistency model (e.g., synchronous parallel processing), a weak consistency model (e.g., BSP processing), or no consistency control at all (e.g., asynchronous parallel processing).

2.1.1 Task Scheduler

In VC graph analytics, the task scheduler is in charge of the strategy used to select the next vertex to run the UDF on. A popular choice is the BSP processing introduced by Pregel [35]. BSP works in iterations. All vertices run the same UDF *in parallel* in each iteration, and each vertex executes the UDF exactly once in one iteration. A barrier is enforced after each iteration to ensure that all workers complete their jobs before entering the next iteration, and synchronization messages are exchanged as well so that vertices reach synchronized states at the beginning of the next iteration. BSP is a coarse-grained scheduling method as illustrated by Fig. 2(a).

In contrast, a fine-grained scheduling method eliminates all barriers. Each execution of the UDF on a vertex becomes a computing task. A (centralized) global scheduler, usually a work queue, is responsible for scheduling. We illustrate its execution in Fig. 2(c).

The main advantage of coarse-grained scheduling like BSP is its simplicity and low overhead in concurrency control, since vertices run in parallel and isolation in one iteration and they only coordinate and send update messages to each other at the end of one iteration. If all tasks in one iteration have similar costs, *and* all iterations have similar number of tasks, as shown in Fig. 2(a), BSP will be highly effective. However, in practice this assumption usually does not hold and the coarse granularity in parallelism may suffer from several drawbacks.

The first limitation is *task straggler*: some tasks may run (much) longer than others. Therefore, as illustrated as Fig. 2(b), the completion time of each iteration is determined by the slowest task.

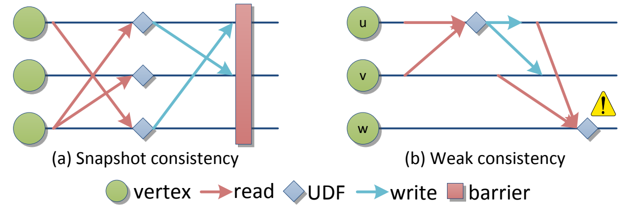


Figure 3: Snapshot and weak data consistency models.

Secondly, the overhead of enforcing the barriers becomes significant especially if not all tasks are active in an iteration. As illustrated in Fig. 2(d), if only a small number of tasks are being executed in each iteration, the workers (e.g., CPU cores) are underutilized and the time spent on barrier synchronization becomes a major bottleneck. Fine-grained scheduling is able to remove this overhead as shown in Fig. 2(e). *Third*, the coarse-grained scheduling prohibits application-specific intra-iteration optimizations. For example, for some problems assigning different (scheduling) priorities to tasks helps speed up the convergence rate [68]. Such prioritized scheduling is only feasible under a fine-grained scheduling methods. *Lastly*, in BSP a new iteration cannot start until all workers finish their communications in the last iteration. In other words cores are idling while waiting for network communication and synchronization. In contrast, finer-grained scheduling is able to reduce CPU idling significantly.

2.1.2 Data Consistency Model

If UDFs on different vertices are allowed to execute concurrently without control, they may interfere with each other and lead to inconsistent states. BSP overcomes this data consistency problem by working on a *snapshot*. In particular, each vertex reads data from last iteration and emits messages (to update other vertices) that will only be visible in next iteration. In other words, each vertex is not aware of other vertices' work (that run concurrently in parallel) until next iteration. This data consistency model is shown in Fig. 3(a). Clearly, this approach eliminates a lot of synchronization overheads, but limits the rate of state synchronization which in turn leads to slow convergence and longer job completion time.

Using the asynchronous model for fine-grained parallelism relaxes the data consistency requirement: there is little or no control on the data consistency at all. Each worker is allowed to *write to shared data anytime in any order*. As illustrated in Fig. 3(b), this approach suffers from the *weak data consistency* problem: the transaction w reads an inconsistent snapshot of the vertex u and v, which one have been update by transaction u and the other not. In order to compute exact and correct results, developers have to maintain data consistency somehow by themselves. This dramatically increases the complexity for programming and code maintenance. Nevertheless, this approach is very useful when an approximate result is acceptable or even desirable for better efficiency, which is usually satisfied by machine learning and data mining applications. However the exact trade-off between efficiency and accuracy is unclear and hard to model and reason about.

Lastly, strong data consistency model in fine-grained parallelism can be achieved by using synchronous parallel processing. In particular, for all workers that are being executed concurrently in parallel, the strong data consistency model requires that they satisfy *conflict serializability*. In other words, the concurrent execution of these workers leads to a state that is the same as the state from one of the serial executions of these workers. We will detail and formalize this framework in next section.

3. SYNCHRONOUS PARALLEL PROCESSING FOR FINE-GRAINED PARALLELISM

In this section we propose the fine-grained parallelism for graph analytics with strong data consistency by using synchronous parallel processing. Following the general VC framework, user will submit an UDF and it is executed over vertices in a graph. UDF (v_j) may read other vertices' attribute values and write values back. The *task scheduler* consists of the following modules:

1) The class `vertex` models the actions related to a vertex. It asks for an implementation of the UDF for this vertex, that may be called multiple times and may have different actions in each execution depending on the current states of the vertices in a graph. It also provides the interfaces for reading and writing values from and to other vertices. `vertex` can access its edge lists. For simplicity we do not distinguish the direction of edges. Instead we model them as undirected ones with tags. In other words, a `vertex` object represents a task related to a specific vertex.

2) `addTask` adds a task into the scheduler's queue, possibly with a priority assignment. Each task is removed from the scheduler's queue before its execution.

3) `barrier` introduces a global barrier; all workers enter this barrier after completing its currently assigned task. And they will execute a user-specified function f before leaving the barrier. The barrier can be used for synchronization. The function f may use `addTask` to add more tasks back to the queue. For example, to implement BSP-style scheduling, user adds each vertex using `addTask` and imposes the barrier based on a global value for iteration that is only updated when all workers in the current round have completed. Tasks in the next iteration are added back to the queue using f. `barrier` is also used for progress monitoring, checking for termination condition, debugging and profiling, etc.

```
1  // global functions and class
2  class vertex {
3      abstract UDF(vertex v);
4      value_t readVertex(vertex v);
5      writeVertex(vertex v, value_t val);}
6
7  addTask(vertex v, priority = 1.0);
8  barrier(function f);
```

Figure 4: The API of fine-grained parallelism.

```
1  in parallel for each worker (executed by a
       thread)
2      if at barrier
3          wait until all workers reach barrier
4          execute the function f
5          continue
6      if no more tasks in the scheduler queue
7          exit
8      retrieve (and remove) task vertex v
9      begin transaction
10         execute UDF(v) as a transaction t(v)
11     end transaction
```

Figure 5: The workflow of fine-grained parallelism.

The workflow of fine-grained parallelism is shown as Fig. 5. Once a worker becomes available, it retrieves a task from the scheduler and executes the task *concurrently in parallel* with other active workers while *ensuring strong data consistency*. The UDF is treated as an *atomic unit*. Atomic units run concurrently, but each of them behaves like *running by its own in isolation*. Although users can define such atomic unit with arbitrary logic, the most common form in graph analytics involves computations over a single vertex and its neighbors and the edges between them if applicable (i.e., its edge list). These computations form an atomic unit and are encapsulated by the UDF. Each such UDF proceeds as long as it does not interfere (i.e., conflict with) other concurrently running UDFs. For example, a conflict is identified if UDF (v_1) tries to update the attribute values of v_3 that is being used by UDF (v_2) running concurrently (suppose v_3 is a neighbor of both v_1 and v_2).

```
1  void UDF(vertex v) {
2      // calculate the page rank value
3      double sum = 0, previous = readVertex(v);
4      for (all neighbors u)
5          sum += readVertex(u);
6      sum = sum * (1 - d) + d;
7      writeVertex(v, sum);
8
9      // schedule v with priority
10     double gap = abs(sum - previous);
11     if (gap > 1e-6)
12         addTask(v, gap);
13
14     // report vertices with top-k page rank
           values periodically
15     if (time elapsed 1 sec since last report)
16         barrier({
17     // find current top-k vertices and print
18     });
19 }
```

Figure 6: PageRank in fine-grained parallelism framework.

This computation model bears great similarity to *transaction processing* in relational databases. In particular, each UDF as described above forms an atomic unit that satisfies the *atomicity* and *isolation* properties that are exactly the same as *'A' and 'I'* requirements in the ACID definition for transactions in RDBMS. Henceforth, we can enforce fine-grained parallelism for active workers in Fig. 5 through *transactions*.

UDF (v) is represented by a *transaction on vertex class* v. Unless specified we will denote fine-grained parallelism as transaction handling in following. Active workers lead to the set of actively running transactions. Using a *transaction scheduler* for concurrency control among these transactions, the system keeps the transactions from interfering each other while running concurrently. Formally speaking, the end effect of running them in parallel is equivalent to the results of a sequential execution of these transactions.

An example. We use the classic PageRank algorithm (PR) (Fig. 6) to illustrate the realization of fine-grained parallelism through this API. This UDF calculates the PageRank value of vertex v by reading neighboring vertices' PageRank values and writing back the updated PageRank value to the vertex v itself. After which, it adds the vertex back to the scheduler's queue, using `addTask`, if and only if the PageRank value for v has changed for more than a threshold. It reports the vertices with top-k PageRank values periodically by using the `barrier` function.

Remarks. The fine-grained parallelism through synchronous parallel processing has several advantages compared to other approaches (namely, BSP and asynchronous processing).

First, it is more flexible compared to BSP and asynchronous models. In fact, we can use the above fine-grained parallelism framework to realize BSP and asynchronous processing, simply by using different `barrier` functions and different concurrency control schemes for the *transactions*. BSP and asynchronous processing also suffer from the snapshot and weak data consistencies as discussed in Section 2.1.2. In particular, asynchronous computing introduces uncertainty to the computations, which brings extra troubles for users to reason the correctness of analytics.

Second, by modeling synchronous processing of active workers as transaction processing, fine-grained parallelism greatly reduce the system complexity. This frees other system modules from considering the concurrency control and synchronization issues. For example, dynamic workload balancing which aims at re-balancing the workloads during the parallel processing is critical to the performance. However such re-balancing is a sophisticated process in BSP: a typical workflow is by exchanging the workload informa-

tion several times before each worker agrees with the workloads to be migrated. Additional communications are needed to perform the migration and notification [54, 23, 65]. Such complexity are due to the staleness introduced by the coarse grain parallelism. For fine-grained parallelism re-balancing is much simpler: underloaded workers can steal the work from overloaded ones on the fly.

Lastly, dissemination of states among vertices is faster and synchronization reaches other vertices much quicker than using BSP, and ensures correctness that might not be possible in asynchronous model. For algorithms on convex problems like stochastic gradient descent (SGD), the flexibility of fine-grained parallelism makes it possible to support arbitrary update orders. Some gradients may get updated more frequently than others, and these updates are instantly visible to others once the transactions commit. Therefore the algorithm may converge faster. Non-convex and/or discrete problems may have problem-specific constrains or dependencies on the update orders. Consider graph coloring as an example: each vertex shall find a color distinct from its neighboring colors. It is important to ensure a vertex can access up-to-date and consistent color values of its neighbors during the computation. Fine-grained parallelism helps enforce such constraints: user program does not have to worry about a vertex u's neighbors concurrently alter their colors and lead to incorrect coloring assignment. Instead, the system itself will take over the concurrent executions while ensuring isolation. In contrast, the BSP model will incur significant overhead, and the asynchronous model may not even be able to implement this correctly because each vertex cannot be sure to read its neighbors' correct and consistent status.

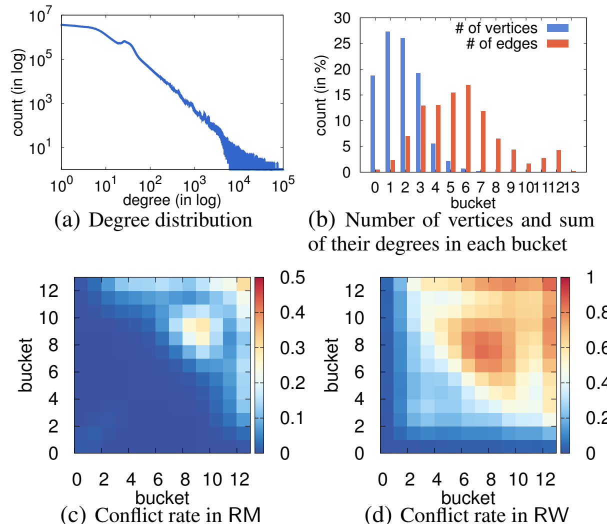
4. THE SCALE-FREE GRAPH CHALLENGE

Although fine-grained parallelism has several significant advantages against other consistency models including BSP and asynchronous processing, it can lead to considerable overheads in order to maintain strong data consistency. In particular, its *transaction scheduler* may perform poorly on large graph analytical processing due to the overhead in doing concurrency control over many vertex transactions. Many large graphs in real applications are *scale-free graphs, i.e., whose degree distribution (for vertices) follows a power law* [6]. In this section, we discuss the properties of scale-free graphs and investigate the impacts of pow law graphs have for the transaction scheduler in fine-grained parallelism.

4.1 Large Scale-Free Graphs

Consider the `twitter-mpi` graph which is a follower network in Twitter, crawled by MPI in 2010, and represents friendships among twitter users. It has more than 52 million vertices (twitter users) and around 2 billion edges (follower-followee). Its degree distribution is shown in Fig. 7(a), where the *x-axis* is the vertex degree, and *y-axis* is the number of vertices with a particular degree value. Both axises are in *log scale*. This is clearly a *power law* distribution. We divide vertices into buckets. In Fig. 7(b), we show the percentage of vertices, and the percentage of edges *(reflects the sum of the number of vertex neighbors, i.e., total degrees of vertices)*, that fall in a bucket in the graph, where a vertex with degree d is assigned to the $\lfloor 2\log_{10}(d) \rfloor$-th bucket. Note that the number of neighbors for a vertex v is a good indicator for the amount of updates and the workload (assuming that the time complexity of UDF is linear) of transaction $t(v)$ in one execution. Clearly, most vertices fall into buckets with small degrees yet the total number of neighbors for these buckets is small compared to those buckets with larger degree vertices (even though very few vertices are in those buckets).

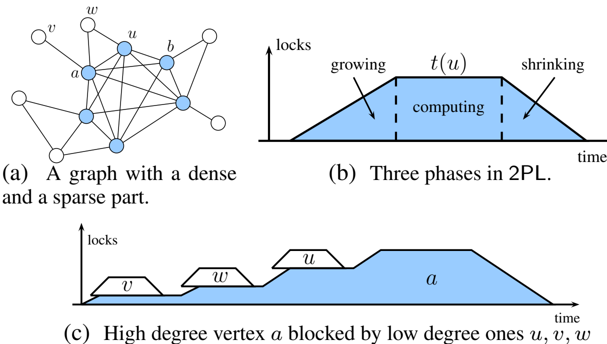
Next, we investigate the likelihood of conflicts by showing the probability of conflicts of two vertex transactions, one from each



(a) Degree distribution (b) Number of vertices and sum of their degrees in each bucket

(c) Conflict rate in RM (d) Conflict rate in RW

Figure 7: The impacts of scale-free graphs.



(a) A graph with a dense and a sparse part. (b) Three phases in 2PL.

(c) High degree vertex a blocked by low degree ones u, v, w

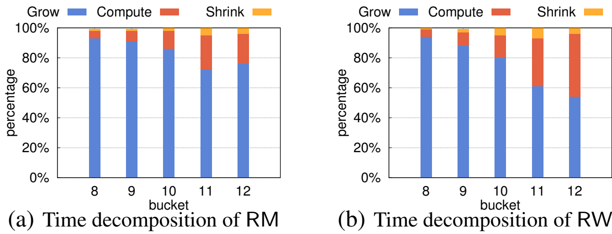
Figure 8: Inefficiency of 2PL.

of every two buckets, for Read Mostly (RM) and Read Write (RW) workloads in Fig. 7(c) and Fig. 7(d) respectively. In Fig. 7(c), the RM workload asks every transaction $t(v)$ writes vertex v and reads its neighbors. In Fig. 7(d), the RW workload asks every transaction $t(v)$ writes vertex v as well as its neighbors. Details of these workloads can be found in Section 6. As can be seen in Fig. 7(c), if two transactions in RM are from buckets 0 to 7, their conflict rates are under 1%. For transactions on vertices with larger degrees, their conflict rates could be as large as 20% even for RM workloads. Conflict rates are higher for RW workloads as shown in Fig. 7(d). Consider two transactions in bucket 3, where transactions are over vertices with a degree about 30. The conflict rate is around 30%. For large degree vertices, the conflict rate could be as high as 100%.

Clearly, the higher the degree of the two vertices, the more likely their transactions are going to conflict, and the conflict rates of RW are much higher than that of RM.

4.2 Limitations of Existing Schedulers

Several existing designs are available to serve as the *transaction scheduler* in the fine-grained parallelism framework as presented in Section 3. Unfortunately, when being applied over large scale-free graphs, they suffer from various limitations as we will investigate next. We use a toy example (Fig. 8(a)) to represent a large scale-free graph. This graph G has a large fully connected component consisting of blue vertices. Based on our analysis from last subsection, the conflict rates among vertex transactions for blue vertices are going to be very high, whereas the conflict rates among white vertices will be fairly small. Note that recent research [34] confirms



(a) Time decomposition of RM (b) Time decomposition of RW

Figure 9: 2PL **on** `twitter-mpi`**:** RM **and** RW**.**

that large scale graphs have both a dense part and a sparse part that are similar to this graph G.

Locking based schedulers. We use two phase lock (2PL) as the representative for locking-based schedulers. 2PL is used in one of the GraphLab's engines [33]. 2PL requires locks to be obtained for all vertices that the transaction needs to access. If all lock requests are successful, the transaction proceeds. Otherwise the transaction must wait till it can acquire these locks.

As illustrated in Fig. 8(b), there are three phases in a transaction following 2PL. The transaction requests all locks in the *growing phase*, and releases all locks in the *shrinking phase*. The computation is performed between the growing and shrinking phases.

In multi-core in-memory graph analytics, the locking overhead becomes a significant overhead. A vertex v with a high degree may require as many as millions of locks in a large graph. There is a very high probability that transaction $t(v)$ will be blocked by the locks held by some of its neighbors and their neighbors. In Fig. 8(c), it shows that the growing phase of $t(a)$ can be very expensive. In particular, a has many white vertices as its neighbors and the lock requests by $t(a)$ on these vertices are likely to be blocked by transactions running concurrently over these white vertices (which have low degrees).

Using both the RM and RW workloads on the `twitter-mpi` graph, we measured the total clock time (over all vertex transactions) of the three phases in Fig. 9 for buckets 8 to 12 (refer to Fig. 7(b)). The growing phase for requesting locks accounts for 60% to 90% of the overall processing time, which is too high.

Optimistic schedulers. In an optimistic scheduler like OCC [28], a transaction $t(v)$ does not lock vertices it needs to access, and is allowed to execute while all its reads and writes are being logged. When $t(v)$ is ready to commit, the scheduler will replay the log to ensure all its operations are still valid in order to commit: for all read operations the current values for objects read by $t(v)$ should be same as the values in the log for those reads; for all write operations the values written by $t(v)$ should not have been overwritten. If this validation succeeds, $t(v)$ commits; otherwise, the scheduler aborts $t(v)$ (even with only one violation) and $t(v)$ will be queued again for execution.

Compared to locking based schedulers, the optimistic scheduler does not block transactions. Thus, the overheads on locking are eliminated. On the other hand, if two transactions conflict with each other, at least one of them will be forced to abort and redo all the work. This is bad news when we apply OCC for the fine-grained parallelism framework on large scale-free graphs, as shown in Figures 7(c) and 7(d), the conflict rates can be very high especially for the dense part of a graph like the graph G.

TO schedulers. Timestamp ordering (TO) schedulers assign a timestamp for each transaction (as ts_v) and each data item (as TS_v). Usually the transaction's timestamp is assigned as the time it starts. During the execution of the $t(v)$, each read/write operation is allowed to operate on an item only if the timestamp of $t(v)$ satisfies a condition determined by the data item's timestamp.

TO scheduler suffers from the following issues over vertex transactions in large scale free graphs. Suppose $t(v)$ reads u. It can perform the read *if and only if* $ts_v \geq TS_u$, where $TS_u = max\{ts_w \mid t(w)$ writes u, $t(w)$ has committed$\}$. Otherwise $t(v)$ will have to abort. In large scale free graphs the degree of v could be millions. It is very likely that one of its neighbors has been overwritten by another $t(w)$ which starts later than $t(v)$ but finishes earlier than $t(v)$. Thus $t(v)$ on vertices with large degree is very likely to abort.

MVCC and SI schedulers. Multi-version concurrency control (MVCC) [63, 30, 31, 9, 44, 27, 13, 53, 32] increases transaction throughput, especially for read-only transactions, by keeping multiple versions of the same object (each update creates a new version). With extra storage cost, the strength of MVCC is to not block any read-only transactions. Snapshot isolation (SI) works on the snapshot of data instead of up-to-date values. Grace [50] is a graph database that supports snapshot isolation, but it does not provide serializable schedule. Serializable snapshot isolation (SSI) [14] builds on top of SI, with an extra component for correcting the errors/anomalies (w.r.t. serializability) brought by SI.

These schedulers work better than other single-versioned schedulers when there are *many read-only transactions*: they handle non-read-only transactions via other schedulers like 2PL and read-only transactions with snapshots. If there are no read-only transactions, these schedulers degenerate to 2PL. However for most analytical graph processing workloads using fine-grained parallelism, most if not all vertex transactions are NOT read-only. A graph analytics job often needs to update values associated with vertices throughout its computation, e.g., PageRank in Figure 6. Thus, MVCC, SI and their variants are not likely to help in this situation.

Chromatic scheduler. First proposed in [33], chromatic scheduler ensures data consistency in graph analytics by *assigning colors to vertices*. Assume that UDF(v) reads/writes vertex v and (only) reads its neighbors (RM), chromatic scheduler colors vertices so that adjacent vertices have different colors. The execution of UDFs consists of the following recursive phases: in each phase, only vertices in one particular color are active (note that UDF(v) may add one or more vertex UDFs back to the queue). Coloring ensures that active workers for computing vertex UDFs will not interfere with each other. This idea is further developed in [22, 21].

Compared to the *transaction scheduler* in fine-grained parallelism, all transaction scheduling overheads are eliminated in this approach. However its performance often suffers from the large number of colors that are needed in real large graphs. It is observed that the large scale graphs or networks usually have one or more dense subgraphs. Say the graph G has a k-clique where any two vertices in this clique are connected. Using the chromatic scheduler, each vertex in the k-clique shall have a unique color. Therefore we have at least k colors for graph G. Chromatic scheduler will have to schedule UDFs for vertices in the clique to run *sequentially*. But a typical value of k could range from several hundreds to tens of thousands in a large graph. Sequential execution on this large sub-graph drastically degrades the performance of chromatic scheduling. Furthermore, for RW workloads, where each vertex reads and writes itself and its neighbors, the graph shall be colored *such that each vertex's color is different from the colors of its 2-hop neighbors*. This is a hard problem to solve, since for each vertex we shall inspect its 2-hop neighbors and the total number of 2-hop neighbors over all vertices is a large search space. The cost of maintaining a valid coloring in case of incorporating changes to the underlying graph is significant too. Hence, it was once employed in GraphLab [33], but has been replaced by a locking-based scheduler like 2PL.

Remarks. Our analyses reveal that existing schedulers are not

Algorithm 1 *TX-START* (v)

Require: vertices values in `D[]` ▷ Graph data
Require: vertices version in `ver[]` ▷ Meta-data for scheduling
Require: vertices locks in `L[]` ▷ Fine-grained locks

1: **if** $d(v) \geq$ **`threshold`** τ **then**
2:     `BIG` $\leftarrow$ **`true`**; *TX-START-B* (v);
3: **else**
4:     `BIG` $\leftarrow$ **`false`**; *TX-START-S* (v);
5: set `version` = concatenation of `ThreadID` and `counter`;
6: `counter++`;

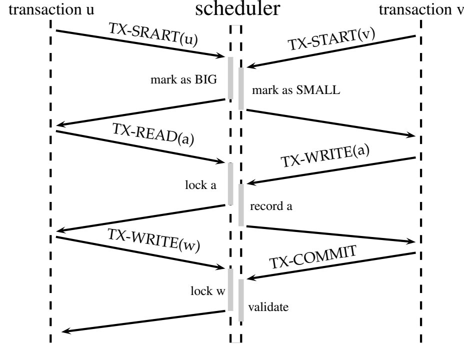


Figure 10: Illustration of transaction-scheduler interaction

likely to perform well for vertex transactions in a large scale-free graph. A better design is to be able to adapt to high conflict rates on high degree vertices and low conflict rates on low degree vertices.

5. HSync: A HYBRID SCHEDULER

The above discussion shows none of the existing schedulers will perform well for graph analytics over large scale-free graphs. This inspires us to design a hybrid scheduler, HSync, which combines the advantages of 2PL and OCC by routing vertex transactions to an appropriate scheduler based on the degrees of the vertices. Intuitively, there are a few high degree vertices in a large scale-free graph and the UDFs on these vertices are more costly to compute, thus we want to ensure that they do not abort using 2PL. On the other hand, there are many low degree vertices and we want to ensure that they don't create too many conflicts that will block the execution of UDFs for high degree vertices, and aborting these vertex transactions is not expensive since low degree vertices have small computation workload. Thus, we use OCC for these vertices.

A hybrid approach. We take a hybrid approach HSync, which uses 2PL for the transactions over *big* vertices and OCC for the transactions over *small* vertices. A vertex v is a big vertex if its degree $d(v)$ is greater than or equal to a threshold τ; otherwise, it is a small vertex. Using HSync in our fine-grained parallelism framework, a UDF on the vertex v initiates a vertex transaction $t(v)$ by *TX-START*, and ends the transaction $t(v)$ by either *TX-COMMIT* or *TX-ABORT*.

We show *TX-START* in Algorithm 1. It checks the degree of the vertex v, if $d(v) \geq \tau$, it will call *TX-START-B*; otherwise it will call *TX-START-S*. A transaction is assigned a unique identifier by concatenating the `ThreadID` of a thread (which is unique) and a local `counter` variable. Note that with the help of the `BIG` variable, all subsequent transaction procedures, e.g. *TX-COMMIT*, will call a procedure with *-B* suffix for big vertices, and a procedure with *-S* suffix for small vertices. That said, procedures for big vertices are shown in Algorithm 2, and procedures for small vertices are shown in Algorithm 3. In all these procedures, the variables `D[]`, `ver[]`, and `L[]` refer to the same data structures in

Algorithm 2 Transaction Processing for Big Vertices

1: **procedure** *TX-START-B*(v)
2:     `locks` $\leftarrow \emptyset$; ▷ Reset locks

3: **procedure** *TX-READ-B*(v)
4:     request lock `L`$[v]$ for *read* mode; ▷ Request lock first
5:     `locks` $\leftarrow$ `locks` $\cup \{v\}$;
6:     **return** value of vertex v;

7: **procedure** *TX-WRITE-B*$(v, value)$
8:     request lock `L`$[v]$ for *exclusive* mode; ▷ Request lock first
9:     `locks` $\leftarrow$ `locks` $\cup \{v\}$;
10:     `D`$[v] \leftarrow value$;
11:     `ver[v]` $\leftarrow$ `version`; ▷ Update the version

12: **procedure** *TX-COMMIT-B*()
13:     **for** v in `locks` **do** ▷ Release locks
14:         release the lock `L`$[v]$;

15: **procedure** *TX-ABORT-B*()
16:     **for** v in `locks` **do** ▷ Recover and release locks
17:         recover the value of v in `D[]` and `ver[]` from the log;
18:         release the locks `L`$[v]$ if held;

Algorithm 3 Transaction Processing for Small Vertices

1: **procedure** *TX-START-S*(v)
2:     `reads` $\leftarrow \emptyset$;
3:     `writes` $\leftarrow \emptyset$;

4: **procedure** *TX-READ-S*(v)
5:     **if** v found in `writes` **then** ▷ Find the vertex in written log
6:         **return** `writes`$[v]$;
7:     **if** v **not** in `reads` **then** ▷ Add the read access to log
8:         `reads` $\leftarrow$ `reads` $\cup \{(v,$ `ver`$[v])\}$;
9:     **return** `D`$[v]$;

10: **procedure** *TX-WRITE-S*$(v, value)$
11:     `writes` $\leftarrow$ `writes` $\cup \{(v, value)\}$ ▷ Append to log

12: **procedure** *TX-COMMIT-S*() ▷ Validation phase
13:     **for** v in `writes` **do** ▷ Request locks for modification
14:         try-request lock `L`$[v]$ for *exclusive* mode;
15:         **if** fails **then**
16:             *TX-ABORT-S*();
17:     **for** v in `reads` **do** ▷ Verify read access
18:         **if** `ver`$[v] \neq$ `reads`$[v]$ **or**
19:             v locked by another thread exclusively **then**
20:             *TX-ABORT-S*()
21:     **for** v in `writes` **do** ▷ Update data and version
22:         `D`$[v] \leftarrow$ `writes`$[v]$;
23:         `ver`$[v] \leftarrow$ `version`;
24:     **for** v in `writes` **do**
25:         release the locks `L`$[v]$ held;

26: **procedure** *TX-ABORT-S*()
27:     **for** v in `writes` **do**
28:         release the locks `L`$[v]$ if held;

Algorithm 1. Basically, the procedures for big vertices follow the 2PL protocol, and the procedures for small vertices follow the OCC protocol. But they share data values, data versions, and data locks. Note that a data version here *does not* represent a multi-version data, rather it simply *shows which transaction has the latest update to the data value in concern.*

User is unaware of such a hybrid approach, nor her involvement is needed. The system decides automatically if a vertex transaction is big or small based on the threshold τ. In other words, the fine-grained parallelism API always calls, for example, *TX-READ*, but not *TX-READ-B* or *TX-READ-S* (and similarly for other API calls

to any transaction procedure). A workflow chart that illustrates the interaction between two transactions and the HSync scheduler is shown as Fig. 10. In this example, when transaction u enters the system, the scheduler marks it as BIG, so for following operations issued from transaction u, HSync will request locks on the corresponding vertices. On the other hand, for transaction v, the scheduler marks it as SMALL, and simply records its write operations `TX-WRITE(a)` on its private write log without requesting any locks and verify if they are still valid at the time of commit.

The details of HSync. We first discuss the procedures for big vertices (Algorithm 2). In *TX-START-B*, it initializes the `locks` to empty. In the growing phase of 2PL, a transaction needs to lock all vertices it tends to access, using shared locks for read access and exclusive locks for write access. In *TX-COMMIT-B*, it releases all locks it has held. All operations are recorded in a log for recovery during transaction abort. In *TX-ABORT-B*, it recovers and restores the value and the version of v back into `D[]` and `ver[]` using the old values kept in the log before releasing the locks held.

Note that in our implementation of 2PL in Algorithm 2, we did not use the traditional data structures used in the lock manager for disk-based RDBMS [63, Chapter 10]. Traditional scheduler registers the transaction $t(v)$ on the waiting list of the lock, suspends the $t(v)$ and switches to another transaction (when $t(v)$ can not acquire a lock). The cost of doing so is relatively small since a dominant cost in disk-based RDBMS is disk I/Os. However, for in-memory graph analytics, the cost of context switching and the cost associated with cache misses hurt the performance significantly [18, 47, 43]. Hence, we use the spinlock instead, i.e., each vertex has a light-weight spinlock to indicate if it is currently locked by a transaction or not. The transaction simply waits for the lock release, when a lock can not be acquired. We resolve deadlocks using deadlock prevention: since the lock set for a vertex transaction for vertex u in a graph analytical workload is known before the execution of the transaction, which is a subset of u's neighbors (may include u itself), we request locks following a global ordering of locks. In our implementation we used the vertex ids as this global ordering for locks. Note that similar approaches have also been adopted in other systems [51, 43, 58].

We next discuss the procedures for small vertices (Algorithm 3). Recall that in OCC, a transaction keeps a local set `reads` to maintain all reads executed, and a local set `writes` to maintain all writes executed. OCC allows all reads/writes to be executed without locking. It will check their validity at the validation phase when the transaction is about to commit. In *TX-START-S*, it initializes both `reads` and `writes` to empty. In *TX-READ-S*, it first checks if v had been updated by the current transaction. If so, it returns the updated value of v from `writes[`v`]`. If v is not in `writes`, it further checks if it is already recorded in `reads`, returns the value kept in `D[`v`]` if not and records it in `reads`. In *TX-WRITE-S*, it keeps the newly updated value in `writes`, and *does not update* the value in `D[`v`]`.

The validation phase is done in *TX-COMMIT-S*. *TX-COMMIT-S* will abort the transaction by calling *TX-ABORT-S*, if one of the following conditions is met: 1) the transaction cannot get exclusive locks for all objects in `writes`. Here the lock requests are non-blocking, i.e., if a lock cannot be acquired, the transaction thread will not wait for its availability, and will return fail and abort the transaction immediately. If all locks are successfully obtained, it will update `D[`v`]` for every vertex v in `writes` before releasing any locks; 2) the transaction finds a mismatch between the version of a vertex v in its `reads` and the version of the vertex from `D[`v`]`. This implies that another transaction has updated v since v was read by the current transaction. Lastly, *TX-ABORT-S* simply releases all locks held by the current transactions.

In traditional implementation, the validation of a transaction in OCC is a non-interruptible critical section, and all other transactions will be declined to enter validation if there is an actively running critical session of a transaction. This is not efficient nor scalable for in-memory graph analytics over large graphs. We adopt an optimization that is similar to Silo [58]. In the validation phase, the transaction requests locks on vertices. If the transactions are less likely to conflict, which is the case for transactions over small vertices, the locks will be in the cache of the workers of these transactions during the whole validation phase. There are two ways to validate a transaction in OCC [63]. One is BOCC (backward-oriented optimistic concurrency control), and the other is FOCC (forward-oriented optimistic concurrency control). In our system, we use FOCC which validates a transaction against all the transactions that run in concurrent but have not committed yet.

To reduce overhead while integrating OCC with 2PL, we let small transactions access vertices' values where possible and then request all locks before their validation phases. Due to the low conflict rates for the small transactions as they do not have a large number of neighbors (see Figures 7(c) and 7(d)), it is likely that most small transactions will successfully commit without being aborted. The main advantage of using OCC for small transactions rather than using 2PL is that a small transaction has a chance to commit much early. This is because that the time taken from requesting locks to releasing locks in OCC is very small (especially for small transactions) which means they will not interfere with the execution of big transactions under 2PL too much. In addition, in the worst case, if a small transaction gets aborted, it is very inexpensive to re-execute it and let it try to commit again.

Note that in HSync, 2PL and OCC share the same locks. 2PL uses locks for transaction scheduling, and OCC utilizes locks to replace the usage of critical section as discussed above and prevents other transactions to write concurrently. By sharing the same set of locks, small transactions under OCC acquire locks to prevent potential data racing from both OCC and big transactions under 2PL. Although small transactions under OCC may request locks that may conflict with big transactions, they hold these locks for very minimum time. Compared to the 2PL-only scheduler, our hybrid scheduler essentially ensures that big transactions have higher priorities against small transactions.

Correctness. We briefly discuss the proof sketch of our theorem showing that the hybrid scheduler HSync ensures serializable. Full details can be found in appendix.

Theorem 1 *HSync permits only conflict serializable schedules.*

We define the *commit time* as the time when a transaction decides to commit. Transactions satisfy **commit order-preserving** (CO) if for any two transactions $t(i)$ and $t(j)$ that are in conflict, $t(i)$'s conflicting operation is earlier than $t(j)$'s, and $t(i)$'s commit time is also earlier than $t(j)$'s. Our proof goes by case analysis; we show that under the hybrid scheduler HSync, any two transactions that are in direct conflict with each other will guarantee to satisfy CO. We enumerate four cases that both $t(i)$ and $t(j)$ could be scheduled by 2PL or OCC. Based on this, we show that all transactions' conflict relationships are ordered by their commit time which has a total order. Therefore there must be no cycles of conflicts in HSync, which is a sufficient condition for conflict serializability. ■

We take one step forward and proved that under some reasonable assumptions, our approach *achieves the optimal concurrency for any two schedulers to form a hybrid scheduler*, where the degree of concurrency is measured by the possible transaction execution

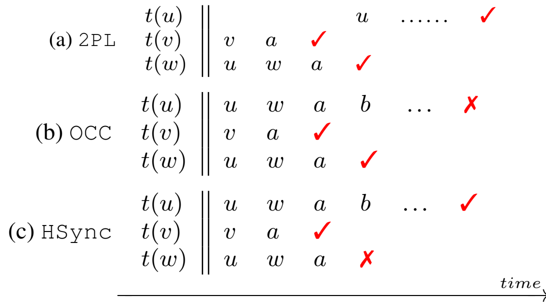


Figure 11: An example using three different schedulers.

traces allowed by the scheduler [63]. The details of this discussion and its proof could be found in appendix.

An example. We show the advantages of our hybrid scheduler by illustrating the execution of three transactions from the graph in Fig. 8(a). Assume three transactions, namely $t(u)$, $t(v)$ and $t(w)$, where u is a big vertex , and v and w are small vertices. We assume all operations are `write`, and their operations follow the order as shown in Figure 11.

In 2PL, a transaction can easily be blocked by other transactions. In this case, transaction $t(w)$ competes for lock on vertex u with transaction $t(u)$. Assume $t(w)$ acquires the lock successfully, then $t(u)$ has to wait for $t(w)$'s completion. In the toy example u has only six neighbors, but in a real large graph, a big vertex like u may have millions of neighbors. Thus, during the execution of a transaction on a big vertex like u, it may be blocked numerous times. To make the situation even worse, it is possible that $t(u)$ has to wait for a lock near the end of its growing phase, while it already holds millions of locks. Many other transactions cannot proceed if they require any one of these locks. In short, a small transaction may block a big transaction, which further blocks millions of small transactions. This exacerbates the congestion among transactions and may even stall the whole system. The failure of 2PL is caused by the fact that big transactions are blocked by small transactions.

For OCC, two small transactions are able to commit but the big transaction $t(u)$ has to abort in its validation phase. The committed transaction $t(v)$ "contaminates" the value read by $t(u)$ and so does $t(w)$. This is a great waste considering such big vertex u may have millions of neighbors in a large graph.

For our HSync approach, $t(u)$ is never blocked nor interrupted by $t(v)$ and $t(w)$. For data racing between $t(u)$ and $t(w)$, $t(u)$ wins due to the hybrid policy: $t(w)$ is scheduled by an optimistic scheduler and it does not acquire any lock; therefore it will not block $t(u)$, nor will it contaminate the value read by $t(u)$ because all modifications it made are recorded in its local private log. During $t(w)$'s validation phase, it will find out that the vertex a's lock is held by another transaction ($t(u)$) so it aborts. This will not affect the execution of $t(u)$ at all. But aborting and re-executing a small transaction is not costly. On the other hand, $t(v)$ is still able to finish and commit before $t(u)$ produces conflicts with it.

Optimization and extension. When a small transaction fails to commit, by default it will be rescheduled for another execution. Most existing optimistic schedulers adopt this policy because they assume that the chance of having conflicts during validation is small and re-trying several times is sufficient for the transactions to eventually progress. However in our problem, although the conflicts among small transactions are indeed infrequent, the rate for a small transaction to be in conflict with a big transaction that is concurrently running is not negligible, and a big transaction could be long standing as well due to its heavier workload. Thus, simply retrying the small transaction is not necessarily a good solution. To address this issue, we adopt a new policy that when a small transaction has failed to commit too many times, we will execute it as a big transaction (using 2PL).

Our discussion so far assumes that the graph edges have no values and they are not involved in the analytics. If they do, we propose two extensions to handle them. Taking big transactions as example (small transactions are handled similarly), the first method is to treat edges as vertices as well. Each edge e has a fine-grained lock `L[e]`. When a big transaction tries to access e, it requests its lock. The second method does not assign a lock to an edge e. Instead, if a big transaction tries to access $e = (u, v)$, it must own both locks for u and v. The first method has a finer locking granularity, and hence, potentially better concurrency, while the second method saves space and reduces the number of locks to manage.

In contrast to classic OLTP environments like online banking, graph analytics, like other OLAP workloads, does not require durability: in case of a system crash, it is safe to restart the computation from the beginning. Thus, HSync eliminates write-ahead-logging at per-operation level (which is only needed for durability). Rather, it periodically records a snapshot of states as a checkpoint for more efficient fault recovery [63]. Users may adjust how frequent the checkpointing process is in HSync. Such fault tolerance avoids re-execution from the beginning in case of a crash.

As shown in our experiments, HSync is not very sensitive to the choice of τ (i.e., there is a relatively wide safe region to choose the value of τ from to define BIG and SMALL). That said, to facilitate choosing a good τ value, HSync uses a sampling approach. It executes the UDF on a small sample graph, sampled from the input graph using weighted sampling over vertices where the sampling weight for a vertex u is proportional to its degree, to estimate the vertex transaction conflict rates and a heat map similar to Fig. 7(c) and 7(d). The value of τ is then chosen to separate the intense conflict and occasionally conflict regions.

Dataset	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert E\rvert/\lvert V\rvert$	Size
`sk-2005`	50,636,154	1,949,412,601	38.50	17G
`twitter-mpi`	52,579,682	1,963,263,821	37.33	18G
`wdc-sd`	89,2472739	2,043,203,933	22.89	20G
`uk-2007-05`	105,896,555	3,738,733,648	35.31	33G

Table 1: Four real graph datasets

6. EXPERIMENTS

Experiment Setting: We have conducted extensive experiments on Amazon EC2 `c4.8xlarge` instance which has 36 cores and 60GB memory. All experiments were repeated 10 times with the average value being reported. We implemented all methods in C++.

Real large graphs: We used 4 real large graph data sets in the experiments. The dataset `twitter-mpi` is a follower network from Twitter, crawled by MPI in 2010.[2] The dataset `wdc-sd` is a hyperlink graph crawled in 2012.[3] All other data sets are large social networks and web page graphs used in different domains which can be downloaded from WebGraph.[4] Table 1 summarizes these datasets. The number of vertices of these graphs are in the range from 50 million to 106 million, the numbers of edges are in the range from 1.9 billion to 3.7 billion, where the average degrees are in the range of 22.89 and 38.50. The sizes of these graphs are from 17 gigabytes to 33 gigabytes, assuming that a graph is represented using the adjacency lists. We also conducted experiments on synthetic graphs at Appendix D.

[2] http://twitter.mpi-sws.org/
[3] http://webdatacommons.org/hyperlinkgraph/
[4] http://law.di.unimi.it/datasets.php

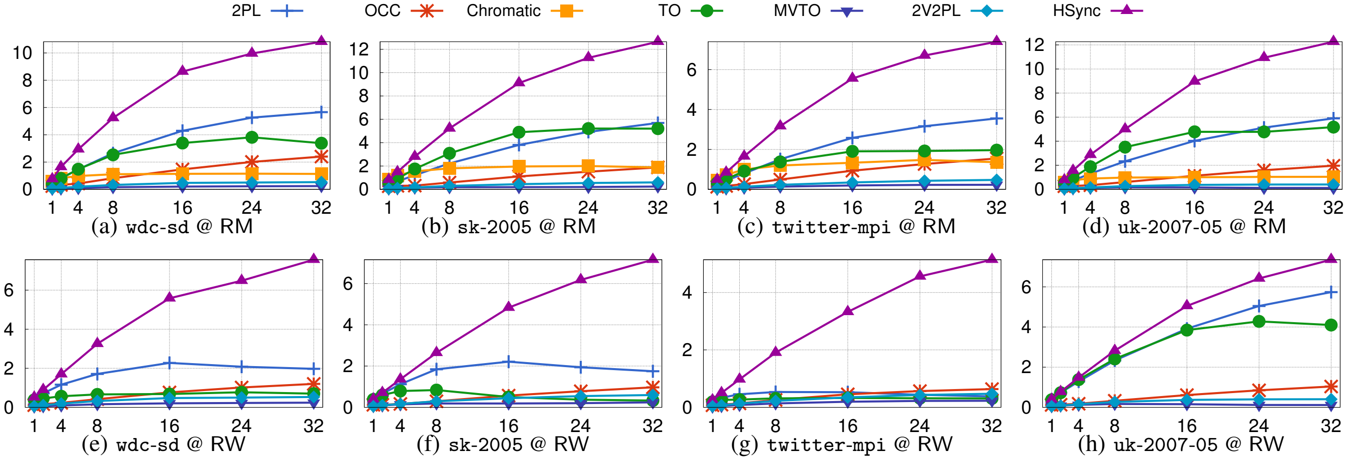


Figure 12: Throughput on RM and RW *(x-axis for number of cores, y-axis for throughput ($\times 10^6$ tps))*

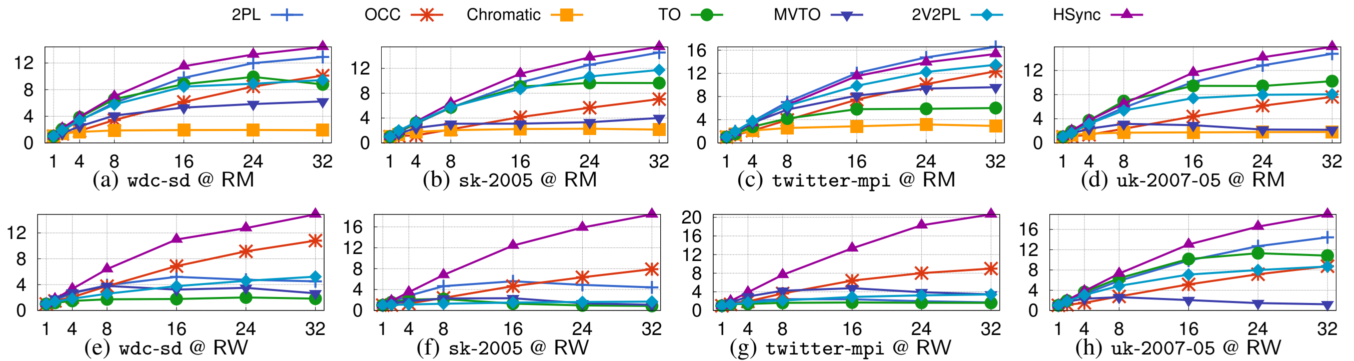


Figure 13: The scalability on RM and RW (normalized as the ratios to the throughput using 1 core)

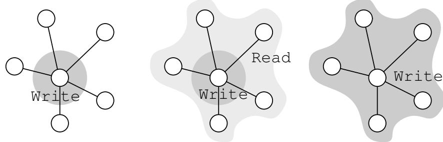


Figure 14: Different workload behavior: WO, RM, RW

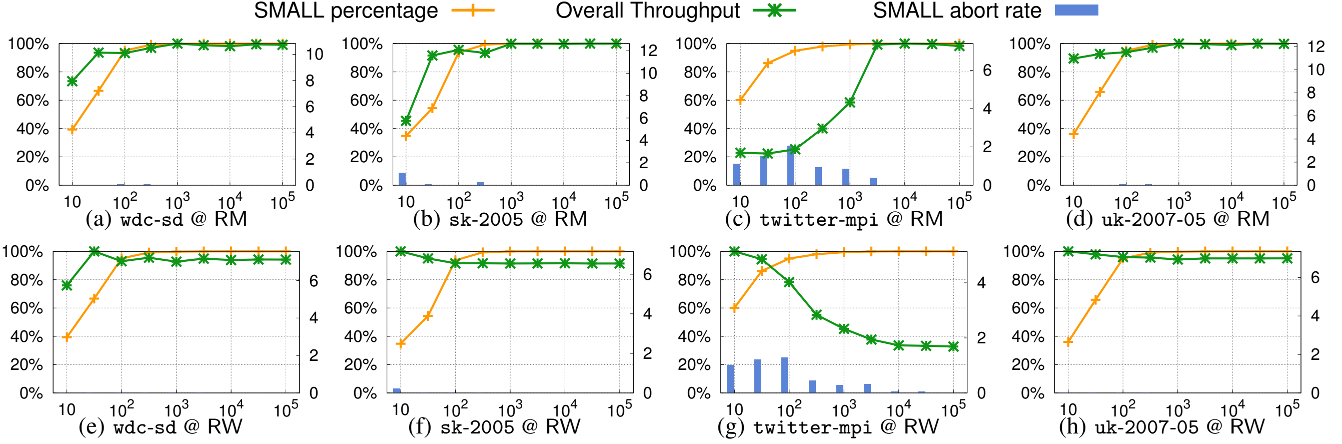
Various Workload Behavior: In practice, for a vertex transaction in $t(v)$, v itself and its neighbors usually play different roles: for example, in some cases we only need to read v's neighbors but write to v. Depending on the amount of different operations, there are three possible modes of vertextransactions, shown in Fig. 14, namely *(i)* write to v; *(ii)* write to v and read from neighbors; *(iii)* write to v and neighbors. A similar consistency model has been proposed by GraphLab in [33], with data item on edges. We name these different workloads WO (Write Once), RM (Read Mostly), RW (Read Write). Among these three, WO is not interesting as it degrades to a key-value database.

6.1 Schedulers Performance

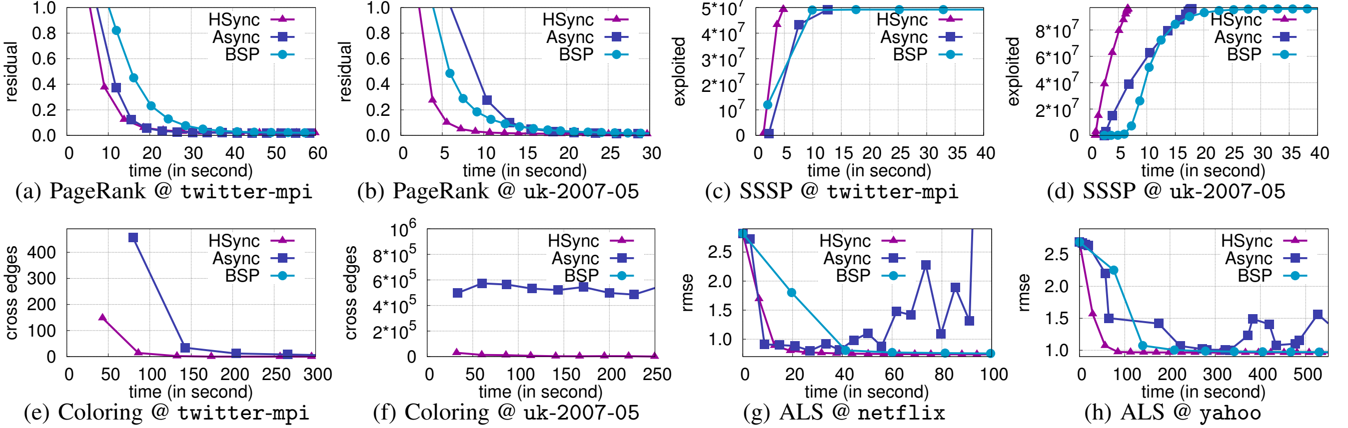
We first study the performance of different schedulers by measuring the throughputs of vertex transactions. The throughput of a scheduler is defined as the number of *t*ransactions successful committed in every *s*econd (*tps*). We define the performance gain as the ration between the throughput of the HSync scheduler vs. *the best throughput of all other schedulers*. Another important metric is the scalability. It measures the improvements on throughput when we increase the computing resources, i.e., the number of cores. To make a fair comparison, each scheduler is assigned to schedule the same set of workloads, and the throughput is measured by the wall clock time taken for each scheduler from start to the completion of last worker. For the RM workload, the sequential greedy coloring algorithm for the chromatic scheduler takes several hours for each graph. For the RW workload, chromatic scheduler is not even available because computing the appropriate graph coloring didn't finish after running for more than 100 hours for each dataset (it needs to explore 2-hop neighbors in this case).

Throughput. The throughput of transaction schedulers are shown in Fig. 12, for workload RM and RW. For RM, with 32 cores, HSync's performance gain ranges from 1.91 to 2.23 for different data sets (note that this is against the best performance from all other schedulers), with an average performance gain of 2.07. For workload RW, HSync's performance gain is from 1.28 to 8.2, with a an average gain of 3.57. In short HSync is on average 2 to 4 times more efficient than the best state-of-the-art graph vertex transaction schedulers. We note although our scheduler performs well on all data sets, the throughput gains on uk-2007-05 are less ideal. For this dataset, the throughput of 2PL is much better than OCC for both workloads. It indicates that this graph is much denser than other graphs. In a denser graph, the transactions are more likely to conflict and the throughputs of all schedulers are limited by the severe degree of conflicting operations.

Scalability. We consider the scalability of different transactions up to 32 cores. The results are shown in Fig. 13. Recall that this is defined as the throughput value of a transaction at 32 core divided by the throughput value for the same transaction at 1 core. For RM, the scalability of HSync ranges from 14.44 to 15.92, with an average 15.28, for the four data sets tested. The second best scheduler on RM, which is 2PL, only has an average scalability of 14.64. For workload RW, the scalability of HSync ranges from 14.88 to 20.65, with an average 18.1 for the same four data



(a) wdc-sd @ RM (b) sk-2005 @ RM (c) twitter-mpi @ RM (d) uk-2007-05 @ RM



(e) wdc-sd @ RW (f) sk-2005 @ RW (g) twitter-mpi @ RW (h) uk-2007-05 @ RW

Figure 15: The effect of degree threshold τ *(x-axis for degree threshold τ, right y-axis for throughput ($\times 10^6$ tps))*

(a) PageRank @ twitter-mpi (b) PageRank @ uk-2007-05 (c) SSSP @ twitter-mpi (d) SSSP @ uk-2007-05

(e) Coloring @ twitter-mpi (f) Coloring @ uk-2007-05 (g) ALS @ netflix (h) ALS @ yahoo

Figure 16: Four graph analytics workloads: PageRank, SSSP, graph coloring and ALS.

sets. The second best scheduler on RW, which is OCC, only has an average scalability of 9.05.

Threshold τ in HSync. Our hybrid scheduler is almost parameter-free. The only parameter that may affect its performance is the threshold of degree τ, which determines whether a transaction is BIG or SMALL. We vary the value of τ from 10 to 100,000, and measure the overall throughput, the percentage of transactions that are processed by the SMALL transaction scheduler (OCC), the abort rate of SMALL and the lock conflict rate for BIG transactions respectively. The results are shown in Fig. 15.

These results show that HSync is relatively insensitive to the value of τ. The workload RM has lower conflict rates (see Fig. 7(c)), therefore it performs better when τ is large, i.e., most of the transactions are processed by OCC. Similarly the workload RW performs better for a lower threshold. Among these figures, the throughput is stable for the most part, when τ changes from 10^2 to 10^4. Compared to the web graphs, social network twitter-mpi is more sensitive to the choice of the parameter. It is due to its higher degree of skewness. In general, we can just use a large τ value for RM workloads and a small τ value for RW workloads respectively.

6.2 Impacts to Graph Analytics

To verify the advantage of fine grained parallelism using synchronous parallel processing (with HSync) against asynchronous parallel and BSP processing for graph analytics, we tested PageRank, single source shortest path (SSSP), graph coloring, and alternative least squares (ALS) for matrix factorization. We implemented each algorithm in these different models and presented how fast they proceed over time. For ALS, we used the Netflix data [2] and the Yahoo! Music User Ratings [25], with 100,480,507 and 717,872,016 reviews respectively. For the rest, we used the social network data twitter-mpi and the web graph data uk-2007-05.

We adopt an "early termination" heuristics for PageRank: a vertex is consider stable after its last value change is less than 10^{-2}. We measured the residual, defined as the absolute difference between a vertex's PageRank value and the sum of its incoming vertices' PageRank values, averaged against all vertices. Although all three models guarantee to produce correct results eventually, as seen in Fig. 16(a) and 16(b), synchronous processing based on HSync has outperformed the BSP and asynchronous counterparts (converging to smaller errors faster).

For SSSP, the edge weights are randomly generated (and then fixed). A random source is selected, and we report the average results over multiple runs. We measured the number of exploited vertices over time (higher is better). An interesting discovery in Fig. 16(c) and 16(d) is that BSP has a "long tail" while the other two terminate quickly. This is due to the fact that BSP has to pay non-negligible overheads even when only a very small fraction of vertices are still "active" (unexplored), due to its barriers. HSync runs faster than asynchronous processing, thanks to its transaction-style synchronization: for asynchronous processing, two active vertices running concurrently may exploit a unvisited vertex u simultaneously, and thus add u into its work queue more than once. This phenomena could occur frequently in the dense "core" area of the graph. Synchronous processing like HSync avoids such overheads.

Graph coloring is to assign the minimum possible number of colors to all vertices such that the adjacent vertices are assigned different colors. Figures 16(e) and 16(f) show how the number of

	PR		SSSP		WCC		Color		ALS	
Dataset	T	U	T	U	T	U	T	U	N	Y
PowerGraph	1268	381	58.8	90.2	197	210	m	m	392	4956
PowerLyra	662	m	37.5	m	107	m	m	m	164	1784
GraphChi	3778	2424	701	2077	589	3090	848	820	>3000	>8000
HSync	181	42.7	48.8	66.7	30.1	31.9	174	59.8	74.6	901

Table 2: Efficiency of different systems. (End-to-end wall clock time in seconds. T: `twitter-mpi`, U: `uk-2007-05`, N: `netflix`, Y: `yahoo`. m: memory limit exceeded.)

	PR		SSSP		WCC		Color		ALS	
Dataset	T	U	T	U	T	U	T	U	N	Y
PowerGraph	7.0	8.9	1.2	1.4	6.5	6.6	-	-	5.3	5.5
PowerLyra	3.7	-	0.8	-	3.6	-	-	-	2.2	2.0
GraphChi	20.9	56.8	14.4	31.1	19.6	96.9	4.9	13.7	>40.2	>8.9

Table 3: Efficiency compared with HSync, running time as a ratio to HSync's running time. (T: `twitter-mpi`, U: `uk-2007-05`, N:`netflix`, Y: `yahoo`)

edges that have violated this rule (two neighbors still have the same color) decreases over time. Note the BSP in this case is too expensive to converge, and its results have been ignored in these figures. Clearly, the fine grained parallelism using HSync converges much faster than asynchronous processing; the latter has difficulty to converge on dataset `uk-2007-05`, possibly due to its smaller diameter and dense "core" (such that inconsistent readings and assignments in asynchronous processing are hard to converge).

ALS can be formalized as a bipartite graph. ALS works by fixing the values on the right side and updating the left side, then vice visa. Note that there is *no* asynchronous parallel processing algorithm that is able to guarantee correct results for ALS [33]. That said, one can still execute a synchronous parallel processing algorithm *in an asynchronous manner, without requiring the correctness of the final output, to understand its behavior*. As seen in Fig. 16(g) and 16(h), where they show how the root mean square error (RMSE) reduces over time, the asynchronous parallel processing approach could not converge at all, whereas both BSP and synchronous parallel processing have shown much better performance. And fine-grained parallelism using synchronous parallel processing with HSync has clearly outperformed BSP.

6.3 System Performance Comparison

We compared the performance of fine-grained parallelism using HSync against other graph analytics systems, including PowerGraph [15], PowerLyra [4], and GraphChi [24], using PageRank, SSSP, weakly connected component (WCC), graph coloring and ALS. We conducted the PowerGraph and PowerLyra experiments with a 16 node Amazon EC2 `m3.2xlarge` (8 cores, 30GB memory) cluster, and GraphChi on a `r3.8xlarge` instance with 32 cores, 244GB memory and 320GB SSD. The memory budget for GraphChi is set to 200GB which is much larger than the dataset. The (one-pass) data preparation time and I/O time are excluded.

We report the end-to-end wall clock running time of each approach in Table 2, and each system's running time as a ratio to HSync in Table 3. HSync outperforms other systems by 3x to 100x folds in most cases. Note that PowerGraph and PowerLyra are cluster-based with 128 cores in total (from the cluster), and Graphchi and HSync run on a single node with 32 cores.

7. RELATED WORK

Many existing graph systems including Neo4j [38], Trinity [55], FlockDB [41], InfiniteGraph [40], Allegrograph [37], NetflixGraph [39], Titan [36] and HyperGraph-DB [19] support transactional graph processing at different levels. Trinity only provides the primitive functions for developers to lock and unlock vertices. DEX allows at most one read-write transaction, plus unlimited number of concurrent read-only transactions. Neo4j and Titan use standard two-phase lock (2PL) to implement the vertex transaction scheduler. The details of other systems are unknown due to lack of publications and the fact that they are not open sourced.

There are also works in database systems where they assume static workloads (in particular, the types of transactions are known a priori) and they rely on static analysis for improving transaction scheduler's performance. For example, H-Store [56, 18, 20] proposes transaction processing systems based on the partition of data such that local transactions do not conflict and they can run concurrently without concurrency control. Calvin [57] is based on a similar approach. These approaches rely on high quality partitions of the data that lead to little or no cross-partition transactions. After that each worker is responsible for processing the transactions in one partition. Thus, they need strong knowledge about the transactions to execute a priori. These approaches work well on business or e-commerce workloads like TPC-C and YCSB since the data can be easily partitioned. However large scale graphs tend to form a "hyper-ball" [61] and are much hard to partition [29] while ensuring little or no cross-partition vertex transactions.

Mixing concurrency control schemes have been explored before in the context of transaction processing in RDBMS [28, 49, 26] where schedulers that combine two or more CC schemes were designed. For example in real-time systems, 2PL is used for the transactions that must be answered in a timely manner, otherwise the system uses OCC [26]. However all these approaches employ *serializable graph testing* to ensure the correct interaction between multiple (e.g., 2PL and OCC) schedulers. This hurts scalability and throughput since testing serializable graph is very expensive. Our approach does not require additional mechanism to maintain the correctness between two different transaction schedulers.

Lastly, note that we have already reviewed the most closely related work on different schedulers and parallel processing models for graph analytics in Sections 2 and 4.2.

8. CONCLUSION AND FUTURE WORK

We introduce a new design for fine grained parallelism using synchronous parallel processing for graph analytics. By exploring the particular properties in large scale free graphs, we designed and implemented the HSync scheduler that shows excellent throughput and scalability for processing vertex transactions. HSync integrates 2PL with OCC, and uses 2PL to schedule vertex transactions from high degree vertices to avoid aborting them, and uses OCC to schedule vertex transactions from low degree vertices to reduce the overall overheads they may introduce to the system. A careful analysis and design show that the interaction between the two types of schedulers in HSync lead to correct results. Experimental evaluation confirms the advantage of HSync over other approaches.

As for future work, it is interesting to explore how to utilize the new hardware, specially hardware transactional memory (HTM), non-volatile random-access memory (NVRAM) and remote direct memory access (RDMA) for even better parallelism. Our focus here is in-memory multi-core architecture on a single node, but it is possible to extend our hybrid scheduler to a distributed in-memory cluster [11, 60] with distributed transaction processing [12, 62, 5] on top of fast network infrastructures like InfiniBand and RoCE (RDMA). In principle, distributed schedulers, like distributed 2PL and distributed OCC can be combined together in a similar fashion to derive a distributed version of HSync. Nevertheless, there are still significant research challenges as how to combine two distributed schedulers safely to satisfy our aforementioned necessary and sufficient condition to achieve conflict serializability, i.e., satisfying commit order-preserving.

Acknowledgments

The authors thank the helpful comments from the anonymous reviewers. The work was supported by grant of Research Grants Council of the Hong Kong SAR, China No. 14209314, The Chinese University of Hong Kong Direct Grant No. 4055048. Feifei Li was supported in part by NSF grants 1251019 and 1302663. Feifei Li was also supported in part by NSFC grant 61428204. We also thank the generous gift from AWS Cloud Credits for Research from Amazon.

9. REFERENCES

[1] R. Angles and C. Gutierrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1):1:1–1:39, Feb. 2008.

[2] J. Bennett, S. Lanning, and Netflix. The Netflix prize. KDD Cup and Workshop, 2007.

[3] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[4] R. Chen, J. Shi, Y. Chen, and H. Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. *EuroSys '15*, pp. 1:1–1:15, 2015.

[5] Y. Chen, X. Wei, J. Shi, R. Chen, and H. Chen. Fast and general distributed transactions using rdma and htm. *EuroSys '16*, page to appear, 2016.

[6] A. Clauset, C. R. Shalizi, and M. E. J. Newman. Power-law distributions in empirical data. *SIAM Rev.*, 51(4):661–703, Nov. 2009.

[7] T. David, R. Guerraoui, and V. Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. *SOSP '13*, pp. 33–48, 2013.

[8] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[9] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: Sql server's memory-optimized oltp engine. *SIGMOD '13*, pp. 1243–1254, 2013.

[10] S. N. Dorogovtsev, J. F. F. Mendes, and A. N. Samukhin. Structure of growing networks with preferential linking. *Phys. Rev. Lett.*, 85:4633–4636, Nov 2000.

[11] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. Farm: Fast remote memory. *NSDI '14*, pp. 401–414, Apr. 2014.

[12] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: Distributed transactions with consistency, availability, and performance. *SOSP '15*, pp. 54–70, 2015.

[13] J. M. Faleiro and D. J. Abadi. Rethinking serializable multiversion concurrency control. *Proc. VLDB Endow.*, 8(11):1190–1201, July 2015.

[14] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, June 2005.

[15] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. *OSDI '12*, pp. 17–30, 2012.

[16] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 1st edition, 1992.

[17] M. Grund, P. Cudré-Mauroux, J. Krueger, and H. Plattner. Hybrid graph and relational query processing in main memory. DESWEB '13, pp. 23–24, April 2013.

[18] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. *SIGMOD '08*, pp. 981–992, 2008.

[19] B. Iordanov. HyperGraphDB: a generalized graph database. WAIM '10, 2010.

[20] E. P. Jones, D. J. Abadi, and S. Madden. Low overhead concurrency control for partitioned main memory databases. *SIGMOD '10*, pp. 603–614, 2010.

[21] T. Kaler. Chromatic scheduling of dynamic data-graph computations. Master's thesis, Massachusetts Institute of Technology, 2013.

[22] T. Kaler, W. Hasenplaugh, T. B. Schardl, and C. E. Leiserson. Executing dynamic data-graph computations deterministically using chromatic scheduling. SPAA '14, 2014.

[23] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. Mizan: A system for dynamic load balancing in large-scale graph processing. *EuroSys '13*, pp. 169–182, 2013.

[24] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. *OSDI '12*, pp. 31–46, 2012.

[25] Y. Labs. http://webscope.sandbox.yahoo.com/catalog.php?datatype=r. [Online; accessed 6-Mar-2016].

[26] K.-Y. Lam, T.-W. Kuo, B. Kao, T. S. Lee, and R. Cheng. Evaluation of concurrency control strategies for mixed soft real-time database systems. *Information Systems*, 27:149, 2002.

[27] P.-A. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. *Proc. VLDB Endow.*, 5(4):298–309, Dec. 2011.

[28] G. Lausen. Concurrency control in database systems: A step towards the integration of optimistic methods and locking. *ACM '82*, pp. 64–68, 1982.

[29] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[30] J. Levandoski, D. Lomet, S. Sengupta, R. Stutsman, and R. Wang. High performance transactions in deuteronomy. CIDR '15, January 2015.

[31] J. Levandoski, D. Lomet, S. Sengupta, R. Stutsman, and R. Wang. Multi-version range concurrency control in deuteronomy. *Proc. VLDB Endow.*, 8(13):2146–2157, Sept. 2015.

[32] D. B. Lomet, A. Fekete, R. Wang, and P. Ward. Multi-version concurrency via timestamp range conflict management. ICDE '12, pp. 714–725, 2012.

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

[34] T. Maehara, T. Akiba, Y. Iwata, and K. Kawarabayashi. Computing personalized pagerank quickly by exploiting graph structures. *PVLDB*, 7(12):1023–1034, 2014.

[35] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. *SIGMOD '10*, pp. 135–146, 2010.

[36] Aurelius Inc. http://thinkaurelius.github.io/titan/.

[37] Franz Inc. http://www.franz.com/agraph/allegrograph/.

[38] Neo Technology, Inc. . http://www.neo4j.org/.

[39] Netflix Inc. https://github.com/Netflix/netflix-graph.

[40] Objectivity Inc. http://www.objectivity.com/infinitegraph.

[41] Twitter Inc. https://github.com/twitter/flockdb.

[42] B. Mcmahan and M. Streeter. Delay-tolerant algorithms for asynchronous distributed online learning. *In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger, editors,* NIPS '14, pp. 2915–2923, 2014.

[43] N. Narula, C. Cutler, E. Kohler, and R. Morris. Phase reconciliation for contended in-memory transactions. *OSDI '14*, pp. 511–524, 2014.

[44] T. Neumann, T. Mühlbauer, and A. Kemper. Fast serializable multi-version concurrency control for main-memory database systems. *SIGMOD '15*, pp. 677–689, 2015.

[45] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. *SOSP '13*, pp. 456–471, 2013.

[46] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for RAMClouds: Scalable high-performance storage entirely in dram. *SIGOPS Oper. Syst. Rev.*, 43(4):92–105, Jan. 2010.

[47] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *Proc. VLDB Endow.*, 3(1-2):928–939, Sept. 2010.

[48] C. Papadimitriou. *The Theory of Database Concurrency Control*. Computer Science Press, Inc., 1986.

[49] J.-F. Pons and J. F. Vilarem. Mixed concurrency control: Dealing with heterogeneity in distributed database systems. *VLDB '88*, pp. 445–456, 1988.

[50] V. Prabhakaran, M. Wu, X. Weng, F. McSherry, L. Zhou, and M. Haridasan. Managing large graphs on multi-cores with graph awareness. *USENIX ATC '12*, pp. 4–4, 2012.

[51] K. Ren, A. Thomson, and D. J. Abadi. Lightweight locking for main memory database systems. *Proc. VLDB Endow.*, 6(2):145–156, Dec. 2012.

[52] A. Rowstron, D. Narayanan, A. Donnelly, G. O'Shea, and A. Douglas. Nobody ever got fired for using hadoop on a cluster. *HotCDP '12*, pp. 2:1–2:5, 2012.

[53] M. Sadoghi, M. Canim, B. Bhattacharjee, F. Nagel, and K. A. Ross. Reducing database locking contention through multi-version concurrency. *Proc. VLDB Endow.*, 7(13):1331–1342, Aug. 2014.

[54] Z. Shang and J. X. Yu. Catch the wind: Graph workload balancing on cloud. *ICDE '13*, pp. 553–564, 2013.

[55] B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. *SIGMOD '13*, pp. 505–516, 2013.

[56] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it 's time for a complete rewrite). *VLDB '07*, pp. 1150–1160, 2007.

[57] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. *SIGMOD '12*, pp. 1–12, 2012.

[58] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. SOSP '13, pp. 18–32, 2013.

[59] G. Wang, W. Xie, A. Demers, and J. Gehrke. Asynchronous large-scale graph processing made easy. *CIDR '13*, 2013.

[60] Y. Wang, L. Zhang, J. Tan, M. Li, Y. Gao, X. Guerin, X. Meng, and S. Meng. Hydradb: A resilient rdma-driven key-value middleware for in-memory cluster computing. *SC '15*, pp. 22:1–22:11, 2015.

[61] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):409–10, 1998.

[62] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using rdma and htm. *SOSP '15*, pp. 87–104, 2015.

[63] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers Inc., 2001.

[64] R. Xin, J. Gonzalez, M. Franklin, and I. Stoica. Graphx: A resilient distributed graph system on spark. *CIDR '13*, 2013.

[65] S. Yang, X. Yan, B. Zong, and A. Khan. Towards effective partition management for large graphs. *SIGMOD '12*, pp. 517–528, 2012.

[66] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. NSDI '12, pp. 15–28, 2012.

[67] H. Zhang, G. Chen, B. C. Ooi, K. L. Tan, and M. Zhang. In-memory big data management and processing: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 27(7):1920–1948, 2015.

[68] Y. Zhang, Q. Gao, L. Gao, and C. Wang. Priter: A distributed framework for prioritized iterative computations. *SOCC '11*, pp. 13:1–13:14, 2011.

[69] M. Zinkevich, J. Langford, and A. J. Smola. Slow learners are fast. NIPS '09, pp. 2331–2339, 2009.

APPENDIX

A. BACKGROUND ON TRANSACTION PROCESSING

In this section, we briefly introduce the basic concepts of transaction processing. We focus on the main intuitions without introducing too many technical details. For readers who are interested in an in-depth discussion on the theory of transaction processing, please refer to excellent discussion on this topic from the following books [3, 16, 48].

We denote the basic unit of a database as a data element. It could be a tuple in a relational database, a vertex in a graph database, and a key-value pair in a key-value database. Each transaction works by reading and writing data elements. It may read and write several data elements. After a number of reads and writes, the last operation of each transaction is either *abort* or *commit*. If a transaction aborts, the transaction must have no side effect on the database at all, like the transaction had never been submitted. On the other hand, if a transaction commits, all actions in the transaction must be executed and reflected by the database.

When transactions execute concurrently, their reads and writes may interleave and interfere. There are several standards to define whether a given execution is considered correct or not. A popular isolation level is serializability. Executing transactions serially (one after another) is considered correct because there is interleaving of actions from different transactions, hence, transactions do not interfere with each other at all. And all schedules that are equivalent to a serial ordering of transactions are considered correct too.

Formally, let T be a set of transactions, $T = \{t(1), t(2), \cdots\}$. A history for T is known when all transactions in T commit, and a serial history is a history in which $t(i) < t(j)$ or $t(j) < t(i)$ for $i \neq j$. By $t(i) < t(j)$ it means that all reads/writes for a transaction $t(i)$ are executed before all reads/writes of $t(j)$. A history is called serializable if it is equivalent to a serial history s' ($s \approx s'$). Several serializability variations are defined. Among them, conflict serializability is defined on the notion of conflict, where two operations p in $t(i)$ and q in $t(j)$ are *conflict*, for $i \neq j$, if both p and q access the same data element, and at least one of them is write. For any two conflict operations p and q and a history s, we say $p <_s q$ if p proceeds q in s, and q *depends on* p (for history s) in this case.

A history s is *conflict serializable* if there exits a serial history s' such that s and s' contain same set of operations and for any two conflict operations p and q, either $p <_s q$ and $p <_{s'} q$, or $q <_s p$ and $q <_{s'} p$. The class of conflict serializable schedules is denoted as *CSR*. For simplicity, we denote transaction $t(j)$ *depends on* $t(i)$ if one of $t(j)$'s operations depends on one of $t(i)$'s operations.

If each transaction is represented as a vertex in a graph, then the dependency relationships can be modeled as directed edges among these vertices. The graph is called *serializable graph*. A schedule of these transactions satisfies *(conflict) serializable* if and only if the corresponding serializable graph is *acyclic*.

Based on CSR, we introduce a subclass of CSR, denoted as *COCSR* for *commit order-preserving* conflict serializable. COCSR says, a history s is $\approx$ to a serial history s' with the additional restriction such that for any two different transactions $t(i)$ and $t(j)$, if $t(i) < t(j)$ in s' then $ct(i) < ct(j)$ in s (and s'), where $ct(i)$ denotes the time $t(i)$ commits, and $ct(i) < ct(j)$ means $t(i)$ commits before $t(j)$ commits. The commit time $ct(i)$ for transaction $t(i)$ is defined as the time when all parties reach a consensus that it shall commit. In other words, before commit time the transaction could commit or abort (depending on the system status, user inputs, and the states of other transactions), and after the commit time it will not be aborted (regardless other transactions' operations).

We note that *commit order preserving* is a sufficient (but not necessary) condition for CSR. If all transactions obey the *commit order preserving* rule, the conflict relationship is ordered by the $ct(\cdot)$s of all transactions, which is a total order, therefore the serializable graph is acyclic (so the schedule satisfies CSR).

B. PROOF OF THEOREM 1

We show that our hybrid approach in Algorithms 1, 2, 3 follows the COCSR, thus follows CSR. We begin with the following lemma.

Lemma 2.1: *Consider two committed transactions $t(i)$ and $t(j)$ in HSync, $ct(i) < ct(j)$ if $t(j)$ depends on $t(i)$.* □

PROOF. For simplicity, we assume $t(i)$ accesses vertex v before $t(j)$ (and at least one of them writes v). We denote the lock on v as L_v.

By the definition of *commit time*, a BIG transaction commits when it just enters *TX-COMMIT-B* and a SMALL transaction commits when it passes validation and is ready to write `D[]` (line 21 in Algorithm 3).

- Case-1: Both $t(i)$ and $t(j)$ are BIG transactions. They cannot acquire L_v simultaneously. Since $t(j)$ depends on $t(i)$, $t(j)$ acquires L_v after $t(i)$. Thus $t(j)$ must wait until $t(i)$ releases L_v. In other words, when $t(j)$ is still in its growing phase, $t(i)$ is already in its shrinking phase. Therefore $ct(i) < ct(j)$.
- Case-2: $t(i)$ is a BIG transaction whereas $t(j)$ is a SMALL transaction. In the validation phase of $t(j)$, if L_v can be acquired by $t(j)$, it means that $t(i)$ has released L_v so it is in its shrinking phase. Therefore $ct(i) < ct(j)$ since $t(j)$ has not started to commit yet. Otherwise, $t(j)$ cannot pass validation and commit. It violates our assumption that both of them had been committed.
- Case-3: $t(i)$ is a SMALL transaction. $t(j)$ could be BIG or SMALL. Note that at least one of $t(i)$ and $t(j)$ writes to v.
  - If $t(i)$ writes to v, at this time $t(i)$ shall already been committed which indicates $ct(i) < ct(j)$. Otherwise $t(j)$ is not able to observe this modification as $t(i)$ has not committed, which contradicts with the assumption that $t(j)$ depends on $t(i)$.
  - If $t(i)$ reads from v and $t(j)$ writes to v. Suppose $ct(i) > ct(j)$. Therefore $t(j)$ successfully acquires the lock and writes to v before $t(i)$ commits. $t(i)$ will never have a chance to pass validation and commit. This violates our assumption that both of them had successfully committed. Thus $ct(i) < ct(j)$.

Therefore we prove the Lemma 2.1. □

Since CO is a sufficient condition of CSR, this completes the proof of Theorem 1.

C. EFFICIENT HYBRID SCHEDULER

C.1 Performance Consideration

We consider a problem that the transactions are scheduled by two schedulers. We assume that scheduler S_h is responsible to process transactions when conflicts are common and scheduler S_l is responsible for transactions that are not likely to conflict. Besides the two schedulers, an additional machinery is used to make sure the combination of them produces serializable schedules.

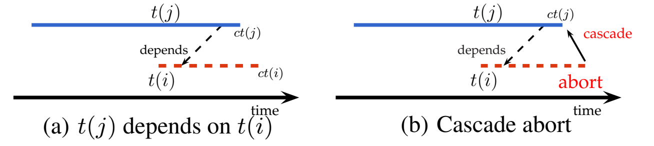
We emphasize that the coordination cost must be taken into consideration. If not, the coordinator itself may simply implement a standalone 2PL so that it ensures serializability anyway regardless how S_h and S_l behave. We highlight three general necessary conditions for efficient main memory computing, which are listed below.

Rule 1: No memory hot-spot. Ensuring strong locality in modern CPU architecture often hinders the scalability and efficiency of implementing a centralized component. For example in the experiments of memory latency [7], process only needs `5` to `44` cycles to access memory address in CPU's cache line, but `355` cycles for any data that are not found in cache. The slowest case is to access data that had just been modified by other CPUs (dirty access), which is as slow as `492` cycles.

Centralized memory access, or informally speaking, memory **hot-spot**, is a small area of memory that is frequently and repeatedly accessed by different CPUs. Having such hot-spot makes almost all access to it dirty. This phenomena is called cache thrashing. Memory hot-spot should be avoided if possible.

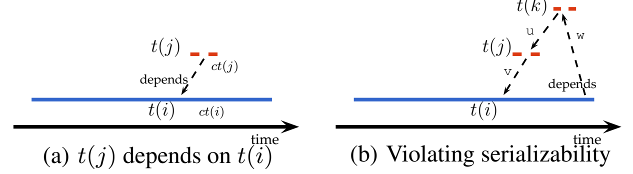
Rule 2: Linear overhead. In hard-disk based systems, disk access cost dominates the overall execution time. CPU and main memory operations are almost free. Hence, the amount of these main memory operations spent on transaction processing can be superliner to the size of data items involved in the transactions if needed. However in a main memory database, CPU and memory operations become dominating. It demands transaction processing schedulers to spend at most linear time, or even sub-linear time to the number of data items in a transaction.

Rule 3: No sacrifice for S_h. The hybrid scheduler should not abort S_h if possible. The reason is straightforward, transactions executed under scheduler S_h have a exponentially larger size like millions. Aborting these transactions are not worthwhile.



(a) $t(j)$ depends on $t(i)$ (b) Cascade abort

Figure 17: Violating commit order: S_h schedules t(j) and S_l schedules t(i).



(a) $t(j)$ depends on $t(i)$ (b) Violating serializability

Figure 18: Violating commit order: S_h schedules t(i) and S_l schedules t(j).

C.2 Key to Success: Commitment Order

We already showed CO is a sufficient condition for CSR and even COCSR in Section B. We will show, CO is also a necessary condition for an efficient hybrid scheduler. In other words, in order to keep the transactions CSR and do not break the aforementioned rules, *the coordination must obey commit order*.

We prove by contradiction. Assume there are two transactions $t(i)$ and $t(j)$, $t(j)$ depends on $t(i)$ and $ct(i) > ct(j)$, which means they do not satisfy CO.

- Suppose $t(j)$ is scheduled by S_h, whereas $t(i)$ is scheduled by S_l, as illustrated in Fig. 17(a). After $t(j)$ commits, if $t(i)$ aborts, then $t(j)$ may access some "dirty" values from $t(i)$, which violates the atomic property of transaction, and may cause cascade abort like shown Fig. 17(b).
- Next, suppose $t(i)$ is scheduled by S_h, whereas $t(j)$ is scheduled by S_l, as shown in Fig. 18(a). As illustrated in Fig. 18(b), after $t(j)$ commits, a transaction $t(k)$, that depends on $t(j)$ via accessing a value u, commits. However, $t(i)$ may depend on $t(k)$ via accessing a value w, which causes a cycle in the conflict graph. This violates CSR.

We show why such cycle is unavoidable. Firstly, when $t(k)$ tries to access the value u that $t(j)$ just committed, $t(k)$ is completely

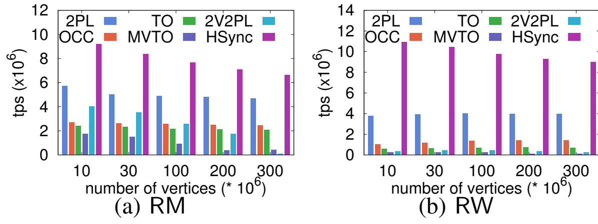


Figure 19: Synthetic graph: vary the number of vertices.

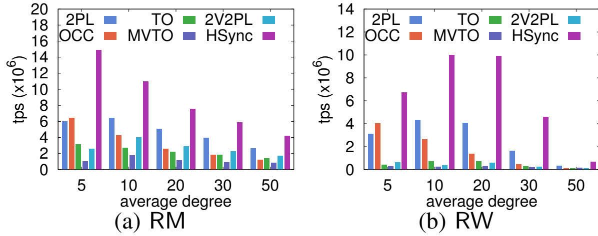


Figure 20: Synthetic graph: vary degree.

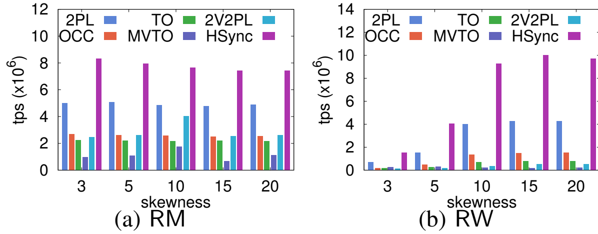


Figure 21: Synthetic graph: vary skewness.

unaware of $t(i)$. It cannot access the metadata of $t(i)$ directly, otherwise it violates the "no memory hot-spot" **Rule 1**. Neither can the liveness of $t(i)$ be stored in places associated with $t(j)$ or u, otherwise when $t(i)$ commits, it has to update its liveness flags there. Since the number of $t(i)$'s operations could be large, the number of such liveness flags could be even larger, this violates the "linear work" **Rule 2**. Thus $t(k)$ cannot infer whether the current value u's previous owner $t(j)$ has active dependencies or not, so with no information, it has to access u and causes conflict with $t(j)$. For the same reason, there is nothing that can prevent $t(k)$ to commit. After $t(k)$ has committed, it is entirely possible that $t(i)$ has to access a value w that $t(k)$ just committed. Allowing $t(i)$'s such access will violate the serializability. Denying it will force $t(i)$ to abort, or force both $t(j)$ and $t(k)$ to abort. For the later case, it is still possible that other transactions scheduled by S_h may have just accessed values updated by $t(j)$ or $t(k)$, hence had to abort. Therefore it violates **Rule 3**.

D. ADDITIONAL EXPERIMENTS

Synthetic Data. We also generate synthetic graph datasets to test different approaches. Given a vertex size V and an average degree D, we generate a graph G using the preference attachment method in [10]. Another parameter A (the "power-law-ness") is also used in the graph generation process. The smaller the value of $\frac{A}{D}$ is, the smaller the fraction of high-degree vertices is in the graph G. The range and default value of all parameters are shown in Table 4. For all experiments over synthetic graphs, we set $\tau = 10^3$.

Parameter	Range	Default
Size of V ($\times 10^6$)	10, 30, 100, 200, 300	100
Average degree D	5, 10, 20, 30, 50	20
Skew-ness A	3, 5, 10, 15, 20	10

Table 4: The parameters of synthetic data.

We measured the performance (tps, transactions per second) of different schedulers by varying the values of these parameters. In Fig. 19 we vary the number of vertices. HSync outperforms other schedulers regardless the size of the graphs for both RM and RW workloads. As the graph becomes bigger, the performance degenerates slightly due to the increasing density of the graph, which inherently limits the concurrency.

In Fig. 20 we vary the average degree of the graph. We note as the degree increases the amount of work in a transactions increases. When the average degree is small, OCC outperforms the 2PL since transactions conflicts are less likely to happen. Our scheduler HSync achieves the best performance among all cases.

In Fig. 21 we vary the skewness. Clearly, the performance gap between HSync and other schedulers increases as the graph degree distribution becomes skewer. It is contributed by the skewness-aware hybrid approach which schedules vertex transactions based on the degree of the vertex with different schedulers.

HSync in GraphLab. Lastly, we implemented HSync in GraphLab using GraphLab's communication primitives, to show that existing systems can also benefit from fine-grained parallelism using synchronous parallel processing with a scheduler like HSync. It is worth noting that the GraphLab is built on message passing, even in single computer mode, which limits the performance of good schedulers, since a scheduler needs to take a message round trip to verify before commitment.

In Fig. 22 we show how the training error reduces over time, while executing ALS on Netflix data. Clearly, the default fine-grained parallelism synchronous parallel processing in GraphLab has the worst performance: it converges much slower than both asynchronous and BSP approaches. This is due the ineffective scheduler it has used. It confirms the observations from GraphLab developers that fine-grained parallelism is not as fast as other engines. However, once using our hybrid scheduler HSync, fine-grained parallelism has outperformed all other engines in GraphLab including the asynchronous engine.

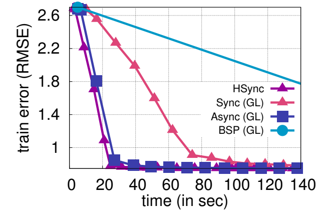


Figure 22: ALS in GraphLab