A Hierarchical Synchronous Parallel Model for Wide-Area Graph Analytics

Shuhao Liu, Li Chen, Baochun Li, Aiden Carnegie
Department of Electrical and Computer Engineering, University of Toronto
{shuhao, lchen, bli}@ece.toronto.edu, *aiden.carnegie*@mail.utoronto.ca

***Abstract*—Graph analytics has emerged as one of the fundamental techniques to support modern Internet applications. As real-world graph data is generated and stored globally, the scale of the graph that needs to be processed keeps growing. It is critical to efficiently process graphs across multiple geographically distributed datacenters, running wide-area graph analytics.**

Existing graph analytics frameworks are not designed to run across multiple datacenters well, as they implement a Bulk Synchronous Parallel model that requires excessive wide-area data transfers. In this paper, we present a new Hierarchical Synchronous Parallel model designed and implemented for synchronization across datacenters with a much improved efficiency in inter-datacenter communication. Our new model requires no modifications to graph analytics applications, yet guarantees their convergence and correctness. Our prototype implementation on Apache Spark can achieve up to 32% lower WAN bandwidth usage, 49% faster convergence, and 30% less total cost for benchmark graph algorithms, with input data stored across five geographically distributed datacenters.

I. Introduction

Graph analytics serves as the foundation of many popular Internet services, including PageRank Web search [4] and social networking [5]. These services are typically deployed at a global scale, with user-related data naturally generated and stored in geographically distributed, commodity datacenters [8], [19], [20], [25]. In fact, popular cloud providers, such as Amazon, Microsoft, and Google, all operate tens of datacenters around the world [11], offering convenient access to both storage and computing resources.

In Internet-scale graph analytics, the production graphs are typically as large as billions of vertices and trillions of edges [14], [23], taking terabytes of storage. For example, it is reported that Web search engines operate on an indexable Web graph consisting of ever-growing 50 billions of websites and one trillion hyperlinks between them [14]. Such large scales with rapid rates of change [11], coupled with high costs of Wide-Area Network (WAN) data transfers [19] and possible regulatory constraints [26], make it expensive, inefficient, or simply infeasible to centralize the entire dataset to a central location, even though this is a commonly-used approach [1], [15] for analytics.

Therefore, it is critical to design efficient mechanisms to run graph analytics applications in a geographically distributed manner across multiple datacenters in a paradigm called *wide-area graph analytics*. In particular, the fundamental challenge is to process the graph with raw input data stored and computing resources distributed in globally operated datacenters, which are inter-connected by wide-area networks (WANs).

Unfortunately, existing distributed graph analytics frameworks are not sufficiently competent to address such a challenge. Representative works in the literature, *e.g.,* Pregel [17], PowerGraph [6] and GraphX [7], are solely designed and optimized for processing graphs *within* a single datacenter. Gemini [33], one of the state-of-the-art solutions, even assume a high-performance cluster with 100 Gbps of bandwidth capacity between worker nodes. Unfortunately, this assumption is far beyond the reality in inter-datacenter WANs, whose available capacity is typically hundreds of Mbps [11].

In this paper, we argue that the inefficiency of wide-area graph analytics stems from the Bulk Synchronous Parallel (BSP) model [24], which is the dominating synchronization model implemented by most of the popular graph analytics engines [28]. The primary reason for its popularity is that BSP works seamlessly with the vertex-centric programming abstraction [17], which eases the development of graph analytics applications. Many graph algorithms and optimizations, namely vertex programs, are exclusively designed with such an abstraction with BSP [21], [29] in mind.

In a vertex program under BSP, the application runs in a sequence of "supersteps," or *iterations*, which apply updates on vertices and edges iteratively. Message passing and synchronization is made between two consecutive supersteps, while performing local computation within each superstep. Since each superstep typically allows communication between neighboring vertices only, it takes at least k supersteps until the algorithm converges on a graph whose diameter is k [29]. Thus, k message passing phases will happen in serial, which incurs excessive — and not always necessary [31] — inter-datacenter traffic in wide-area graph analytics.

One possible solution is to loosen the BSP model, by allowing asynchronous updates on different graph partitions. This way, new iterations of computation are able to proceed with partially staled vertex/edge properties, relaxing the hard requirement on inter-datacenter communication to a best-effort model. Existing systems implementing such an asynchronous parallel model include GraphUC [9] and Maiter [31]. However, neither system can guarantee the convergence or the correct-

This work is supported by a research contract with Huawei Corp. and an NSERC Collaborative Research and Development (CRD) Grant.

ness of graph applications [28].

Our objective is to design a new synchronization model for wide-area graph analytics, which satisfies three important requirements:

1) *WAN efficiency.* The new model should require fewer rounds of inter-datacenter communication and generate less inter-datacenter traffic.
2) *Correctness.* The new model should ensure the application can return the same result as if it was executed under BSP.
3) *Transparency.* The new model should require absolutely no change to the existing applications, by retaining the same set of vertex-centric abstraction APIs.

In this paper, we introduce Hierarchical Synchronous Parallel (HSP), a novel synchronization model designed for efficiency in wide-area graph analytics. In contrast to BSP, which requires complete, global synchronizations among all worker nodes in all datacenters, HSP allows partial, local synchronizations within each datacenter as additional updates. Specifically, HSP automatically switches between two modes of execution, `global` and `local`, like a two-level hierarchical organization. The `global` mode is the same as BSP, where all datacenters respond to central coordination. The `local` mode, on the other hand, allows each datacenter to work autonomously without coordinating with others. Our theoretical analysis shows that, if the mode switch happens strategically, HSP can guarantee the convergence and correctness of all vertex programs. In addition, if the implementation of the vertex program is considered practical [29], HSP can ensure a much higher rate of convergence, as compared to BSP with the same amount of inter-datacenter traffic generated.

We have implemented the HSP model on GraphX [7], an open-source general graph analytics framework built on top of Apache Spark [30]. The original implementation of GraphX supports the BSP vertex-centric programming abstraction. In our prototype implementation, we have extended the framework with HSP, by allowing synchronization to be bounded within a single datacenter, and by implementing the feature that automatically switches the mode of execution on a central coordinator. With our implementation, we have performed an extensive evaluation of HSP in five real geographically distributed datacenters on Google Cloud. Three empirical benchmark workloads on two large-scale real-world graph datasets have been experimented on. The results show that HSP is efficient in running wide-area graph analytics. It requires significantly fewer cross-datacenter synchronizations until a guaranteed algorithm convergence, and reduces WAN bandwidth usage by 22.4% to 32.2%. The monetary cost of running graph applications can be reduced by up to 30.4%.

II. Background and Motivation

Graphs in production are typically too large to be efficiently processed by a single machine [17]. Distributed graph analytics frameworks are thus developed to run graph analytics in parallel on multiple worker nodes. Before running the actual analytics, the input graph is divided into several partitions, each of which is held and processed by a worker. The frameworks will then handle the synchronization and necessary message passing among workers automatically, allowing developers to work solely on the analytic logic itself.

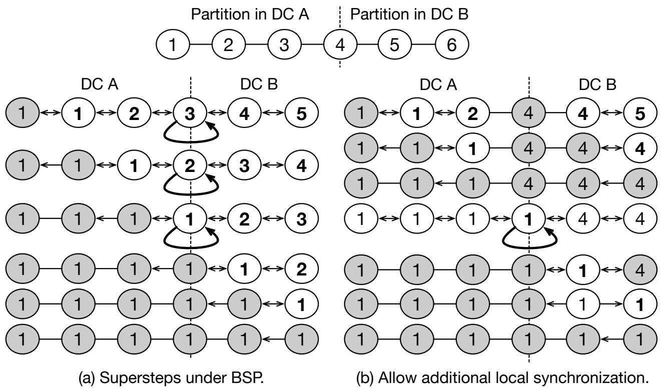

Partition in DC A
Partition in DC B
DC A
DC B
DC A
DC B
(a) Supersteps under BSP.
(b) Allow additional local synchronization.

Fig. 1: A Connected-Component algorithm executed under different synchronization models. White circles indicate active vertices, and the arrows represent message passing.

Most of the state-of-the-art solutions [6], [7], [33] provide a vertex-centric abstraction for developers to work on — similar to Google's Pregel — and implement the Bulk Synchronous Parallel (BSP) model for inter-node synchronization [28]. Such an integration of programming abstraction and synchronization model allows developers to "think like a vertex," making the development of graph analytics applications intuitive and easy to debug.

Even though it is well-known that the BSP model requires excessive communication, the bandwidth capacity among workers is seldom considered a system bottleneck [33]. When deployed within a high-performance cluster where bandwidth is readily abundant, BSP performs well with a large number of system optimization techniques, such as advanced graph partitioning strategies and load balancing. Unfortunately, this is no longer true in wide-area data analytics, where inter-datacenter data transfers can incur a much higher cost, in terms of both time and monetary expenses.

Fig. 1a illustrates a sample execution of the Connected-Component (CC) algorithm under BSP. The algorithm runs on a six-vertex graph, which is cut into two partitions. Within a superstep, each vertex tries to update itself with the smallest vertex ID seen so far at all its neighbors. A vertex becomes inactive as soon as it cannot get further updates. The algorithm converges until no vertex is active, and we can then compute the number of connected components by counting the remaining vertex IDs. Fig. 1a shows that CC converges in a total of 6 supersteps, while the first three supersteps require message passing from DC A to DC B.

However, it is easy to observe that the first two inter-datacenter messages are unnecessary in this example. These two messages are in fact transferring values that will be immediately overridden in the next superstep. The insight behind it is that we can sometimes hold inter-datacenter synchronization until multiple cycles of synchronization within a single datacenter have been performed, for the sole purpose of minimizing cross-datacenter traffic. For example, one possible optimization is illustrated in Fig. 1b. It allows updates of vertex IDs to happen within a datacenter, without updating the vertex that is owned by both datacenters. Inter-datacenter communication happens once at the fourth step, when both partitions has already converged locally. This new principle of synchronization reaches the same result of the CC algorithm, yet generating only 1/3 of the inter-datacenter traffic as compared to BSP.

The example shown in Fig. 1 inspires us to explore such a new principle, for the sake of minimizing inter-datacenter traffic. To achieve this objective, we wish to carefully design a new synchronization model, called the Hierarchical Synchronous Parallel (HSP) model. As an alternative to BSP, it needs to guarantee that the correctness of any vertex program should be retained, which may be way more complex than the Connected-Component algorithm.

III. Hierarchical Synchronous Parallel Model

In this section, we introduce the Hierarchical Synchronous Parallel (HSP) model. We will first explain the high-level principle of its design and the general idea behind its correctness guarantee. We will then formulate HSP model theoretically and explain it in greater detail. With our formulation, we present a formal proof of its correctness and rate of convergence in wide-area graph analytics. Finally, we use a simple PageRank application as an example to illustrate the effectiveness of HSP.

A. Overview

Generally speaking, HSP is an extension to the BSP model in wide-area graph analytics, by performing synchronization in a two-level hierarchy. In addition to BSP, HSP allows local synchronization among worker nodes located in a single datacenter, completely avoiding inter-datacenter communication. To achieve this, HSP introduces two modes of execution, `global` and `local`, and switches between them strategically and frequently.

In the `global` mode, HSP has exactly the same behavior as BSP, where each synchronization is a global, all-to-all communication among all worker nodes, regardless of which datacenter they are located in. We call one iteration of the execution in HSP a "global update," which is equivalent to a *superstep* in BSP. Global updates are essential to the correctness of graph analytics, because it is necessary to spread the information outside of individual datacenters.

In the `local` mode, the worker nodes are organized in different autonomous datacenters. Workers housed in the same datacenter work synchronously. In particular, they still run in iterations, or "local updates," as if they are running the vertex program under BSP. The difference is that, if a vertex has mirrors in multiple datacenters, called a "global vertex," we mark it *inactive* and do not update it until switching back to the `global` mode. Since synchronizing the property of these global vertices is the only source of inter-datacenter traffic, we completely eliminate the need for inter-datacenter communication in the `local` mode. In addition, it is worth noting that without the need for global synchronization, execution at different datacenters can be asynchronous.

Without running in the `local` mode, HSP is equivalent to BSP. Thus, it still guarantees correctness. However, running in the `local` mode takes advantage of low-cost synchronization within a datacenter, allowing more updates in the same amount of time. It is an interesting question when to make the switching between these two modes.

Our mode-switching strategy is designed based on our theoretical analysis in the next subsection for the sake of the algorithm convergence guarantee. Here we introduce its general principle. On the one hand, local updates should allow at least one information exchange between any pair of vertices. Thus, the number of local updates should be higher than the diameter of the local partition. On the other hand, we should not leave any worker idle before reaching global convergence. As a result, HSP will switch away from the `local` mode as soon as all datacenters execute more local updates than its partition diameter or the local update in any datacenter converges. Then, while HSP running in the `global` mode, it will switch back to `local` immediately as soon as the algorithm is considered "more converged," whose metric will be introduced later.

The intuition behind HSP is that, in general, graph algorithms need to spread the information on a vertex to all vertices of the entire graph. In other words, every vertex has to "get its voice heard." In wide-area data analytics, communicating with neighbors does not always come at similar prices. Therefore, instead of requiring every vertex to talk to its neighbors in every iteration, HSP organizes communication hierarchically. It allows information to spread well within a closed neighborhood, before inter-neighborhood communication. This way, the entire graph can still converge, but at a much lower cost.

B. Model Formulation and Description

Before formulating the HSP model, we first give a formal definition of a vertex program. Given a graph $G = (V, E)$ with initial properties on all vertices $\boldsymbol{x}^{(0)} \in \mathbb{R}^{|V|}$, a vertex programming application defines the function to update each vertex in a superstep. Specifically, a combiner function $g(\cdot)$ is defined to combine the received message to each vertex, and a `compute` function $f(\cdot)$ is defined to compute the updated property on a vertex using the old property and the combined incoming message. Without loss of generality, let $f_i : \mathbb{R}^{|V|} \to \mathbb{R}$ denote the update function defined on vertex $i \in \{1, 2, \dots, |V|\}$, such that, in BSP, we have

$$x_i^{(k+1)} = f\big(x_i^{(k)}, g_i(\boldsymbol{x}^{(k)})\big) := f_i(\boldsymbol{x}^{(k)}). \tag{1}$$

Or equivalently, define $\boldsymbol{F} : \mathbb{R}^{|V|} \to \mathbb{R}^{|V|}$ such that

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{F}(\boldsymbol{x}^{(k)}).$$

The objective of the vertex programming application is to compute $\boldsymbol{x}^* \in \mathbb{R}^{|V|}$ which satisfies $\boldsymbol{x}^* = \boldsymbol{F}(\boldsymbol{x}^*)$, by iteratively applying the update defined in Eq. (1) until convergence under BSP. By definition, $\boldsymbol{x}^*$ is a *fixed point* under operator $\boldsymbol{F}$. In practice, the application is considered converged and an

Procedure 1 Execution of a vertex programming application under the Hierarchical Synchronous Parallel (HSP) model.

```
1: Set execution mode to global, global update counter $k \leftarrow 0$, current error $\delta_0 \leftarrow \infty$;
2: while $\delta_k > \delta$ do
3:     if Execution mode is global then
4:         Perform a global update: $\boldsymbol{x}^{(k+1)} \leftarrow \boldsymbol{F}(\boldsymbol{x}^{(k)})$;
5:         $\delta_{k+1} \leftarrow D(\boldsymbol{x}^{(k+1)}, \boldsymbol{x}^{(k)})$;
6:         if $\delta_{k+1} < \delta_k$ then
7:             Switch execution mode to local;
8:         else
9:             $\delta_{k+1} \leftarrow \delta_k$;
10:        $k \leftarrow k + 1$;
11:    else
12:        Apply local updates in each datacenter concurrently (as in Procedure 2), until any datacenter calls forceModeSwitch() or all datacenters call voteModeSwitch().
13:        Switch execution mode to global;
14: return $\boldsymbol{x}^{(k)}$.
```

approximation $\boldsymbol{x}^{(N)}$ is obtained after N supersteps, if the following condition is satisfied:

$$D(\boldsymbol{x}^{(N-1)}, \boldsymbol{x}^{(N)}) \leq \delta \tag{2}$$

where $D : (\mathbb{R}^{|V|}, \mathbb{R}^{|V|}) \rightarrow \mathbb{Q}_{\geq 0}$ is a distance metric (i.e., $(\mathbb{R}^{|V|}, D)$ is a metric space), and $\delta \in \mathbb{Q}_{\geq 0}$ is a pre-defined error bound. For example, Chebyshev norm is commonly used as the metric, *i.e.,*

$$D(\boldsymbol{x}, \boldsymbol{y}) = \max\{|x_1 - y_1|, |x_2 - y_2|, \ldots, |x_{|V|} - y_{|V|}|\},$$

because it does not require global knowledge to compute.

To process the graph in d geographically distributed datacenters, it is partitioned into d large subgraphs. We define each vertex either a *local vertex* or a *global vertex*. Specifically, vertex i is a *local vertex of datacenter* j if datacenter j stores its original copy and its property x_i is not required to update any vertex stored in a different datacenter. In other words, x_i should never be delivered outside of datacenter j when running the application under BSP. Let I_j denote the index set of all local vertices of datacenter j. Otherwise, if x_i is required to update vertices stored in multiple datacenters, we define vertex i a *global vertex*. Let I_G denote the index set of all global vertices. Apparently, the defined $d + 1$ index sets $I_1, I_2, \ldots, I_d, I_G$ are mutually exclusive and their union is $\{1, 2, \ldots, |V|\}$.

With a globally partitioned graph, we define $\boldsymbol{F}_j(\cdot)$, the *local update function* in datacenter j, as follows:

$$\left[\boldsymbol{F}_j(\boldsymbol{x})\right]_i = \begin{cases} f_i(\boldsymbol{x}) & i \in I_j \\ x_i & \text{otherwise} \end{cases}. \tag{3}$$

Using the notations introduced above, we present our detailed description of the HSP model in Procedure 1. The procedure for local updates in a single datacenter is listed separately. Since the local updates in different datacenters are asynchronous, it requires a mechanism of central coordination that decides mode switch to `global`. This is achieved by two methods, `voteModeSwitch()` and `forceModeSwitch()`.

Procedure 2 Local updates in datacenter j.

```
1: Set local iteration counter $n_j \leftarrow 0$, local error $\delta_0 \leftarrow \infty$; $d$ denotes the diameter of the local partition;
2: repeat
3:     $n_j \leftarrow n_j + 1$;
4:     Perform an in-place local update: $\boldsymbol{x}^{(k,n_j)} \leftarrow \boldsymbol{F}_j\big(\boldsymbol{x}^{(k,n_j-1)}\big)$;
5:     $\delta_{n_j} \leftarrow D(\boldsymbol{x}^{(k,n_j)}, \boldsymbol{x}^{(k,n_j-1)})$;
6:     if $\delta_{n_j} < \delta$ then
7:         forceModeSwitch();
8:     if $n_j == d$ then
9:         voteModeSwitch();
10: until Mode switch is forced by any or voted by all datacenters;
11: Execution mode switched to global.
```

The prior is called once the number of local updates reaches the subgraph diameter, while the latter is called upon local convergence. The central coordinator receives the signals triggered by these two methods, and enforces mode switch when appropriate (line 12 of Prcedure 1).

C. Proof of Convergence and Correctness

To prove the convergence and correctness guarantee of the proposed HSP model, the only requirement for the vertex programming application is that the iterative computation is correctly implemented; that is, the application can converge within a finite number of iterations under BSP. Formally, we have the following assumption:

***Assumption** 1 (Convergence under BSP):* Given any arbitrary initial value of $\boldsymbol{x}^{(0)}$ and a distance metric $D(\cdot, \cdot)$, the sequence of successive approximations $\{\boldsymbol{x}^{(k)}\}$ approaches $\boldsymbol{x}^*$, a fixed point under operator $\boldsymbol{F}$, as more iterations of global updates Eq. (1) are applied. That is,

$$\lim_{k\to\infty} D\big(\boldsymbol{x}^{(k)}, \boldsymbol{x}^*\big) = 0, \text{where } \boldsymbol{x}^* \in \text{Fix}\boldsymbol{F}. \tag{4}$$

***Theorem** 1 (Convergence and correctness guarantee of HSP):* If a vertex programming application satisfies Assumption 1, it will also return a valid approximation of $\boldsymbol{x}^*$ in finite time under Procedure 1.

Proof: Since δ_k is overridden at line 9 using the previous value in the sequence, it is valid to consider their original value before being overridden.

$$\begin{aligned}\delta_{k+1} &= D\big(\boldsymbol{x}^{(k+1)}, \boldsymbol{x}^{(k)}\big) \\ &\leq \big[D\big(\boldsymbol{x}^{(k+1)}, \boldsymbol{x}^*\big) + D\big(\boldsymbol{x}^{(k)}, \boldsymbol{x}^*\big)\big] \text{ (triangle inequality)},\end{aligned}$$

which approaches 0 as $k \rightarrow \infty$ given Eq. (4). Equivalently,

$$\forall \delta_k > 0 \text{ and } \forall \boldsymbol{x}^{(k)}, \exists n \in \mathbb{N} \text{ s.t. } \delta_{n+k} < \delta_k.$$

Considering the `global` mode in Procedure 1, a mode switch to `local` (line 7) will happen after a finite number of global updates. This process will result in a non-increasing sequence $\{\delta_k\}$, which strictly decreases after each mode switch to `local`.

In a real-world implementation where precision of any number is bounded, the sequence $\{\delta_k\}$ will eventually approach to 0, *i.e.,* $\exists k^* < \infty$, s.t. $\delta_{k^*} < \delta$, and Procedure 1 will return an estimation $\boldsymbol{x}^{(k^*)}$.

$\boldsymbol{x}^{(k^*)}$ satisfies Eq. (2), making it a valid estimation of $\boldsymbol{x}^*$.

■

As is shown in the proof, the convergence guarantee of HSP relies heavily upon its global updates. As long as the application can converge, applying local updates in the middle does not affect the result of the vertex programming algorithm.

D. Rate of Convergence

Even with the convergence and the correctness guarantee, one may still be skeptical about the *effectiveness* of HSP. How could the additional local updates help with the application execution? Is it a guarantee that it will generate less inter-datacenter traffic?

It is difficult to answer these questions without any prior knowledge about the actual application itself, since the vertex programming model provides developers with a substantial amount of flexibility. Generally speaking, developers are able to code whatever they desire, making it difficult to reach any useful conclusion about such applications.

However, to ensure the scalability while processing very large datasets, vertex programming applications can share some characteristics in practice. Yan *et al.* [29], in particular, investigated some well-implemented vertex programming algorithms, namely *practical Pregel algorithms*. Common characteristics of a practical vertex programming application were summarized under BSP. These applications require linear space usage, linear computing complexity and linear communication cost per superstep. In addition, practical Pregel algorithms require at most a logarithmic number of supersteps till their convergence, *i.e.,* at least a linear rate of convergence.

With the latter characteristic as an assumption, HSP can ensure effectiveness by allowing additional local updates in different datacenters.

***Assumption** 2 (Practical implementation):* The vertex programming application converges at a linear or a superlinear rate under BSP, *i.e.,*

$$\exists \mu \in [0,1), \text{s.t. } \lim_{k\to\infty} \frac{D(\boldsymbol{x}^{(k+1)}, \boldsymbol{x}^*)}{D(\boldsymbol{x}^{(k)}, \boldsymbol{x}^*)} \le \mu. \tag{5}$$

To study the rate of convergence, we consider each *cycle* of synchronization in HSP. A cycle of synchronization is defined as the interval between two consecutive mode switches from `local` to `global`. In other words, a cycle includes several consecutive global updates and the subsequent local updates, until a switch back to the `global` mode.

In particular, during the `local` mode within each synchronization cycle, several iterations of local updates are applied in each individual datacenter at the same time. For the convenience of our subsequent proof, we collectively formulate these local updates as a function

$$\bar{\boldsymbol{F}}^{(n_1,n_2,\ldots,n_d)}(\boldsymbol{x}) := \big(\prod_{j=1}^{d} \boldsymbol{F}_j^{n_j}\big)(\boldsymbol{x}),$$

where n_j denotes the number of iterations of local updates applied in datacenter j.

***Lemma** 1:* $\boldsymbol{x}^*$ is a fixed point under operator $\bar{\boldsymbol{F}} := \bar{\boldsymbol{F}}^{(n_1,n_2,\ldots,n_d)}$ for any $n_1, n_2, \ldots, n_d$.

Proof: Given $j \in \{1,2,\ldots,d\}$ and $i \in I_j$, according to Eq. (3), we have $\big[\boldsymbol{F}_j(\boldsymbol{x}^*)\big]_i = f_i(\boldsymbol{x}^*) = \boldsymbol{x}_i^*$. Also, given $i \notin I_G$, $\big[\boldsymbol{F}_j(\boldsymbol{x}^*)\big]_i = \boldsymbol{x}_i^*$ by definition. Thus, $\boldsymbol{x}^*$ is a fixed point under $\boldsymbol{F}_j$.

Since no global update is applied, the properties of global vertices remain the same after $\bar{\boldsymbol{F}}(\cdot)$ is applied, *i.e.,*

$$\big[\bar{\boldsymbol{F}}(\boldsymbol{x})\big]_i = x_i, \forall i \in I_G.$$

Further, $\boldsymbol{F}_j(\cdot)$ depends on the local vertices of datacenter j only. Thus, the individual functions $\boldsymbol{F}_j (j = 1,2,\ldots,d)$ are commutative and associative in Eq. (3). Therefore,

$$\begin{aligned}\bar{\boldsymbol{F}}^{(n_1,n_2,\ldots,n_d)}(\boldsymbol{x}^*) &= \bar{\boldsymbol{F}}^{(n_1,\ldots,n_j-1,\ldots,n_d)}\big(\boldsymbol{F_j}(\boldsymbol{x}^*)\big)\\ = \ \bar{\boldsymbol{F}}^{(n_1,\ldots,n_j-1,\ldots,n_d)}(\boldsymbol{x}^*) &= \cdots = \bar{\boldsymbol{F}}^{(0,\ldots,0)}(\boldsymbol{x}^*) = \boldsymbol{x}^*\end{aligned}$$

■

***Lemma** 2:* $\bar{\boldsymbol{F}} : \mathbb{R}^{|V|} \to \mathbb{R}^{|V|}$ is a contraction mapping, *i.e.,*

$$D(\bar{\boldsymbol{F}}(\boldsymbol{x}), \bar{\boldsymbol{F}}(\boldsymbol{y})) < D(\boldsymbol{x}, \boldsymbol{y}), \forall \boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^{|V|}.$$

Proof: According to the Banach fixed-point theorem [2], an equivalent condition of Eq. (5) is that $\boldsymbol{F} : \mathbb{R}^{|V|} \to \mathbb{R}^{|V|}$ is a contraction mapping. Given $\forall \boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^{|V|}$, we have $D(\boldsymbol{F}(\boldsymbol{x}), \boldsymbol{F}(\boldsymbol{y})) \le \mu D(\boldsymbol{x}, \boldsymbol{y})$.

Construct $\hat{\boldsymbol{y}} \in \mathbb{R}^{|V|}$ by letting $\hat{y}_i = \begin{cases} y_i & i \in I_j \\ x_i & i \notin I_j \end{cases}$. Thus,

$$\begin{aligned}D(\boldsymbol{F}_j(\boldsymbol{x}), \boldsymbol{F}_j(\boldsymbol{y})) = D(\boldsymbol{F}_j(\boldsymbol{x}), \boldsymbol{F}_j(\hat{\boldsymbol{y}})) &= D(\boldsymbol{F}(\boldsymbol{x}), \boldsymbol{F}(\hat{\boldsymbol{y}}))\\ \le \mu D(\boldsymbol{x}, \hat{\boldsymbol{y}}) &\le \mu D(\boldsymbol{x}, \boldsymbol{y}).\end{aligned}$$

Therefore,

$$D(\bar{\boldsymbol{F}}(\boldsymbol{x}), \bar{\boldsymbol{F}}(\boldsymbol{y})) \le \mu^{\sum_{j=1}^{d} n_j} D(\boldsymbol{x}, \boldsymbol{y}).$$

■

***Theorem** 2 (Rate of convergence of HSP):* If a vertex programming application satisfies Assumption 1 and 2, HSP will converge to $\boldsymbol{x}^*$ at a higher rate than BSP, given the same amount of inter-datacenter traffic generated.

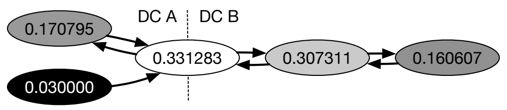
Proof: Consider a HSP synchronization cycle that includes n global updates and a set of local updates denoted by $\bar{\boldsymbol{F}}$. After the synchronization cycle, the original estimation $\boldsymbol{x}^{(k)}$ is updated to $\bar{\boldsymbol{x}}^{(k+n)} = (\bar{\boldsymbol{F}} \cdot \boldsymbol{F}^n)(\boldsymbol{x}^{(k)})$.

As a valid comparison, with the same amount of inter-datacenter traffic generated (*i.e.,* same number of global synchronization), BSP will get $\boldsymbol{x}^{(k+n)} = \boldsymbol{F}^n(\boldsymbol{x}^{(k)})$.

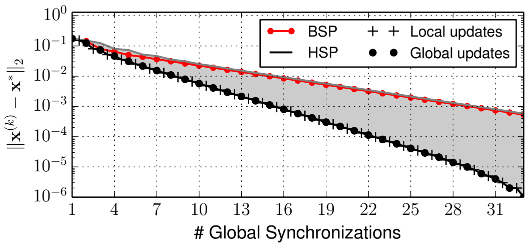
The average rate of convergence of HSP μ_{HSP} satisfies

$$\begin{aligned}\mu_{HSP}^n = \lim_{k\to\infty} \frac{D(\bar{\boldsymbol{x}}^{(k+n)}, \boldsymbol{x}^*)}{D(\boldsymbol{x}^{(k)}, \boldsymbol{x}^*)} &= \lim_{k\to\infty} \frac{D(\bar{\boldsymbol{F}}(\boldsymbol{x}^{(k+n)}), \boldsymbol{x}^*)}{D(\boldsymbol{x}^{(k)}, \boldsymbol{x}^*)}\\ &= \lim_{k\to\infty} \frac{D(\bar{\boldsymbol{F}}(\boldsymbol{x}^{(k+n)}), \bar{\boldsymbol{F}}(\boldsymbol{x}^*))}{D(\boldsymbol{x}^{(k)}, \boldsymbol{x}^*)} \text{ (Lemma 1)}\\ &\le \mu^{\sum_{j=1}^{d} n_j} \cdot \lim_{k\to\infty} \frac{D(\boldsymbol{x}^{(k+n)}, \boldsymbol{x}^*)}{D(\boldsymbol{x}^{(k)}, \boldsymbol{x}^*)} \text{ (Lemma 2)}\\ &= \mu^{\sum_{j=1}^{d} n_j} \cdot \mu_{BSP}^n < \mu_{BSP}^n\end{aligned}$$

Therefore, HSP provides a much higher average rate of convergence as compared to BSP. ■



(a) The example direct graph. The numbers on the vertices represent the final ranks that we used as $\boldsymbol{x}^*$.



(b) Convergence of PageRank under BSP and HSP.

Fig. 2: A PageRank example with damping factor set to 0.15 [4]. All values used in computation are rounded to the sixth decimal place; therefore, norm lower than 10^{-6} makes little sense and is ignored in the figure.

E. PageRank Example: a Numerical Verification

To verify our findings in the previous theorems, we compare the convergence under HSP and BSP using a simple PageRank example shown in Fig. 2. Fig. 2a shows the 5-vertex graph in the example, while the graph is partitioned into two datacenters by cutting the central vertex. Note that the diameter in each partition is 1 (ignoring the global vertex); therefore, HSP runs local updates only once in every `local` mode of execution. We plot every single estimation achieved by both synchronization models in Fig. 2b, whose y-axis shows the Euclidean norm between estimations and $\boldsymbol{x}^*$ in log scale.

Since PageRank is a practical Pregel algorithm by definition [29], BSP shows a perfectly linear rate of convergence (the red line in Fig. 2b). As a comparison, HSP depicted by the black line, the lower bound of the gray area, shows a much higher rate of convergence, given the same number of global updates as the number of supersteps in BSP.

If we consider x-axis as algorithm run time in real systems, the gray area indicates the possible convergence rate of HSP. The reason is that the lower bound, shown by the black line, assumes no cost incurred by local updates because they do not introduce inter-datacenter traffic. The upper bound, in the other hand, assumes that a local update takes exactly the same amount of time as a superstep in BSP. In reality, the time needed by a local update is in between of these two extremes, and HSP can always converge faster.

IV. Prototype Implementation

We have implemented a prototype of the HSP synchronization model on GraphX [7], and it retains full compatibility with existing analytics applications. Our prototype implementation is non-trivial to complete, yet it makes a strong case that HSP can be seamlessly integrated with existing BSP-based graph analytics engines.

Systems design. GraphX is an open-source graph analytics engine built on top of Apache Spark [30]. It is an ideal platform for us to implement our prototype, due to its full interoperability with general dataflow analytics and machine learning applications in the popular Spark framework.

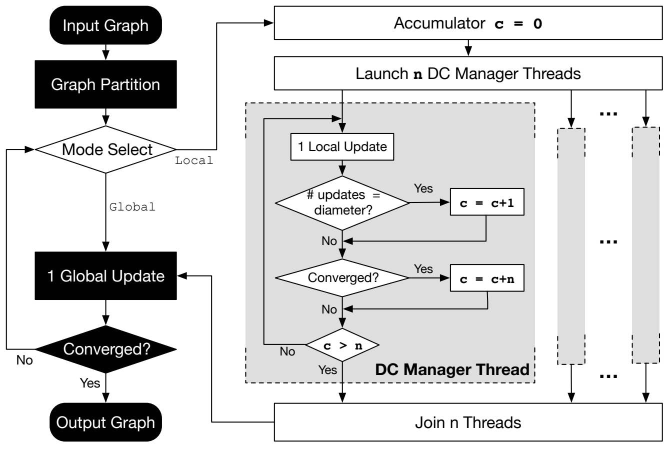


Fig. 3: The flow chart that shows central coordination in HSP. Blocks in black indicate the original `Pregel` implementation in GraphX.

In GraphX, the vertex-centric programming abstraction is supported via a `Pregel` API, which allows developers to pass in their customized vertex update and message passing functions. The graph analytics applications are executed under the BSP model, with all workers proceeding in supersteps. It takes advantage of the Resilient Distributed Dataset (RDD) programming abstraction [30]. An RDD represents a collection of data stored on multiple workers. Parallel computation on the data can be modeled as sequential transformations made on the RDD, allowing developers to program as if it is running on a single machine. In particular, `Pregel` models a graph using a `VertexRDD` and an `EdgeRDD`, while a superstep is modeled as a series of sequential transformations on them.

Our implementation retains the original `Pregel` API, providing full compatibility with existing applications and requiring no change to their code. To enable HSP instead of BSP, users can simply pass an additional option, `-Dspark.graphx.hsp.enabled=true`, along with the application submission command.

Under the hood, our modifications to the original GraphX codebase take place within the `Pregel` implementation and include two separate components: the central logic to switch between `local` and `global` execution modes, as well as the RDD transformation sequence that actually implements the local updates.

Mode switches. We have implemented Procedure 1 and Procedure 2 for mode switching. It runs on the Spark "driver" program, which centrally manages all workers acquired by the application. The flow chart is shown in Fig. 3. We utilize two `Accumulators` to help decision making. An `Accumulator` can serve as a global variable, which is addable by workers and readable by the central coordinator.

To switch from the `global` to the `local` mode, one `Accumulator` is used to track the differences (*i.e., distance*) between consecutive updates. A mode switch will be made once the difference decreases. On the other hand, to switch back to `global`, we use the other `Accumulator` to track the progress made by individual local updates. `voteModeSwitch()` and forceModeSwitch() are implemented by adding to the `Accumulator`. Each individual datacenter

TABLE I: Summary of the used datasets.

Dataset	# Vertices	# Edges	Harmonic Diameter	Edgelist Size (MB)
enwiki-2013	4,206,785	101,355,853	5.24	1556.9
uk-2014-host	4,769,354	50,829,923	21.48	801.4

will check the value of the global variable after each local update to decide whether to proceed, and change the value once it converges or reaches a preset number.

Local updates. Implementing local updates in GraphX is challenging, because the RDD transformations on a graph are designed to hide some runtime details from the developers. These details include data distributions across the workers, which are essential to the concept of local updates. We need to dig deep into the RDD internals for our implementation.

One of the major challenges is to identify `global vertices` and `local vertices`. In Spark, the actual dataset placement is decided at runtime depending on the worker availabilities. Thus, the co-located graph partitions in the same datacenters can only be identified at runtime. Fortunately, such runtime information is accessible via the `preferredLocations` feature in an RDD, which provides insights about the actual worker where a data partition is placed. We make full use of this feature, creating a set containing all global vertices once the graph is fully loaded to the available workers. Since graph partitioning remains unchanged throughout the application execution, the global vertex set can be cached in the memory without being computed again.

With knowledge about all global vertices, we create a `SubGraph` out of the co-located graph partitions in each datacenter. The local updates will be carried out asynchronously on different `SubGraph` instances, and they will be controlled by `DC Manager Threads` on the Spark driver as is depicted in Fig. 3. All transformations made on the `VertexRDD` and the `EdgeRDD` of a `SubGraph` are similar to the original BSP transformations. However, they have been carefully rewritten such that no global vertex is modified during local updates.

V. Experimental Evaluation

We have evaluated the effectiveness and correctness of our prototype implementation in real datacenters, with empirical benchmarking workloads and real-world graph datasets. In this section, we summarize and analyze our experimental results.

A. Methodology

Experiment platforms. We deploy a 10-worker Spark cluster on Google Cloud. Each worker is a regular Ubuntu 16.04 LTS instance, with 2 CPU cores and 7.5GB of memory. Our modified version of GraphX is based on Spark v2.2.0, and the cluster is deployed in the standalone mode.

In particular, two workers are employed in each of the five geographical regions including N. Virginia, Oregon, Tokyo, Belgium, and Sydney. Preliminary measurements on the available bandwidth show $\sim$3Gbps of capacity within a datacenter. Inter-datacenter bandwidth is more than a magnitude lower, ranging from 50Mbps to 230Mbps. The findings are similar to the measurements reported in [11].

TABLE II: WAN bandwidth usage comparison.

Workload		# HSP Global Sync.	# BSP Super-step	HSP Usage (GB)	BSP Usage (GB)	Reduction (%)
enwiki-2013	**PR**	46	74	18.39	27.14	**32.2**
	CC	5	7	0.69	0.91	**23.4**
	SP	7	10	0.59	0.84	**30.6**
uk-2014-host	**PR**	35	52	21.48	31.47	**31.7**
	CC	12	20	0.71	0.95	**25.4**
	SP	15	23	0.50	0.64	**22.4**

Applications. We use three benchmarking applications to evaluate the effectiveness of HSP, including *PageRank (PR)*, *ConnectedComponents (CC)*, and *ShortestPaths (SP)*. We use the default implementations provided by GraphX *without changing a single line of code*. Also, the default graph partitioning strategy is used, which preserves the original edge partitioning in the HDFS input file.

PR represents random walk algorithms, an important category of algorithms that seek to find the steady state in the graph. CC and SP represent graph traversal algorithms. These two categories of algorithms cover the most common vertex programs in practice [22]. Three applications show different degrees of network intensiveness. PR requires more time for synchronization, while SP is more computation-intensive.

Input datasets. We use two web datasets from WebGraph [3]. The key features of the datasets are summarized in Table I. Both datasets have more than 4 million vertices. However, `uk-2014-host` has much fewer edges, making the diameter of the graph much higher. In other words, `enwiki-2013` is more “dense” in terms of vertex connectivity. Experimenting on these two datasets makes a strong case that HSP can work well on natural, real-world graphs.

B. WAN Bandwidth Usage

Apart from the correctness guarantee and the API transparency, the design objective of HSP is WAN efficiency in wide-area graph analytics. As compared to BSP, HSP is expected to significantly reduce the required number of global synchronizations as well as the WAN bandwidth usage. These statistics in the experiments are calculated and summarized in Table II, with all combinations of benchmarks and datasets.

In general, HSP has met our expectations; more than 22% reduction in WAN bandwidth usage can be observed in all workloads. Among the applications, PR benefits most from running under HSP, enjoying an over 30% reduction in total inter-datacenter traffic. CC and SP require fewer global synchronizations before convergence even in BSP, allowing less room for improvement.

HSP also works well on both datasets, despite the differences in graph diameters. Because `uk-2014-host` is partitioned with less fragmentation in 5 datacenters due to a larger diameter, graph traversal applications (CC and SP) can see a higher reduction in the number of global synchronizations under HSP. Another interesting finding is that running CC on `enwiki-2013` under HSP takes only 5 global synchronizations, which reaches the expected minimum in a 5-datacenter setting.

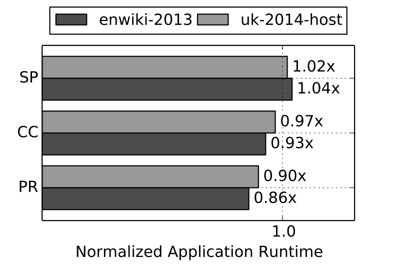


Fig. 4: Application runtime under HSP, normalized by the runtime under BSP.

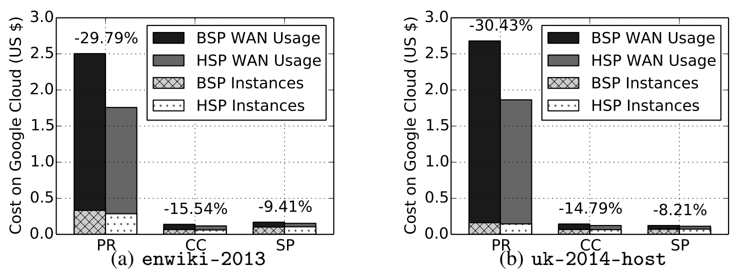


Fig. 5: Estimated cost breakdown for running applications. The calculation follows the Google Cloud pricing model as of July 2017, where 10 instances cost \$0.95/h and WAN traffic is \$0.08/GB.

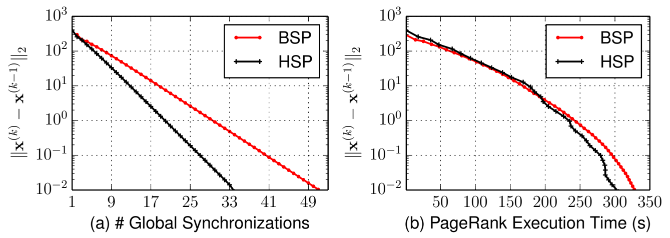


Fig. 6: Rate of convergence analysis for PageRank on `uk-2014-host`. The delta is organized by the number of global synchronizations and the application execution time, respectively.

C. *Performance and Total Cost Analysis*

WAN bandwidth usage, along with instance usage, directly contribute to the monetary cost of running analytics in public cloud. In most cloud pricing models, inter-datacenter traffic is charged by GBs of usage, while instances are charged by hours of machine time.

In our experiments, the runtime of each workload is summarized in Fig. 4. It shows the normalized time for running applications under HSP as compared to BSP. The performance vary on different applications, due to the different degrees of network intensiveness. PR, for example, can achieve 14% less application runtime because it originally spent more time transferring a huge amount of data across-datacenters. SP, on the other hand, has a slightly degraded performance, since the extra computation time incurred by local updates exceeds the reduction in network transfers.

However, we argue that the possible performance degradation is acceptable in considering total cost. We illustrate the cost breakdown of our experiments in Fig. 5.

For PR, WAN usage cost contributes a majority proportion to the final monetary cost. Since both machine time and inter-datacenter traffic have been reduced, HSP is about 30% cheaper than BSP. The rest two applications are relatively less network-intensive, but the WAN usage still constitutes a large proportion. Even though the application runtimes are similar, HSP can still save about 10%.

D. *Rate of Convergence*

To further verify the theoretical proof of a higher rate of convergence in HSP (Sec. III-D), we study the convergence speed of PageRank in our experiments. The results are shown in Fig. 6. Different from Fig. 2b, we measure the ranks' "delta" (in the form of Euclidean norm) between two consecutive global synchronizations instead of the distance to the real ranks, because the real ranks are unknown.

We may observe that Fig. 6a matches the numerical analysis in Fig. 2b. HSP converges linearly, yet at a rate that is 1.49x of BSP with the same number of global synchronizations.

In Fig. 6b we plot the deltas by the end times of global synchronizations. It shows a similar speed of convergence in early stages of execution, while HSP accelerates more in later stages. The reason is that, in the beginning, HSP takes more time running local updates. These local updates usually double the time interval between global synchronizations. However, local updates take much less time later because the local vertices are be considered "more converged," and the progress of HSP accelerates significantly.

VI. Related Work

Wide-area data analytics. New optimization techniques for running big data analytics across geographically distributed datacenters have recently been proposed in the literature. Workloads of interest include general batch jobs [10], [12], [13], [19], [27], streaming analytics [20] and SQL queries [25]. These works established similar background and system settings; however, most of the proposed techniques cannot be applied directly to graph analytics due to the unique characteristics of graph algorithms.

In the context of machine learning applications, Gaia [11] proposed a similar synchronization model to minimize inter-datacenter traffic in machine learning systems by selectively sending updates across datacenter boundaries. However, Gaia focused on the parameter server architecture with the stochastic gradient descent algorithm only, which works in a fundamentally different manner from graph analytics.

Towards optimizing wide-area graph analytics, Mayer *et al.* [18] and Zhou *et al.* [32] proposed different graph partitioning methods that aim to reduce the generated inter-datacenter traffic between supersteps. These works are orthogonal to our work, which focuses on a new synchronization model, given any partitioned graph.

Vertex-centric graph analytics. A variety of graph analytics systems, most of which implemented the BSP model, have been proposed in the literature. Representatives include [17], [33]. These systems focused on computing environments within a high-performance cluster, with the abundantly available bandwidth between worker nodes. Our work proposes a new synchronization model that can be integrated seamlessly with these systems, serving as an alternative of BSP when running across multiple datacenters.

Algorithm-level optimizations such as [21] can certainly reduce the required inter-datacenter traffic. They can be applied to specific categories of graph algorithms, and are orthogonal to optimizations on the system or the synchronization model.

As closely related works, asynchronous (GraphLab [16], PowerGraph [6]) or partial asynchronous (GraphUC [9], Maiter [31]) synchronization models can potentially reduce the need for inter-datacenter communications, but they cannot always guarantee algorithm convergence [28]. Such guarantees in our work root in the strategical switches between synchronous (`global`) and asynchronous (`local`) modes. The limited extent of asynchrony achieves a sweet spot with both minimal inter-datacenter traffic and convergence guarantees.

VII. Concluding Remarks

We introduce Hierarchical Synchronous Parallel (HSP), a new synchronization model that is designed to run graph analytics on geographically distributed datasets efficiently.We have proved that, theoretically and experimentally, HSP guarantees the convergence and correctness of existing graph applications without change. Our prototype implementation and evaluation show that HSP can reduce the WAN bandwidth usage by up to 32%, leading to a significant reduction in monetary cost for analyzing graph data in the cloud. We conclude that HSP is a general, efficient, and readily implementable synchronization model that can benefit wide-area graph analytics systems.

References

[1] A. Auradkar, C. Botev, S. Das, D. De Maagd, A. Feinberg, P. Ganti, L. Gao, B. Ghosh, K. Gopalakrishna, B. Harris, et al. Data Infrastructure at LinkedIn. In *Proc. IEEE International Conference on Data Engineering (ICDE)*, 2012.

[2] S. Banach. Sur les Opérations Dans les Ensembles Abstraits et Leur Application aux Équations Intégrales. *Fundamenta Mathematicae*, 3(1):133–181, 1922.

[3] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *Proc International World Wide Web Conference (WWW)*, 2004.

[4] S. Brin and L. Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Computer Networks and ISDN Systems*, 30(1):107–117, 1998.

[5] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. C. Li, et al. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proc. USENIX Annual Technical Conference (ATC)*, 2013.

[6] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[7] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[8] A. Gupta, F. Yang, J. Govig, A. Kirsch, K. Chan, K. Lai, S. Wu, S. Dhoot, A. Kumar, A. Agiwal, et al. Mesa: Geo-Replicated, Near Real-Time, Scalable Data Warehousing. *Proc. VLDB Endowment*, 7(12):1259–1270, 2014.

[9] M. Han and K. Daudjee. Giraph Unchained: Barrierless Asynchronous Parallel Execution in Pregel-Like Graph Processing Systems. *Proc. VLDB Endowment*, 8(9):950–961, 2015.

[10] B. Heintz, A. Chandra, R. Sitaraman, and J. Weissman. End-to-End Optimization for Geo-Distributed MapReduce. *IEEE Transactions on Cloud Computing*, 2015.

[11] K. Hsieh, A. Harlap, N. Vijaykumar, D. Konomis, G. R. Ganger, P. B. Gibbons, and O. Mutlu. Gaia: Geo-Distributed Machine Learning Approaching LAN Speeds. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[12] C. Hung, L. Golubchik, and M. Yu. Scheduling Jobs Across Geo-Distributed Datacenters. In *Proc. ACM Symposium on Cloud Computing (SoCC)*, 2015.

[13] K. Kloudas, M. Mamede, N. Preguiça, and R. Rodrigues. Pixida: Optimizing Data Parallel Jobs in Wide-Area Data Analytics. *Proc. VLDB Endowment*, 9(2):72–83, 2015.

[14] R. Kumar, P. Raghavan, S. Rajagopalan, D. Sivakumar, A. Tompkins, and E. Upfal. The Web as a Graph. In *Proc. ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, 2000.

[15] G. Lee, J. Lin, C. Liu, A. Lorek, and D. Ryaboy. The Unified Logging Infrastructure for Data Analytics at Twitter. *Proc. VLDB Endowment*, 5(12):1771–1780, 2012.

[16] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: a Framework for Machine Learning and Data Mining in the Cloud. *Proc. VLDB Endowment*, 5(8):716–727, 2012.

[17] 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. In *Proc. ACM SIGMOD*, 2010.

[18] C. Mayer, M. A. Tariq, C. Li, and K. Rothermel. Graph: Heterogeneity-Aware Praph Computation with Adaptive Partitioning. In *Proc. IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2016.

[19] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, P. Bahl, and I. Stoica. Low Latency Geo-Distributed Data Analytics. In *Proc. ACM SIGCOMM*, 2015.

[20] A. Rabkin, M. Arye, S. Sen, V. Pai, and M. Freedman. Aggregation and Degradation in JetStream: Streaming Analytics in the Wide Area. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[21] S. Salihoglu and J. Widom. Optimizing Graph Algorithms on Pregel-Like Systems. *Proc. VLDB Endowment*, 7(7):577–588, 2014.

[22] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson. From Think Like a Vertex to Think Like a Graph. *Proc. VLDB Endowment*, 7(3):193–204, 2013.

[23] J. Ugander, B. Karrer, L. Backstrom, and C. Marlow. The Anatomy of the Facebook Social Graph. *arXiv preprint arXiv:1111.4503*, 2011.

[24] L. G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, 1990.

[25] R. Viswanathan, G. Ananthanarayanan, and A. Akella. Clarinet: Wan-Aware Optimization for Analytics Queries. In *Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[26] A. Vulimiri, C. Curino, P. Godfrey, T. Jungblut, J. Padhye, and G. Varghese. Global Analytics in the Face of Bandwidth and Regulatory Constraints. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[27] A. Vulimiri, C. Curino, P. Godfrey, K. Karanasos, and G. Varghese. WANalytics: Analytics for a Geo-Distributed Data-Intensive World. In *Proc. Conference on Innovative Data Systems Research (CIDR)*, 2015.

[28] D. Yan, Y. Bu, Y. Tian, A. Deshpande, et al. Big Graph Analytics Platforms. *Foundations and Trends® in Databases*, 7(1-2):1–195, 2017.

[29] D. Yan, J. Cheng, K. Xing, Y. Lu, W. Ng, and Y. Bu. Pregel Algorithms for Graph Connectivity Problems with Performance Guarantees. *Proc. VLDB Endowment*, 7(14):1821–1832, 2014.

[30] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[31] Y. Zhang, Q. Gao, L. Gao, and C. Wang. Maiter: an Asynchronous Graph Processing Framework for Delta-Based Accumulative Iterative Computation. *IEEE Transactions on Parallel and Distributed Systems*, 25(8):2091–2100, 2014.

[32] A. Zhou, S. Ibrahim, and B. He. On Achieving Efficient Data Transfer for Graph Processing in Geo-Distributed Datacenters. In *Proc. IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2017.

[33] X. Zhu, W. Chen, W. Zheng, and X. Ma. Gemini: A Computation-Centric Distributed Graph Processing System. In *Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.