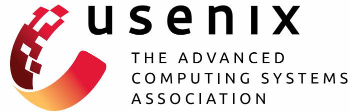


Gaia: Geo-Distributed Machine Learning Approaching LAN Speeds

Kevin Hsieh, Aaron Harlap, Nandita Vijaykumar, Dimitris Konomis, Gregory R. Ganger, and Phillip B. Gibbons, ***Carnegie Mellon University;*** **Onur Mutlu,** ***ETH Zurich and Carnegie Mellon University***

https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/hsieh

This paper is included in the Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17).

March 27–29, 2017 • Boston, MA, USA

ISBN 978-1-931971-37-9

Open access to the Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation is sponsored by USENIX.

Gaia: Geo-Distributed Machine Learning Approaching LAN Speeds

Kevin Hsieh† Aaron Harlap† Nandita Vijaykumar† Dimitris Konomis†
Gregory R. Ganger† Phillip B. Gibbons† Onur Mutlu§†
†Carnegie Mellon University §ETH Zürich

Abstract

Machine learning (ML) is widely used to derive useful information from large-scale data (such as user activities, pictures, and videos) generated at increasingly rapid rates, *all over the world*. Unfortunately, it is infeasible to move all this globally-generated data to a centralized data center before running an ML algorithm over it—moving large amounts of raw data over wide-area networks (WANs) can be extremely slow, and is also subject to the constraints of privacy and data sovereignty laws. This motivates the need for a geo-distributed ML system spanning *multiple data centers*. Unfortunately, communicating over WANs can significantly degrade ML system performance (by as much as 53.7× in our study) because the communication overwhelms the limited WAN bandwidth.

Our goal in this work is to develop a *geo-distributed* ML system that (1) employs an intelligent communication mechanism over WANs to efficiently utilize the scarce WAN bandwidth, while retaining the accuracy and correctness guarantees of an ML algorithm; and (2) is generic and flexible enough to run a wide range of ML algorithms, without requiring any changes to the algorithms.

To this end, we introduce a new, general geo-distributed ML system, *Gaia*, that decouples the communication *within* a data center from the communication *between* data centers, enabling different communication and consistency models for each. We present a new ML synchronization model, *Approximate Synchronous Parallel (ASP)*, whose key idea is to dynamically eliminate *insignificant* communication between data centers while still guaranteeing the correctness of ML algorithms. Our experiments on our prototypes of Gaia running across 11 Amazon EC2 global regions and on a cluster that emulates EC2 WAN bandwidth show that Gaia provides 1.8–53.5× speedup over two state-of-the-art distributed ML systems, and is within 0.94–1.40× of the speed of running the same ML algorithm on machines on a local area network (LAN).

1. Introduction

Machine learning (ML) is very widely used across a variety of domains to extract useful information from large-scale data. It has many classes of applications such as image or video classification (e.g., [24,39,65]), speech recognition (e.g., [4]), and topic modeling (e.g., [10]). These applications analyze massive amounts of data from user activities, pictures, videos, etc., which are generated at very rapid rates, *all over the world*. Many large organizations, such as Google [28], Microsoft [51], and Amazon [7], operate tens of data centers globally to minimize their service latency to end-users, and store massive quantities of data all over the globe [31,33,36, 41,57,58,71–73,76].

A commonly-used approach to run an ML application over such rapidly generated data is to *centralize* all data into *one data center* over wide-area networks (WANs) before running the ML application [9,12,44,68]. However, this approach can be prohibitively difficult because: (1) WAN bandwidth is a scarce resource, and hence moving all data can be extremely slow [12,57]. Furthermore, the fast growing rate of image and video generation will eventually saturate the total WAN bandwidth, whose growth has been decelerating for many years [67,73]. (2) Privacy and data sovereignty laws in some countries prohibit transmission of *raw data* across national or continental borders [12,72,73].

This motivates the need to *distribute* an ML system across *multiple data centers*, globally. In such a system, large amounts of raw data are stored locally in different data centers, and the ML algorithms running over the distributed data communicate between data centers using WANs. Unfortunately, existing large-scale distributed ML systems [5,13,45,47,50,77] are suitable only for data residing *within* a single data center. Our experiments using three state-of-the-art distributed ML systems (Bösen [74], IterStore [17], and GeePS [18]) show that operating these systems across as few as two data centers (over WANs) can cause a slowdown of 1.8–53.7× (see Section 2.3 and Section 6) relative to their performance within a data center (over LANs). Existing systems that do address challenges in geo-distributed data analytics [12,33,36, 41,57,58,71–73] do *not* consider the broad class of important, sophisticated ML algorithms commonly run on ML systems — they focus instead on other types of computation, e.g., map-reduce or SQL.

Our goal in this work is to develop a geo-distributed ML system that (1) minimizes communication over WANs, so that the system is not bottlenecked by the scarce WAN bandwidth; and (2) is general enough to be applicable to a wide variety of ML algorithms, without requiring any changes to the algorithms themselves.

To achieve these goals, such a system needs to address two key challenges. First, to efficiently utilize the limited (and heterogeneous) WAN bandwidth, we need to find an effective communication model that minimizes communication over WANs but still retains the correctness guarantee for an ML algorithm. This is difficult because ML algorithms typically require extensive communication to exchange updates that keep the global ML model sufficiently consistent across data centers. These updates are

required to be timely, irrespective of the available network bandwidth, to ensure algorithm correctness. Second, we need to design a *general* system that effectively handles WAN communication for ML algorithms without requiring any algorithm changes. This is challenging because the communication patterns vary significantly across different ML algorithms [37, 54, 60, 64, 66, 69]. Altering the communication across systems can lead to different tradeoffs and consequences for different algorithms [83].

In this work, we introduce *Gaia*, a new general, geo-distributed ML system that is designed to efficiently operate over a collection of data centers. Gaia builds on the widely used *parameter server* architecture (e.g., [5, 6, 13, 16, 17, 20, 34, 45, 74, 77]) that provides ML worker machines with a distributed global shared memory abstraction for the ML model parameters they collectively train until *convergence* to fit the input data. The key idea of Gaia is to maintain an approximately-correct copy of the global ML model *within* each data center, and dynamically eliminate any unnecessary communication *between* data centers. Gaia enables this by *decoupling* the synchronization (i.e., communication/consistency) model within a data center from the synchronization model between different data centers. This differentiation allows Gaia to run a conventional synchronization model [19, 34, 74] that maximizes utilization of the more-freely-available LAN bandwidth *within* a data center. At the same time, across different data centers, Gaia employs a new synchronization model, called *Approximate Synchronous Parallel (ASP)*, which makes more efficient use of the scarce and heterogeneous WAN bandwidth. By ensuring that each ML model copy in different data centers is *approximately correct* based on a precise notion defined by ASP, we guarantee ML algorithm convergence.

ASP is based on a key finding that the vast majority of updates to the global ML model parameters from each ML worker machine are *insignificant*. For example, our study of three classes of ML algorithms shows that more than 95% of the updates produce less than a 1% change to the parameter value. With ASP, these insignificant updates to the same parameter within a data center are *aggregated* (and thus not communicated to other data centers) until the aggregated updates are significant enough. ASP allows the ML programmer to specify the *function* and the *threshold* to determine the significance of updates for each ML algorithm, while providing default configurations for unmodified ML programs. For example, the programmer can specify that all updates that produce more than a 1% change are significant. ASP ensures all significant updates are synchronized across all model copies in a timely manner. It dynamically adapts communication to the available WAN bandwidth between pairs of data centers and uses special *selective barrier* and *mirror clock* control messages to ensure algorithm convergence even during a period of sudden fall (negative spike) in available WAN bandwidth.

In contrast to a state-of-the-art communication-efficient synchronization model, Stale Synchronous Parallel (SSP) [34], which bounds how *stale (i.e., old)* a parameter can be, ASP bounds how *inaccurate* a parameter can be, in comparison to the most up-to-date value. Hence, it provides high flexibility in performing (or not performing) updates, as the server can delay synchronization *indefinitely* as long as the aggregated update is insignificant.

We build two prototypes of Gaia on top of two state-of-the-art parameter server systems, one specialized for CPUs [17] and another specialized for GPUs [18]. We deploy Gaia across 11 regions on Amazon EC2, and on a local cluster that emulates the WAN bandwidth across different Amazon EC2 regions. Our evaluation with three popular classes of ML algorithms shows that, compared to two state-of-the-art parameter server systems [17, 18] deployed on WANs, Gaia: (1) significantly improves performance, by 1.8–53.5×, (2) has performance within 0.94–1.40× of running the same ML algorithm on a LAN in a single data center, and (3) significantly reduces the monetary cost of running the same ML algorithm on WANs, by 2.6–59.0×.

We make three major contributions:

- To our knowledge, this is the first work to propose a general geo-distributed ML system that (1) differentiates the communication over a LAN from the communication over WANs to make efficient use of the scarce and heterogeneous WAN bandwidth, and (2) is general and flexible enough to deploy a wide range of ML algorithms while requiring *no* change to the ML algorithms themselves.
- We propose a new, efficient ML synchronization model, Approximate Synchronous Parallel (ASP), for communication between parameter servers across data centers over WANs. ASP guarantees that each data center's view of the ML model parameters is approximately the same as the "fully-consistent" view and ensures that all significant updates are synchronized in time. We prove that ASP provides a theoretical guarantee on algorithm convergence for a widely used ML algorithm, stochastic gradient descent.
- We build two prototypes of our proposed system on CPU-based and GPU-based ML systems, and we demonstrate their effectiveness over 11 globally distributed regions with three popular ML algorithms. We show that our system provides significant performance improvements over two state-of-the-art distributed ML systems [17,18], and significantly reduces the communication overhead over WANs.

2. Background and Motivation

We first introduce the architectures of widely-used distributed ML systems. We then discuss WAN bandwidth constraints and study the performance implications of running two state-of-the-art ML systems over WANs.

2.1. Distributed Machine Learning Systems

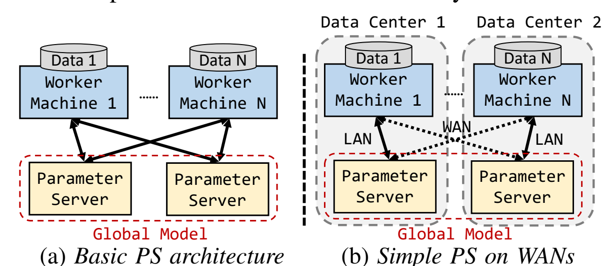
While ML algorithms have different types across different domains, almost all have the same goal—searching for

the best *model* (usually a set of *parameters*) to describe or explain the input *data* [77]. For example, the goal of an image classification neural network is to find the parameters (of the neural network) that can most accurately classify the input images. Most ML algorithms iteratively refine the ML model until it *converges* to fit the data. The correctness of an ML algorithm is thus determined by whether or not the algorithm can *accurately converge* to the best model for its input data.

As the input data to an ML algorithm is usually enormous, processing all input data on a single machine can take an unacceptably long time. Hence, the most common strategy to run a large-scale ML algorithm is to distribute the input data among *multiple* worker machines, and have each machine work on a *shard* of the input data in parallel with other machines. The worker machines communicate with each other periodically to *synchronize* the updates from other machines. This strategy, called *data parallelism* [14], is widely used in many popular ML systems (e.g., [1, 2, 5, 13, 45, 47, 50, 77]).

There are many large-scale distributed ML systems, such as ones using the MapReduce [14] abstraction (e.g., MLlib [2] and Mahout [1]), ones using the graph abstraction (e.g., GraphLab [47] and PowerGraph [26]), and ones using the parameter server abstraction (e.g., Petuum [77] and TensorFlow [5]). Among them, the parameter server architecture provides a performance advantage[1] over other systems for many ML applications and has been widely adopted in many ML systems.

Figure 1a illustrates the high-level overview of the parameter server (PS) architecture. In such an architecture, each parameter server keeps a shard of the global model parameters as a key-value store, and each worker machine communicates with the parameter servers to `READ` and `UPDATE` the corresponding parameters. The major benefit of this architecture is that it allows ML programmers to view all model parameters as a global shared memory, and leave the parameter servers to handle the synchronization.



(a) *Basic PS architecture* (b) *Simple PS on WANs*

Figure 1: Overview of the parameter server architecture

Synchronization among workers in a distributed ML system is a critical operation. Each worker needs to see other workers' updates to the global model to compute more accurate updates using fresh information. However, synchronization is a high-cost operation that can significantly slow down the workers and reduce the benefits of parallelism. The trade-off between *fresher updates* and *communication overhead* leads to three major synchronization models: (1) **Bulk Synchronous Parallel (BSP)** [70], which synchronizes all updates after each worker goes through its shard of data; all workers need to see the most up-to-date model before proceeding to the next iteration, (2) **Stale Synchronous Parallel (SSP)** [34], which allows the fastest worker to be ahead of the slowest worker by up to a bounded number of iterations, so the fast workers may proceed with a *bounded stale* (i.e., old) model, and (3) **Total Asynchronous Parallel (TAP)** [59], which removes the synchronization between workers completely; all workers keep running based on the results of best-effort communication (i.e., each sends/receives as many updates as possible). Both BSP and SSP guarantee algorithm convergence [19, 34], while there is no such guarantee for TAP. Most state-of-the-art parameter servers implement both BSP and SSP (e.g., [5, 16–18, 34, 45, 77]).

As discussed in Section 1, many ML applications need to analyze geo-distributed data. For instance, an image classification system would use pictures located at different data centers as its input data to keep improving its classification using the pictures generated continuously all over the world. Figure 1b depicts the straightforward approach to achieve this goal. In this approach, the worker machines in each data center (i.e., within a LAN) handle the input data stored in the corresponding data center. The parameter servers are evenly distributed across multiple data centers. Whenever the communication between a worker machine and a parameter server crosses data centers, it does so on WANs.

2.2. WAN Network Bandwidth and Cost

WAN bandwidth is a very scarce resource [42, 58, 73] relative to LAN bandwidth. Moreover, the high cost of adding network bandwidth has resulted in a deceleration of WAN bandwidth growth. The Internet capacity growth has fallen steadily for many years, and the annual growth rates have lately settled into the low-30 percent range [67].

To quantify the scarcity of WAN bandwidth between data centers, we measure the network bandwidth between all pairs of Amazon EC2 sites in 11 different regions (Virginia, California, Oregon, Ireland, Frankfurt, Tokyo, Seoul, Singapore, Sydney, Mumbai, and São Paulo). We use `iperf3` [23] to measure the network bandwidth of each pair of different regions for five rounds, and then calculate the average bandwidth. Figure 2 shows the average network bandwidth between each pair of different regions. We make two observations.

First, the WAN bandwidth between data centers is 15× smaller than the LAN bandwidth within a data center on average, and up to 60× smaller in the worst case (for Singapore ↔ São Paulo). Second, the WAN bandwidth *varies significantly* between different regions. The WAN bandwidth between geographically-close regions (e.g., Oregon ↔ California or Tokyo ↔ Seoul) is up to 12× of the bandwidth between distant regions (e.g., Singapore

[1] For example, a state-of-the-art parameter server, IterStore [17], is shown to outperform PowerGraph [26] by 10× for Matrix Factorization. In turn, PowerGraph is shown to match the performance of GraphX [27], a Spark [79] based system.

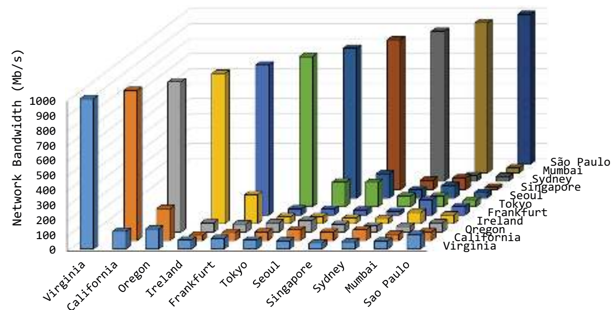


Figure 2: Measured network bandwidth between Amazon EC2 sites in 11 different regions

↔ São Paulo). As Section 2.3 shows, the scarcity and variation of the WAN bandwidth can significantly degrade the performance of state-of-the-art ML systems.

Another important challenge imposed by WANs is the *monetary cost* of communication. In data centers, the cost of WANs far exceeds the cost of a LAN and makes up a significant fraction of the overall cost [29]. Cloud service providers, such as Amazon EC2, charge an extra fee for WAN communication while providing LAN communication free of charge. The cost of WAN communication can be much higher than the cost of the machines themselves. For example, the cost of two machines in Amazon EC2 communicating at the rate of the average WAN bandwidth between data centers is up to 38× of the cost of renting these two machines [8]. These costs make running ML algorithms on WANs much more expensive than running them on a LAN.

2.3. ML System Performance on WANs

We study the performance implications of deploying distributed ML systems on WANs using two state-of-the-art parameter server systems, IterStore [17] and Bösen [74]. Our experiments are conducted on our local 22-node cluster that emulates the WAN bandwidth between Amazon EC2 data centers, the accuracy of which is validated against a real Amazon EC2 deployment (see Section 5.1 for details). We run the same ML application, *Matrix Factorization* [25] (Section 5.2), on both systems.

For each system, we evaluate both BSP and SSP as the synchronization model (Section 2.1), with four deployment settings: (1) *LAN*, deployment within a single data center, (2) *EC2-ALL*, deployment across 11 aforementioned EC2 regions, (3) *V/C WAN*, deployment across two data centers that have the same WAN bandwidth as that between Virginia and California (Figure 2), representing a distributed ML setting within a continent, and (4) *S/S WAN*, deployment across two data centers that have the same WAN bandwidth as that between Singapore and São Paulo, representing the lowest WAN bandwidth between any two Amazon EC2 regions.

Figure 3 shows the normalized execution time until algorithm convergence across the four deployment settings. All results are normalized to IterStore using BSP on a LAN. The data label on each bar represents how much slower the WAN setting is than its *respective* LAN setting

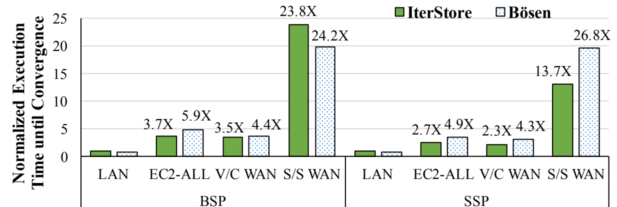


Figure 3: Normalized execution time until ML algorithm convergence when deploying two state-of-the-art distributed ML systems on a LAN and WANs

for the given system, e.g., Bösen-BSP on EC2-ALL is 5.9× slower than Bösen-BSP on LAN.

As we see, both systems suffer significant performance degradation when deployed across multiple data centers. When using BSP, IterStore is 3.5× to 23.8× slower on WANs than it is on a LAN, and Bösen is 4.4× to 24.2× slower. While using SSP can reduce overall execution times of both systems, both systems still show significant slowdown when run on WANs (2.3× to 13.7× for IterStore, and 4.3× to 26.8× for Bösen). We conclude that simply running state-of-the-art distributed ML systems on WANs can seriously slow down ML applications, and thus we need a new distributed ML system that can be effectively deployed on WANs.

3. Our Approach: Gaia

We introduce Gaia, a general ML system that can be effectively deployed on WANs to address the increasing need to run ML applications *directly* on geo-distributed data. We identify two key challenges in designing such a system (Section 3.1). We then introduce the system architecture of Gaia, which differentiates the communication *within* a data center from the communication *between* different centers (Section 3.2). Our approach is based on the key empirical finding that the vast majority of communication within an ML system results in *insignificant* changes to the state of the global model (Section 3.3). In light of this finding, we design a new ML synchronization model, called *Approximate Synchronous Parallel (ASP)*, which can eliminate the insignificant communication while ensuring the convergence and accuracy of ML algorithms. We describe ASP in detail in Section 3.4. Finally, Section 3.5 summarizes our theoretical analysis of how ASP guarantees algorithm convergence for a widely-used ML algorithm, stochastic gradient descent (SGD) (the full proof is in Appendix A).

3.1. Key Challenges

There are two key challenges in designing a general and effective ML system on WANs.

Challenge 1. *How to effectively communicate over WANs while retaining algorithm convergence and accuracy?* As we see above, state-of-the-art distributed ML systems can overwhelm the scarce WAN bandwidth, causing significant slowdowns. We need a mechanism that significantly reduces the communication between data centers so that the system can provide competitive performance. However, reducing communication can affect the accuracy of an ML algorithm. A poor choice

of synchronization model in a distributed ML system can prevent the ML algorithm from converging to the optimal point (i.e., the best model to explain or fit the input data) that one can achieve when using a proper synchronization model [11, 59]. Thus, we need a mechanism that can reduce communication intensity while ensuring that the communication occurs in a *timely* manner, even when the network bandwidth is extremely stringent. This mechanism should provably guarantee algorithm convergence *irrespective* of the network conditions.

Challenge 2. *How to make the system generic and work for ML algorithms without requiring modification?* Developing an effective ML algorithm takes significant effort and experience, making it a large burden for the ML algorithm developers to change the algorithm when deploying it on WANs. Our system should work across a wide variety of ML algorithms, preferably *without any change* to the algorithms themselves. This is challenging because different ML algorithms have different communication patterns, and the implication of reducing communication can vary significantly among them [37, 54, 60, 64, 66, 69, 83].

3.2. Gaia System Overview

We propose a new ML system, Gaia, that addresses the two key challenges in designing a general and effective ML system on WANs. Gaia is built on top the popular parameter server architecture, which is proven to be effective on a wide variety of ML algorithms (e.g., [5, 6, 13, 16, 17, 20, 34, 45, 74, 77]). As discussed in Section 2.1, in the parameter server architecture, *all* worker machines synchronize with each other through parameter servers to ensure that the global model state is up-to-date. While this architecture guarantees algorithm convergence, it also requires substantial communication between worker machines and parameter servers. To make Gaia effective on WANs while fully utilizing the abundant LAN bandwidth, we design a new system architecture to *decouple* the synchronization within a data center (LANs) from the synchronization across different data centers (WANs).

Figure 4 shows an overview of Gaia. In Gaia, each data center has some worker machines and parameter servers. Each worker machine processes a shard of the input data stored in its data center to achieve data parallelism (Section 2.1). The parameter servers in each data center collectively maintain a version of the *global model copy* (❶), and each parameter server handles a shard of this global model copy. A worker machine *only* `READs` and `UPDATEs` the global model copy in its data center.

To reduce the communication overhead over WANs, the global model copy in each data center is only *approximately correct*. This design enables us to eliminate the insignificant, and thus unnecessary, communication across different data centers. We design a new synchronization model, called Approximate Synchronous Parallel (ASP ❷), between parameter servers *across* different data centers to ensure that each global model copy is approximately correct even with very low WAN bandwidth.

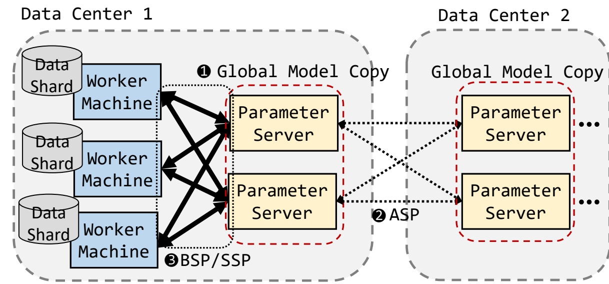


Figure 4: Gaia system overview

Section 3.4 describes the details of ASP. On the other hand, worker machines and parameter servers *within* a data center synchronize with each other using the conventional BSP (Bulk Synchronous Parallel) or SSP (Stale Synchronous Parallel) models (❸). These models allow worker machines to quickly observe fresh updates that happen *within* a data center. Furthermore, worker machines and parameter servers within a data center can employ more aggressive communication schemes such as sending updates early and often [19, 74] to fully utilize the abundant (and free) network bandwidth on a LAN.

3.3. Study of Update Significance

As discussed above, Gaia reduces the communication overhead over WANs by eliminating insignificant communication. To understand the benefit of our approach, we study the *significance* of the updates sent from worker machines to parameter servers. We study three classes of popular ML algorithms: *Matrix Factorization (MF)* [25], *Topic Modeling (TM)* [10], and *Image Classification (IC)* [43] (see Section 5.2 for descriptions). We run all the algorithms until convergence, analyze all the updates sent from worker machines to parameter servers, and compare the change they cause on the parameter value when the servers receive them. We define an update to be *significant* if it causes $S\%$ change on the parameter value, and we vary S, the significance threshold, between 0.01 and 10. Figure 5 shows the percentage of insignificant updates among all updates, for different values of S.

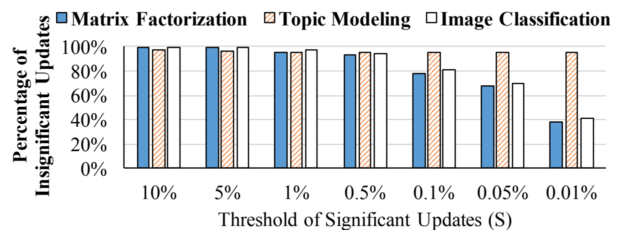


Figure 5: Percentage of insignificant updates

As we see, the vast majority of updates in these algorithms are *insignificant*. Assuming the significance threshold is 1%, 95.2% / 95.6% / 97.0% of all updates are insignificant for *MF* / *TM* / *IC*. When we relax the significance threshold to 5%, 98.8% / 96.1% / 99.3% of all updates are insignificant. Thus, most of the communication changes the ML model state only very slightly.

It is worth noting that our finding is consistent with the findings of prior work [21, 22, 40, 47, 80] on other

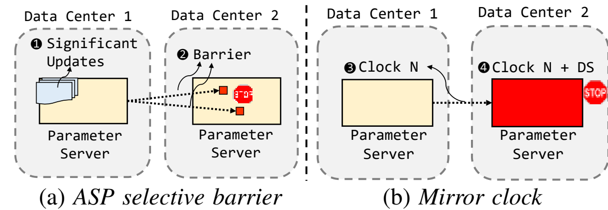
ML algorithms, such as PageRank and Lasso. These works observe that in these ML algorithms, not all model parameters converge to their optimal value within the same number iterations — a property called *non-uniform convergence* [78]. Instead of examining the convergence rate, we *quantify* the *significance* of updates with various significance thresholds, which provides a unique opportunity to reduce the communication over WANs.

3.4. Approximate Synchronous Parallel

The goal of our new synchronization model, Approximate Synchronous Parallel (ASP), is to ensure that the global model copy in each data center is approximately correct. In this model, a parameter server shares only the significant updates with other data centers, and ASP ensures that these updates can be seen by all data centers in a timely fashion. ASP achieves this goal by using three techniques: (1) the significance filter, (2) ASP selective barrier, and (3) ASP mirror clock. We describe them in order.

The significance filter. ASP takes two inputs from an ML programmer to determine whether or not an update is significant. They are: (1) a *significance function* and (2) an *initial significance threshold*. The significance function returns the significance of each update. We define an update as significant if its significance is larger than the threshold. For example, an ML programmer can define the significance function as the update's magnitude relative to the current value ($|\frac{Update}{Value}|$), and set the initial significance threshold to 1%. The significance function can be more sophisticated if the impact of parameter changes to the model is not linear, or the importance of parameters is non-uniform (see Section 4.3). A parameter server aggregates updates from the local worker machines and shares the aggregated updates with other data centers when the aggregated updates become significant. To ensure that the algorithm can converge to the optimal point, ASP automatically reduces the significance threshold over time (specifically, if the original threshold is v, then the threshold at iteration t of the ML algorithm is $v/\sqrt{t}$).

ASP selective barrier. While we can greatly reduce the communication overhead over WANs by sending only the significant updates, the WAN bandwidth might still be insufficient for such updates. In such a case, the significant updates can arrive too late, and we might not be able to bound the deviation between different global model copies. ASP handles this case with the *ASP selective barrier* (Figure 6a) control message. When a parameter server receives the significant updates (❶) at a rate that is higher than the WAN bandwidth can support, the parameter server first sends the indexes of these significant updates (as opposed to sending both the indexes and the update values together) via an ASP selective barrier (❷) to the other data centers. The receiver of an ASP selective barrier blocks its local worker from reading the specified parameters until it receives the significant updates from the sender of the barrier. This technique ensures that all worker machines in each data center are aware of the significant updates after a bounded network latency, and they wait *only* for these updates. The worker machines can make progress as long as they do not depend on any of these parameters.



(a) *ASP selective barrier* (b) *Mirror clock*

Figure 6: The synchronization mechanisms of ASP

Mirror clock. The ASP select barrier ensures that the latency of the significant updates is no more than the network latency. However, it assumes that 1) the underlying WAN bandwidth and latency are fixed so that the network latency can be bounded, and 2) such latency is short enough so that other data centers can be aware of them in time. In practice, WAN bandwidth can fluctuate over time [35], and the WAN latency can be intolerably high for some ML algorithms. We need a mechanism to guarantee that the worker machines are aware of the significant updates in time, irrespective of the WAN bandwidth or latency.

We use the *mirror clock* (Figure 6b) to provide this guarantee. When each parameter server receives all the updates from its local worker machines at the end of a clock (e.g., an iteration), it reports its clock to the servers that are in charge of the same parameters in the other data centers. When a server detects its clock is ahead of the slowest server that shares the same parameters by a predefined threshold DS (data center staleness), the server blocks its local worker machines from reading its parameters until the slowest mirror server catches up. In the example of Figure 6b, the server clock in Data Center 1 is N, while the server clock in Data Center 2 is $(N+DS)$. As their difference reaches the predefined limit, the server in Data Center 2 blocks its local worker from reading its parameters. This mechanism is similar to the concept of SSP [34], but we use it only as the last resort to guarantee algorithm convergence.

3.5. Summary of Convergence Proof

In this section, we summarize our proof showing that a popular, broad class of ML algorithms are guaranteed to converge under our new ASP synchronization model. The class we consider are ML algorithms expressed as convex optimization problems that are solved using distributed stochastic gradient descent.

The proof follows the outline of prior work on SSP [34], with a new challenge, i.e., our new ASP synchronization model allows the synchronization of insignificant updates to be delayed indefinitely. To prove algorithm convergence, our goal is to show that the distributed execution of an ML algorithm results in a set of parameter values that are very close (practically identical) to the values that would be obtained under a serialized execution.

Let f denote the objective function of an optimization problem, whose goal is to minimize f. Let $\tilde{\boldsymbol{x}}_t$ denote the sequence of noisy (i.e., inaccurate) *views* of the parameters, where $t = 1, 2, ..., T$ is the index of each view over time. Let $\boldsymbol{x}^*$ denote the value that minimizes f. Intuitively, we would like $f_t(\tilde{\boldsymbol{x}}_t)$ to approach $f(\boldsymbol{x}^*)$ as $t \to \infty$. We call the difference between $f_t(\tilde{\boldsymbol{x}}_t)$ and $f(\boldsymbol{x}^*)$ *regret*. We can prove $f_t(\tilde{\boldsymbol{x}}_t)$ approaches $f(\boldsymbol{x}^*)$ as $t \to \infty$ by proving that the *average regret*, $\frac{R[X]}{T} \to 0$ as $T \to \infty$.

Mathematically, the above intuition is formulated with Theorem 1. The details of the proof and the notations are in Appendix A.

Theorem 1. ***(Convergence of SGD under ASP).*** *Suppose that, in order to compute the minimizer x^* of a convex function $f(\boldsymbol{x}) = \sum_{t=1}^{T} f_t(\boldsymbol{x})$, with $f_t, t = 1, 2, \ldots, T$, convex, we use stochastic gradient descent on one component ∇f_t at a time. Suppose also that 1) the algorithm is distributed in D data centers, each of which uses P machines, 2) within each data center, the SSP protocol is used, with a fixed staleness of s, and 3) a fixed mirror clock difference Δ_c is allowed between any two data centers. Let $\boldsymbol{u}_t = -\eta_t \nabla f_t(\tilde{\boldsymbol{x}}_t)$, where the step size η_t decreases as $\eta_t = \frac{\eta}{\sqrt{t}}$ and the significance threshold v_t decreases as $v_t = \frac{v}{\sqrt{t}}$. If we further assume that: $\|\nabla f_t(\boldsymbol{x})\| \leq L$, $\forall \boldsymbol{x} \in dom(f_t)$ and $\max(D(\boldsymbol{x}, \boldsymbol{x}')) \leq \Delta^2, \forall \boldsymbol{x}, \boldsymbol{x}' \in dom(f_t)$. Then, as $T \to \infty$, the regret $R[X] = \sum_{t=1}^{T} f_t(\tilde{\boldsymbol{x}}_t) - f(\boldsymbol{x}^*) = O(\sqrt{T})$ and therefore $\lim_{T \to \infty} \frac{R[X]}{T} \to 0$.*

4. Implementation

We introduce the key components of Gaia in Section 4.1, and discuss the operation and design of individual components in the remaining sections.

4.1. Gaia System Key Components

Figure 7 presents the key components of Gaia. All of the key components are implemented in the parameter servers, and can be transparent to the ML programs and the worker machines. As we discuss above, we decouple the synchronization within a data center (LANs) from the synchronization across different data centers (WANs). The *local server* (❶) in each parameter server handles the synchronization between the worker machines in the same data center using the conventional BSP or SSP models. On the other hand, the *mirror server* (❷) and the *mirror client* (❸) handle the synchronization with other data centers using our ASP model. Each of these three components runs as an individual thread.

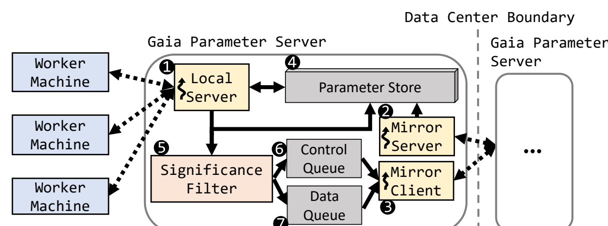


Figure 7: Key components of Gaia

4.2. System Operations and Communication

We present a walkthrough of major system operations and communication.

`UPDATE` from a worker machine. When a *local server* (❶) receives a parameter update from a worker machine, it updates the parameter in its *parameter store* (❹), which maintains the parameter value and its accumulated update. The local server then invokes the *significance filter* (❺) to determine whether or not the accumulated update of this parameter is significant. If it is, the significance filter sends a `MIRROR UPDATE` request to the mirror client (❸) and resets the accumulated update for this parameter.

Messages from the significance filter. The significance filter sends out three types of messages. First, as discussed above, it sends a `MIRROR UPDATE` request to the mirror client through the data queue (❼). Second, when the significance filter detects that the arrival rate of significant updates is higher than the underlying WAN bandwidth that it monitors at every iteration, it first sends an `ASP Barrier` (Section 3.4) to the control queue (❻) before sending the `MIRROR UPDATE`. The mirror client (❸) prioritizes the control queue over the data queue, so that the barrier is sent out earlier than the update. Third, to maintain the mirror clock (Section 3.4), the significance filter also sends a `MIRROR CLOCK` request to the control queue at the end of each clock in the local server.

Operations in the mirror client. The mirror client thread wakes up when there is a request from the control queue or the data queue. Upon waking up, the mirror client walks through the queues, packs together the messages to the same destination, and sends them.

Operations in the mirror server. The mirror server handles above messages (`MIRROR UPDATE`, `ASP BARRIER`, and `MIRROR CLOCK`) according to our ASP model. For `MIRROR UPDATE`, it applies the update to the corresponding parameter in the parameter store. For `ASP BARRIER`, it sets a flag in the parameter store to block the corresponding parameter from being read until it receives the corresponding `MIRROR UPDATE`. For `MIRROR CLOCK`, the mirror server updates its local mirror clock state for each parameter server in other data centers, and enforces the predefined clock difference threshold *DS* (Section 3.4).

4.3. Advanced Significance Functions

As we discuss in Section 3.4, the significance filter allows the ML programmer to specify a custom *significance function* to calculate the significance of each update. By providing an advanced significance function, Gaia can be more effective at eliminating the insignificant communication. If several parameters are always referenced together to calculate the next update, the significance function can take into account the values of all these parameters. For example, if three parameters a, b, and c are always used as $a \cdot b \cdot c$ in an ML algorithm, the significance of a, b, and c can be calculated as the change on $a \cdot b \cdot c$. If one of them is 0, any change in another parameter, however large it may be, is insignificant. Similar principles can be applied to model parameters that are non-linear or

non-uniform. For unmodified ML programs, the system applies default significance functions, such as the relative magnitude of an update for each parameter.

4.4. Tuning of Significance Thresholds

The user of Gaia can specify two different goals for Gaia: (1) speed up algorithm convergence by fully utilizing the available WAN bandwidth and (2) minimize the communication cost on WANs. In order to achieve either of these goals, the significance filter maintains two significance thresholds and dynamically tunes these thresholds. The first threshold is the *hard* significance threshold. The purpose of this threshold is to guarantee ML algorithm convergence. As we discuss in our theoretical analysis (Section 3.5), the initial threshold is provided by the ML programmer or a default system setting, and the significance filter reduces it over time. Every update whose significance is above the hard threshold is guaranteed to be sent to other data centers. The second threshold is the *soft* significance threshold. The purpose of it is to use underutilized WAN bandwidth to speed up convergence. This threshold is tuned based on the arrival rate of the significant updates and the underlying WAN bandwidth. When the user chooses to optimize the first goal (speed up algorithm convergence), the system lowers the soft significance threshold whenever there is underutilized WAN bandwidth. The updates whose significance is larger than the soft significance threshold are sent in a best-effort manner. On the other hand, if the goal of the system is to minimize the WAN communication costs, the soft significance threshold is not activated.

While the configuration of the initial hard threshold depends on how error tolerant each ML algorithm is, a simple and conservative threshold (such as 1%–2%) is likely to work in most cases. This is because most ML algorithms initialize their parameters with random values, and make large changes to their model parameters at early phases. Thus, they are more error tolerant at the beginning. As Gaia reduces the threshold over time, its accuracy loss is limited. An ML expert can choose a more aggressive threshold based on domain knowledge of the ML algorithm.

4.5. Overlay Network and Hub

While Gaia can eliminate the insignificant updates, each data center needs to *broadcast* the significant updates to all the other data centers. This broadcast-based communication could limit the scalability of Gaia when we deploy Gaia to many data centers. To make Gaia more scalable with more data centers, we use the concept of overlay networks [48].

As we discuss in Section 2.2, the WAN bandwidth between geographically-close regions is much higher than that between distant regions. In light of this, Gaia supports having geographically-close data centers form a *data center group*. Servers in a data center group send their significant updates only to the other servers in the same group. Each group has *hub* data centers that are in charge of aggregating all the significant updates within the group, and sending to the hubs of the other groups. Similarly, a hub data center broadcasts the aggregated significant updates from other groups to the other data centers within its group. Each data center group can designate different hubs for communication with different data center groups, so the system can utilize more links within a data center group. For example, the data centers in Virginia, California, and Oregon can form a data center group and assign the data center in Virginia as the hub to communicate with the data centers in Europe and the data center in Oregon as the hub to communicate with the data centers is Asia. This design allows Gaia to broadcast the significant updates with lower communication cost.

5. Methodology

5.1. Experiment Platforms

We use three different platforms for our evaluation.

Amazon-EC2. We deploy Gaia to 22 machines spread across 11 EC2 regions as we show in Figure 2. In each EC2 region we start two instances of type `c4.4xlarge` or `m4.4xlarge` [8], depending on their availability. Both types of instances have 16 CPU cores and at least 30GB RAM, running 64-bit Ubuntu 14.04 LTS (HVM). In all, our deployment uses 352 CPU cores and 1204 GB RAM.

Emulation-EC2. As the monetary cost of running all experiments on EC2 is too high, we run some experiments on our local cluster that emulates the computation power and WAN bandwidth of EC2. We use the same number of machines (22) in our local cluster. Each machine is equipped with a 16-core Intel Xeon CPU (E5-2698), an NVIDIA Titan X GPU, 64GB RAM, a 40GbE NIC, and runs the same OS as above. The computation power and the LAN speeds of our machines are higher than the ones we get from EC2, so we slow down the CPU and LAN speeds to match the speeds on EC2. We model the measured EC2 WAN bandwidth (Figure 2) with the Linux Traffic Control tool [3]. As Section 6.1 shows, our emulation platform gives very similar results to the results from our real EC2 deployment.

Emulation-Full-Speed. We run some of our experiments on our local cluster that emulates the WAN bandwidth of EC2 at *full speed*. We use the same settings as **Emulation-EC2** except we do not slow down the CPUs and the LAN. We use this platform to show the results of deployments with more powerful nodes.

5.2. Applications

We evaluate Gaia with three popular ML applications.

Matrix Factorization (MF) is a technique commonly used in recommender systems, e.g., systems that recommend movies to users on Netflix (a.k.a. collaborative filtering) [25]. Its goal is to discover latent interactions between two entities, such as users and movies, via matrix factorization. For example, input data can be a partially filled matrix X, where every entry is a user's rating for a movie, each row corresponding to a user, and each column corresponding to a specific movie. Matrix factor-

ization factorizes X into factor matrices L and R such that their product approximates X (i.e., $X \approx LR$). Like other systems [17,32,83], we implement *MF* using the stochastic gradient descent (SGD) algorithm. Each worker is assigned a portion of the known entries in X. The L matrix is stored locally in each worker, and the R matrix is stored in parameter servers. Our experiments use the *Netflix* dataset, a 480K-by-18K sparse matrix with 100M known entries. They are configured to factor the matrix into the product of two matrices, each with rank 500.

Topic Modeling (TM) is an unsupervised method for discovering hidden semantic structures (*topics*) in an unstructured collection of *documents*, each consisting of a bag (multi-set) of *words* [10]. *TM* discovers the topics via word co-occurrence. For example, "policy" is more likely to co-occur with "government" than "bacteria", and thus "policy" and "government" are categorized to the same topic associated with political terms. Further, a document with many instances of "policy" would be assigned a topic distribution that peaks for the politics-related topics. *TM* learns the hidden topics and the documents' associations with those topics jointly. Common applications for *TM* include community detection in social networks and news categorizations. We implement our *TM* solver using collapsed Gibbs sampling [30]. We use the *Nytimes* dataset [53], which has 100M words in 300K documents with a vocabulary size of 100K. Our experiments classify words and documents into 500 topics.

Image Classification (IC) is a task to classify images into categories, and the state-of-the-art approach is using deep learning and convolutional neural networks (CNNs) [43]. Given a set of images with known categories (training data), the ML algorithm trains a CNN to learn the relationship between the image features and their categories. The trained CNN is then used to predict the categories of another set of images (test data). We use GoogLeNet [65], one of the state-of-the-art CNNs as our model. We train GoogLeNet using stochastic gradient descent with back propagation [61]. As training a CNN with a large number of images requires substantial computation, doing so on CPUs can take hundreds of machines over a week [13]. Instead, we use distributed GPUs with a popular deep learning framework, Caffe [38], which is hosted by a state-of-the-art GPU-specialized parameter server system, GeePS [18]. Our experiments use the ImageNet Large Scale Visual Recognition Challenge 2012 (ILSVRC12) [62] dataset, which consists of 1.3M training images and 50K test images. Each image is labeled as one of the 1,000 pre-defined categories.

5.3. Performance Metrics and Algorithm Convergence Criteria

We use two performance metrics to evaluate the effectiveness of a globally distributed ML system. The first metric is the *execution time until algorithm convergence*. We use the following algorithm convergence criterion, based on guidance from our ML experts: if the value of the objective function (the *objective value*) in an algorithm changes by less than 2% over the course of 10 iterations, we declare that the algorithm has converged [32]. In order to ensure that each algorithm *accurately* converges to the optimal point, we first run each algorithm on our local cluster until it converges, and we record the absolute objective value. The execution time of each setting is the time it takes to converge to this absolute objective value. The second metric is the *cost of algorithm convergence*. We calculate the cost based on the cost model of Amazon EC2 [8], including the cost of the server time and the cost of data transfer on WANs. We provide the details of the cost model in Appendix C.

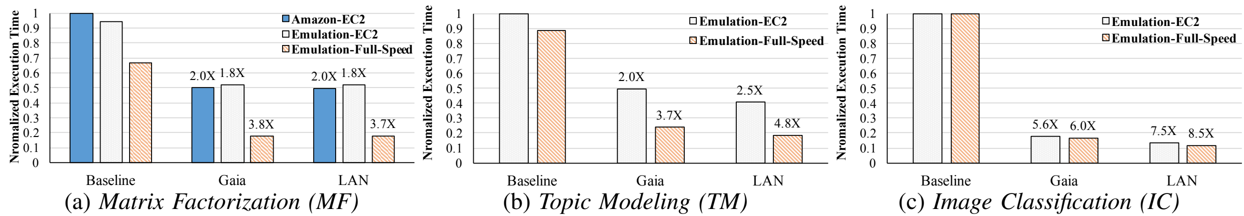
6. Evaluation Results

We evaluate the effectiveness of `Gaia` by evaluating three types of systems/deployments: (1) `Baseline`, two state-of-the-art parameter server systems (IterStore [17] for *MF* and *TM*, GeePS [18] for *IC*) that are deployed across multiple data centers. Every worker machine handles the data in its data center, while the parameter servers are distributed evenly across all the data centers; (2) `Gaia`, our prototype systems based on IterStore and GeePS, deployed across multiple data centers; and (3) `LAN`, the baseline parameter servers (IterStore and GeePS) that are deployed within a single data center (also on 22 machines) that already hold all the data, representing the ideal case of all communication on a LAN. For each system, we evaluate two ML synchronization models: *BSP* and *SSP* (Section 2.1). For `Baseline` and `LAN`, BSP and SSP are used among all worker machines, whereas for `Gaia`, they are used only within each data center. Due to limited space, we present the results for BSP in this section and leave the results for SSP to Appendix B.

6.1. Performance on EC2 Deployment

We first present the performance of `Gaia` and `Baseline` when they are deployed across 11 EC2 data centers. Figure 8 shows the normalized execution time until convergence for our ML applications, normalized to `Baseline` on EC2. The data label on each bar is the speedup over `Baseline` for *the respective deployment*. As Section 5.1 discusses, we run only *MF* on EC2 due to the high monetary cost of WAN data transfer. Thus, we present the results of *MF* on all three platforms, while we show the results of *TM* and *IC* only on our emulation platforms. As Figure 8a shows, our emulation platform (*Emulation-EC2*) matches the execution time of our real EC2 deployment (*Amazon-EC2*) very well. We make two major observations.

First, we find that `Gaia` significantly improves the performance of `Baseline` when deployed globally across many EC2 data centers. For *MF*, `Gaia` provides a speedup of 2.0× over `Baseline`. Furthermore, the performance of `Gaia` is very similar to the performance of `LAN`, indicating that `Gaia` almost attains the performance *upper bound* with the given computation resources. For *TM*, `Gaia` delivers a similar speedup (2.0×) and is within 1.25× of the ideal speed of `LAN`. For *IC*, `Gaia` provides a speedup



(a) *Matrix Factorization (MF)* (b) *Topic Modeling (TM)* (c) *Image Classification (IC)*

Figure 8: Normalized execution time until convergence when deployed across 11 EC2 regions and our emulation cluster

of 5.6× over `Baseline`, which is within 1.32× of the `LAN` speed, indicating that `Gaia` is also effective on a GPU-based ML system. The gap between `Baseline` and `LAN` is larger for *IC* than for the other two applications. This is because the GPU-based ML system generates parameter updates at a higher rate than the CPU-based one, and therefore the limited WAN bandwidth slows it down more significantly.

Second, `Gaia` provides a higher performance gain when deployed on a more powerful platform. As Figure 8 shows, the performance gap between `Baseline` and `LAN` significantly increases on *Emulation-Full-Speed* compared to the slower platform *Emulation-EC2*. This is expected because the WAN bandwidth becomes a more critical bottleneck when the computation time reduces and the LAN bandwidth increases. `Gaia` successfully mitigates the WAN bottleneck in this more challenging *Emulation-Full-Speed* setting, and improves the system performance by 3.8× for *MF*, 3.7× for *TM*, and 6.0× for *IC* over `Baseline`, approaching the speedups provided by `LAN`.

6.2. Performance and WAN Bandwidth

To understand how `Gaia` performs under different amounts of WAN bandwidth, we evaluate two settings where `Baseline` and `Gaia` are deployed across two data centers with two WAN bandwidth configurations: (1) *V/C WAN*, which emulates the WAN bandwidth between Virginia and California, representing a setting within the same continent; and (2) *S/S WAN*, which emulates the WAN bandwidth between Singapore and São Paulo, representing the lowest WAN bandwidth between any two Amazon EC2 sites. All the experiments are conducted on our emulation platform at full speed. Figures 9 and 10 show the results. Three observations are in order.

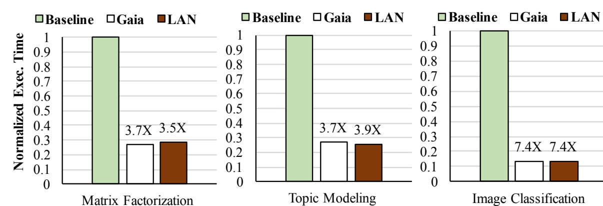


Figure 9: Normalized execution time until convergence with the WAN bandwidth between Virginia and California

First, `Gaia` successfully matches the performance of `LAN` when WAN bandwidth is high (*V/C WAN*). As Figure 9 shows, `Gaia` achieves a speedup of 3.7× for *MF*, 3.7× for *TM*, and 7.4× for *IC*. For all three ML applications, the performance of `Gaia` on WANs is almost the same as `LAN` performance.

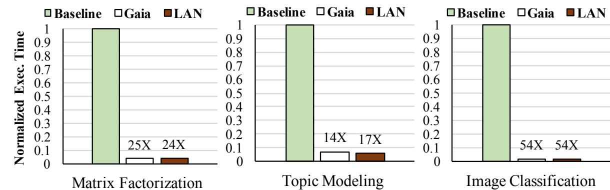


Figure 10: Normalized execution time until convergence with the WAN bandwidth between Singapore and São Paulo

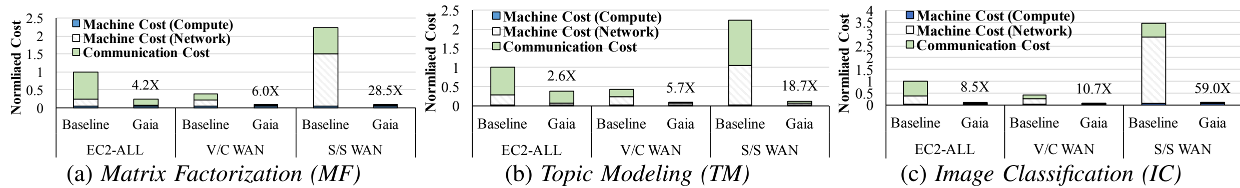
Second, `Gaia` still performs very well when WAN bandwidth is low (*S/S WAN*, Figure 10): `Gaia` provides a speedup of 25.4× for *MF*, 14.1× for *TM*, and 53.5× for *IC*, and successfully approaches `LAN` performance. These results show that our design is robust for both CPU-based and GPU-based ML systems, and it can deliver high performance even under scarce WAN bandwidth.

Third, for *MF*, the performance of `Gaia` (on WANs) is slightly better than `LAN` performance. This is because we run ASP between different data centers, and the workers in each data center need to synchronize only with each other locally in each iteration. As long as the mirror updates on WANs are timely, each iteration of `Gaia` can be faster than that of `LAN`, which needs to synchronize across all workers. While `Gaia` needs more iterations than `LAN` due to the accuracy loss, `Gaia` can still outperform `LAN` due to the faster iterations.

6.3. Cost Analysis

Figure 11 shows the monetary cost of running ML applications until convergence based on the Amazon EC2 cost model, normalized to the cost of `Baseline` on 11 EC2 regions. Cost is divided into three components: (1) the cost of machine time spent on computation, (2) the cost of machine time spent on waiting for networks, and (3) the cost of data transfer across different data centers. As we discuss in Section 2.2, there is no cost for data transfer within a single data center in Amazon EC2. The data label on each bar shows the factor by which the cost of `Gaia` is cheaper than the cost of *each respective* `Baseline`. We evaluate all three deployment setups that we discuss in Sections 6.1 and 6.2. We make two major observations.

First, `Gaia` is very effective in reducing the cost of running a geo-distributed ML application. Across all the evaluated settings, `Gaia` is 2.6× to 59.0× cheaper than `Baseline`. Not surprisingly, the major cost saving comes from the reduction of data transfer on WANs and the reduction of machine time spent on waiting for networks. For the S/S WAN setting, the cost of waiting for networks is a more important factor than the other



(a) *Matrix Factorization (MF)* (b) *Topic Modeling (TM)* (c) *Image Classification (IC)*

Figure 11: Normalized monetary cost of `Gaia` vs. `Baseline`

two settings, because it takes more time to transfer the same amount of data under low WAN bandwidth. As `Gaia` significantly improves system performance and reduces data communication overhead, it significantly reduces both cost sources. We conclude that `Gaia` is a cost-effective system for geo-distributed ML applications.

Second, `Gaia` reduces data transfer cost much more when deployed on a smaller number of data centers. The reason is that `Gaia` needs to broadcast the significant updates to *all* data centers, so communication cost is higher as the number of data centers increases. While we employ network overlays (Section 4.5) to mitigate this effect, there is still more overhead with more than two data centers. Nonetheless, the cost of `Gaia` is still much cheaper (4.2×/2.6×/8.5×) than `Baseline` even when deployed across 11 data centers.

6.4. Comparisons with Centralized Data

Gaia obtains its good performance without moving any raw data, greatly reducing WAN costs and respecting privacy and data sovereignty laws that *prohibit* raw data movement. For settings in which raw data movement is *allowed*, Table 1 summarizes the performance and cost comparisons between `Gaia` and the centralized data approach (`Centralized`), which moves *all* the geo-distributed data into a single data center and then runs the ML application over the data. We make `Centralized` very cost efficient by moving the data into the *cheapest* data center in each setting, and we use low cost machines (`m4.xlarge` [8]) to move the data. We make two major observations.

Table 1: Comparison between `Gaia` and `Centralized`

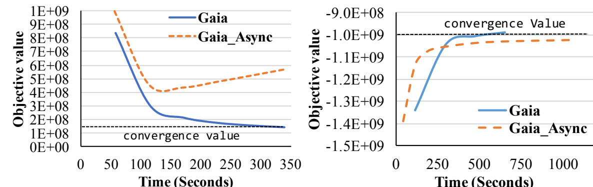
Application	Setting	`Gaia` Speedup over `Centralized`	`Gaia` cost / `Centralized` cost
MF	EC2-ALL	1.11	3.54
	V/C WAN	1.22	1.00
	S/S WAN	2.13	1.17
TM	EC2-ALL	0.80	6.14
	V/C WAN	1.02	1.26
	S/S WAN	1.25	1.92
IC	EC2-ALL	0.76	3.33
	V/C WAN	1.12	1.07
	S/S WAN	1.86	1.08

First, `Gaia` outperforms `Centralized` for most settings, except for *TM* and *IC* in the EC2-ALL setting. Other than these two cases, `Gaia` provides a 1.02–2.13× speedup over `Centralized`. This is because `Gaia` does not need to wait for data movement over WANs, and the performance of `Gaia` is very close to that of `LAN`. On the other hand, `Centralized` performs better when there is a performance gap between `Gaia` and `LAN`, especially in the setting of all 11 data centers for *TM* and *IC*. The data movement overhead of `Centralized` is smaller in this setting because each data center has only a small fraction of the data, and `Centralized` moves the data from all data centers in parallel.

Second, `Centralized` is more cost-efficient than `Gaia`, but the gap is small in the two data centers setting. This is because the total WAN traffic of `Gaia` is still larger than the size of the training data, even though `Gaia` significantly reduces the communication overhead over `Baseline`. The cost gap is larger in the setting of 11 data centers (3.33–6.14×) than in two data centers (1.00–1.92×), because the WAN traffic of `Gaia` is positively correlated with the number of data centers (Section 4.5).

6.5. Effect of Synchronization Mechanisms

One of the major design considerations of ASP is to ensure that the significant updates arrive in a timely manner to guarantee algorithm convergence. To understand the effectiveness of our proposed synchronization mechanisms (i.e., ASP selective barrier and mirror clock), we run *MF* and *TM* on `Gaia` with both mechanisms disabled across 11 EC2 regions. Figure 12 shows the progress toward algorithm convergence with the synchronization mechanisms enabled (`Gaia`) and disabled (`Gaia_Async`). For *MF*, lower object value is better, while for *TM*, higher is better.



(a) *Matrix Factorization (MF)* (b) *Topic Modeling (TM)*

Figure 12: Progress toward algorithm convergence with and without Gaia's synchronization mechanisms

As Figure 12 shows, `Gaia` steadily reaches algorithm convergence for both applications. In contrast, `Gaia_Async` diverges from the optimum point at ~100 seconds for *MF*. For *TM*, `Gaia_Async` looks like it makes faster progress at the beginning of execution because it eliminates the synchronization overhead. However, it makes very slow progress after ~200 seconds and does not reach the value that results in convergence until 1100 seconds. It may take a long time for `Gaia_Async` to reach that point, if ever. Thus, the lack of synchronization leads to worse model quality than that achieved by using proper synchronization mechanisms. Both results demonstrate that the synchronization mechanisms we introduce in ASP are effective and vital for deploying ML algorithms on Gaia on WANs.

7. Related Work

To our knowledge, this is the first work to propose a globally distributed ML system that is (1) designed to run effectively over WANs while ensuring the convergence and accuracy of an ML algorithm, and (2) generic and flexible to run a wide range of ML algorithms *without* requiring modification. In this section, we discuss related works on 1) WAN-aware data analytics and ML systems, 2) distributed ML systems, 3) non-uniform convergence in ML algorithms, 4) communication-efficient ML algorithms, and 5) approximate queries on distributed data, all of which are related to our proposed system.

WAN-aware Data Analytics and ML Systems. Prior work establishes the emerging problem of analyzing the globally-generated data in the context of data analytics systems (e.g., [33, 36, 41, 57, 58, 71–73]). These works show very promising WAN bandwidth reduction and/or system performance improvement with a WAN-aware data analytics framework. However, their goal is *not* to run an *ML system* effectively on WANs, which has very different challenges from a data analytics system. Cano et al. [12] first discuss the problem of running an ML system on geo-distributed data. They show that a geo-distributed ML system can perform significantly better by leveraging a communication-efficient algorithm [49] for logistic regression models. This solution requires the ML programmer to change the ML algorithm, which is challenging and algorithm-specific. In contrast, our work focuses on the design of communication-efficient mechanisms at the system level and requires no modification to the ML algorithms.

Distributed ML Systems. There are many distributed ML systems that aim to enable large-scale ML applications (e.g., [1, 2, 5, 6, 13, 16–18, 20, 34, 40, 45–47, 74, 77]). These systems successfully demonstrate their effectiveness on a large number of machines by employing various synchronization models and system optimizations. However, all of them assume that the network communication happens *within* a data center and do not tackle the challenges of scarce WAN bandwidth. As our study shows, state-of-the-art parameter server systems suffer from significant performance degradation and data transfer cost when deployed across multiple data centers on WANs. We demonstrate our idea on both CPU-based and GPU-based parameter server systems [17, 18], and we believe our proposed general solution can be applied to all distributed ML systems that use the parameter server architecture.

Non-uniform Convergence in ML Algorithms. Prior work observes that, in many ML algorithms, not all model parameters converge to their optimal value within the same number of computation iterations [21, 22, 40, 47, 80]. Several systems exploit this property to improve the algorithm convergence speed, e.g., by prioritizing the computation of important parameters [40, 47, 80], communicating the important parameters more aggressively [74], or sending fewer updates with user-defined filters [45, 46].

Among these, the closest to our work is Li et al.'s proposal for a communication-efficient parameter server system [45, 46], which employs various filters to reduce communication between worker machines and parameter servers. In contrast to our work, this work 1) does not differentiate the communication on LANs from the communication on WANs, and thus it cannot make efficient use of the abundant LAN bandwidth or the scarce and heterogeneous WAN bandwidth, and 2) does not propose a general synchronization model across multiple data centers, a key contribution of our work.

Communication-Efficient ML Algorithms. A large body of prior work proposes ML algorithms to reduce the dependency on intensive parameter updates to enable more efficient parallel computation (e.g., [37, 52, 63, 66, 81, 82, 84]). These works are largely orthogonal to our work, as we focus on a generic system-level solution that does not require any changes to ML algorithms. These ML algorithms can use our system to further reduce communication overhead over WANs.

Approximate Queries on Distributed Data. In the database community, there is substantial prior work that explores the trade-off between precision and performance for queries on distributed data over WANs (e.g., [15, 55, 56, 75]). The idea is to reduce communication overhead of moving up-to-date data from data sources to a query processor by allowing some user-defined precision loss for the queries. At a high level, our work bears some resemblance to this idea, but the challenges we tackle are fundamentally different. The focus of distributed database systems is on approximating queries (e.g., simple aggregation) on raw data. In contrast, our work focuses on retaining ML algorithm convergence and accuracy by using approximate ML models during training.

8. Conclusion

We introduce Gaia, a new ML system that is designed to efficiently run ML algorithms on globally-generated data over WANs, without any need to change the ML algorithms. Gaia decouples the synchronization within a data center (LANs) from the synchronization across different data centers (WANs), enabling flexible and effective synchronization over LANs and WANs. We introduce a new synchronization model, Approximate Synchronous Parallel (ASP), to efficiently utilize the scarce and heterogeneous WAN bandwidth while ensuring convergence of the ML algorithms with a theoretical guarantee. Using ASP, Gaia dynamically eliminates insignificant, and thus unnecessary, communication over WANs. Our evaluation shows that Gaia significantly outperforms two state-of-the-art parameter server systems on WANs, and is within 0.94–1.40$\times$ of the speed of running the same ML algorithm on a LAN. Gaia also significantly reduces the monetary cost of running the same ML algorithm on WANs, by 2.6–59.0$\times$. We conclude that Gaia is a practical and effective system to enable globally-distributed ML applications, and we believe the ideas behind Gaia's system design for communication across WANs can be applied to many other large-scale distributed ML systems.

Acknowledgments

We thank our shepherd, Adam Wierman, and the reviewers for their valuable suggestions. We thank the SAFARI group members and Garth A. Gibson for their feedback. Special thanks to Henggang Cui for his help on IterStore and GeePS, Jinliang Wei for his help on Bösen, and their feedback. We thank the members and companies of the PDL Consortium (including Broadcom, Citadel, EMC, Facebook, Google, HP Labs, Hitachi, Intel, Microsoft Research, MongoDB, NetApp, Oracle, Samsung, Seagate, Tintri, Two Sigma, Uber, Veritas, and Western Digital) for their interest, insights, feedback, and support. We acknowledge the support of our industrial partners: Google, Intel, NVIDIA, Samsung and VMWare. This work is supported in part by NSF grant 1409723, Intel STC on Cloud Computing (ISTC-CC), Intel STC on Visual Cloud Systems (ISTC-VCS), and the Dept of Defense under contract FA8721-05-C-0003. Dimitris Konomis is partially supported by Onassis Foundation.

References

[1] "Apache Mahout." http://mahout.apache.org/

[2] "Apache Spark MLlib." http://spark.apache.org/mllib/

[3] "Linux Traffic Control." http://tldp.org/HOWTO/Traffic-Control-HOWTO/intro.html

[4] "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal Process. Mag.*, 2012.

[5] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. http://tensorflow.org/

[6] A. Ahmed, M. Aly, J. Gonzalez, S. M. Narayanamurthy, and A. J. Smola, "Scalable inference in latent variable models," in *WSDM*, 2012.

[7] Amazon, "AWS global infrastructure." https://aws.amazon.com/about-aws/global-infrastructure/

[8] Amazon, "Amazon EC2 pricing," Janurary 2017. https://aws.amazon.com/ec2/pricing/

[9] A. Auradkar, C. Botev, S. Das, D. D. Maagd, A. Feinberg, P. Ganti, L. Gao, B. Ghosh, K. Gopalakrishna, B. Harris, J. Koshy, K. Krawez, J. Kreps, S. Lu, S. Nagaraj, N. Narkhede, S. Pachev, I. Perisic, L. Qiao, T. Quiggle, J. Rao, B. Schulman, A. Sebastian, O. Seeliger, A. Silberstein, B. Shkolnik, C. Soman, R. Sumbaly, K. Surlaker, S. Topiwala, C. Tran, B. Varadarajan, J. Westerman, Z. White, D. Zhang, and J. Zhang, "Data infrastructure at LinkedIn," in *ICDE*, 2012.

[10] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent Dirichlet allocation," *JMLR*, 2003.

[11] J. K. Bradley, A. Kyrola, D. Bickson, and C. Guestrin, "Parallel coordinate descent for L1-regularized loss minimization," in *ICML*, 2011.

[12] I. Cano, M. Weimer, D. Mahajan, C. Curino, and G. M. Fumarola, "Towards geo-distributed machine learning," *CoRR*, 2016.

[13] T. M. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project Adam: Building an efficient and scalable deep learning training system," in *OSDI*, 2014.

[14] C. Chu, S. K. Kim, Y. Lin, Y. Yu, G. R. Bradski, A. Y. Ng, and K. Olukotun, "Map-Reduce for machine learning on multicore," in *NIPS*, 2006.

[15] G. Cormode, M. N. Garofalakis, S. Muthukrishnan, and R. Rastogi, "Holistic aggregates in a networked world: Distributed tracking of approximate quantiles," in *SIGMOD*, 2005.

[16] H. Cui, J. Cipar, Q. Ho, J. K. Kim, S. Lee, A. Kumar, J. Wei, W. Dai, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing, "Exploiting bounded staleness to speed up big data analytics," in *USENIX ATC*, 2014.

[17] H. Cui, A. Tumanov, J. Wei, L. Xu, W. Dai, J. Haber-Kucharsky, Q. Ho, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing, "Exploiting iterative-ness for parallel ML computations," in *SoCC*, 2014, software available at https://github.com/cuihenggang/iterstore.

[18] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing, "GeePS: Scalable deep learning on distributed GPUs with a GPU-specialized parameter server," in *EuroSys*, 2016, software available at https://github.com/cuihenggang/geeps.

[19] W. Dai, A. Kumar, J. Wei, Q. Ho, G. Gibson, and E. P. Xing, "Analysis of high-performance distributed ML at scale through parameter server consistency models," in *AAAI*, 2015.

[20] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. W. Senior, P. A. Tucker, K. Yang, and A. Y. Ng, "Large scale distributed deep networks," in *NIPS*, 2012.

[21] B. Efron, T. Hastie, I. Johnstone, and R. Tibshirani, "Least angle regression," in *The Annals of Statistics*, 2004.

[22] G. Elidan, I. McGraw, and D. Koller, "Residual belief propagation: Informed scheduling for asynchronous message passing," in *UAI*, 2006.

[23] ESnet and Lawrence Berkeley National Laboratory, "iperf3." http://software.es.net/iperf/

[24] A. Frome, G. S. Corrado, J. Shlens, S. Bengio, J. Dean, M. Ranzato, and T. Mikolov, "DeViSE:

A deep visual-semantic embedding model," in *NIPS*, 2013.

[25] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis, "Large-scale matrix factorization with distributed stochastic gradient descent," in *SIGKDD*, 2011.

[26] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *OSDI*, 2012.

[27] 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 *OSDI*, 2014.

[28] Google, "Google data center locations." https://www.google.com/about/datacenters/inside/locations/index.html

[29] A. G. Greenberg, J. R. Hamilton, D. A. Maltz, and P. Patel, "The cost of a cloud: research problems in data center networks," *Computer Communication Review*, 2009.

[30] T. L. Griffiths and M. Steyvers, "Finding scientific topics," *Proceedings of the National Academy of Sciences of the United States of America*, 2004.

[31] A. Gupta, F. Yang, J. Govig, A. Kirsch, K. Chan, K. Lai, S. Wu, S. G. Dhoot, A. R. Kumar, A. Agiwal, S. Bhansali, M. Hong, J. Cameron, M. Siddiqi, D. Jones, J. Shute, A. Gubarev, S. Venkataraman, and D. Agrawal, "Mesa: Geo-replicated, near real-time, scalable data warehousing," *PVLDB*, 2014.

[32] A. Harlap, H. Cui, W. Dai, J. Wei, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing, "Addressing the straggler problem for iterative convergent parallel ML," in *SoCC*, 2016.

[33] B. Heintz, A. Chandra, and R. K. Sitaraman, "Optimizing grouped aggregation in geo-distributed streaming analytics," in *HPDC*, 2015.

[34] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. R. Ganger, and E. P. Xing, "More effective distributed ML via a stale synchronous parallel parameter server," in *NIPS*, 2013.

[35] C. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving high utilization with software-driven WAN," in *SIGCOMM*, 2013.

[36] C. Hung, L. Golubchik, and M. Yu, "Scheduling jobs across geo-distributed datacenters," in *SoCC*, 2015.

[37] M. Jaggi, V. Smith, M. Takác, J. Terhorst, S. Krishnan, T. Hofmann, and M. I. Jordan, "Communication-efficient distributed dual coordinate ascent," in *NIPS*, 2014.

[38] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. B. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *CoRR*, 2014.

[39] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and F. F. Li, "Large-scale video classification with convolutional neural networks," in *CVPR*, 2014.

[40] J. K. Kim, Q. Ho, S. Lee, X. Zheng, W. Dai, G. A. Gibson, and E. P. Xing, "STRADS: a distributed framework for scheduled model parallel machine learning," in *EuroSys*, 2016.

[41] K. Kloudas, R. Rodrigues, N. M. Preguiça, and M. Mamede, "PIXIDA: optimizing data parallel jobs in wide-area data analytics," *PVLDB*, 2015.

[42] N. Laoutaris, M. Sirivianos, X. Yang, and P. Rodriguez, "Inter-datacenter bulk transfers with netstitcher," in *SIGCOMM*, 2011.

[43] Y. LeCun, B. E. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. E. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural Computation*, 1989.

[44] G. Lee, J. J. Lin, C. Liu, A. Lorek, and D. V. Ryaboy, "The unified logging infrastructure for data analytics at Twitter," *PVLDB*, 2012.

[45] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B. Su, "Scaling distributed machine learning with the parameter server," in *OSDI*, 2014.

[46] M. Li, D. G. Andersen, A. J. Smola, and K. Yu, "Communication efficient distributed machine learning with the parameter server," in *NIPS*, 2014.

[47] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Distributed GraphLab: A framework for machine learning in the cloud," *VLDB*, 2012.

[48] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim, "A survey and comparison of peer-to-peer overlay network schemes," *IEEE Communications Surveys and Tutorials*, 2005.

[49] D. Mahajan, S. S. Keerthi, S. Sundararajan, and L. Bottou, "A functional approximation based distributed learning algorithm," *CoRR*, 2013.

[50] X. Meng, J. K. Bradley, B. Yavuz, E. R. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. B. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar, "MLlib: Machine learning in Apache Spark," *CoRR*, 2015.

[51] Microsoft, "Azure regions." https://azure.microsoft.com/en-us/region

[52] W. Neiswanger, C. Wang, and E. P. Xing, "Asymptotically exact, embarrassingly parallel MCMC," in *UAI*, 2014.

[53] "New York Times dataset," http://www.ldc.upenn.edu/.

[54] D. Newman, A. U. Asuncion, P. Smyth, and M. Welling, "Distributed algorithms for topic models," *JMLR*, 2009.

[55] C. Olston, J. Jiang, and J. Widom, “Adaptive filters for continuous queries over distributed data streams,” in *SIGMOD*, 2003.

[56] C. Olston and J. Widom, “Offering a precision-performance tradeoff for aggregation queries over replicated data,” in *VLDB*, 2000.

[57] Q. Pu, G. Ananthanarayanan, P. Bodík, S. Kandula, A. Akella, P. Bahl, and I. Stoica, “Low latency geo-distributed data analytics,” in *SIGCOMM*, 2015.

[58] A. Rabkin, M. Arye, S. Sen, V. S. Pai, and M. J. Freedman, “Aggregation and degradation in Jet-Stream: Streaming analytics in the wide area,” in *NSDI*, 2014.

[59] B. Recht, C. Ré, S. J. Wright, and F. Niu, “Hogwild: A lock-free approach to parallelizing stochastic gradient descent,” in *NIPS*, 2011.

[60] P. Richtárik and M. Takác, “Distributed coordinate descent method for learning with big data,” *CoRR*, 2013.

[61] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” *Cognitive modeling*, 1988.

[62] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, “ImageNet large scale visual recognition challenge,” *IJCV*, 2015.

[63] O. Shamir, N. Srebro, and T. Zhang, “Communication-efficient distributed optimization using an approximate Newton-type method,” in *ICML*, 2014.

[64] A. J. Smola and S. M. Narayanamurthy, “An architecture for parallel topic models,” *VLDB*, 2010.

[65] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, “Going deeper with convolutions,” in *CVPR*, 2015.

[66] M. Takác, A. S. Bijral, P. Richtárik, and N. Srebro, “Mini-batch primal and dual methods for SVMs,” in *ICML*, 2013.

[67] TeleGeography, “Global Internet geography.” https://www.telegeography.com/research-services/global-internet-geography/

[68] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. S. Sarma, R. Murthy, and H. Liu, “Data warehousing and analytics infrastructure at Facebook,” in *SIGMOD*, 2010.

[69] K. I. Tsianos, S. F. Lawlor, and M. G. Rabbat, “Communication/computation tradeoffs in consensus-based distributed optimization,” in *NIPS*, 2012.

[70] L. G. Valiant, “A bridging model for parallel computation,” *Commun. ACM*, 1990.

[71] R. Viswanathan, A. Akella, and G. Ananthanarayanan, “Clarinet: WAN-aware optimization for analytics queries,” in *OSDI*, 2016.

[72] A. Vulimiri, C. Curino, B. Godfrey, K. Karanasos, and G. Varghese, “WANalytics: Analytics for a geo-distributed data-intensive world,” in *CIDR*, 2015.

[73] A. Vulimiri, C. Curino, P. B. Godfrey, T. Jungblut, J. Padhye, and G. Varghese, “Global analytics in the face of bandwidth and regulatory constraints,” in *NSDI*, 2015.

[74] J. Wei, W. Dai, A. Qiao, Q. Ho, H. Cui, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing, “Managed communication and consistency for fast data-parallel iterative analytics,” in *SoCC*, 2015.

[75] O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang, “Moving objects databases: Issues and solutions,” in *SSDBM*, 1998.

[76] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha, “SPANStore: Cost-effective geo-replicated storage spanning multiple cloud services,” in *SOSP*, 2013.

[77] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu, “Petuum: A new platform for distributed machine learning on big data,” in *SIGKDD*, 2015.

[78] E. P. Xing, Q. Ho, P. Xie, and W. Dai, “Strategies and principles of distributed machine learning on big data,” *CoRR*, 2015.

[79] 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,” in *NSDI*, 2012.

[80] Y. Zhang, Q. Gao, L. Gao, and C. Wang, “PrIter: A distributed framework for prioritized iterative computations,” in *SoCC*, 2011.

[81] Y. Zhang, J. C. Duchi, and M. J. Wainwright, “Communication-efficient algorithms for statistical optimization,” *JMLR*, 2013.

[82] Y. Zhang and X. Lin, “DiSCO: Distributed optimization for self-concordant empirical loss,” in *ICML*, 2015.

[83] M. Zinkevich, A. J. Smola, and J. Langford, “Slow learners are fast,” in *NIPS*, 2009.

[84] M. Zinkevich, M. Weimer, A. J. Smola, and L. Li, “Parallelized stochastic gradient descent,” in *NIPS*, 2010.

Appendix

A. Convergence Proof of SGD under ASP

Stochastic Gradient Descent is a very popular algorithm, widely used for finding the minimizer/maximizer of a criterion (sum of differentiable functions) via iterative steps. The intuition behind the algorithm is that we randomly select an initial point $\boldsymbol{x}_0$ and keep moving toward the negative direction of the gradient, producing a sequence of points $\boldsymbol{x}_i, i = 1, ...n$ until we detect that moving further decreases (increases) the minimization (maximization, respectively) criterion only negligibly.

Formally, step t of the SGD algorithm is defined as:

$$\boldsymbol{x}_t = \boldsymbol{x}_{t-1} - \eta_t \nabla f_t(\boldsymbol{x}_t) = \boldsymbol{x}_{t-1} - \eta_t \boldsymbol{g}_t = \boldsymbol{x}_{t-1} + \boldsymbol{u}_t \quad (1)$$

where η_t is the step size at step t, $\nabla f_t(\boldsymbol{x}_t)$ or $\boldsymbol{g}_t$ is the gradient at step t, and $\boldsymbol{u}_t = \eta_t \boldsymbol{g}_t$ is the update of step t.

Let us define an order of the updates up to step t. Suppose that the algorithm is distributed in D data centers, each of which uses P machines, and the logical clocks that mark progress start at 0. Then,

$$\boldsymbol{u}_t = \boldsymbol{u}_{d,p,c} = \boldsymbol{u}_{\lfloor \frac{t}{P} \rfloor \bmod D,\ t \bmod P,\ \lfloor \frac{t}{DP} \rfloor} \quad (2)$$

represents a mapping that loops through clocks ($c = \lfloor \frac{t}{DP} \rfloor$) and for each clock loops through data centers ($d = \lfloor \frac{t}{P} \rfloor \bmod D$) and for each data center loops through its workers ($p = t \bmod P$).

We now define a reference sequence of states that a single machine serial execution would go through if the updates were observed under the above ordering:

$$\boldsymbol{x}_t = \boldsymbol{x}_0 + \sum_{t'=1}^{t} \boldsymbol{u}_{t'} \quad (3)$$

Let Δ_c denote the threshold of mirror clock difference between different data centers. At clock c, let $A_{d,c}$ denote the $(c - \Delta_c)$-width window of updates at data center d: $A_{d,c} = [0, P-1] \times [0, c - \Delta_c - 1]$. Also, let $K_{d,c}$ denote the subset of $A_{d,c}$ of significant updates (i.e., those broadcast to other data centers) and $L_{d,c}$ denote the subset of $A_{d,c}$ of the insignificant updates (not broadcast) from this data center. Clearly, $K_{d,c}$ and $L_{d,c}$ are disjoint and their union equals $A_{d,c}$.

Let s denote a user-chosen staleness threshold for SSP. Let $\tilde{\boldsymbol{x}}_t$ denote the sequence of noisy (i.e., inaccurate) *views* of the parameters $\boldsymbol{x}_t$. Let $B_{d,c}$ denote the $2s$-width window of updates at data center d: $B_{d,c} = [0, P-1] \times [c - s, c + s - 1]$. A worker p in data center d will definitely see its own updates and may or may not see updates from other workers that belong to this window. Then, $M_{d,c}$ denotes the set of updates that are not seen in $\tilde{\boldsymbol{x}}_t$ and are seen in $\boldsymbol{x}_t$, whereas $N_{d,c}$ denotes the updates that are seen in $\tilde{\boldsymbol{x}}_t$ and not seen in $\boldsymbol{x}_t$. The sets $M_{d,c}$ and $N_{d,c}$ are disjoint and their union equals the set $B_{d,c}$.

We define the noisy view $\tilde{\boldsymbol{x}}_t$ using the above mapping:

$$\tilde{\boldsymbol{x}}_{d,p,c} = \sum_{p'=0}^{P-1} \sum_{c'=0}^{c-s-1} \boldsymbol{u}_{d,p',c'} + \sum_{c'=c-s}^{c-1} \boldsymbol{u}_{d,p,c'} + \sum_{(p',c') \in B'_{d,c} \subset B_{d,c}} \boldsymbol{u}_{d,p',c'} + \sum_{d' \neq d} \Big[\sum_{(p',c') \in K_{d',c'}} \boldsymbol{u}_{d',p',c'} \Big] \quad (4)$$

The difference between the reference view $\boldsymbol{x}_t$ and the noisy view $\tilde{\boldsymbol{x}}_t$ becomes:

$$\tilde{\boldsymbol{x}}_t - \boldsymbol{x}_t = \tilde{\boldsymbol{x}}_{d,p,c} - \boldsymbol{x}_t = \tilde{\boldsymbol{x}}_{\lfloor \frac{t}{P} \rfloor \bmod D, t \bmod P, \lfloor \frac{t}{DP} \rfloor} - \boldsymbol{x}_t = - \sum_{i \in M_{d,c}} \boldsymbol{u}_i + \sum_{i \in N_{d,c}} \boldsymbol{u}_i - \sum_{d' \neq d} \sum_{i \in L_{d',c}} \boldsymbol{u}_i + \sum_{d' \neq d} \Big[- \sum_{i \in M_{d',c}} \boldsymbol{u}_i + \sum_{i \in N_{d',c}} \boldsymbol{u}_i \Big] \quad (5)$$

Finally, let $D(\boldsymbol{x}, \boldsymbol{x'})$ denote the distance between points $\boldsymbol{x}, \boldsymbol{x'} \in \mathbb{R}^n$:

$$D(\boldsymbol{x}, \boldsymbol{x'}) = \frac{1}{2} \|\boldsymbol{x} - \boldsymbol{x'}\|^2. \quad (6)$$

We now prove the following lemma:

Lemma. *For any* $\boldsymbol{x}^*, \tilde{\boldsymbol{x}}_t \in \mathbb{R}^n$,

$$\langle \tilde{\boldsymbol{x}}_t - \boldsymbol{x}^*, \tilde{\boldsymbol{g}}_t \rangle = \frac{1}{2} \eta_t \|\tilde{\boldsymbol{g}}_t\|^2 + \frac{D(\boldsymbol{x}^*, \boldsymbol{x}_t) - D(\boldsymbol{x}^*, \boldsymbol{x}_{t+1})}{\eta_t} + \Big[- \sum_{i \in M_{d,c}} \eta_i \langle \tilde{\boldsymbol{g}}_i, \tilde{\boldsymbol{g}}_t \rangle + \sum_{i \in N_{d,c}} \eta_i \langle \tilde{\boldsymbol{g}}_i, \tilde{\boldsymbol{g}}_t \rangle \Big] + \sum_{d' \neq d} \Big[- \sum_{i \in L_{d',c}} \eta_i \langle \tilde{\boldsymbol{g}}_i, \tilde{\boldsymbol{g}}_t \rangle \Big] + \sum_{d' \neq d} \Big[- \sum_{i \in M_{d',c}} \eta_i \langle \tilde{\boldsymbol{g}}_i, \tilde{\boldsymbol{g}}_t \rangle + \sum_{i \in N_{d',c}} \eta_i \langle \tilde{\boldsymbol{g}}_i, \tilde{\boldsymbol{g}}_t \rangle \Big] \quad (7)$$

Proof.

$$\begin{aligned}
D(\boldsymbol{x}^*, \boldsymbol{x}_{t+1}) - D(\boldsymbol{x}^*, \boldsymbol{x}_t) &= \frac{1}{2} \|\boldsymbol{x}^* - \boldsymbol{x}_{t+1}\|^2 - \frac{1}{2} \|\boldsymbol{x}^* - \boldsymbol{x}_t\|^2 \\
&= \frac{1}{2} \|\boldsymbol{x}^* - \boldsymbol{x}_t + \boldsymbol{x}_t - \boldsymbol{x}_{t+1}\|^2 - \frac{1}{2} \|\boldsymbol{x}^* - \boldsymbol{x}_t\|^2 \\
&= \frac{1}{2} \|\boldsymbol{x}^* - \boldsymbol{x}_t + \eta_t \tilde{\boldsymbol{g}}_t\|^2 - \frac{1}{2} \|\boldsymbol{x}^* - \boldsymbol{x}_t\|^2 \\
&= \frac{1}{2} \langle \boldsymbol{x}^* - \boldsymbol{x}_t + \eta_t \tilde{\boldsymbol{g}}_t, \boldsymbol{x}^* - \boldsymbol{x}_t + \eta_t \tilde{\boldsymbol{g}}_t \rangle - \frac{1}{2} \langle \boldsymbol{x}^* - \boldsymbol{x}_t, \boldsymbol{x}^* - \boldsymbol{x}_t \rangle \\
&= \frac{1}{2} \langle \boldsymbol{x}^* - \boldsymbol{x}_t, \boldsymbol{x}^* - \boldsymbol{x}_t \rangle + \frac{1}{2} \langle \eta_t \tilde{\boldsymbol{g}}_t, \eta_t \tilde{\boldsymbol{g}}_t \rangle + \langle \boldsymbol{x}^* - \boldsymbol{x}_t, \eta_t \tilde{\boldsymbol{g}}_t \rangle \\
&\quad - \frac{1}{2} \langle \boldsymbol{x}^* - \boldsymbol{x}_t, \boldsymbol{x}^* - \boldsymbol{x}_t \rangle \\
&= \frac{1}{2} \eta_t{}^2 \|\tilde{\boldsymbol{g}}_t\|^2 + \eta_t \langle \boldsymbol{x}^* - \boldsymbol{x}_t, \tilde{\boldsymbol{g}}_t \rangle \\
&= \frac{1}{2} \eta_t{}^2 \|\tilde{\boldsymbol{g}}_t\|^2 - \eta_t \langle \boldsymbol{x}_t - \boldsymbol{x}^*, \tilde{\boldsymbol{g}}_t \rangle \\
&= \frac{1}{2} \eta_t{}^2 \|\tilde{\boldsymbol{g}}_t\|^2 - \eta_t \langle \boldsymbol{x}_t - \tilde{\boldsymbol{x}}_t + \tilde{\boldsymbol{x}}_t - \boldsymbol{x}^* \rangle \\
&= \frac{1}{2} \eta_t{}^2 \|\tilde{\boldsymbol{g}}_t\|^2 - \eta_t \langle \boldsymbol{x}_t - \tilde{\boldsymbol{x}}_t, \tilde{\boldsymbol{g}}_t \rangle - \eta_t \langle \tilde{\boldsymbol{x}}_t - \boldsymbol{x}^*, \tilde{\boldsymbol{g}}_t \rangle \implies
\end{aligned}$$

$$\langle \tilde{\boldsymbol{x}}_t - \boldsymbol{x}^*, \tilde{\boldsymbol{g}}_t \rangle = \frac{1}{2}\eta_t \|\tilde{\boldsymbol{g}}_t\|^2 + \frac{D(\boldsymbol{x}^*, \boldsymbol{x}_t) - D(\boldsymbol{x}^*, \boldsymbol{x}_{t+1})}{\eta_t} - \langle \boldsymbol{x}_t - \tilde{\boldsymbol{x}}_t, \tilde{\boldsymbol{g}}_t \rangle \quad (8)$$

Substituting the RHS of Equation 5 into Equation 8 completes the proof. □

Theorem 1. ***(Convergence of SGD under ASP).***

Suppose that, in order to compute the minimizer x^* of a convex function $f(\boldsymbol{x}) = \sum_{t=1}^{T} f_t(\boldsymbol{x})$, with $f_t, t = 1, 2, \ldots, T$, convex, we use stochastic gradient descent on one component ∇f_t at a time. Suppose also that 1) the algorithm is distributed in D data centers, each of which uses P machines, 2) within each data center, the SSP protocol is used, with a fixed staleness of s, and 3) a fixed mirror clock difference Δ_c is allowed between any two data centers. Let $\boldsymbol{u}_t = -\eta_t \nabla f_t(\tilde{\boldsymbol{x}}_t)$, where the step size η_t decreases as $\eta_t = \frac{\eta}{\sqrt{t}}$ and the significance threshold v_t decreases as $v_t = \frac{v}{\sqrt{t}}$. If we further assume that: $\|\nabla f_t(\boldsymbol{x})\| \leq L$, $\forall \boldsymbol{x} \in dom(f_t)$ and $\max(D(\boldsymbol{x}, \boldsymbol{x}')) \leq \Delta^2, \forall \boldsymbol{x}, \boldsymbol{x}' \in dom(f_t)$. Then, as $T \to \infty$,

$$R[X] = \sum_{t=1}^{T} f_t(\tilde{\boldsymbol{x}}_t) - f(\boldsymbol{x}^*) = O(\sqrt{T})$$

and therefore

$$\lim_{T \to \infty} \frac{R[X]}{T} \to 0$$

Proof.

$$\begin{aligned}
R[X] &= \sum_{t=1}^{T} f_t(\tilde{\boldsymbol{x}}_t) - f_t(\boldsymbol{x}^*) \\
&\leq \sum_{t=1}^{T} \langle \nabla f_t(\tilde{\boldsymbol{x}}_t), \tilde{\boldsymbol{x}}_t - \boldsymbol{x}^* \rangle \quad \text{(convexity of } f_t\text{)} \\
&= \sum_{t=1}^{T} \langle \tilde{\boldsymbol{g}}_t, \tilde{\boldsymbol{x}}_t - \boldsymbol{x}^* \rangle \\
&= \sum_{t=1}^{T} \Bigg[\frac{1}{2}\eta_t \|\tilde{\boldsymbol{g}}_t\|^2 + \frac{D(\boldsymbol{x}^*, \boldsymbol{x}_t) - D(\boldsymbol{x}^*, \boldsymbol{x}_{t+1})}{\eta_t} \\
&\quad + \sum_{d' \neq d} \Big[- \sum_{i \in L_{d',c}} \eta_i \langle \tilde{\boldsymbol{g}}_i, \tilde{\boldsymbol{g}}_t \rangle \Big] \\
&\quad + \Big[- \sum_{i \in M_{d,c}} \eta_i \langle \tilde{\boldsymbol{g}}_i, \tilde{\boldsymbol{g}}_t \rangle + \sum_{i \in N_{d,c}} \eta_i \langle \tilde{\boldsymbol{g}}_i, \tilde{\boldsymbol{g}}_t \rangle \Big] \\
&\quad + \sum_{d' \neq d} \Big[- \sum_{i \in M_{d',c}} \eta_i \langle \tilde{\boldsymbol{g}}_i, \tilde{\boldsymbol{g}}_t \rangle + \sum_{i \in N_{d',c}} \eta_i \langle \tilde{\boldsymbol{g}}_i, \tilde{\boldsymbol{g}}_t \rangle \Big] \Bigg] \quad (9)
\end{aligned}$$

We first bound the upper limit of the term: $\sum_{t=1}^{T} \frac{1}{2}\eta_t \|\tilde{\boldsymbol{g}}_t\|^2$:

$$\begin{aligned}
\sum_{t=1}^{T} \frac{1}{2}\eta_t \|\tilde{\boldsymbol{g}}_t\|^2 &\leq \sum_{t=1}^{T} \frac{1}{2}\eta_t L^2 \qquad (\|\nabla f_t(\boldsymbol{x})\| \leq L) \\
&= \sum_{t=1}^{T} \frac{1}{2} \frac{\eta}{\sqrt{t}} L^2 \\
&= \frac{1}{2}\eta L^2 \sum_{t=1}^{T} \frac{1}{\sqrt{t}} \qquad (\sum_{t=1}^{T} \frac{1}{\sqrt{t}} \leq 2\sqrt{T}) \\
&\leq \frac{1}{2}\eta L^2 2\sqrt{T} = \eta L^2 \sqrt{T} \quad (10)
\end{aligned}$$

Second, we bound the upper limit of the term: $\sum_{t=1}^{T} \frac{D(x^*, x_t) - D(x^*, x_{t+1})}{\eta_t}$:

$$\begin{aligned}
&\sum_{t=1}^{T} \frac{D(\boldsymbol{x}^*, \boldsymbol{x}_t) - D(\boldsymbol{x}^*, \boldsymbol{x}_{t+1})}{\eta_t} \\
&= \frac{D(\boldsymbol{x}^*, \boldsymbol{x}_1)}{\eta_1} - \frac{D(\boldsymbol{x}^*, \boldsymbol{x}_{T+1})}{\eta_T} + \sum_{t=2}^{T} D(\boldsymbol{x}^*, \boldsymbol{x}_t)(\frac{1}{\eta_t} - \frac{1}{\eta_{t-1}}) \\
&\leq \frac{\Delta^2}{\eta} - 0 + \frac{\Delta^2}{\eta} \sum_{t=2}^{T} [\sqrt{t} - \sqrt{t-1}] \quad (\max(D(\boldsymbol{x}, \boldsymbol{x}')) \leq \Delta^2) \\
&= \frac{\Delta^2}{\eta} + \frac{\Delta^2}{\eta} [\sqrt{T} - 1] \\
&= \frac{\Delta^2}{\eta} \sqrt{T} \quad (11)
\end{aligned}$$

Third, we bound the upper limit of the term: $\sum_{t=1}^{T} \sum_{d' \neq d} \Big[- \sum_{i \in L_{d',c}} \eta_i \langle \tilde{\boldsymbol{g}}_i, \tilde{\boldsymbol{g}}_t \rangle \Big]$:

$$\begin{aligned}
&\sum_{t=1}^{T} \sum_{d' \neq d} \Big[- \sum_{i \in L_{d',c}} \eta_i \langle \tilde{\boldsymbol{g}}_i, \tilde{\boldsymbol{g}}_t \rangle \Big] \\
&\leq \sum_{t=1}^{T} (D-1) \Big[- \sum_{i \in L_{d',c}} \eta_i \langle \tilde{\boldsymbol{g}}_i, \tilde{\boldsymbol{g}}_t \rangle \Big] \leq \sum_{t=1}^{T} (D-1) v_t \\
&= (D-1) \sum_{t=1}^{T} \frac{v}{\sqrt{t - (s + \Delta_c + 1)P}} \\
&\leq (D-1) v \sum_{t=(s+\Delta_c+1)P+1}^{T} \frac{1}{\sqrt{T - (s + \Delta_c + 1)P}} \\
&\leq 2(D-1) v \sqrt{T - (s + \Delta_c + 1)P} \\
&\leq 2(D-1) v \sqrt{T} \\
&\leq 2Dv\sqrt{T} \quad (12)
\end{aligned}$$

where the fourth inequality follows from the fact that: $\sum_{t=(s+\Delta_c+1)P+1}^{T} \frac{1}{\sqrt{T-(s+\Delta_c+1)P}} \leq \sqrt{T - (s + \Delta_c + 1)P}$.

Fourth, we bound the upper limit of the term: $\sum_{t=1}^{T} \Big[- \sum_{i \in M_{d,c}} \eta_i \langle \tilde{\boldsymbol{g}}_i, \tilde{\boldsymbol{g}}_t \rangle + \sum_{i \in N_{d,c}} \eta_i \langle \tilde{\boldsymbol{g}}_i, \tilde{\boldsymbol{g}}_t \rangle \Big]$:

$$
\begin{aligned}
&\sum_{t=1}^{T}\Big[-\sum_{i\in M_{d,c}}\eta_i\langle\tilde{\boldsymbol{g}}_i,\tilde{\boldsymbol{g}}_t\rangle+\sum_{i\in N_{d,c}}\eta_i\langle\tilde{\boldsymbol{g}}_i,\tilde{\boldsymbol{g}}_t\rangle\Big]\\
&\le\sum_{t=1}^{T}[|M_{d,c}|+|N_{d,c}\,|]\eta_{max(1,t-(s+1)P)}L^2\\
&=L^2\Big[\sum_{t=1}^{(s+1)P}[|M_{d,c}|+|N_{d,c}|]\eta\\
&\quad+\sum_{t=(s+1)P+1}^{T}[|M_{d,c}|+|N_{d,c}|]\eta_{t-(s+1)P}\Big]\\
&=L^2\Big[\sum_{t=1}^{(s+1)P}[|M_{d,c}|+|N_{d,c}|]\eta\\
&\quad+\sum_{t=(s+1)P+1}^{T}[|M_{d,c}|+|N_{d,c}|]\frac{\eta}{\sqrt{t-(s+1)P}}\Big]\\
&\le\eta L^2\Big[\sum_{t=1}^{(s+1)P}2s(P-1)\\
&\quad+\sum_{t=(s+1)P+1}^{T}2s(P-1)\frac{1}{\sqrt{t-(s+1)P}}\Big]\\
&=2\eta L^2 s(P-1)\Big[(s+1)P+\sum_{t=(s+1)P+1}^{T}\frac{1}{\sqrt{T-(s+1)P}}\Big]\\
&\le2\eta L^2 s(P-1)\Big[(s+1)P+2\sqrt{T-(s+1)P}\Big]\\
&\le2\eta L^2 s(P-1)[(s+1)P+2\sqrt{T}]\\
&=2\eta L^2 s(s+1)(P-1)P+4\eta L^2 s(P-1)\sqrt{T}\\
&\le2\eta L^2(s+1)(s+1)(P-1)P+4\eta L^2(s+1)(P-1)\sqrt{T}\\
&=2\eta L^2(s+1)^2(P-1)P+4\eta L^2(s+1)(P-1)\sqrt{T}\\
&\le2\eta L^2(s+1)^2PP+4\eta L^2(s+1)P\sqrt{T}\\
&=2\eta L^2[(s+1)P]^2+4\eta L^2(s+1)P\sqrt{T} \qquad (13)
\end{aligned}
$$

where the first inequality follows from the fact that $\eta_{max(1,t-(s+1)P)}\ge\eta_t, t\in M_{d,t}\cup N_{d,t}$, the second inequality follows from the fact that $|M_{d,t}|+|N_{d,t}|\le 2s(P-1)$, and the third inequality follows from the fact that $\sum_{t=(s+1)P+1}^{T}\left[\frac{1}{\sqrt{T-(s+1)P}}\right]\le 2\sqrt{T-(s+1)P}$.

Similarly, $\forall d'\in D'=D\setminus\{d\}$, we can prove that:

$$
\sum_{t=1}^{T}\Big[-\sum_{i\in M_{d',c}}\eta_i\langle\tilde{\boldsymbol{g}}_i,\tilde{\boldsymbol{g}}_t\rangle+\sum_{i\in N_{d',c}}\eta_i\langle\tilde{\boldsymbol{g}}_i,\tilde{\boldsymbol{g}}_t\rangle\Big]\le 2\eta L^2[(s+\Delta_c+1)P]^2+4\eta L^2(s+\Delta_c+1)P\sqrt{T}
$$

which implies:

$$
\sum_{t=1}^{T}\sum_{d'\neq d}\Big[-\sum_{i\in M_{d',c}}\boldsymbol{u}_i+\sum_{i\in N_{d',c}}\boldsymbol{u}_i\Big]\le D\Big[2\eta L^2[(s+\Delta_c+1)P]^2+4\eta L^2(s+\Delta_c+1)P\sqrt{T}\Big]
$$

By combining all the above upper bounds, we have:

$$
\begin{aligned}
R[X]&\le\eta L^2\sqrt{T}+\frac{\Delta^2}{\eta}\sqrt{T}+2Dv\sqrt{T}+2\eta L^2[(s+1)P]^2\\
&+4\eta L^2(s+1)P\sqrt{T}\\
&+D\Big[2\eta L^2[(s+\Delta_c+1)P]^2+4\eta L^2(s+\Delta_c+1)P\sqrt{T}\Big]\\
&=O(\sqrt{T}) \qquad (14)
\end{aligned}
$$

and thus $\lim_{T\to\infty}\frac{R[X]}{T}\to 0$. □

B. Performance Results of SSP

Due to limited space, we present the performance results of SSP for *MF (Matrix Factorization)* and *TM (Topic Modeling)* here. We do not present the results of SSP for *IC (Image Classification)* because SSP has worse performance than BSP for *IC* [18]. In our evaluation, BSP and SSP are used among all worker machines for `Baseline` and `LAN`, whereas for `Gaia`, they are used only within each data center. To show the performance difference between BSP and SSP, we show both results together.

B.1. SSP Performance on EC2 Deployment

Similar to Section 6.1, Figures 13 and 14 show the execution time until convergence for *MF* and *TM*, normalized to `Baseline` with BSP on EC2. The data label above each bar shows the speedup over `Baseline` for *the respective deployment and synchronization model*.

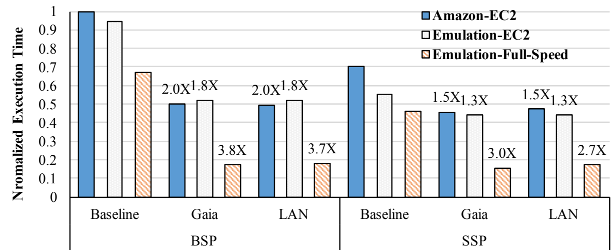


Figure 13: Normalized execution time of *MF* until convergence when deployed across 11 EC2 regions

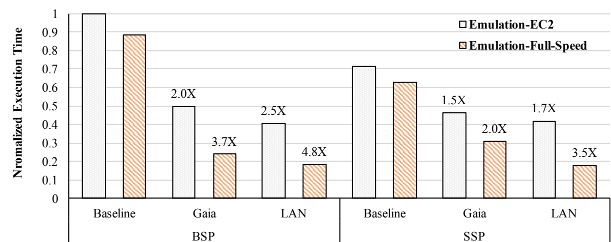


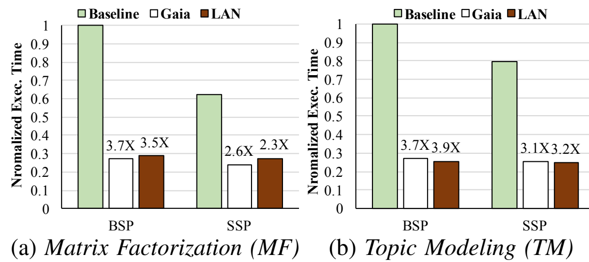
Figure 14: Normalized execution time of *TM* until convergence when deployed across 11 EC2 regions

We see that `Gaia` significantly improves the performance of `Baseline` with SSP. For *MF*, `Gaia` provides a speedup of 1.3–3.0× over `Baseline` with SSP, and successfully approaches the speedups of `LAN` with SSP. For *TM*, `Gaia` achieves speedups of 1.5–2.0× over `Baseline`. Note that for *TM*, `Gaia` with BSP outperforms `Gaia` with SSP. The reason is that SSP allows using stale, and thus

inaccurate, values in order to get the benefit of more efficient communication. However, compared to Baseline, the benefit of employing SSP to reduce communication overhead is much smaller for Gaia because it uses SSP only to synchronize a small number of machines within a data center. Thus, the cost of inaccuracy outweighs the benefit of SSP in this case. Fortunately, Gaia decouples the synchronization model within a data center from the synchronization model across different data centers. Thus, we can freely choose the combination of synchronization models that works better for Gaia.

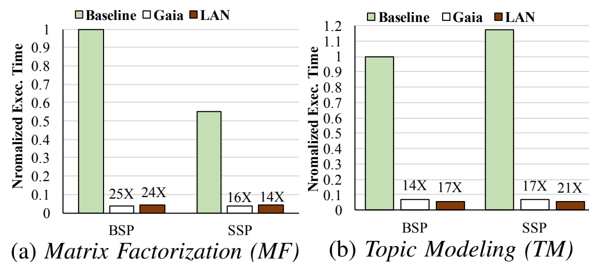
B.2. SSP Performance and WAN Bandwidth

Similar to Section 6.2, Figures 15 and 16 show the normalized execution time until convergence on two deployments: V/C WAN and S/S WAN. The data label above each bar shows the speedup over Baseline for *the respective deployment and synchronization model*.



(a) *Matrix Factorization (MF)* (b) *Topic Modeling (TM)*

Figure 15: Normalized execution time until convergence with the WAN bandwidth between Virginia and California



(a) *Matrix Factorization (MF)* (b) *Topic Modeling (TM)*

Figure 16: Normalized execution time until convergence with the WAN bandwidth between Singapore and São Paulo

We find that Gaia performs very well compared to Baseline with SSP in both high WAN bandwidth (V/C WAN) and low WAN bandwidth (S/S WAN) settings. For V/C WAN, Gaia achieves a speedup of 2.6× for *MF* and 3.1× for *TM* over Baseline with SSP. For both applications, the performance of Gaia is almost the same as the performance of LAN. For S/S WAN, Gaia provides a speedup of 15.7× / 16.8× for *MF* / *TM* over Baseline with SSP, and successfully approaches the LAN speedups. We conclude that Gaia provides significant performance improvement over Baseline, irrespective of the synchronization model used by Baseline.

C. EC2 Cost Model Details

We use the on-demand pricing of Amazon EC2 published for January 2017 as our cost model [8]. As the pricing might change over time, we provide the details of the cost model in Table 2.

The CPU instance is `c4.4xlarge` or `m4.4xlarge`, depending on the availability in each EC2 region. The GPU instance is `g2.8xlarge`. The low-cost instance (`m4.xlarge`) is the one used for centralizing input data. All the instance costs are shown in USD per hour. All WAN data transfer costs are shown in USD per GB.

Table 2: Cost model details

Region	CPU Instance	GPU Instance	Low-cost Instance	Send to WANs	Recv. from WANs
Virginia	$0.86	$2.60	$0.22	$0.02	$0.01
California	$1.01	$2.81	$0.22	$0.02	$0.01
Oregon	$0.86	$2.60	$0.22	$0.02	$0.01
Ireland	$0.95	$2.81	$0.24	$0.02	$0.01
Frankfurt	$1.03	$3.09	$0.26	$0.02	$0.01
Tokyo	$1.11	$3.59	$0.27	$0.09	$0.01
Seoul	$1.06	$3.59	$0.28	$0.08	$0.01
Singapore	$1.07	$4.00	$0.27	$0.09	$0.01
Sydney	$1.08	$3.59	$0.27	$0.14	$0.01
Mumbai	$1.05	$3.59	$0.26	$0.09	$0.01
São Paulo	$1.37	$4.00	$0.34	$0.16	$0.01