Asynchronous Approximate Data-parallel Computation

Asim Kadav and Erik Kruus
NEC Labs America

Abstract

Emerging workloads, such as graph processing and machine learning are approximate because of the scale of data involved and the stochastic nature of the underlying algorithms. These algorithms are often distributed over multiple machines using bulk-synchronous processing (BSP) or other synchronous processing paradigms such as map-reduce. However, data-parallel processing primitives such as repeated `barrier` *and* `reduce` *operations introduce high synchronization overheads. Hence, many existing data-processing platforms use asynchrony and staleness to improve data-parallel job performance. Often, practitioners simply change the synchronous communication to asynchronous between the worker nodes in the cluster. This improves the throughput of data processing but results in a poor accuracy of the final output since different workers may progress at different speeds and process inconsistent intermediate outputs.*

In this paper, we present ASAP, a model that provides asynchronous and approximate processing semantics for data-parallel computation. ASAP provides fine-grained worker synchronization using `NOTIFY-ACK` *semantics that allows independent workers to run asynchronously. ASAP also provides* stochastic reduce *that provides approximate but guaranteed convergence to the same result as an aggregated all-reduce. In our results, we show that ASAP can reduce synchronization costs and provides 2-10X speedups in convergence and up to 10X savings in network costs for distributed machine learning applications and provides strong convergence guarantees.*

1. Introduction

Large-scale distributed data-parallel computation often provides two fundamental constructs to scale-out local data processing. First, a *merge* or a *reduce* operation to allow the workers to merge updates from all other workers and second, a *barrier* or an implicit *wait* operation to ensure that all workers can synchronize and operate at similar speeds. For example, the bulk-synchronous model (BSP) is a general paradigm to model data-intensive distributed processing [62]. Here, each node after processing a specific amount of data synchronizes with the other nodes using `barrier` and `reduce` operations. The BSP model is widely used to implement many big data applications, frameworks and libraries such as in the areas of graph processing and machine learning [1, 28, 33, 36, 45, 55]. Other synchronous paradigms such as the map-reduce [26, 68], the parameter server [25, 43] and dataflow based systems [6, 39, 48] use similar constructs to synchronize outputs across multiple workers.

There is an emerging class of big data applications such as graph-processing and machine learning (ML) that are approximate because of the stochastic nature of the underlying algorithms that converge to the final solution in an iterative fashion. These iterative-convergent algorithms operate on large amounts of data and unlike traditional TPC style workloads that are CPU bound [53], these iterative algorithms incur significant network and synchronization costs by communicating large vectors between their workers. These applications can gain an increase in performance by reducing the synchronization costs in two ways. First, the workers can operate over stale intermediate outputs. The stochastic algorithms operate over input data in an iterative fashion to produce and communicate intermediate outputs with other workers. However, it is not imperative that the workers perform a `reduce` on all the intermediate outputs at every iteration. Second, the synchronization requirements between the workers may be relaxed, allowing partial, stale or overwritten outputs. This is possible because in some cases the iterative nature of data processing and the stochastic nature of the algorithms may provide an opportunity to correct any errors introduced from staleness or incorrect synchronization.

There has been recent research that uses this property by processing stale outputs, or by removing all synchronization [56, 64]. However, naïvely converting the algorithms from synchronous to asynchronous can increase the throughput but may not improve the convergence speeds. This is because an increase in data processing speeds may not produce the final output with same accuracy and in some cases may even converge the underlying algorithm to an incorrect value [47].

Hence, to provide asynchronous and approximate semantics with reasonable correctness guarantees for iterative convergent algorithms, we present Asynchronous and Approximate abstractions for data-parallel computation. To facilitate approximate processing, we describe *stochastic reduce*, a sparse reduce primitive, that mitigate the communication and synchronization costs by performing the `reduce` operation with fewer workers. We construct a reduce operator by choosing workers based on sparse, directed *expander* graphs on underlying communication nodes that mitigates CPU and network costs during `reduce` for iterative convergent algorithms. Furthermore, stochastic reduce convergence is directly proportional to the spectral gap of the underlying machine graph that allows practitioners to introduce adjust network structure based on available network bandwidth.

To reduce synchronization costs, we propose a fine-grained `NOTIFY-ACK` based synchronization that provides

performance improvement over `barrier` based methods. `NOTIFY-ACK` allows independent worker threads (such as those in stochastic reduce) to run asynchronously instead of blocking on a coarse-grained global `barrier` at every iteration. Additionally, `NOTIFY-ACK` provides stronger consistency than simply using a `barrier` to implement synchronous data-parallel processing.

ASAP is not a programming model (like map-reduce [26]) or is limited to a set of useful mechanisms. It introduces semantics for approximate and asynchronous execution which are often amiss in the current flurry of distributed machine learning systems which often use asynchrony and staleness to trade-off the input processing throughput with output accuracy. The contributions of this paper are as follows:

- We present ASAP, an asynchronous approximate computation model for large scale data parallel applications. We present an inter-discplinary approach where we combine algorithm design with the hardware properties of low latency networks to design a distributed data-parallel system. We use stochastic reduce for approximate semantics with fine-grained synchronization based on `NOTIFY-ACK` to allow independent threads run asynchronously.
- With stochastic reduce, we present a spectral gap measure that allows developers to reason why some node communication graphs may converge faster than others and can be a better choice for connecting machines with stochastic reduce. This allows developers to compare the flurry of recent papers that propose different network topologies for gradient propagation such as ring, tree etc.
- We apply ASAP to build a distributed learning framework over RDMA. In our results, we show that our system can achieve strong consistency, provable convergence, and provides 2-10X in convergence and up to 10X savings in network costs.

2. Background

Large-scale problems such as training image classification models, page-rank computation and matrix factorization operate on large amounts of data. As a result, many stochastic algorithms have been proposed that make these problem tractable for large data by iteratively approximating the solution over small batches of data. For example, to scale better, matrix factorization methods have moved from direct and exact factorization methods such as singular value decomposition to iterative and approximate factorization using gradient descent style algorithms [29]. Hence, many algorithms that discover relationships within data have been re-written in the form of distributed optimization problems that iterate over the input data and approximate the solution. In this paper, we focus on how to provide asynchronous and approximate semantics to distributed machine learning applications. These optimizations can be beneficial to end users when they run their distributed machine learning jobs across multiple cloud instances [38] or by the cloud provider themselves when training a model for vision or speech recognition [19, 34, 35].

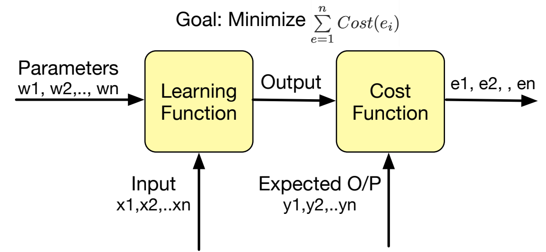


Figure 1: The machine learning training process.

2.1. Distributed Machine Learning

Machine learning algorithms process data to build a training model that can generalize over new data. The training output model, or the parameter vector (represented by w) is computed to perform future predictions over new data. To train over large data, ML methods often use the Stochastic Gradient Descent (SGD) algorithm that can train over a single (or a batch) of examples over time. The SGD algorithm processes data examples to compute the gradient of a loss function. The parameter vector is then updated based on this gradient value after processing each training data example. After a number of iterations, the parameter vector or the model converges towards acceptable error values over test data.

Figure 1 shows the ML training process. To scale out the computation over multiple machines, the SGD algorithm can be distributed over a cluster by using data parallelism, by splitting the input ($x1$,$x2$,..,xn) or by model parallelism, by splitting the model ($w1$, $w2$,..,wn). The goal of parallelization is not just to improve throughput but also to maintain low error rates ($e1$, $e2$,..,en). In data-parallel learning using BSP, the parallel model replicas train over different machines. After a fixed number of iterations, these machines synchronize the parameter models that have been trained over the partitioned data with one-another using a `reduce` operation. For example, each machine may perform an average of all incoming models with its own model, and proceed to train over more data. In the BSP model, there is a global barrier that ensures that models train and synchronize intermediate inputs at the same speeds. Hence, distributed data-parallel ML suffers from additional synchronization and communication costs over a single thread.

Figure 2 shows the interconnect speeds for a single computer buses, ethernet and infiniBand. Modern buses can send data at higher throughputs with increasingly lower latencies. As a result, software costs for synchronization, such as in a `reduce` operation have begun to contribute significantly towards overall job times in distributed machine learning and other iterative data-parallel tasks. The `reduce` operation requires communicating updates to all other machines when training using the BSP or the map-reduce model. However, since these algorithms are *iterative-convergent*, and can tolerate errors in the synchronization step, there has been re-

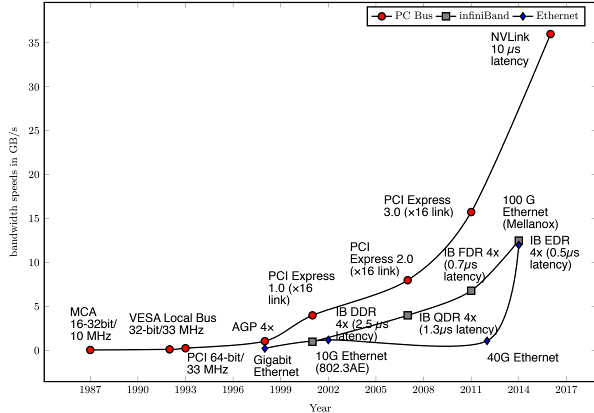


Figure 2: Decreasing latency and increasing local and network bandwidths over the years. The latency measurements are for 64B packets.

cent work on communicating stale intermediate parameter updates and exchanging parameters with little or no synchronization [11, 18, 56].

Past research has found that simply removing the barrier may speed up the throughput of the system [15, 64]. However, this may not always improve the convergence speed and may even converge the system to an incorrect final value [47]. Since the workers do not synchronize, and communicate model parameters at different speeds, the workers process the data examples at a higher throughput. However, since the different workers train at different speeds, the global model skews towards the workers that are able to process and communicate their models. Similarly, using bounds for asynchrony may appear to work for some workloads. But determining these bounds can be empirically difficult and for some datasets these bounds may be no better than synchronous. Furthermore, if a global single model is maintained and updated without locks (as in [56]), a global convergence may only be possible if the parameter vector is sparse. Finally, maintaining a global single model in a distributed setting results in lots of wasted communication since a lot of the useful parameter updates are overwritten [15, 51].

The distributed parameter-server architecture limits network traffic by maintaining a central master [6, 25, 43]. Here, the server coordinates the parameter consistency amongst all other worker machines by resetting the workers' model after every iteration and ensures global consensus on the final model. Hence, a single server communicates with a large number of workers that may result in network congestion at the edges which can be mitigated using a distributed parameter server [43]. However, the parameter server suffers from similar synchronization issues as BSP style systems – a synchronous server may spend a significant amount of time at the `barrier` while an asynchronous server may `reduce` with few workers' models and produce inconsistent intermediate outputs and this can slow down convergence. Hence, the parameter server architecture can benefit from a fine-grained synchronization mechanisms that have low overheads.

To provide asynchronous and approximate processing semantics with consistency and convergence guarantees, we introduce ASAP that provides approximate processing by synchronizing each worker with a subset of workers at each iteration. Additionally, ASAP provides fine-grained synchronization that improves convergence behavior and reduces synchronization overheads over a `barrier`. We describes both these techniques next.

3. Stochastic reduce for approximate processing

In this section, we describe how data-parallel applications can use stochastic reduce to mitigate network and processing times for iterative ML algorithms. More importantly, we prove that convergence speeds of algorithms depend on the *spectral-gap values* of underlying node communication graph of the cluster.

With distributed ML parallel workers train on input data over model replicas. They synchronize with one-another after few iterations and perform a `reduce` over intermediate model updates and continue to train. This synchronization step is referred to as the *all-reduce* step. In the parameter server model, this synchronization is performed at a single or distributed master [25, 44]. To mitigate the reduce overheads, efficient all-reduce has been explored in the map-reduce context where nodes perform partial aggregation in a tree-style to reduce network costs [17, 12, 66]. However, these methods decrease the network and processing costs at the cost of increasing the latency of the `reduce` operation proportional to the height of the tree.

The network of worker nodes, i.e. the node communication graph of the cluster determines how rapidly the intermediate model parameters are propagated to all other machines and also determines the associated network and processing costs. This network information diffusion or connectedness should be high for faster convergence but imposes higher network costs. For example, all-reduce and the parameter server represent different types of communication graphs that describe how the workers communicate the intermediate results as shown in Figure 3. In the all-reduce architecture, all machines exchange parameters while in a parameter server architecture a central node coordinates the parameter update. Intuitively, when the workers communicate with all machines at every `reduce` step, this network is densely connected and convergence is rapid since all the machines get the latest intermediate updates. However, if the network of nodes is sparsely connected, the convergence may be slow due to stale, indirect updates being exchanged between machines. However, with sparse connectivity, there are savings in network and CPU costs (fewer updates to process at each node), that can result in an overall speedup in job completion times. Furthermore, if there is a heterogeneity in communication bandwidths between the workers (or between the workers and the master, if a master is used), many workers may end up waiting. As an example, if one is training using GPUs over a cluster, GPUs within one machine can synchronize at far lower costs over

the PCI bus than over the network. Hence, frequent `reduce` across interconnects with varying latency may introduce a large synchronization cost for all workers. Hence, we propose using sparse or *stochastic reduce* based on sparse node graphs with strong connectivity properties.

The goal of stochastic reduce is to improve performance by reducing network and processing costs by using sparse reduce graphs. Recent work has shown that every dense graph can be reduced to a sparse graph with fewer edges [9] with similar network information diffusion properties. This is a significant result since it implies that stochastic reduce can be applied to save network costs for almost any network topology. Expander graphs, which are sparse graphs with strong connectivity properties have been explored in the context of data centers and distributed communities to communicate data with low overheads [60, 61]. An expander graph has fixed out-degrees as the number of vertices increase while maintaining approximately the same connectivity between the vertices. Hence, using directed expander graphs for stochastic reduce provides approximately the same convergence as all-reduce while keeping network costs low as the number of nodes increase. Directed graphs can use the one-sided properties of RDMA networks that allow for fast, asynchronous communication. Furthermore, we use low-degree graphs for less overall communication costs. Both these properties ensure better scalability for peer-to-peer communication fabrics like RDMA over Infiniband.

To measure the convergence of algorithms that use stochastic reduce, i.e. to compare the sparsity of the adjacency graph of communication, we calculate the *spectral gap* of the adjacency matrix of the network of workers. The spectral gap is the difference between the two largest singular values of the adjacency matrix normalized by the in-degree of every node. The spectral gap of a communication graph determines how rapidly a distributed, iterative sparse reduce converges when performing distributed optimization over a network of nodes represented by this graph. For faster convergence, this value should be as high as possible. Hence, densely-connected graphs have a high spectral gap value and converge faster but can have high communication costs. Conversely, if the graph is disconnected, the spectral gap value is *zero*, and with partitioned data, the model may never converge to a correct value. Hence, there is a research opportuniy in designing networks that have high-spectral gap but have low communication costs by using low-degree nodes.

Past work using partial-reduce for optimization problems has explored fixed communication graphs such as butterfly or logarithmic sequences [11, 42]. These communication sequences provide fixed network costs but may not generalize to networks with complex topologies or networks with dissimilar bandwidth edges such as distributed GPU environments. More importantly, ASAP introduces the ability to *reason* convergence using the spectral gap of a network graph and developers can reason why some node graphs have stronger convergence properties. Finally, existing approaches use a global barrier after each iteration, incurring extra synchronization overheads which can be reduced using ASAP's fine-grained synchronization described in the next section. In the next section, we prove that this convergence can be compared using the spectral gap of the node graphs.

3.1. Stochastic reduce convergence analysis

In this section, we analyze the conditions for convergence for stochastic reduce for any distributed optimization problem that achieves consensus by communicating information between nodes. We show that the rate of convergence for a set of nodes is dependent on the spectral gap of the transition matrix of the nodes. The transition matrix represents the probability of transition of gradient updates from one node to another and is the ratio of the adjacency matrix/in-degree of each node. Mathematically, the optimization problem is defined on a connected directed network and solved by n nodes collectively,

$$\min_{\boldsymbol{x} \in \mathscr{X} \subseteq \mathbb{R}^d} \quad \bar{f}(\boldsymbol{x}) := \sum_{i=1}^{n} f_i(\boldsymbol{x}). \tag{3.1}$$

The feasible set $\mathscr{X}$ is a closed and convex set in $\mathbb{R}^d$ and is known by all nodes, whereas $f_i : \mathscr{X} \in \mathbb{R}$ is a convex function known by the node i. We also assume that f_i is L-Lipschitz continuous over $\mathscr{X}$ with respect to the Euclidean norm $\|\cdot\|$. The network $\mathscr{G} = (\mathscr{N}, \mathscr{E})$, with the node set $\mathscr{N} = [n] := \{1, 2, \cdots, n\}$ and the edge set $\mathscr{E} \subseteq \mathscr{N} \times \mathscr{N}$, specifies the topological structure on how the information is spread amongst the nodes through local node interactions over time. Each node i can only send and retrieve information as defined by the node communication graph $\mathscr{N}(i) := \{j \mid (j,i) \in \mathscr{E}\}$ and itself.

Our goal is to measure the overall convergence time (also known as mixing time in Markov chain literature) based on a specific network topology. We specifically use *directed* graphs as using undirected graph such as in the case of parameter servers [25, 43] and exchange based peer-to-peer protocols [65] require synchronization. Furthermore, directed graph style communication improves performance in-case of RDMA networks using one-sided communication.

In this algorithm, each node i keeps a local estimate $\boldsymbol{x}_i$ and a model variable $\boldsymbol{w}_i$ to maintain an accumulated sub-gradient. At iteration t, to update $\boldsymbol{w}_i$, each node needs to collect the model update values of its neighbors which is the gradient value denoted by $\nabla f_i(\boldsymbol{w}_t^i)$, and forms a convex combination with an equal weight of the received information. The learning rate is denoted by η_t. Hence, the updates received by each machine can be expressed as:

$$\begin{aligned} \boldsymbol{w}_{t+1/2}^i &\leftarrow \boldsymbol{w}_t^i - \eta_t \nabla f_i(\boldsymbol{w}_t^i) \\ \boldsymbol{w}_{t+1}^i &\leftarrow \frac{1}{|\mathscr{N}_{in}^i|} \sum_{j \in \mathscr{N}_{in}^i} \boldsymbol{w}_{t+1/2}^j \end{aligned} \tag{3.2}$$

In order to make the above algorithm converge correctly, we need a network over which each node has the same influence. To understand and quantify this requirement, we denote the adjacency matrix as $\boldsymbol{A}$, i.e. $A_{ij} = 1$ if $(j,i) \in \mathscr{A}$ and 0 otherwise, and denote $\boldsymbol{P}$ as the transition matrix after scaling each i-th row of $\boldsymbol{A}$ by the in-degree of node i, i.e. $\boldsymbol{P} = \text{diag}(\boldsymbol{d}_{in})^{-1}\boldsymbol{A}$, where $\boldsymbol{d}_{in} \in \mathbb{R}^k$ and $d_{in}(i)$ equals the in-degree of node i. For ease of illustration, we assume that degree $d = 1$ and initial $w_0^i = 0$. We denote sub-gradient and model updates over time as $\boldsymbol{g}_t = (g_t^1, g_t^2, \dots, g_t^k)$, and $\boldsymbol{w}_t = (w_t^1, w_t^2, \dots, w_t^k)^\top$. Then the updates can be expressed as :

$$
\begin{aligned}
\boldsymbol{w}_1 &= -\eta_0 \boldsymbol{P}\boldsymbol{g}_0 \\
\boldsymbol{w}_2 &= -\eta_0 \boldsymbol{P}^2\boldsymbol{g}_0 - \eta_1 \boldsymbol{P}\boldsymbol{g}_1 \\
&\vdots \\
\boldsymbol{w}_t &= -\eta_0 \boldsymbol{P}^t\boldsymbol{g}_0 - \eta_1 \boldsymbol{P}^{t-1}\boldsymbol{g}_1 - \dots - \eta_{t-1}\boldsymbol{P}\boldsymbol{g}_{t-1} \\
\boldsymbol{w}_t &= -\sum_{k=0}^{t-1} \eta_k \boldsymbol{P}^{t-k}\boldsymbol{g}_k \qquad (3.3)
\end{aligned}
$$

It can be easily verified that $\boldsymbol{P}^\infty := \lim_{t\to\infty} \boldsymbol{P}^t = \boldsymbol{1}\boldsymbol{\pi}^\top$, where $\boldsymbol{\pi}$ is a probability distribution (known as stationary distribution). Thus, π_i here represents the influence that node i plays in the network. Therefore, to take a fair treatment of each node, $\pi_i = \frac{1}{k}$ is desired, which is equivalent to when the row sums and column sums of the transition matrix $\boldsymbol{P}$ are all equal to one, i.e. $\boldsymbol{P}$ is a doubly stochastic matrix. Hence, in the context of our network setting, we need a network whose nodes have the same in-degree. Intuitively, this means that as long as all nodes are connected and each node has the same influence i.e. sends its gradients to equal number of neighbors, the overall problem will converge correctly even if sparsely connected graphs may require a large number of iterations. Indeed, when f_i is convex, the convergence results have been established under this assumption [27, 49].

Besides being regular, the network $\mathscr{N}(i)$ should also be constructed in a way such that the information can be effectively spread out over the entire network, which is closely related with the concept of spectral gap. We calculate the **spectral gap** of the network as $1 - \sigma_2(\boldsymbol{P})$, where $\sigma_2(\boldsymbol{P})$ is the second largest singular value of P. The transition matrix P is defined as A/d, where A is the adjacency matrix (including self-loop) and d is the in-degree (including self-loop). The spectral gap here is defined as $\sigma_1(\boldsymbol{P}) - \sigma_2(\boldsymbol{P})$. But $\sigma_1(\boldsymbol{P})$ the largest singular value should be 1. So the gap equals $1 - \sigma_2(\boldsymbol{P})$, where $\sigma_2(\boldsymbol{P})$ is the second largest singular value of P. We denote $\sigma_1(\boldsymbol{P}) \geq \sigma_2(\boldsymbol{P}) \geq \dots \geq \sigma_k(\boldsymbol{P}) \geq 0$, where $\sigma_i(\boldsymbol{P})$ is the ith largest singular value of P. Clearly, $\sigma_1(\boldsymbol{P}) = 1$. From the expression (3.3), we see that the speed of convergence depends on how fast $\boldsymbol{P}^t$ converges to $\frac{1}{k}\boldsymbol{1}\boldsymbol{1}^\top$, and based on the Perron-Frobenius theory [57], we have,

$$
\left\| \boldsymbol{P}^t \boldsymbol{x} - \frac{1}{n}\boldsymbol{1} \right\|_2 \leq \sigma_2(\boldsymbol{P})^t,
$$

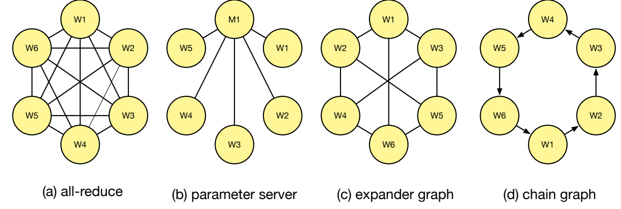


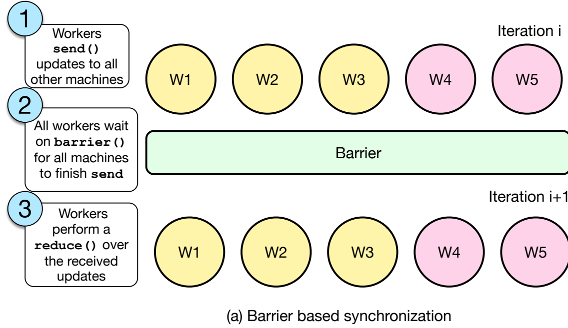
Figure 3: Figure (a) shows all-reduce, Spectral Gap (SG) for 6/25 nodes, SG-6:1.00, SG-25:1.00. Figure (b) shows parameter server, SG-6:0.75, SG-25:0.68. Figure (c) shows an expander graph with SG-6:0.38, SG-25:0.2 Figure (d) shows a chain graph with SG-6: 0.1, SG-25: 0.002. To the best of our knowledge, our work is the first to quantify convergence rates of commonly used distributed learning node communication graphs.

for any $\boldsymbol{x}$ in the k-dimensional probability simplex. Therefore, the network with large spectral gap, $1 - \sigma_2(\boldsymbol{P})$, is greatly desired. Hence, the spectral gap represents how rapidly the mixing time or the variance between the model at different nodes decreases. This should be as high as possible for a given network budget. A spectral gap of 1 indicates all nodes are inter-connected, providing the fastest convergence but possibly high network costs. A spectral gap of 0 indicates a disconnected or partitioned network graph. We now compare this spectral gap values for commonly used distributed learning architectures and construct a low-degree, high-spectral gap node communication graph.

3.2. Stochastic reduce using expander graphs

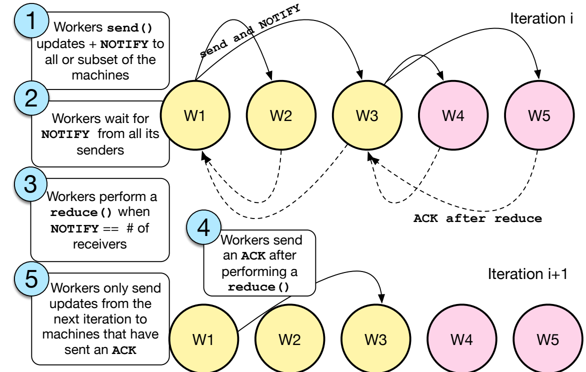
Figure 3 shows six nodes connected using four distributed ML training architectures, the all-reduce, the parameter server (non-distributed), an expander graph with a fixed out-degree of two, and a chain like architecture and their respective spectral-gap values for 6 and 25 nodes.

As expected, architectures that contain nodes with more edges (higher-indegree) have a higher spectral gap. Figure 3(a) shows the all-reduce, where all machines communicate with one-another and may incur significant network costs. Figure 3(b) shows the parameter server has a reasonably high spectral gap but using a single master with a high fanout requires considerable network bandwidth and Paxos-style reliability for the master. Figure 3(c) shows a root expander graph has a fixed out-degree of two, and in a network of N total nodes, each node i sends the intermediate output to its neighbor $(i+1)$ (to ensure connectivity) and to $i + \sqrt{N}$th node. Such root expander graphs ensure that the updates are spread across the network as N scales since the root increases with N. Finally, figure 3(d), shows a chain like graph, where the nodes are connected in a chain-like fashion, the intermediate parameter updates from node i may spread to $i+1$ in a single time step, but will require N time steps to reach to the last node in the cluster and has low spectral gap values. In the rest of the paper, we use the *root* sparse communication graph, as shown in Figure 3(c), with a fixed out-degree of two, to evaluate stochastic reduce.



(a) Barrier based synchronization

Figure 4: This figure shows sparse synchronization semantics where workers W1, W2 and W3 synchronize with one another and W3, W4 and W5 synchronize with one another. With a `barrier`, *all* workers wait for every other worker and then proceed to the next iteration.



(b) NOTIFY-ACK based synchronization

Figure 5: Fine-grained synchronization in ASAP. The solid lines show `NOTIFY` operation and the dotted lines show the corresponding `ACK`. Workers only wait for intermediate outputs from dependent workers to perform a `reduce`. After `reduce`, workers push more data out when they receive an `ACK` from receivers signaling that the sent parameter update has been consumed.

We find that by using sparse communication reduce graphs, with just two out-degrees often provides good convergence and speedup over all-reduce i.e. high enough spectral gap values with reasonably low communication costs. Using sparse reduce graphs with stochastic reduce results in faster model training times because: First, the amount of network time is reduced. Second, the synchronization time is reduced since each machine communicates with fewer nodes. Finally, the CPU costs at each node are reduced since they process fewer incoming model updates.

For stochastic reduce to be effective, the following properties are desirable. First, the generated node communication graph should have a **high-spectral gap**. This ensures that the model updates from each machine are diffused across the network rapidly. Second, the node-communication graphs should have **low communication costs**. For example, the out-degrees of each node in the graph should have small out-degrees. Finally, the graph should be **easy to generate** such as using a sequence to accommodate variable number of machines or a possible re-configuration in case of a failure. These properties can be used to guide existing data-parallel optimizers [37] to reduce the data shuffling costs by constructing sparse reduce graphs to accommodate available network costs and other constraints such as avoiding cross-rack `reduce`. We now discuss how to reduce the synchronization costs with fine-grained communication.

4. Fine-grained synchronization

Barrier based synchronization is an important and widely used operation for synchronizing parallel ML programs across multiple machines. After executing a `barrier` operation, a parallel worker waits until *all* the processes in the system have reached a `barrier`. Parallel computing libraries like MPI, as well as data parallel frameworks such as BSP systems and some parameter servers expose this primitive to the developers [14, 33, 45, 43]. Furthermore, ML systems based on map-reduce use the stage barrier between the `map` and `reduce` tasks to synchronize intermediate outputs across machines [16, 30].

Figure 4 shows a parallel training system on n processes. Each process trains on a subset of data and compute the intermediate model update and issues a `send` to other machines and the wait on the `barrier` primitive. When all processes arrive at the barrier, the workers perform a `reduce` operation over the incoming output and continue processing more input.

However, using the `barrier` as a synchronization point in the code suffers from several problems: First, the BSP protocol described above, suffers from mixed-version issues i.e. in the absence of additional synchronization or serialization at the receive side, a receiver may perform a `reduce` with partial or torn model updates (or skip them if a consistency check is enforced). This is because just using a `barrier` gives no information if the recipient has finished receiving and consuming the model update. Second, most `barrier` implementations synchronize with all other processes in the computation. In contrast, with stochastic reduce, finer grained synchronization primitives are required that will block on only the required subset of workers to avoid unnecessary synchronization costs. A global `barrier` operation is slow and removing this operation can reduce synchronization costs, but makes the workers process inconsistent data that may slow down the overall time to achieve the final accuracy. Finally, using a `barrier` can cause network resource spikes if all the processes send their parameters at the same time.

Adding extra barriers before/after push and reduce, does not produce a strongly consistent BSP that can incorporate model updates from all replicas since the actual send operation may be asynchronous and there is no guarantee the receivers receive these messages when the perform a reduce. Unless a blocking receive is added after every send, the consistency is not guaranteed. However, this introduces a significant synchronization overhead.

Hence, to provide efficient coordination among parallel model replicas, we require the following three properties

from any synchronization protocol. First, the synchronization should be *fine-grained*. Coarse-grained synchronization such as `barrier` impose high overheads as discussed above. Second, the synchronization mechanism should provide *consistent* intermediate outputs. Strong consistency methods avoid torn-reads and mixed version parameter vectors, and improve performance [13, 67]. Finally, the synchronization should be *efficient*. Excessive receive-side synchronization for every `reduce` and `send` operation can significantly increase blocking times.

In data-parallel systems with `barrier` based synchronization, there is often no additional explicit synchronization between the sender and receiver when an update arrives. Furthermore, any additional synchronization may reduce the performance especially when using low latency communication hardware such as RDMA that allow one-sided writes without interrupting the receive-side CPU [40]. In the absence of synchronization, a fast sender can overwrite the receive buffers or the receiver may perform a `reduce` with a fewer senders instead of consuming each worker's output hurting convergence. This is especially problematic with RDMA memory that can only pin a finite amount of memory on the device.

Naiad, a dataflow based data-parallel system, provides a `notify` mechanism to inform the receivers about the incoming model updates [48]. This ensures that when a node performs a local `reduce`, it consumes the intermediate outputs from all machines. Hence, a per-receiver notification allows for finer-grained synchronization. However, simply using a `notify` is not enough since a fast sender can overwrite the receive queue of the receiver and a `barrier` or any other style of additional synchronization is required to ensure that the parallel workers process incoming model parameters at the same speeds.

To eliminate the `barrier` overheads for stochastic reduce and to provide strong consistency, we propose using a `NOTIFY-ACK` based synchronization mechanism that gives stricter guarantees than using a coarse grained `barrier`. This can also improve convergence times in some cases since it facilitates using consistent data from dependent workers during the `reduce` step.

In ASAP, with `NOTIFY-ACK`, the parallel workers compute and send their model parameters with notifications to other workers. They then proceed to `wait` to receive notifications from all its senders as defined by their node communication graphs as shown in figure 5. The `wait` operation counts the `NOTIFY` events and invokes the `reduce` when a worker has received notifications from all its senders as described by the node communication graph. Once all notifications have been received, it can perform a consistent `reduce`.

After performing a `reduce`, the workers sends `ACK`s, indicating that the intermediate output in previous iteration has been consumed. Only when a sender receives this `ACK` for a previous send, it may proceed to send the data for the next iteration. Unlike a `barrier` based synchronization, where there is no guarantee that a receiver has consumed the intermediate outputs from all senders, waiting on `ACK`s from receivers ensures that a sender never floods the receive side queue and avoids any mixed version issues from overlapping intermediate outputs. Furthermore, fine-grained synchronization allows efficient implementation of stochastic reduce since each sender is only blocked by dependent workers and other workers may run asynchronously.

`NOTIFY-ACK` requires no additional receive-side synchronization making it ideal for direct-memory access style protocols such as RDMA or GPU Direct [3]. However, `NOTIFY-ACK` requires ordering guarantees of the underlying implementation to guarantee that a `NOTIFY` arrives after the actual data. Furthermore, in a `NOTIFY-ACK` based implementation, the framework should ensure that the workers send their intermediate updates and then `wait` on their reduce inputs to avoid any deadlock from a cyclic node communication graphs.

5. Implementation

We develop our second generation distributed ML framework using the ASAP model and incorporate stochastic reduce and fine-grained synchronization. We implement distributed data-parallel model averaging over stochastic gradient descent (SGD). We implement our reference framework with stochastic reduce and `NOTIFY-ACK` support in C++ and provide Lua bindings to run our Lua based deep learning networks [20]. As compared to our first-generation system [8], we use zero copy for Lua tensors, use `NOTIFY-ACK` instead of BSP for sparse graphs. Furthermore, we use low-degree sparse graph to connect machine nodes rather than fixed-degree fully-connected graphs.

For distributed communication, we use MPI and create model parameters in distributed shared memory. In our implementation, the parallel model replicas create a model vector in the shared memory and train on a portion of the dataset using the SGD algorithm. The model replicas process the partitioned dataset to compute and communicate the gradient and perform a `reduce` periodically.

We use the infiniBand transport and each worker directly writes the intermediate model to its senders without interrupting the receive side CPU, using one-sided RDMA operations. To reduce synchronization overheads, each machine maintains a fixed size per-sender receive queue to receive the model updates from other machines [55]. Hence, after computing the gradient, parallel replicas write their updates to this queue using one-sided RDMA writes. Each replicas performs a `reduce` of its model and all the models in the queues following `NOTIFY-ACK` semantics. Hence, our system only performs RDMA writes which have half the latency of RDMA reads. The queues and the shared memory communication between the model replicas are created based on a node communication graph provided as an input when launching a job. After the `reduce` operation, each machine sends out the model updates to other machines' queues as defined by the

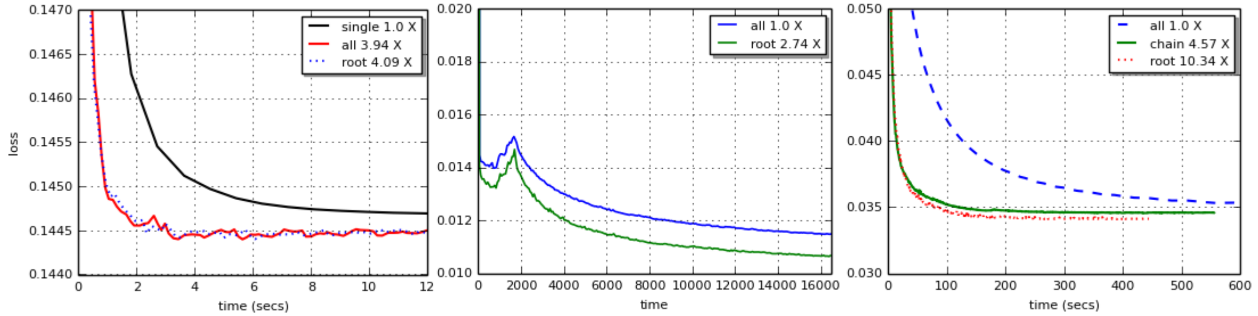


Figure 6: Figure (a) compares the speedup of a root expander graph and an all-reduce graph for 4 workers a single machine SGD. Each machine in root expander graph transmits 56 MB data while all-reduce transmits 84 MB of data to converge. Figure (b) shows the convergence of a root expander graph with all-reduce graph for the splice-site dataset over 8 workers. Each machine in root graph transmits 219 GB of data while all-reduce transmits 2.08 TB/machine to reach the desired accuracy. Figure (c) shows the convergence of a chain graph, a root expander graph and an all-reduce implementation for the webspam dataset over 25 workers.

communication graph. Our system can perform `reduce` over any user-provided node communication graph allowing us to evaluate stochastic reduce for different node communication graphs.

Furthermore, we also implement the synchronous, asynchronous and `NOTIFY-ACK` based synchronization. We implement synchronous (BSP) training by using the `barrier` primitive. We use low-level distributed `wait` and `notify` primitives to implement `NOTIFY-ACK`. We maintain ACK counts at each node and send all outputs before waiting for ACKs across iterations to avoid deadlocks. We use separate infiniBand queues for transmitting short messages (ACKs and other control messages) and large model updates (usually a fixed size for a specific dataset). For Ethernet based implementation, separate TCP flows can be used to reduce the latency of control messages [50].

Fault Tolerance: We provide a straight-forward model for fault tolerance by simply check-pointing the trained model periodically to the disk. Additionally, for synchronous methods, we implement a dataset specific dynamic timeout which is a function of the time taken for the `reduce` operation.

6. Evaluation

We evaluate (i) What is the benefit of stochastic reduce? Do networks with a higher spectral gap exhibit better convergence? and (ii) What is the speedup of using fine-grained synchronization over a `barrier`? Is it consistent?

We evaluate the ASAP model for applications that are commonly used today including text classification, spam classification, image classification and Genome detection. Our baseline for evaluation is our BSP/synchronous and asynchronous based all-reduce implementations which is provided by many other distributed big data or ML systems [1, 6, 24, 28, 33, 36, 45, 55]. We use an efficient infiniBand implementation stack that improves performance for all methods. However, stochastic reduce and `NOTIFY-ACK` can be implemented and evaluated over any existing distributed learning platform such as GraphLab or TensorFlow.

We run our experiments on eight Intel Xeon 8-core, 2.2 GHz Ivy-Bridge processors and 64 GB DDR3 DRAM. All connected via a Mellanox Connect-V3 56 Gbps infiniBand cards. Our network achieves a peak throughput of 40 Gbps after accounting for the bit-encoding overhead for reliable transmission. All machines load the input data from a shared NFS partition. We sometimes run multiple processes on each machine, especially for models with less than 1M parameters, where a single model replica is unable to saturate the network and CPU. All reported times do not account for the initial one-time cost for the loading the data-sets in memory. All times are reported in seconds.

We evaluate two ML methods – (a) SVM: We test ASAP on distributed SVM based on Bottou's SVM-SGD [10]. Each machine computes the model parameters and communicates them to other machines as described in the machine communication graph. We train SVM over the RCV1 dataset (document classification), the webspam dataset (webspam detection) and the splice-site dataset (Genome classification) [4]; and(b) Convolutional Neural Networks (CNNs): We train CNNs for image classification over the CIFAR-10 dataset [5]. The dataset consists of 50K train and 10K test images and the goal is to classify an input image within 10 classes. We use the VGG network to train 32x32 CIFAR-10 images with 11 layers that has 7.5M parameters [59]. We use `OMP_PARALLEL_THREADS` to parallelize the convolutional operations within a single machine.

6.1. Approximate processing benefits

Speedup with stochastic reduce We measure the speedups of all applications under test as the time to reach a specific accuracy. We first evaluate a small dataset (RCV1, 700MB, document classification) for the SVM application. The goal here is to demonstrate that for problems that fit in one machine, our data-parallel system outperforms a single thread for each workload [46]. Figure 6(a) shows the convergence speedup for 4 machines for the RCV1 dataset. We compare the performance for all-reduce against a root graph with a fixed out degree of two, where each node sends the model updates to two nodes – its neighbor and *rootNth* node. We find that

the root expander graph converges marginally faster than the all-reduce primitive, owing to marginally lower network and CPU costs since the number of machines is small.

Figure 6(b) shows the convergence for the SVM application using the splice-site dataset on 8 machines with 8 processes. The splice site training dataset is about 250GB, and does not fit in-memory in any one of our machines. This is one of the largest public dataset that cannot be sub-sampled and requires training over the entire dataset to converge correctly [7]. Figure 6(b) compares the two communication graphs – an all-reduce graph and a root expander graph with an out-degree of 2. We see that the expander graph can converge faster, and requires about 10X lower network bandwidth. Figure 6(c) shows the convergence for the SVM application on 25 processes using the webspam dataset that consists of 250K examples. We compare three node communication graphs – a root expander graph (with a spectral gap of 0.2 for 25 nodes) and a chain-like architecture where each machine maintains a model and communicates its updates in the form a chain to one other machine, with the all-reduce implementation. The chain node graph architecture has lower network costs, but very low spectral values (≈ 0.008). Hence, it converges slower than the root expander graph. Both the sparse graphs provide a speedup over all-reduce since the webspam dataset has a large dense parameter vector of about 11M `float` values. However, the chain graph requires more epochs to train over the dataset and sends 50433 MB/node to converge while the root node requires 44351 MB/node even though the chain graph transmits less data per epoch. Hence, one should avoid sparse reduce graphs with very low spectral gap values and use expander graphs that provide good convergence for a given network bandwidth.

To summarize, we find that stochastic reduce provides significant speedup in convergence (by 2-10X) and reduces the network and CPU costs. However, if the node communication graph is sparse and has low spectral gap values (usually less than 0.01), the convergence can be slow. Hence, stochastic reduce provides a quantifiable measure using the spectral gap values to sparsify the node communication graphs. For models where the network capacity and CPU costs can match the model costs, using a network that can support the largest spectral gap is recommended.

6.2. Fine-grained synchronization benefits

We compare the performance of `NOTIFY-ACK`, synchronous and asynchronous implementation. We implement the BSP algorithm using a `barrier` in the training loop, and all data-parallel models perform each iteration concurrently. However, simply using a `barrier` may not ensure consistency at the receive queue. For example, the models may invoke a `barrier` after sending the models and then perform a `reduce`. This does not guarantee that each machine receives *all* the intermediate outputs during reduce. Hence, we perform a consistency check on the received intermediate outputs and adjust the number of intermediate models to compute the model average correctly. Each parameter update carries a unique version number in the header and footer we verify the version values in the header and footer are identical before and after reading the data from the input buffers.

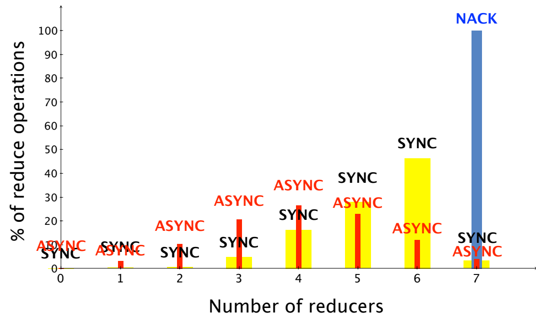


Figure 7: This bar graph shows the percentage of consistent `reduce` operations with `NOTIFY-ACK` vs BSP vs ASYNC for 8 machines for the RCV1 dataset. `NOTIFY-ACK` provides the strongest consistency allowing 100% of `reduce` operations to be performed with other 7 nodes.

For asynchronous processing, we perform no synchronization between the workers once they finish loading data and start training. We check the incoming intermediate model updates for consistency as described above. For the `NOTIFY-ACK` implementation, we send intermediate models with notifications and `wait` for the ACK before performing a `reduce`. Often communication libraries over network interconnects may not guarantee that the notifications arrive with the data, and we additionally check the incoming model updates and adjust the number of reducers to compute the model average correctly. We first perform a micro-benchmark to measure how consistency of `reduce` operations in synchronous (SYNC), asynchronous (ASYNC) and `NOTIFY-ACK`.

Figure 7 shows, for a graph of 8 nodes with each machine having an in-degree 7 , the distribution of correct buffers reduced. `NOTIFY-ACK`, reduces with all 7 inputs and is valid 100% of the time. BSP has substantial torn reads, and 77% of the time performs a `reduce` with 5 or more workers. ASYNC can only perform 39% of the `reduce` operations correctly with 5 or more workers. Hence, we find that `NOTIFY-ACK` provides the most consistent data during `reduce` and fewest torn buffers followed by BSP using a `barrier`, followed by ASYNC.

Figure 8 shows the convergence for the CIFAR-10 dataset for eight machines in an all-reduce setup. We calculate the time to reach 99% training accuracy on the VGG network which corresponds to an approximately 84% test accuracy. We train our network with a mini-batch size of 1, with no data augmentation and a network wide learning rate schedule that decays every epoch. We find that `NOTIFY-ACK` provides superior convergence over BSP and ASYNC. Even with a dense communication graph, we find that `NOTIFY-ACK` reduces `barrier` times and provides stronger consistency and faster convergence. Furthermore, we find that ASYNC ini-

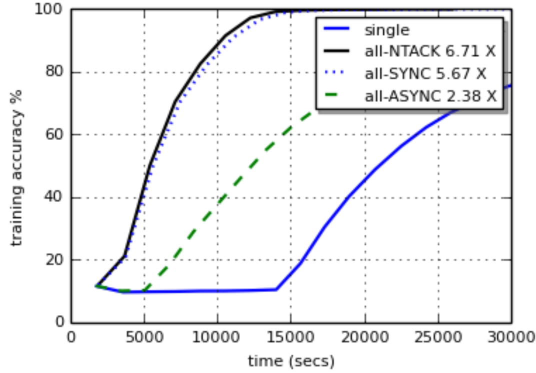


Figure 8: This figure shows the convergence of NOTIFY-ACK, BSP, ASYNC with eight machines using all-reduce for training CNNs over the CIFAR-10 dataset with a VGG network. Speedups are measured over a single machine implementation.

tializes slowly and converges slower than synchronous and NOTIFY-ACK methods.

We also measure the throughput (examples/second) for synchronous (BSP), asynchronous and the NOTIFY-ACK synchronous methods. NOTIFY-ACK, avoids coarse-grained synchronization and achieves an average throughput of 229.3 images per second for eight machines. This includes the time for forward and backward propagation, adjusting the weights and communicating and averaging the intermediate updates. With BSP, we achieve 217.8 fps. Finally, we find that even though ASYNC achieves the highest throughput of 246 fps, Figure 8 shows that the actual convergence is poor. Hence, to understand the benefits of approaches like relaxed consistency, one must consider speedup towards a (good) final accuracy. However, ASYNC may converge fastest when the datasets are highly redundant. ASYNC can also provide a speedup if the model updates are sparse, the reduce operation becomes commutative and reduces conflicting updates.

Figure 9 shows the convergence for the SVM application using the webspam dataset on 25 processes. We use the root-expander graph as described earlier.We find that using fine-grained NOTIFY-ACK improves the convergence performance and is about 3*X* faster than BSP for the webspam dataset. Furthermore, the asynchronous algorithm does not converge to the correct value even though it operates at a much higher throughput. NOTIFY-ACK provides good performance for three reasons. First, NOTIFY-ACK provides stronger consistency than BSP implemented using a barrier. In the absence of additional expensive synchronization with each sender, the model replicas may reduce with fewer incoming model updates. NOTIFY-ACK provides stronger guarantees since each worker waits for the NOTIFYs before performing the reduce and sends out additional data with after receiving the ACK messages. Second, for approximate processing i.e. when the communication graph is sparse, a barrier blocks all parallel workers while when using fine-grained communication with NOTIFY-ACK independent workers run asynchronously with respect to one another. To summarize, we find that NOTIFY-ACK eliminates

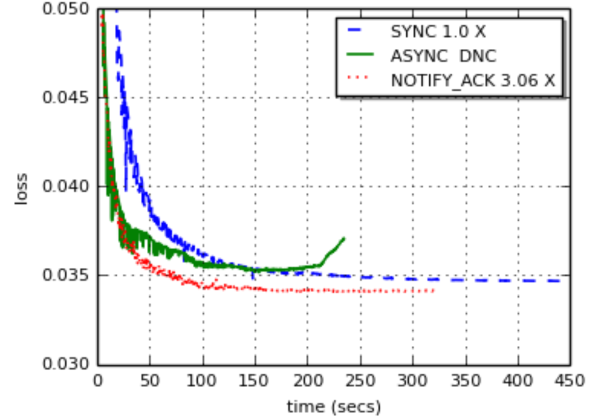


Figure 9: This figure shows the convergence of NOTIFY-ACK vs BSP and ASYNC for the root expander graph over the webspam dataset. The asynchronous implementation does not converge (DNC) to the final value.

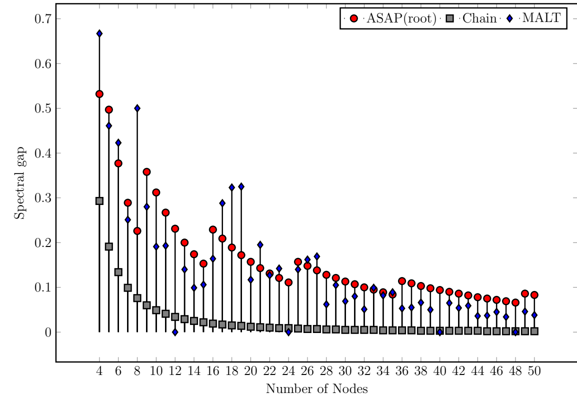


Figure 10: This figure compares the spectral gap values for expander graph (degree 2), chain graph (degree 1) and MALT (degree 2). Higher spectral gap values indicate faster convergence.

torn buffers and provides faster convergence over existing consistency mechanisms for dense as well as sparse node graphs. We describe our results for the BSP/all-reduce model. However, these results can also be extended to bi-directional communication architectures such as the parameter server or the butterfly architecture.

Comparision of expander graph with other architectures: Finally, we compare the convergence of expander graphs with a chain node communication and MALT Halton graphs [42] in Figure 10. We compute the spectral graphs for degree 2 for MALT and ASAP as well as for a chain (or ring graph). The spectral gap represents how fast these node communications graphs converge as the number of nodes (on x-axis) increases. We found that ASAP converges faster than MALT for most of network graphs and chain for every network graph. The ASAP expander graph of *N* nodes sends its update to next neighbor and the root(*N*) node. The MALT graph follows the Halton series to send its updates to another two nodes as computed by this series [2]. In some cases, this may create nodes that are partitioned into two groups, and spectral gap falls to zero, resulting in no convergence.

7. Related Work

Batch systems: The original map-reduce uses a stage barrier i.e. all mapper tasks synchronize intermediate outputs over a distributed file-system [26]. This provides synchronization and fault tolerance for the intermediate job state but can hurt performance due to frequent disk I/O. Spark [68] implements the map-reduce model in-memory using copy-on-write data-structures (RDDs). RDDs provide fault tolerance and enforce determinism preventing applications from running asynchronously. ASIP [32] adds asynchrony support in Spark by decoupling fault tolerance and coarse-grained synchronization. However, ASIP finds that asynchronous execution may lead to incorrect convergence, and presents the case for running asynchronous machine learning jobs using second order methods that provides stronger guarantees but can be extremely CPU intensive [54]. Finally, there are many existing general purpose dataflow based systems that use barriers or block on all inputs to arrive [21, 39, 48, 52]. ASAP uses fine-grained synchronization with partial reduce to mitigate communication and synchronization overheads. We use application level checkpoints for fault tolerance of intermediate states.

Approximate Processing: Recent work on partial aggregation uses efficient tree-style all-reduce primitive to mitigate communication costs in batch systems by combining results at various levels such as machine, rack etc. [12, 66]. However, the `reduce` operation suffers from additional latency proportional to the height of the tree. Furthermore, when partial aggregation is used with iterative-convergent algorithms, the workers wait for a significant aggregated latency time which can be undesirable. Other work on partial-aggregation produces variable accuracy intermediate outputs over different computational budgets [41, 58]. Other methods to reduce network costs include using lossy compression [6] or KKT filters [43] which computes the benefit of updates before sending them over the network. These methods can be applied with stochastic reduce even though they may incur additional CPU costs unlike stochastic reduce.

Asynchronous Processing: Past work has explored removing barriers in Hadoop to start reduce operations as soon as some of the mappers finish execution [31, 63]. HogWild [56] uses a single shared parameter vector and allows parallel threads to update model parameters without locks, thrashing one another's updates. However, HogWild may not converge to a correct final value if the parameter vector is dense and the updates from different threads overlap frequently. Project Adam [15], DogWild [51] and other systems that use HogWild in a distributed setting often cause wasteful communication especially when used to communicate dense parameter updates. Several systems propose removing the barriers which may provide faster throughput but may lead to slower or incorrect convergence towards the final optimization goal. To overcome this problem, bounded-staleness [22] provides asynchrony within bounds for the parameter server i.e. the fast running threads wait for the stragglers to catch up. However, determining these bounds empirically can be difficult, and in some cases they may not be more relaxed than synchronous. ASAP instead proposes using fine-grained synchrony that reduces synchronization overhead with strong consistency.

ML frameworks: Parameter Servers [25, 43] provide a master-worker style communication framework. Here, workers compute the parameter updates and send it to one or more central servers. The parameter servers compute and update the global model and sends it to the workers and they continue to train new data over this updated model. On the contrary, all-reduce based systems may operate fully asynchronously since unlike parameter server there is no consensus operation to exchange the gradients. ASAP reduces communication overheads in the all-reduce model and by proposing partial-reduce based on information dispersal properties of underlying nodes. TensorFlow runs a dataflow graph across a cluster and uses the asynchronous parameter server to train large models [6, 13]. For larger models, the large fanout of the master can be a bottleneck and the model parameters are aggregated at bandwidth hierarchies [13]. Using ASAP's stochastic reduce to improve the convergence behavior of such network architectures can reduce the wait times. The parameter server architecture has also been proposed over GPUs [23, 69] and the communication and synchronization costs can be reduced in these systems by using the ASAP model.

8. Conclusion

Practitioners often use approximation and asynchrony to accelerate job processing throughput (i.e. examples/second) in data parallel frameworks. However, these optimizations may not achieve a speedup over BSP to reach high accuracy output.

In this paper, we present stochastic reduce that provides tunable approximation based on available network bandwidth. We also introduce `NOTIFY-ACK` that provides fine-grained yet stronger consistency than BSP. In our results, we demonstrate that our model can achieve 2-10X speedups in convergence and up to 10X savings in network costs for distributed ML applications. Other optimization problems such as graph-processing face similar trade-offs, and can benefit from using the ASAP model. A github link of our data-parallel system including the code to compute spectral gap for different node topologies will be provided in the final version of our paper.

Finally, there are other sources of synchrony in the system that can be relaxed. For example, we find that loading data into memory consumes a significant portion of job times. For training tasks with significant CPU times, such as processing images through deep networks, having a separate worker that loads and sends data to various workers over low latency networks allows overlapping data loading with model training, and can remove the initial data load wait times.

References

[1] Apache Hama: Framework for Big Data Analytics. http://hama.apache.org/.

[2] Halton sequence. en.wikipedia.org/wiki/Halton_sequence.

[3] NVIDIA GPUDirect. https://developer.nvidia.com/gpudirect.

[4] PASCAL Large Scale Learning Challenge. http://largescale.ml.tu-berlin.de, 2009.

[5] The CIFAR-10 dataset. https://www.cs.toronto.edu/ kriz/cifar.html, 2009.

[6] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[7] Alekh Agarwal, Olivier Chapelle, Miroslav Dudík, and John Langford. A reliable effective terascale linear learning system. *JMLR*, 2014.

[8] Anonymous. Anonymized for review. In *Unknown*. XXX, XXX.

[9] Joshua Batson, Daniel A Spielman, Nikhil Srivastava, and Shang-Hua Teng. Spectral sparsification of graphs: theory and algorithms. *Communications of the ACM*, 56(8):87–94, 2013.

[10] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Springer COMPSTAT*, pages 177–187, Paris, France, 2010.

[11] John Canny and Huasha Zhao. Butterfly mixing: Accelerating incremental-update algorithms on clusters. In *SDM*, pages 785–793, 2013.

[12] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment*, 1(2):1265–1276, 2008.

[13] Jianmin Chen, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. Revisiting distributed synchronous sgd. *ICLR Workshop*, 2016.

[14] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[15] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project Adam: Building an Efficient and Scalable Deep Learning Training System. In *USENIX OSDI*, 2014.

[16] Cheng Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Andrew Y Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. *NIPS*, 19:281, 2007.

[17] Lingkun Chu, Hong Tang, Tao Yang, and Kai Shen. Optimizing data aggregation for cluster-based internet services. In *ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP) , 2003*. ACM.

[18] James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Gregory R Ganger, Garth Gibson, Kimberly Keeton, and Eric Xing. Solving the straggler problem with bounded staleness. In *USENIX HotOS*, 2013.

[19] Clarifai. Clarifai Visual Search. https://www.clarifai.com/visual-search.

[20] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, 2011.

[21] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *NSDI*, volume 10, page 20, 2010.

[22] Henggang Cui, James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Abhimanu Kumar, Jinliang Wei, Wei Dai, Gregory R Ganger, Phillip B Gibbons, et al. Exploiting bounded staleness to speed up big data analytics. In *USENIX ATC*, 2014.

[23] Henggang Cui, Hao Zhang, Gregory Ganger, Phillip Gibbons, and Eric Xing. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 2016.

[24] Wei Dai, Jinliang Wei, Xun Zheng, Jin Kyu Kim, Seunghak Lee, Junming Yin, Qirong Ho, and Eric P Xing. Petuum: A framework for iterative-convergent distributed ml. *arXiv preprint arXiv:1312.7651*, 2013.

[25] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. Large scale distributed deep networks. In *NIPS*, 2012.

[26] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[27] John C Duchi, Alekh Agarwal, and Martin J Wainwright. Dual averaging for distributed optimization: convergence analysis and network scaling. *Automatic control, IEEE Transactions on*, 57(3):592–606, 2012.

[28] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 97–104. Springer, 2004.

[29] Rainer Gemulla, Erik Nijkamp, Peter J Haas, and Yannis Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *ACM KDD*, pages 69–77, 2011.

[30] Amol Ghoting, Rajasekar Krishnamurthy, Edwin Pednault, Berthold Reinwald, Vikas Sindhwani, Shirish Tatikonda, Yuanyuan Tian, and Shivakumar Vaithyanathan. Systemml: Declarative machine learning on mapreduce. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 231–242. IEEE, 2011.

[31] Ínigo Goiri, Ricardo Bianchini, Santosh Nagarakatte, and Thu D Nguyen. Approxhadoop: Bringing approximations to mapreduce frameworks. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 383–397. ACM, 2015.

[32] Joseph E Gonzalez, Peter Bailis, Michael I Jordan, Michael J Franklin, Joseph M Hellerstein, Ali Ghodsi, and Ion Stoica. Asynchronous complex analytics in a distributed dataflow architecture. *arXiv preprint arXiv:1510.07092*, 2015.

[33] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *USENIX OSDI*, 2012.

[34] Google Cloud Platform. CLOUD SPEECH API. https://cloud.google.com/speech/.

[35] Google Cloud Platform. CLOUD VISION API. https://cloud.google.com/vision/.

[36] Douglas Gregor and Andrew Lumsdaine. The parallel BGL: A generic library for distributed graph computations. *Parallel Object-Oriented Scientific Computing (POOSC)*, 2:1–18, 2005.

[37] Zhenyu Guo, Xuepeng Fan, Rishan Chen, Jiaxing Zhang, Hucheng Zhou, Sean McDirmid, Chang Liu, Wei Lin, Jingren Zhou, and Lidong Zhou. Spotting code optimizations in data-parallel pipelines through periscope. In *USENIX OSDI*, 2012.

[38] Aaron Harlap, Alexey Tumanov, Andrew Chung, Gregory R Ganger, and Phillip B Gibbons. Proteus: agile ml elasticity through tiered reliability in dynamic resource markets. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 589–604. ACM, 2017.

[39] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM EuroSys*, 2007.

[40] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using RDMA efficiently for key-value services. In *SIGCOMM*, pages 295–306. ACM, 2014.

[41] Gautam Kumar, Ganesh Ananthanarayanan, Sylvia Ratnasamy, and Ion Stoica. Hold'em or fold'em? aggregation queries under performance variations. 2016.

[42] Hao Li, Asim Kadav, Erik Kruus, and Cristian Ungureanu. Malt: distributed data-parallelism for existing ml applications. In *Eurosys*. ACM, 2015.

[43] Mu Li, David Andersen, Alex Smola, Junwoo Park, Amr Ahmed, Vanja Josifovski, James Long, Eugene Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *USENIX OSDI*, 2014.

[44] Mu Li, David G Andersen, and Alexander Smola. Distributed delayed proximal gradient methods. In *NIPS Workshop on Optimization for Machine Learning*, 2013.

[45] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[46] Frank McSherry, Michael Isard, and Derek G Murray. Scalability! but at what cost? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.

[47] McSherry, Frank. Progress in graph processing: Synchronous vs asynchronous graph processing. https://github.com/frankmcsherry/blog/blob/master/posts/2015-12-24.md.

[48] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martin Abadi. Naiad: A timely dataflow system. In *ACM SOSP*, 2013.

[49] Angelia Nedic and Asuman Ozdaglar. Distributed subgradient methods for multi-agent optimization. *IEEE Transactions on Automatic Control*, pages 48–61, 2009.

[50] Edmund B Nightingale, Jeremy Elson, Jinliang Fan, Owen Hofmann, Jon Howell, and Yutaka Suzue. Flat datacenter storage. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 1–15, 2012.

[51] Cyprien Noel and Simon Osindero. Dogwild!—distributed hogwild for cpu & gpu. In *NIPS workshop on Distributed Machine Learning and Matrix Computations*, 2014.

[52] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

[53] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, Byung-Gon Chun, and VMware ICSI. Making sense of performance in data analytics frameworks. In *NSDI*, 2015.

[54] Hua Ouyang, Niao He, Long Tran, and Alexander Gray. Stochastic alternating direction method of multipliers. In *Proceedings of the 30th International Conference on Machine Learning*, pages 80–88, 2013.

[55] Russell Power and Jinyang Li. Piccolo: Building fast, distributed programs with partitioned tables. In *USENIX OSDI*, pages 293–306, 2010.

[56] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, 2011.

[57] Eugene Seneta. *Non-negative matrices and Markov chains*. Springer Science & Business Media, 2006.

[58] Ambuj Shatdal and Jeffrey F Naughton. Adaptive parallel aggregation algorithms. In *ACM SIGMOD Record*, volume 24, pages 104–114. ACM, 1995.

[59] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[60] Dinh Nguyen Tran, Bonan Min, Jinyang Li, and Lakshminarayanan Subramanian. Sybil-resilient online content voting. In *NSDI*, volume 9, pages 15–28, 2009.

[61] Asaf Valadarsky, Michael Dinitz, and Michael Schapira. Xpander: Unveiling the secrets of high-performance datacenters. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, page 16. ACM, 2015.

[62] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[63] Abhishek Verma, Brian Cho, Nicolas Zea, Indranil Gupta, and Roy H Campbell. Breaking the mapreduce stage barrier. *Cluster computing*, 16(1):191–206, 2013.

[64] Guozhang Wang, Wenlei Xie, Alan J Demers, and Johannes Gehrke. Asynchronous large-scale graph processing made easy. In *CIDR*, 2013.

[65] Pijika Watcharapichat, Victoria Lopez Morales, Raul Castro Fernandez, and Peter Pietzuch. Ako: Decentralised deep learning with partial gradient exchange. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pages 84–97. ACM, 2016.

[66] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *USENIX OSDI*, 2008.

[67] Hyokun Yun, Hsiang-Fu Yu, Cho-Jui Hsieh, SVN Vishwanathan, and Inderjit Dhillon. NOMAD: Non-locking, stOchastic Multi-machine algorithm for Asynchronous and Decentralized matrix completion. In *ACM VLDB*, 2014.

[68] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *USENIX NSDI*, 2012.

[69] Hao Zhang, Zhiting Hu, Jinliang Wei, Pengtao Xie, Gunhee Kim, Qirong Ho, and Eric Xing. Poseidon: A system architecture for efficient gpu-based deep learning on multiple machines. *arXiv preprint arXiv:1512.06216*, 2015.