Elastic Bulk Synchronous Parallel Model for Distributed Deep Learning

Xing Zhao*, Manos Papagelis*, Aijun An*, Bao Xin Chen*, Junfeng Liu†, Yonggang Hu†
*Department of Electrical Engineering and Computer Science, York University, Toronto, Canada
†Platform Computing, IBM Canada, Markham, Canada
{xingzhao, papaggel, aan, baoxchen}@eecs.yorku.ca;{jfliu, yhu}@ca.ibm.com

***Abstract*—The *bulk synchronous parallel* (BSP) is a celebrated *synchronization model* for general-purpose parallel computing that has successfully been employed for distributed training of machine learning models. A prevalent shortcoming of the BSP is that it requires workers to wait for the straggler at every iteration. To ameliorate this shortcoming of classic BSP, we propose ELASTICBSP a model that aims to relax its strict synchronization requirement. The proposed model offers more flexibility and adaptability during the training phase, without sacrificing on the accuracy of the trained model. We also propose an efficient method that materializes the model, named ZIPLINE. The algorithm is tunable and can effectively balance the trade-off between quality of convergence and iteration throughput, in order to accommodate different environments or applications. A thorough experimental evaluation demonstrates that our proposed ELASTICBSP model converges faster and to a higher accuracy than the classic BSP. It also achieves comparable (if not higher) accuracy than the other sensible synchronization models.**

***Index Terms*—Distributed deep learning, parameter server framework, GPU cluster, data parallelism, BSP, SSP, ASP**

I. INTRODUCTION

The *parameter server framework* [1] [2] has been widely adopted to distributing the training of large deep neural network (DNN) models [3] [4]. The framework consists of multiple *workers* and a logical *server* that maintains globally shared parameters, typically represented as dense or sparse vectors and matrices [5], and it supports two approaches: *model parallelism* and *data parallelism* [6]. In this paper we focus on data parallelism. Data parallelism refers to partitioning (sharding) of large training data into smaller equal size shards and assigning them to workers. Then, the entire DNN model is replicated to each worker. During the training, each worker trains the replica model using its assigned data shard, sends the locally computed gradients (via `push` operation) to the server that maintains globally shared parameters (weights) and receives back updated global weights from the server (via `pull` operation). That *weight synchronization step* is critical as it provides to the server a means of controlling the iteration throughput (to boost the *convergence speed* in wall-clock time) and the quality of convergence (i.e., the *accuracy*).

Due to its importance a number of *synchronization models* have been proposed, the most important of which are the *asynchronous parallel* (ASP), the *bulk synchronous parallel* (BSP), and the *stale synchronous parallel* (SSP). ASP [1] is the simplest model as it assumes no weight synchronization

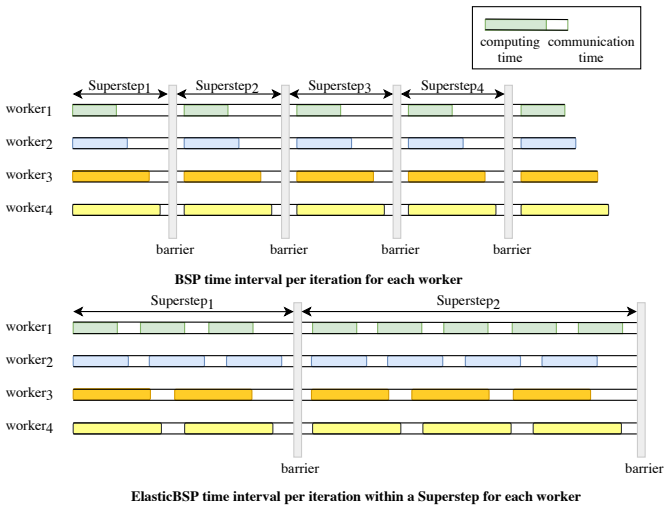


Fig. 1. Vanilla BSP and our proposed ELASTICBSP. Each *barrier* represents the time of weight synchronization among workers and a *superstep* represents the time between barriers. In BSP the superstep is fixed to a number of k iterations and all workers have to wait for each other at the end of their k iterations ($k = 1$ is shown, which is typical). In ELASTICBSP, the time the barrier is imposed varies and each superstep can allow a different number of iterations per worker. These values are determined at runtime by our proposed ZIPLINE method that achieves minimum overall waiting time of all workers.

— workers always receive different versions of weights from the server at every iteration. BSP [7] is the most celebrated synchronization model. A critical component of it is the *barrier synchronization*, where workers reaching a *barrier* have to wait until all other workers have reached it, as well (see Figure 1). During the training phase of a DNN model, each worker, at each iteration, computes the model gradients based on the local data shard and the local weights (originally from the server) and sends the gradients to the server. The server aggregates the gradients of all workers, performs weight update (as one synchronization) and signals the workers to retrieve the latest weights for the next iteration. The workers replace their local weights with the latest weights from the server and start a new iteration. SSP [2] provides an intermediate approach to the two extremes achieved by the ASP and the BSP models. It performs synchronization, but mitigates the strict synchronization requirement of BSP. In principle, it monitors the iteration difference between the fastest and the slowest workers and restricts it to be within a threshold via enforcing synchronization on both workers upon the excess of the threshold.

The aforementioned models exhibit certain limitations. In ASP there is no need for synchronization, so the waiting time overhead of the workers is eliminated. However, the convergence in the training might be dramatically affected due to inconsistent weight updates. On the other hand, a prevalent shortcoming of the BSP is the strict synchronization requirement it imposes. As shown in Figure 1, all workers are waiting for each other by a synchronization barrier. Each *barrier* represents the time of the weight synchronization among workers and a *superstep* represents the time between subsequent barriers. In BSP-like models the superstep is fixed to a number of k iterations and all workers have to wait for the straggler at the end of their k iterations ($k = 1$ is typical), such as in [8]. In SSP, while the strict synchronization requirement of BSP is removed, there is still a requirement to manually set the threshold that controls the iteration difference among workers, which remains fixed throughout the training period. Further, SSP does not consider the computational capacity of each worker but merely count on the number of iterations of each worker.

To ameliorate the shortcoming of current synchronization models, we propose ELASTICBSP, a model that aims to relax the strict synchronization requirement of the classic BSP for better convergence. Contrary to SSP, the proposed model considers the computational capacity of workers, accordingly, offers more flexibility and adaptability during the training phase, without sacrificing on the accuracy of the trained model. The key idea of ELASTICBSP is that the time the barrier is imposed varies and each superstep can permit a different number of iterations per worker, offering *elasticity* (see Figure 1). We also propose an efficient method that materializes the model, named ZIPLINE. ZIPLINE consists of two phases. First, k future iteration intervals (timestamps) of each worker are predicted at run time based on their most recent intervals, assuming a stable environment. Then, a one-pass algorithm operates over the predicted intervals of all workers and performs a lookahead greedy algorithm to determine the next synchronization time (i.e., a time that the overall workers' waiting time overhead is minimized). The algorithm can effectively balance the trade-off between accuracy and convergence speed, in order to accommodate different environments or applications. The major contributions of this work are as follows:

- we propose ELASTICBSP, a novel synchronization model for scaling the training of distributed deep learning models. ELASTICBSP replaces the strict synchronization requirement of other BSP-like models with an online decision making about the best time to impose the next synchronization barrier. The model guarantees the convergence for a large number of iterations.
- we design and develop ZIPLINE, a one-pass algorithm that can efficiently materialize the ELASTICBSP model. ZIPLINE performs online optimization with lookahead to predict the next best synchronization time. It also outperforms sensible baselines.

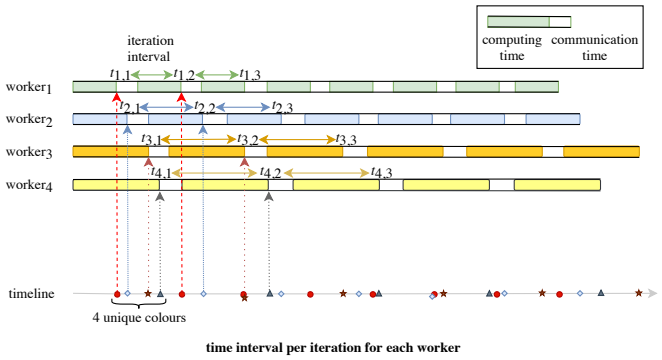


Fig. 2. Iteration intervals measured by timestamps of push requests from workers. A dotted line represents the time a push request arrives at the server from a worker. An iteration interval consists of gradient computing period (solid block) and communication period (blank block). All workers' ending timestamps can be mapped onto a timeline. Each timestamp on the timeline is associated to one of the workers. A set which is represented by the bracket always keep n unique values (colors) of workers. ZIPLINE scans the points from left to right on the timeline, takes one color point into the set per iteration.

- we present a thorough experimental evaluation of our ELASTICBSP model materialized by the ZIPLINE on two deep learning models on two popular image classification datasets. The results demonstrate that ELASTICBSP converges much faster than BSP and to a higher accuracy than BSP and other state-of-the-art alternatives.

The remainder of the paper is organized as follows. Section II introduces our proposed ELASTICBSP model and its properties. Section III formally defines the problem of interest. In Section IV, we present algorithmic details of sensible baselines and our proposed method ZIPLINE to materialize ELASTICBSP. Section V presents an experimental evaluation of the methods. We review the related work in Section VI and conclude in Section VII.

II. ELASTIC BULK SYNCHRONOUS PARALLEL MODEL

In this section, we propose a novel synchronization model that has the premise to ameliorate drawbacks of current models, without sacrificing their benefits.

The BSP model guarantees the convergence on training the DNN models since it is logically functioning as a single server. However, it introduces a large *waiting time overhead* due to having to wait for the slowest worker in every single iteration (a mini-batch). On the other hand, the ASP model does not perform any synchronization, so waiting time for synchronization is minimal, however, it is risky to be used due to its asynchronous scheme that renders the convergence uncertain [9]. The SSP model offers an intermediate solution to the above two extremes. It guarantees the convergence [2] when the number of iterations is large and the user specified threshold β is small. However, it depends on manually fine-tuning the β hyper-parameter which is non-trivial.

Motivated by the limitations of the current state-of-the-art synchronization models, we propose ELASTICBSP. ELASTICBSP aims to relax the strict synchronization requirement of BSP. The key properties of ELASTICBSP are the following:

- The server deals with sequential decision making regarding the *best* time that the next synchronization barrier

should be imposed (a time when the minimum waiting time for the entire system is achieved). The decision is based on a prediction model that utilizes information about the most recent time interval of each worker available to the server to predict its future intervals. The prediction is based on an online optimization with lookahead and assumes a specific limit R on how many future intervals for each worker should be considered. The need for a specific limit comes from the need to control the algorithm's run time, since that can increase exponentially as the lookahead limit R increases.

- The convergence guarantee of the model follows the theoretical analysis of SSP [2], where a small iteration difference β exists in some period τ (a superstep). In the case of ELASTICBSP, the iteration difference is bounded by the lookahead limit R in some period τ that is defined by the next best synchronization time. By the end of the period τ, the synchronization barrier is posed to all the workers where gradients aggregation is carried out on the server, similarly to BSP, the weights are synchronized.

ELASTICBSP offers elasticity in the sense that the distance between two consequent synchronization barriers is not fixed, but it is determined online. In addition, the waiting time is not determined by a fixed iteration difference between the fastest and the slowest workers (as in SSP), but based on the optimal time to synchronize in order to minimize the waiting time. Moreover, the synchronization time is always bounded within the lookahead limit R, so it will not simulate the ASP model.

III. PROBLEM FRAMEWORK

Most data centers follow the high availability criteria practise [10], it is realistic to assume that the cluster is running in a stable environment where each iteration time interval (including batch processing and gradient computing) of a worker is similar in a short period. If the worker is not responding in a reasonable time, it will be taken out from the distributed system (and the algorithm in our case). Note that our algorithm is orthogonal to the fault torrent problem. Then, we can heuristically predict the future iteration intervals for workers (see Figure 2) based on their most recent iterations.

The Problem. For n workers in a cluster, each worker p has to process many iterations in a training where each iteration time interval on the same worker p is similar. Each iteration interval is measured by the starting and the ending timestamps of processing an iteration. Suppose we predict R future iterations for each worker. For any worker p, it has a set S^p containing a list of starting and ending timestamps of iterations. Most of both timestamps are overlapped for the subsequent iterations. Thus, we only need to use the ending timestamps e^p_{i-1}, e^p_i to measure an iteration i. Mathematically we define the set $S^p = \{e^p_1, e^p_2, ..., e^p_R\}$ where $e^p_i, i \in [1, R]$ stands for an ending timestamp of worker p and $p \in [1, n]$. The set S^p contains R iterations of worker p. We need to find a set Z containing n ending timestamps, one from each set S^p, are closest to each other on the timeline. The maximum and minimum difference of these n ending timestamps is the waiting time for a synchronization. The smallest timestamp indicates the time-spot for the fastest worker starts waiting whereas the largest timestamp indicates the synchronization barrier to which all workers have to stop for the synchronization.

From each of these sets $S^p, p \in [1, n]$, we pick one element $e^p_j, j \in [1, R]$ to form a new set $Z = \{e^p_j\}, p \in [1, n]$. The difference between the maximum and the minimum numbers of the set Z is defined as $d_Z = \max(Z) - \min(Z)$. The slowest worker and the fastest worker finish their current iteration at time $\max(Z)$ and $\min(Z)$ respectively. d_Z is the waiting time of the fastest worker. Thus, d_Z dominates the overall waiting time for a synchronization since other workers' waiting time were overlapped by the fastest worker's. We are looking for the optimal set Z^* which gives the minimum d_{Z^*} from all possible combinations of Z. Hence, our objective function is:

$$Z^* = \underset{Z}{\operatorname{argmin}}\, d_Z$$

IV. METHODOLOGY

To solve the proposed problem, we first investigate the brute force approach. We analyze the naive brute force searching, *naive search* and develop an optimized version of brute force algorithm named *FullGridScan* since it is infeasible to implement the *naive search* as scaling the number of workers. We next introduce our approach ZIPLINE to bring down the computation complexity. Lastly, we show the computation and space complexity of the two approaches in Table I.

Naive search. In order to find the minimum difference d_{Z^*}, a straightforward approach is to use Brute Force. It first checks all possible combinations of selecting a single element from n sets where each set S has R elements. There are $(C^R_1)^n$ combinations. Second, it computes their d_Z values and finds the minimum value d_{Z^*} from all d_Z values. The set Z^* which yields the minimum value d_{Z^*} is the object we are looking for. The computation complexity of this approach is $\mathcal{O}(R^n)$. The space complexity is $\mathcal{O}(R^n)$ to hold the $(C^R_1)^n$ combinations.

GridScan. An optimized heuristic brute force algorithm (Algorithm 1) as a basis component for *FullGridScan*. We consider the predicted R iterations' timestamps for n workers form a $n \times R$ matrix $\mathcal{M}$ where each row of the matrix $\mathcal{M}_p$ represents a worker $p, p \in [1, n]$ and each row $\mathcal{M}_p$ has R predicted iteration points (timestamps) $\mathcal{M}_{p,i} = e^p_i, i \in [1, R]$ for worker p. *Designate* any point in $\mathcal{M}$, we can always find a point from other rows with the *shortest distance* to it. Let these closest points from other rows along with the *designated point* in set Z and we obtain d_Z. Accordingly, designate a row of points, we can find R sets of Zs associated to every point of the *designated row*. Finally, we can find the set Z^* from R sets of Zs with the minimum d_{Z^*}. To guarantee we do not miss any early point (on the timeline), we designate the row with the minimum (earliest) timestamp (i.e., $\mathcal{M}_{p,1}$) as the designated row to start the search which costs $\Theta(n)$. The total computation complexity is $\mathcal{O}(R^2 n)$. The outer loop over the points on the designated row costs R iterations and the inner loop over each points in $n-1$ rows (workers) constructs one combination Z of distinct p *value* points costs $(n-1) \cdot R$

Algorithm 1 GridScan - search the set Z^* with minimum d_{Z^*}

1: **procedure** MIN$_d$SET($\mathcal{M}$)
▷ the $n \times R$ Matrix $\mathcal{M}$ with predicted points
2: $Z^* \leftarrow \varnothing$
▷ the set Z^* takes n elements with unique worker id p, $p \in [1, n]$
3: $d_{Z^*} \leftarrow \infty$
4: find the row $\mathcal{M}_{p_b}$ with the smallest initial time $\mathcal{M}_{p_b,1}$ from set $\{\mathcal{M}_{p,1}\}$
▷ $\mathcal{M}_{p_b,1} = \min(\{\mathcal{M}_{p,1}\}), p \in [1, n]$, $\{\mathcal{M}_{p,1}\}$ the first column of $\mathcal{M}$
5: **for each** point $e \in$ worker $\mathcal{M}_{p_b}$ **do**
6: $Z \leftarrow \varnothing$
7: add e to Z
8: **for each** worker $\mathcal{M}_p \in \mathcal{M}, \mathcal{M}_p \neq \mathcal{M}_{p_b}$ **do**
9: **for each** point $\mathcal{M}_{p,i} \in \mathcal{M}_p$ **do**
10: $\mathcal{M}_{p,min} \leftarrow \text{argmin}_{\mathcal{M}_{p,i}} |\mathcal{M}_{p,i} - e|$
▷ the shortest distance point to e
11: add $\mathcal{M}_{p,min}$ to Z
12: $d_Z \leftarrow \max(Z) - \min(Z)$
13: **if** $d_Z < d_{Z^*}$ **then**
14: $Z^* \leftarrow Z$; $d_{Z^*} \leftarrow d_Z$
15: **return** Z^* ▷ the set with d_{Z^*}

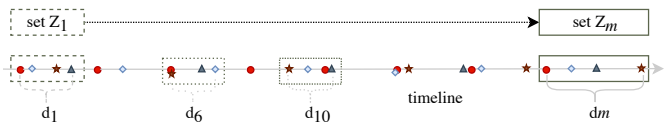
iterations as there is R points per row. During the search, we only need to keep the set Z^* with the minimum waiting time d_{Z^*} per point in the designated row which requires storage space $\theta(n)$. Along with the storage for Rn points, the space complexity is $\mathcal{O}(Rn)$.

FullGridScan. In *GridScan*, R combinations (of Z) are constructed and each of which waiting time d_Z is computed. We expect some critical combinations (containing the smaller waiting time d_Z) may be missed. In order to cover more useful combinations during the search, *FullGridScan* rotates the designated row of *GridScan* in turn to repeat Algorithm 1 without the line 4 till all n rows (workers) in $\mathcal{M}$ are covered. It rapidly increases the computation complexity to $\mathcal{O}(R^2n^2)$. *FullGridScan* therefore covers Rn combinations in total versus R combinations explored in *GridScan*. The storage complexity however remains the same as *GridScan*.

ZipLine. ZIPLINE scans through the data points only once in linear complexity $\Theta(Rn)$ as shown in Figure 3. In ZIPLINE (Algorithm 2), we first merge all n sets into one large set Ω and sort its elements in ascending order by their value e_i^p (ending timestamps) where $i \in [1, R]$ and $p \in [1, n]$. We consider the elements are sorted from left to right in position of the set Ω. Second, we define a set Z with the constraint that it contains one timestamp from each worker p at any time as we will use Z to scan every element of Ω following the timeline from left to right. Intuitively, the set Z only checks the superscript value p of each element e_i^p to prevent duplication of the same worker p. If the new timestamp from worker p is added, the old (*duplicate*) timestamp of worker p in Z is removed. Third, we let the set Z scan the set Ω by iterating one element from Ω at a time. At the beginning of the scanning procedure, we initialize Z by filling elements from the very left of Ω to Z while satisfying its constraint till Z has n timestamps from n workers. Then, we compute the minimum and maximum difference (i.e., waiting time) d_Z of Z based on the element value e_i^p. Assuming Z^* is Z at the initialization, we store Z

Algorithm 2 ZipLine - search the set Z^* with minimum d_{Z^*}

1: **procedure** MIN$_d$SET(Ω) ▷ the merged set Ω
2: $Z \leftarrow \varnothing$
▷ the set Z takes n elements with unique p value, $p \in [1, n]$
3: $\Omega \leftarrow$ sort(Ω)
▷ sort Ω in ascending order by element's value (timestamp)
4: **while** $|Z| < n$ **do**
5: $\omega \leftarrow$ very left element of Ω ▷ ω is e_i^p where $i \in [1, R]$
6: add ω to Z
▷ old element of Z is removed if it has the same p value as ω
7: $\Omega \leftarrow \Omega - \omega$
8: $d_Z \leftarrow \max(Z) - \min(Z)$
9: $Z^* \leftarrow Z$; $d_{Z^*} \leftarrow d_Z$
10: **while** $\Omega \neq \varnothing$ **do**▷ the solution is obtained when Ω is empty
11: $Z \leftarrow Z - \min(Z)$ ▷ Z is in ascending order as of Ω
12: **while** $|Z| < n$ **do**
13: $\omega \leftarrow$ very left element of Ω
14: add ω to Z
15: $\Omega \leftarrow \Omega - \omega$
16: $d_Z \leftarrow \max(Z) - \min(Z)$
17: **if** $d_Z < d_{Z^*}$ **then**
18: $Z^* \leftarrow Z$; $d_{Z^*} \leftarrow d_Z$
19: **return** Z^* ▷ the set with d_{Z^*}



ZipLine searching for the set Z* with the minimum difference d*

Fig. 3. The set Z zips from left to right on the timeline one data point at a time. When Z has n distinct elements, d_Z, the difference of minimum and maximum elements of Z is computed. When the set Z reaches to the end of the time line, the minimum d_Z is attained. If multiple minimum d_Zs are found, the first minimum d_Z is selected. In the above case, d_6 and d_{10} have the same minimum value — d_6 is chosen.

TABLE I
SUMMARY OF COMPUTATION AND SPACE COMPLEXITIES.

Algorithm	Computation	Space
GridScan (heuristic)	$\mathcal{O}(R^2n)$	$\mathcal{O}(Rn)$
FullGridScan	$\mathcal{O}(R^2n^2)$	$\mathcal{O}(Rn)$
Zipline	$\mathcal{O}(Rn^2)$	$\mathcal{O}(Rn)$

to Z^* and d_Z to d_{Z^*}. Next, we add one element from the left of Ω to Z per iteration till Ω is empty. In each iteration, we compute d_Z and compare its value with d_{Z^*}. If d_Z is smaller than d_{Z^*}, we store Z to Z^* and d_Z to d_{Z^*}. After Rn iterations, we attain the optimal set Z^*. The algorithm only uses $\Theta(n)$ space to store Z^*. In each iteration, we also iterate through the set Z to remove the *duplicate* element as the new one is added. This operation maintains the invariant (constraint) of Z and costs $\Theta(n)$. Therefore, the total computation complexity is $\mathcal{O}(Rn^2)$ and the space complexity is $\mathcal{O}(Rn)$ for storing Ω.

V. EXPERIMENTAL EVALUATION

In this section, we run experiments that aim to evaluate:

A. The runtime performance of ZIPLINE to the FullGridScan baseline algorithm. The scalability of ZIPLINE as a function of the number of workers and the parameter R.
B. The performance of ELASTICBSP compared to the classic BSP and other state-of-the-art synchronization models. Which one converges faster and to a higher accuracy? Which one reaches to a fixed number of epochs faster?

TABLE II
COMPUTATION TIME OF ALGORITHMS IN MICROSECONDS/μs.

Algorithm	10 Workers		100 Workers		1000 Workers	
	R=15	R=150	R=15	R=150	R=15	R=150
ZipLine	**1.49e2**	**1.32e3**	**6.37e3**	**4.99e4**	**2.53e5**	**2.38e6**
FullGridScan	1.54e3	4.67e4	8.13e4	2.15e6	4.04e6	2.07e8
GridScan	1.68e2	5.50e3	***1.11e3***	***4.38e4***	***7.45e3***	***2.57e5***

Dataset: We generate the datasets based on realistic scenarios to evaluate the performance of algorithms. Table II lists the different scales of configurations of datasets for the evaluation.

Environment: The overhead experiments of ZIPLINE and baseline algorithms are running on a server with 24x Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz and 64GB ram.

A. ZIPLINE *Performance Comparison*

In Table II, we evaluate the algorithms with 15 predicted iterations for each worker. We use 150 predicted iterations to evaluate the scalability of the algorithms to R. The computation time cost of each algorithm is the average of 10 trials.

The combinations of elements from Matrix $\mathcal{M} : n \times R$ increases in exponential as the number of workers n scales or in polynomial as predicted iterations R increments since the combinations is $(C_1^R)^n$ which we described in section IV. Table II shows that as the number of workers increases the computation time of *FullGridScan* increases much faster than ZIPLINE. For a fixed number of workers, when the number of predicted iterations per worker increases, the computation time of *FullGridScan* increases much faster than others. *GridScan* can be an alternative when the heuristic result is acceptable and the number of workers is larger than 10.

B. Distributed Deep Learning using ELASTICBSP

We compare the performance of ELASTICBSP with BSP, SSP and ASP by training DNN models from scratch under each of them on a distributed environment. We set a small threshold s=3 for SSP to ensure the convergence and achieve higher accuracy [2]. For ELASTICBSP, we set R, the number of predicted future iterations per worker, to 15, 30, 60, 120 and 240 respectively. We ran each experiment three trails and chose the medium result based on the test accuracy.

Environment: We implement ELASTICBSP into MXNet [3] which supports BSP and ASP models. The experiments are running on 4 IBM POWER8 machines. Each machine has 4 NVIDIA P100 GPUs, 512 GB ram and 2×10 cores.

Datasets & DNN models: We train downsized AlexNet [11], ResNet-50 [12] on datasets CIFAR-10 and CIFAR-100 [13].

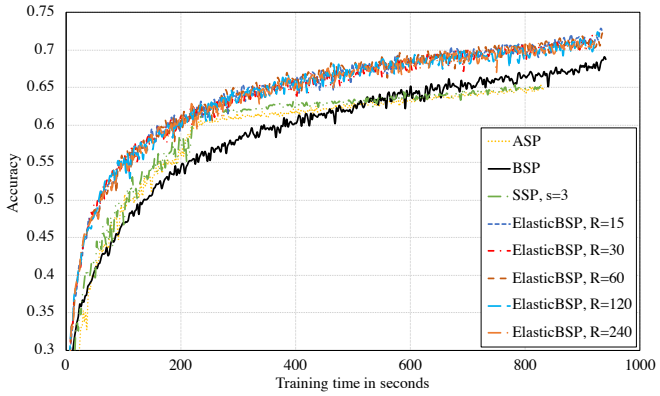
1) Downsized AlexNet: We set mini-batch size to 128, epoch to 400, learning rate 0.001 and weight decay 0.0005. ELASTICBSP converges faster and to a higher accuracy than other distributed paradigms (see Figure 4(a)). BSP converges slower than ASP and SSP but reaches to higher accuracy than both. The increase of R introduces more predicted elements to be computed by ZipLine to determine the optimal synchronization time, therefore, increases the computation overhead. As a result, when R becomes larger, it offers nothing but consumes more training time. To this model training, R=240 costs extra training time to finish 400 epochs compared to the smaller values. On this model, SSP, ASP, ELASTICBSP (R=15,30) and BSP complete the fixed 400 epochs in ascending order.

2) ResNet-50: We set mini-batch size to 128, epoch to 300, learning rate 0.5 and decay 0.1 at epoch 200. The results are shown in Figure 4(b). ELASTICBSP converges faster and to a slightly higher accuracy than BSP. Although ASP and SSP converge faster than ELASTICBSP and BSP, both cost much more training time to complete 300 epochs. Besides, ELASTICBSP converges to a slightly higher accuracy than ASP and SSP. ASP and SSP have no bulk synchronization barriers thus have more iteration throughput causing faster convergence. But larger iteration throughput introduces more frequent communications between workers and server and so increases the number of weight updates. However, weight update has to be computed in sequence (as mentioned in Section I). Thus, their tasks are queued on the server which introduces extra delay. A thorough discussion on why ASP and SSP converge faster but take more training time than BSP can be read in [14]. On this model, ELASTICBSP, BSP, SSP and ASP complete the fixed 300 epochs in ascending order.

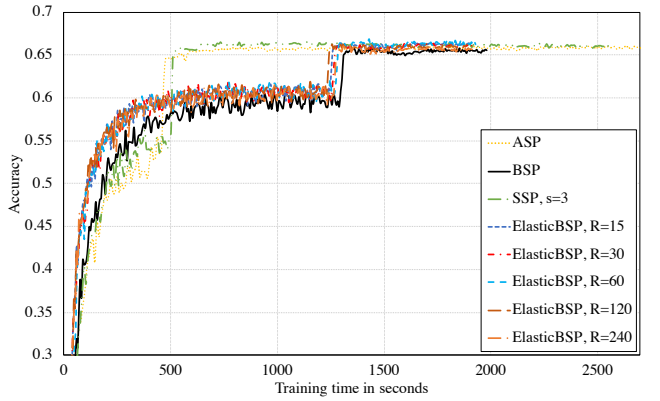
Discussion: Above DNN models show that ELASTICBSP converges to higher accuracy than BSP and takes less training time when R is not too large. Note the different performances of ELASTICBSP on the two DNN models are expected since AlexNet contains 2 fully connected layers whereas ResNets has no fully connected layers. Fully connected layers require much less computation time compared to convolutional layers while their representation requires much more parameters than convolutional layers which leads to a large model size. Convolutional networks without fully connected layers such as ResNets takes much more computing time but consumes less communication time due to its smaller model size as to fully connected layer networks. When the ratio of communication time and computation time is small, there is less training time can be saved. More detailed analysis of the different behavior on DNN models with different ratio of computation time and communication time can be read in [8]. [14] also provides detailed rationality on the different performances of distributed training using ASP, BSP and SSP on different DNN models.

VI. RELATED WORK

A number of important works closely related to our research has already been cited throughout the manuscript. Here, we elaborate on three alternative models that have been proposed to mitigate the slow down caused by the straggler problem of the classic BSP. A-BSP [15] handles the straggler problem by terminating the iteration job corresponding to the slowest worker once the fastest workers have completed their jobs. That way, the waiting time is eliminated. The remaining data of the terminated job of the slowest worker is prioritized in the next iteration. This design is limited to the CPU cluster where samples are processed one after another. But in a GPU cluster, a batch of samples are processed all at once in parallel; GPU takes a batch of samples per iteration and computes the gradients. Decreasing the data of a batch (iteration) does



(a) Downsized AlexNet on CIFAR-10 dataset



(b) ResNet-50 on CIFAR-100 dataset

Fig. 4. Comparison of synchronization models ($n = 4$)

not reduce the computation time of GPU. Furthermore, GPU does not support preempt [16]. Terminating the job (iteration) means losing all the computed result on that batch of data. Chen et al. [17] deal with the straggler problem by adding k extra backup workers to the distributed training with n workers. In this approach, $k + n$ workers are running for the model training. For each iteration, the server only accepts the first n randomly arrived gradient updates from the n faster workers and moves on to the next iteration. The gradients from the k slower workers are dropped. It does save on waiting time of the faster workers but the computing resources of the k slower workers in random iterations are wasted during the training. ADACOMM [8] uses periodic-averaging SGD (PASGD) for bulk synchronization in which workers are doing local updates for τ iterations before a weight synchronization. That way, the communication time of both uploading gradients and downloading weights from the server per iteration is saved for $\tau - 1$ iterations. The straggler problem is not addressed in this work. ADACOMM estimates the optimal τ for a bulk synchronization of local weights based on the training loss. Our ELASTICBSP predicts the optimal synchronization time for all workers where each worker has different τ as opposed to in ADACOMM τ is uniformly assigned to all workers.

VII. CONCLUSION

In this paper, we proposed ELASTICBSP for distributed DNN model training using the parameter server framework. ELASTICBSP is relaxing the bulk synchronization requirement of classic BSP and allows asynchronous gradient updates to a certain extent to ensure the quality of convergence and achieve higher accuracy. As a result, it increases the iteration throughput of the workers. ELASTICBSP operates in two phases per weight synchronization; first future R iterations for each worker are predicted. Then, ZIPLINE is applied to determine the optimal next synchronization barrier that minimizes the overall workers' waiting time overhead. ZIPLINE is a greedy one-pass algorithm and adds a minimal overhead on the server, so it can be easily ported in popular distributed machine learning frameworks. The experimental results show that ELASTICBSP provides faster convergence than classic BSP and achieves higher (or comparable) accuracy on the test data sets than other state-of-the-art synchronization models.

ACKNOWLEDGEMENT

This work is funded by the Natural Sciences and Engineering Research Council of Canada (NSERC), IBM Canada and the Big Data Research Analytics and Information Network (BRAIN) Alliance established by Ontario Research Fund - Research Excellence Program (ORF-RE).

REFERENCES

[1] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le *et al.*, "Large scale distributed deep networks," in *NIPS*, 2012, pp. 1223–1231.

[2] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing, "More effective distributed ml via a stale synchronous parallel parameter server," in *NIPS*, 2013, pp. 1223–1231.

[3] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao *et al.*, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[4] H. Zhang, Z. Zheng, S. Xu, W. Dai, Q. Ho, X. Liang *et al.*, "Poseidon: An efficient communication architecture for distributed deep learning on {GPU} clusters," in *USENIX*, 2017, pp. 181–193.

[5] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *OSDI*, 2014, pp. 583–598.

[6] X.-W. Chen and X. Lin, "Big data deep learning: challenges and perspectives," *IEEE access*, vol. 2, pp. 514–525, 2014.

[7] A. V. Gerbessiotis and L. G. Valiant, "Direct bulk-synchronous parallel algorithms," *Par. and Distr. Comput.*, vol. 22, no. 2, pp. 251–267, 1994.

[8] J. Wang and G. Joshi, "Adaptive communication strategies to achieve the best error-runtime trade-off in local-update sgd," *SysML Conf*, 2019.

[9] Z. Zhou, P. Mertikopoulos, N. Bambos, P. W. Glynn, Y. Ye, L.-J. Li, and F.-F. Li, "Distributed asynchronous optimization with unbounded delays: How slow can you go?" in *ICML*, 2018, pp. 1–10.

[10] K. Benz and T. Bohnert, "Dependability modeling framework: A test procedure for high availability in cloud operating systems," in *VTC*. IEEE, 2013, pp. 1–8.

[11] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *NIPS*, '12, pp. 1097–1105.

[12] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *IEEE CVPR*, 2016, pp. 770–778.

[13] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," Citeseer, Tech. Rep., 2009.

[14] X. Zhao, A. An, J. Liu, and B. X. Chen, "Dynamic stale synchronous parallel distributed training for deep learning," in *ICDCS*, 2019, pp. 1507–1517.

[15] S. Wang, W. Chen, A. Pi, and X. Zhou, "Aggressive synchronization with partial processing for iterative ml jobs on clusters," in *Proceedings of the 19th Int. Middleware Conference*. ACM, 2018, pp. 253–265.

[16] M. Bauer, H. Cook, and B. Khailany, "Cudadma: optimizing gpu memory bandwidth via warp specialization," in *SC*. ACM, 2011, p. 12.

[17] J. Chen, X. Pan, R. Monga, S. Bengio, and R. Jozefowicz, "Revisiting distributed synchronous sgd," *arXiv preprint arXiv:1604.00981*, 2016.