Sync-on-the-fly: A Parallel Framework for Gradient Descent Algorithms on Transient Resources

Guoyi Zhao, Lixin Gao and David Irwin
Dept. of Electrical and Computer Engineering
University of Massachusetts Amherst
guoyi@umass.edu, lgao@umass.edu and irwin@umass.edu

***Abstract*—Many cloud service providers offer transient resources (e.g., spare servers) for a fraction of the cost of on-demand servers. The iterative computations for big data analytic tasks are ideal to run on transient resources. However, the modern distributed data processing systems such as MapReduce or Spark provide little support for running iterative computation on transient resources. The fault-tolerant mechanism provided in MapReduce or Spark typically leads to cascading re-computations after revocations on transiently available resources. In this paper, we propose a distributed framework, Sync-on-the-fly, that takes advantage of the fact that many machine learning algorithms do not require fixed synchronization barriers. These synchronization barriers can be established at any time, and under the situation that some of the workers are revoked. We use the widely used gradient descent algorithms as examples to illustrate the design of Sync-on-the-fly. We implement several popular gradient descent algorithms, Logistic Regression and Matrix Factorization, under Sync-on-the-fly. Our evaluation shows that Sync-on-the-fly can achieve up to 5x speedup over Spark and reduce 85% of the costs.**

***Index Terms*—flexible synchronous parallel, transient resources, machine learning, gradient descent, distributed iterative computation**

I. Introduction

Many cloud service providers offer transient resources such as spare servers only 10% to 20% of the cost of on-demand servers. Users and companies can reduce the cost by using transient servers to run their machine learning algorithms in big data analytics. Where these big data analysis typically requires a large amount of resources to process the data and might need several iterations of processing.

The cheap price does not come for free, the transient resources may be revoked at any time [7] since the demand might fluctuate. Traditional distributed data processing systems such as Hadoop [6], Spark [12], and Pregel [4] are designed to run data analytic jobs on on-demand servers. These systems can tolerate failures as rare events, but not tolerant frequent revocations. They might suffer the excessive cost of cascading re-computations in case of revocations on transiently available resources.

Since most machine learning algorithms are not latency critical, it is a natural fit for the transient resources. Some recent works show the great potential to use transient resources in the big data analysis. Flint [5] and TR-Spark [9] leverage additional nodes of on-demand resources as storages to checkpoint intermediate results. After a revocation, computations can be resumed from the last checkpointed data. Pado [10] use the additional reserved resources to selectively run the computations that are most likely to cause high recomputation costs once revoked. Although such systems introduced various techniques to decide the optimal frequency of checkpointing, it can still be quite expensive for data-intensive workloads. It requires large amounts of data to be transferred back and forth which incurs substantial network and disk overhead.

In addition to the overhead of checkpointing, transient resources make it hard to perform synchronization. When a revocation occurs to a worker, all other workers have to wait for the recovery before performing synchronization. This entails a large amount of resource waste. This is particularly expensive when a large cluster of servers involved in one data analytic job. Current distributed frameworks such as Spark or Pregel use the bulk synchronous parallel (BSP) model where all workers have to synchronize at a predefined barrier. Given the unpredicted nature of server revocations, it is challenging to deploy BSP-based distributed frameworks on transient resources for large-scale machine learning algorithms.

The recent parameter server framework [3] applies the asynchronous communication model to avoid block on computation. Although, the asynchronous model reduce the wait for revoked workers, the update of model parameter will suffer the issue of staleness. The parameters we update will no longer be the one we use, which may make the machine learning algorithms suffer great fluctuations in convergence. In the recent stale synchronous parallel model (SSP) [1], they reduced the impact of the staleness in updating parameters by bounding the maximum staleness. However, a synchronous model do not even have a staleness issue. In flexible synchronous parallel (FSP) framework [8], they break the fixed synchronous barrier and apply the flexible synchronous parallel model in EM algorithms. The dynamic synchronous intervals speedup the computation while maintaining the convergence guarantee.

In this paper, we propose Sync-on-the-fly, a distributed framework for machine learning algorithms. It enables machine learning computations to establish synchronization barriers during runtime. Sync-on-the-fly exploits the fact that synchronization in machine learning algorithms does not have to be performed after a full pass of the data or a fixed set of data points. Synchronization barriers are established for building consistent model parameters, and thus can be performed at any time during the computation, even under

the situation that one or several servers are revoked and have not been recovered.

We design and implement the distributed framework using the gradient descent algorithms as examples. Sync-on-the-fly provides the capability to initiate synchronizations at runtime. When there is a revocation, Syn-on-the-fly ensures all remaining workers can continue to synchronization without waiting for the recovery of the revoked workers. We evaluate Sync-on-the-fly with several well-known machine learning algorithms, Logistic Regression and Non-negative Matrix Factorization, on a cluster of Amazon EC2 instances. We perform experiments on simulated revocation scenarios and revocation scenarios on the spot market of Amazon EC2. The results show that we can achieve up to 5x speedup over Spark and reduce 76%-85% of the cost from using the on-demand severs even under high revocation situation.

The remainder of this paper is organized as follows. Section II formally introduces the sync-on-the-fly programming model and suitable algorithms. Section III presents the design of the Sync-on-the-fly framework. Section IV reports extensive evaluation results. Section V finally concludes this work.

II. Sync-on-the-fly Programming Model

In this section, we first introduce the Sync-on-the-fly programming model for gradient descent algorithms. We then illustrate a series of machine learning problems that can be expressed under the Sync-on-the-fly model.

A. Sync-on-the-fly

Gradient Descent Algorithms are widely used in supervised machine learning. It considers the problem of minimizing an objective function that has the form of a sum as

$$Q(\vec{w}) = \sum_{i=1}^{N} Q_i(\vec{w}, x_i) \tag{1}$$

where $\vec{w}$ is the model parameter vector to be learned, and the function Q_i is associated with the i-th observation, x_i, from the input data set.

Starting with an initial guess of the model parameters, GD algorithms iteratively update model parameter with a gradient vector. The gradient vector is computed as a summation of the gradient at each data point.

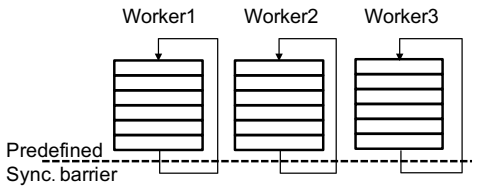
$$g(\vec{w}) = \frac{1}{|X|} \sum_{x_i \in X} \frac{\partial}{\partial \vec{w}} Q_i(\vec{w}, x_i) \tag{2}$$

Then the model parameter $\vec{w}$ can be updated with the gradient as follows.

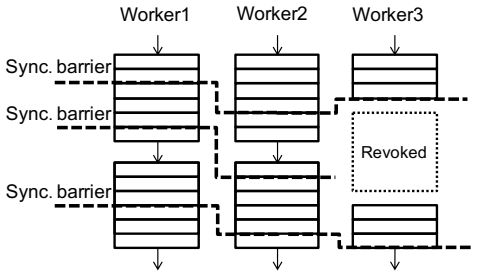
$$\vec{w} = \vec{w} - \eta g(\vec{w}) \tag{3}$$

where η is the learning rate.

To implement GD in a distributed environment, it is common to use the Bulk Synchronous Parallel (BSP) model. That is, the input data points are distributed among workers. Each worker computes the gradient from a mini-batch of the data points assigned to the worker. These gradients are aggregated together at a synchronization barrier. Each worker then updates the model parameter with the gradient and computes the gradient for the next iteration. We show an example of GD under the BSP model in Figure 1(a).



(a) BSP model



(b) Sync-on-the-fly in transient resources

Fig. 1: Illustration of the process order for BSP model (a) and Sync-on-the-fly model (b). The dashed lines indicate synchronous barriers. Each block represents one data point. The dashed box in (b) indicates that a revocation occurs on Worker3

GD under the BSP model has several drawbacks. First, it requires a full pass of the input data set before performing synchronization. In reality, it is possible to compute the gradient based on a subset of input data points, This is commonly referred to as mini-batch GD. Second, even under min-batch GD, the synchronization barrier is predefined. This is problematic since stragglers are common in practice [11]. Predefined synchronization barrier can result that fast workers wait for a straggler. Third, predefined synchronization barriers make it hard to take advantage of transient resources in the cloud. Transient resources can be revoked at any time. A revocation to one worker might lead to all other workers wait in the synchronization barrier.

In this paper, we propose Sync-on-the-fly for Gradient Descent algorithms. Under Sync-on-the-fly, synchronization can be performed at any time. At each synchronization point, the gradient from each worker is aggregated and broadcast to all workers. Then each worker updates the model parameter with the gradient and computes the gradient at each input data point until a synchronization barrier is established. This process repeats. Figure 1(b) shows GD under the Sync-on-the-fly model. When using transient resources, the Sync-on-the-fly model can greatly reduce the impact of the revoked workers.

For the convergence of GD algorithm, our proposed Sync-

on-the-fly model is equivalent to for a mini-batch SGD algorithm. Because in each synchronization, the gradients are computed from a set of data points among all workers. Consider the smallest set of data points from all the synchronizations as the mini-batch size, we can achieve the same convergence property as the mini-batch GD.

B. GD algorithms under Sync-on-the-fly

Formally, the gradient of the objective function at input data point i is computed as

$$g(\vec{w}, x_i) = \frac{\partial}{\partial \vec{w}} Q_i(\vec{w}, x_i) \tag{4}$$

The gradient in each worker will be aggregated as a *local gradient*. At a synchronization point, the *global gradient* is computing by summing up all the local gradients

$$\theta(\vec{w}) = \frac{1}{|B|} \sum_{x_i \in B} g(\vec{w}, x_i) \tag{5}$$

where B is the set of the immutable variables that contribute the gradient in this synchronization. The gradient is then used to update the model parameter as in Equation (3).

Many machine learning algorithms can be written in a form of GD under our Sync-on-the-fly model. For Logistic regression(LogR), which is a supervised machine learning algorithm that derived the relationship between input data points and outcomes. We model each data point $x_i \in X$ contains d attributes and one additional value y_i which indicates outcomes as a boolean variable. The model parameters w is derived to make the values from logistic function $P()$ in Equation $P(x_i) = \frac{1}{1+exp(-x_i^T w)}$ close to y_i for data point x_i. That is, we aim to derive w that minimizes the loss function $Q(w) = \sum_{x_i \in X} ((y_i - 1) * \log(1 - P(x_i)) - y_i * \log P(x_i))$.

To compute the gradient at each data point i, we have the jth dimension of the gradient as $g_j(w) = (P(x_{ij}) - y_{ij}) * x_{ij}$. Since computation of the gradient at each data point requires all dimensions of the model parameter, we will distribute the model parameter to all workers and consequently, the model parameter will be updated at each worker.

Take Nonnegative Matrix Factorization (NMF) as another example. NMF aims to factorize a given matrix $A \in \mathbb{R}_+^{m \times n}$ with observed entries into two nonnegative low-rank factor matrices $W \in \mathbb{R}_+^{m \times k}$ and $H \in \mathbb{R}_+^{k \times n}$, where $A \approx W \cdot H$ and the positive integer $k \ll \min\{m, n\}$. It minimize a loss function $L(A, W, H) = ||A - W \cdot H||_F^2$ based on Frobenius norm $|| \cdot ||_F$.

The matrix loss function L can be written as a series of independent functions for parallel optimization. Towards this end, let W_I denotes the I-th row of W, H_J denotes the J-th column of H, and $A_{I,J}$ denotes the entry of A at row I and column J. Then L can be expressed as $L(A, W, H) = \sum_J \sum_I (A_{I,J} - W_I H_J)$.

Under the Sync-on-the-fly model, W and H are two parameters to update. They can be updated alternatively by fixing one variable and update the other one. So every time we can use the latest version of the feature vector to compute the gradients. So for the gradient computation in Equation (4), we can rewritten specifically for W or H at each data point $A_{I,J}$ as $g_{W_I}(A_{I,J}, W) = (A_{I,J} - W_I H_J) H_J^T$ and $g_{H_J}(A_{I,J}, H) = W_I^T (A_{I,J} - W_I H_J)$.

Note here to figure out the gradient at the point $A_{I,J}$ for W requires only H_J and W_I. We therefore store W_I and H_J at the worker where $A_{I,J}$ resides. Therefore, the model parameter does not have to be stored (and updated) at all workers and rather at workers where their corresponding input data points reside.

C. Programming Interfaces

To program GD algorithms under Sync-on-the-fly model, we have the following APIs.

- *initialization()*: Initialize the model parameter $\vec{w}$.
- *partition(X, $\vec{w}$)*: Indicate the location of the workers where input data points and model parameters reside.
- *gradient(x_i, $\vec{w}_k$)*: Function for computing the gradient of objective function at x_i for dimension k of $\vec{w}$. It is possible to have several of these functions, each of which is for one model parameter.
- *scheduleUpdate($\vec{w}$)*: Indicate the order of updating model parameters. When computing gradient, we go through input data points in a round-robin fashion. This scheduler determines the order we update model parameters and the corresponding gradient computation function used.
- *progress(x_i)*: Function for computing objective function for data point x_i. This function is used for determining termination condition and when to synchronize.

III. System Design

A. System Overview

In the Sync-on-the-fly model, a global view of the worker status and progress is necessary to identify the revocation and determine a proper synchronous barrier. So we introduce a centralized *coordinator* in our system to keep track of the progress of the computation on each worker and monitor the health status of workers. As shown in Figure 2, the coordinator communicated with workers through signals. When enough progress has been made by active workers, it initiates a synchronization by broadcasting *Sync* signals to active workers. When a worker gets revoked, the revocation signal from the transient servers is passed to the coordinator through *WRevok* signal. The future synchronization will exclude that worker until it is recovered. When an available worker is found, the coordinator starts a recovery process. The coordinator and the worker communicate through *Recv* and *paraReq* signals to recover and catch up the computation.

The worker mainly takes charge of scheduling the computation of gradients and updating of the model parameter. The workers schedule the computation of gradients in a round-robin fashion in the computation module. When the interruption signals such as synchronization or revocation occur, the worker interrupt the update or computation. As the worker receives a new global gradient, it updates the model parameter and then continues the computation of gradients.

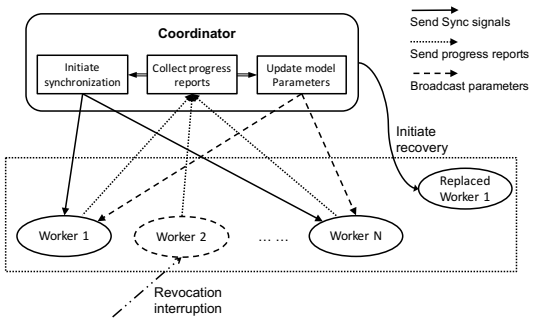


Fig. 2: Sync-on-the-fly system overview. The arrow lines indicate the gradients that are sent from the workers to the coordinator and the communication signals.

The coordinator monitors the status of workers and controls synchronization among active workers. It maintains a *worker status table* and keeps track of the progress of computation on each worker from the worker reports since the last synchronization. The *progress* quantifies the contributes to the convergence of objective function between two synchronizations. The coordinator accumulates the progress of each active worker and decides when to synchronize the workers. After a synchronization is initiated, it waits for all the local gradients from the active workers and aggregates them as global gradients. Then it distributes the global gradient based on the partition function of model parameter.

B. Synchroization on-the-fly

Frequent synchronizations make the model parameter keep being updated and increase the computation quality. However, the synchronization is not free. The more synchronization will decrease the quantity of computations per unit time. So identifying a proper synchronization point is extremely important for the efficiency of our system.

The observation of the convergence curve for the objective functions of GD algorithms shows that the progress from the first a few epochs are much larger than the later ones. The increasing updates in the first few epochs will accelerate the convergence speed for updating the model parameter. Because the updates will make use of the most updated value. Our intuitive is to frequently update the model parameter in the first a few epochs. We slow down the synchronization rate in the later epochs gradually based on the accumulated progress reports from workers. When some workers get revoked, we will need the active workers to make more progress rather than having a fixed synchronous barrier.

The coordinator aggregates the progress of the worker reports to $p^{(t)}$ at iteration t. The $p^{(t)}$ will keep increasing after more worker reports have received. The progress $p^{(t)}$ will be compared with the previous progress $p^{(t-1)}$. Since the value difference of the objective function is getting smaller and smaller, we predefined a percentage λ to reduce the progress needed for the later synchronization. When $p^{(t)} > \lambda p^{(t-1)}$, we assume enough progress has been made and we can initiate a new synchronous barrier. By default, λ is set to 0.8.

Generally, the progress score can reflect the data amount that contributes to the gradients. However, the update of the model parameter from a small set of the data points or outliers can make the objective function value fluctuate. An accumulation of the gradients can reduce the fluctuation by adding gradients from more data points. So as long as we reach a certain progress on the objective function, it worths a new synchronization.

Then the coordinator broadcasts the *sync* signal to all the active workers. Initially, the $p^{(0)}$ will be computed based on the first r reports from the workers, where r is the same as the unit of the data points for the progress report.

When the worker receives a *Sync* signal from the coordinator, it will first finish the computation of the gradient on the current data point as an atomic operation. Then it sends the local gradients to the coordinator and waits for the new global gradients. In the meantime, the local context will point to next data point. If the *Sync* signal arrives when the worker is still in a model parameter update stage, the worker will restart the model parameter update stage with the new global gradient.

C. Revocation handling

Revocation: When a worker gets revoked, it first receives a *Revok* signal from the transient server. The revocation requires an immediate response so that it can be handled within the advanced warning time. Therefore, the worker sends a *WRevok* signal to the coordinator first. If the worker is in the stage of local gradient computation, it interrupts the current computation immediately and sends the local gradients to the coordinator. Then it archives the local context which points to the current data point. When the worker is waiting for the next synchronization, it just archives the local context.

The coordinator is a key component in our system to control the synchronization. It requires an immediate recovery if the coordinator gets revoked to prevent the waiting for all the workers. Since the total progress from the last synchronization and worker status table are relatively small, we can easily archive them and select a worker to replace the revoked coordinator. But when the global gradients are large or the advanced warning time is very limited, it is better to place the coordinator in an steady servers to prevent possible revocation.

Recovery: A recovery process is initiated by the coordinator when an available worker is found. After the new worker receives a *WorkRecv* signal with the location of the input data and archived data, it reloads the input data and sets the local context as the archived one. Then it requests the current model parameter by sending a *paraReq* signal to the coordinator. Since the coordinator does not maintain the model parameter, it will send requests to active workers for the gradients.

If a recovery of coordinator is required, the selected worker first reloads the lists of workers and the historical progress scores. Then the coordinator notifies all the active workers with a *coordRevok* signal. So the workers can send their local gradients or resend the local gradients from the last

synchronization. After receiving all the local gradients, it resumes the function.

IV. Experiments

We evaluate Sync-on-the-fly on several algorithms to show the benefits of our dynamic synchronous barrier. By comparing with the state-of-art system, we illustrate the speedup by Sync-on-the-fly on handling different revocation situations using transient resources. We further calculate the expense we can save by using our system on transient resources in the real spot market.

A. Experiment Setup

We first describe our experiment environment, the data processing engines that we compare, the revocation scenarios and the measurement of the evaluation.

We conduct our experiments on an AWS EC2 cluster with two types of instances to test the performance and scalability of our algorithms. One type is EC2 m4.large instance which contains 2 CPU cores at 2.3-GHz and 8GB memory. Another one is EC2 t2.micro instance which contains 1 CPU core with high frequency Intel Xeon processors and 1GB memory. Each experiment is conducted with half the instances as m4.large and the other half are t2.micro. The free tier t2.micro instance is treated as a straggler.

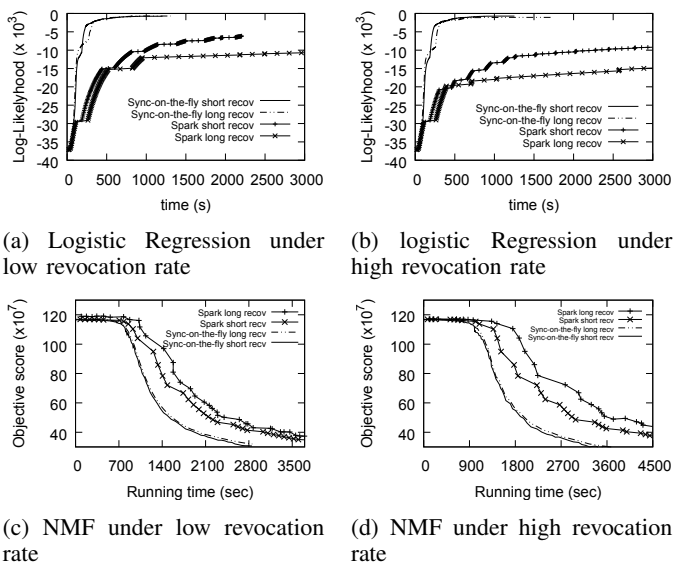
We test two representative applications, Logistic Regression (LogR), and NMF to explore the performance features of all tested frameworks. For regression, we tested on the YearPredictionMSD DataSet from UCI Machine Learning Repository which contains 0.5 million with 90 dimensions. The NMF test is performed under the user-movie matrix from the Netflix prize [2]. The matrix contains 480,190 rows and 17,770 columns with 100 million non-zero elements.

We simulate two revocation models to test low or high revocation ratio. The low rate one takes an average 10 minutes revocation time, while the high rate one takes an average 2 minutes revocation time. The restarting time for one worker is set as 3 minutes, which is built on the assumption that a spot instance with a higher bid price or other types will always available to us. And the restart of an instance takes roughly 2-3 minutes to deploy our system.

Two metrics are evaluated in experiments, *running time* and *objective function value*. Running time is defined as the elapsed time from the point when computation starts to the point when an algorithm converges. The overheads of loading data and dumping results are excluded since they are the same for all compared frameworks. Objective function value, i.e., $Q()$, has been given when introducing tested algorithms. In particular, $Q()$ needs to be minimized for Nonnegative Matrix Factorization and the log-likelihood of Logistic Regression need to be maximized.

B. Experiment Results

1) Performance Evaluation: We first evaluate the convergence acceleration (objective function value decrease) for the two example algorithms. We vary the revocation rate and the recovery time to test the performance. Considering the running time of each algorithms, we select the "short recov", shown in the Figure fig:converg, with 3 minutes to find a new replaced worker. While the "long recov" requires 6 minutes for a new worker.



(a) Logistic Regression under low revocation rate

(b) logistic Regression under high revocation rate

(c) NMF under low revocation rate

(d) NMF under high revocation rate

Fig. 3: Running time comparison between Sync-on-the-fly and Spark

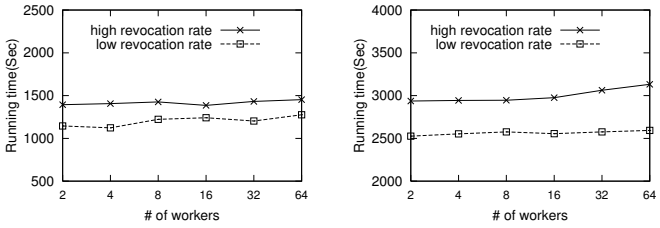
The results of Logistic Regression are shown in Figure Figure 3(a) and 3(b). In the Spark case, for an average of 5 to 6 epochs, there will be one revocation, which randomly revokes 1 to 8 workers. After the revocation, Spark reloads the RDD from the last checkpoint and recomputes the gradient. Under low revocation rate, the Sync-on-the-fly can improve 54% of the final convergence time towards the Spark implementation. It saved 85% of the running time to reach an objective function value around 5×10^7 to 8.5×10^7 in the middle of the computation.

In Figure 3(c) and 3(d), we show the results for NMF algorithm. In the first a few epoch of the run, the convergence is relatively slow because the initial feature vector is quite different from the final vector. To avoid the divergence of the algorithm, the step size is usually not very large. After around 5 to 10 epochs of one run, the convergence speed increases much more. Generally, the Sync-on-the-fly model can reach 23% and 39% improvement for low revocation rate and high revocation rate.

2) Scalability Evaluation: We further evaluate Sync-on-the-fly on the large-scale cluster to test its scalability. With the number of workers increases, the probability that a revocation occurs will increase in an exponential rate. With a low rate revocation, the estimate revocation of one worker occurs every 10 minutes. But with 64 workers, the expected time that at least one revocation occurs reduces to 9.4 seconds. So it is important to test the scalability of Sync-on-the-fly.

In the experiments, we scale the dataset with the number of

workers instead of using the same dataset for more workers. This is because using the same dataset, the short the running time, the more impact of the time from recovery when the number of workers increases. When the dataset scales at the same ratio of the workers, we can expect the same behavior of revocation. And the running time should be the same for the ideal case.



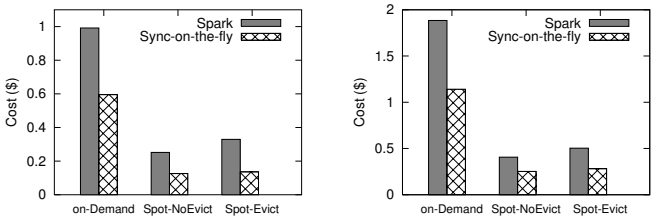
(a) Logistic Regression (b) NMF

Fig. 4: Scalability evaluation on Sync-on-the-fly

In Figure 5, we show the performance of Logistic Regression and NMF under sync-on-the-fly. We test the scalability for both low rate and high rate revocation. When the number of workers increases, we observe very good scalability results. The running time increase at most 10% compared to a two worker case.

3) Evaluation on Spot Market: To evaluate our framework in the real Amazon EC2 spot market, we test the performance of machine learning algorithms under different revocation scenarios. Since the revocation is affected by the fluctuation of the price which we are unable to control, we repeat each experiment several times and then category the results based on different recovery time after a revocation.

When no available workers can be found in a short time, the spark can not converge due to the lack of revoked data. But the Sync-on-the-fly can still finish the job with around 30%-50% more time than run without revocation.



(a) Logistic Regression (b) NMF

Fig. 5: Cost comparison between Sync-on-the-fly and Spark

We compare the cost for different experiment settings for Spark and Sync-on-the-fly using on-demand instance and spot instance in Figure 5. During the testing, the price for on-demand instance m4.large is $0.1, while the average hourly rate of spot price is US east zones are $0.0189 to $0.0244. So we will get 20% of the price to request a spot instance. From the experiments, we can reduce 76%-85% of the cost from using the on-demand instances even with high revocation rate.

V. Conclusion

This paper investigates the problem of machine learning algorithms using transient resources. We propose a sync-on-the-fly programming model to initiate flexible synchronous barriers afforded by many gradient descent algorithms. The model abstract the key components in gradient descent algorithms and enable flexible computation in gradients and update on model parameters. The framework enables a coordinator to actively synchronize workers when necessary and support frequent revocation and recovery. The Sync-on-the-fly can smoothly handle the revocation and reduce the recomputation time. The progress based on synchronization decision can further reduce the overhead of synchronizations. Extensive experiments show that the Sync-on-the-fly consistently outperforms the state-of-the-art solutions. In future work, we plan to investigate the influence of spot market price from the ordering more instances and apply the model to more machine learning algorithms.

Acknowledgment

This work is partially supported by NSF grants CNS-1815412 and CNS-1802523.

References

[1] 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 *Advances in neural information processing systems*, pages 1223–1231, 2013.

[2] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8), 2009.

[3] 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*, volume 14, pages 583–598, 2014.

[4] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[5] P. Sharma, T. Guo, X. He, D. Irwin, and P. Shenoy. Flint: batch-interactive data-intensive processing on transient servers. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 6. ACM, 2016.

[6] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*, pages 1–10. Ieee, 2010.

[7] R. Singh, P. Sharma, D. Irwin, P. Shenoy, and K. Ramakrishnan. Here today, gone tomorrow: Exploiting transient servers in datacenters. *IEEE Internet Computing*, 18(4):22–29, 2014.

[8] Z. Wang, L. Gao, Y. Gu, Y. Bao, and G. Yu. Fsp: towards flexible synchronous parallel framework for expectation-maximization based algorithms on cloud. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 1–14. ACM, 2017.

[9] Y. Yan, Y. Gao, Y. Chen, Z. Guo, B. Chen, and T. Moscibroda. Tr-spark: Transient computing for big data analytics. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pages 484–496. ACM, 2016.

[10] Y. Yang, G.-W. Kim, W. W. Song, Y. Lee, A. Chung, Z. Qian, B. Cho, and B.-G. Chun. Pado: A data processing engine for harnessing transient resources in datacenters. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 575–588. ACM, 2017.

[11] J. Yin, Y. Zhang, and L. Gao. Accelerating expectation-maximization algorithms with frequent updates. In *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*, pages 275–283. IEEE, 2012.

[12] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.