Aggressive Synchronization with Partial Processing for Iterative ML Jobs on Clusters

Shaoqi Wang, Wei Chen, Aidi Pi, Xiaobo Zhou
University of Colorado, Colorado Springs
Colorado Springs, Colorado
{swang,cwei,epi,xzhou}@uccs.edu

ABSTRACT

Executing distributed machine learning (ML) jobs on Spark follows Bulk Synchronous Parallel (BSP) model, where parallel tasks execute the same iteration at the same time and the generated updates must be synchronized on parameters when all tasks are finished. However, the parallel tasks rarely have the same execution time due to sparse data so that the synchronization has to wait for tasks finished late. Moreover, running Spark on heterogeneous clusters makes it even worse because of stragglers, where the synchronization is significantly delayed by the slowest task.

This paper attacks the fundamental BSP model that supports iterative ML jobs. We propose and develop a novel BSP-based Aggressive synchronization (A-BSP) model based on the convergent property of iterative ML algorithms, by allowing the algorithm to use the updates generated based on partial input data for synchronization. Specifically, when the fastest task completes, A-BSP fetches the current updates generated by the rest tasks that have partially processed their input data to push for aggressive synchronization. Furthermore, unprocessed data is prioritized for processing in the subsequent iterations to ensure algorithm convergence rate. Theoretically, we prove the algorithm convergence for gradient descent under A-BSP model. We have implemented A-BSP as a light-weight BSP-compatible mechanism in Spark and performed evaluations with various ML jobs. Experimental results show that compared to BSP, A-BSP speeds up the execution by up to 2.36x. We have also extended A-BSP onto Petuum platform and compared to the Stale Synchronous Parallel (SSP) and Asynchronous Synchronous Parallel (ASP) models. A-BSP performs better than SSP and ASP for gradient descent based jobs. It also outperforms SSP for jobs on physical heterogeneous clusters.

1 INTRODUCTION

Bulk Synchronous Parallel (BSP) model provides a simple and easy-to-use model for parallel data processing. For example, built on BSP model, Apache Spark [42] has evolved to be a widely used computing platform for distributed processing of large data sets in clusters. It is designed with generality to cover a wide range of workloads, including batch jobs [42], graph-parallel computation [20], SQL queries [8], machine learning (ML) jobs [24, 32], and streaming applications [43]. Particularly, Spark powers the library called MLlib [32] that is well-suited for iterative ML jobs [9]. With BSP model, all tasks in the job execute the same iteration at the same time and their generated updates must be synchronized on the solution (i.e., parameters) when all are finished.

BSP implicitly assumes that the parallel tasks have the same execution time. However, input data with sparse features (e.g., sparse matrix) induces imbalanced load among parallel tasks, leading to different task execution time. Thus, the synchronization has to wait for tasks finished late. Moreover, heterogeneous environments (e.g., physical heterogeneous clusters or multi-tenant clouds) worsen this situation, because of stragglers that run significantly slower than others [12]. Such sluggish tasks can take two to five times longer to complete than the fastest one in a heterogeneous production cluster [23], which could severely delay the synchronization.

Previous efforts addressed the straggler problem in several approaches. To improve the performance of MapReduce jobs in heterogeneous clusters, speculative task execution allows fast nodes to run a speculative copy of the straggler task [6, 16, 44]. In work [11] and [19], researchers provision tasks with adaptive input data sizes and match them with heterogeneous machines based on various capabilities. These approaches conform to BSP model and are applied to BSP-based computing platforms such as MapReduce, Hadoop, and Spark. Unfortunately, they become less effective when the heterogeneity varies during task execution, especially in a multi-tenant cloud with dynamic heterogeneity.

Recent efforts explored alternative models with loose synchronization. Asynchronous Synchronization Parallel (ASP) [1, 31] model enables that tasks on the fast machine can be processed without waiting for others. Stale Synchronous Parallel (SSP) [13, 14, 39] model allows the fastest task to be up to a bounded number of iterations ahead of the slowest one. While these models are able to address the effect of stragglers on job execution time to some extent, they are incompatible with BSP model and can only be applied to ML-specified platforms, e.g., Petuum [40]. The approaches cannot be easily integrated into Spark, lacking the generality.

In this paper, we aim for a BSP-compatible synchronization model for Spark and possible extension to other platforms such as Petuum. Towards this end, we propose and develop BSP-based Aggressive synchronization (A-BSP) model. The opportunity lies in the fact that ML algorithms can use updates from partial input data for synchronization. Figure 1 shows the comparison among the three models. Specifically, BSP waits for the straggler task, i.e., $Task_{3,i}$, to finish all input data, and then synchronize updates. SSP only synchronizes updates from two finished tasks (i.e., $Task_{1,i}$ and

Middleware '18, December 10–14, 2018, Rennes, France

ACM ISBN 978-1-4503-5702-9/18/12…$15.00
https://doi.org/10.1145/3274808.3274828

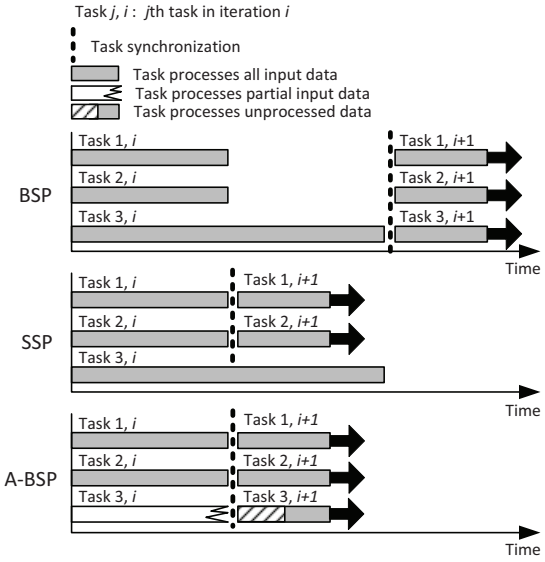


Figure 1: Three synchronization models.

$Task_{2,i}$), leaving the straggler task running, and it then launches two tasks in the next iteration. In A-BSP, when the fastest tasks are finished, it terminates the straggler $Task_{3,i}$ that has only partially processed its input. It then fetches the current updates generated from it to synchronize with other tasks. Finally, it launches all three tasks in the next iteration. Unprocessed data in $Task_{3,i}$ is processed by its subsequent $Task_{3,i+1}$ based on the updated parameters after the synchronization.

A-BSP exploits the convergent property of iterative ML algorithms. In iterative ML jobs, each iteration trains parameters on the input data to obtain updates, and parameters are adjusted using updates by the synchronization to better fit the input data. Thus, in A-BSP, even though the updates in a straggler task are generated from training parameters on its partial input in one iteration, the algorithm can reach convergence because parameters are trained on the entire input data across multiple iterations.

The challenge of partial processing lies in when and how to terminate input data processing. For instance, each task is unaware of the progress of others in the existing computing platforms. Also, the existing algorithms in ML libraries (e.g., Spark MLlib) require that the task has to process all input data to finish. To solve the challenge, A-BSP realizes partial processing through the cooperation between the task communication and ML algorithm augmentation. It also prioritizes unprocessed data to make sure that it will be processed in subsequent iterations, e.g., $Task_{3,i+1}$ in Figure 1.

Note that for iterative ML jobs based on gradient descent algorithm, mini-batch gradient descent approaches [29, 34] were proposed to use partial data for model learning and stochastic optimization. The mini-batch approaches pre-define a mini-batch size based on the characterization of datasets. Spark provides a hyper-parameter for setting the mini-batch size for all parallel tasks. It does not support a different mini-batch size for every task. Even if it does, it can only address static heterogeneity by profiling machine capacities, but it cannot address dynamic heterogeneity in multi-tenant clusters since the heterogeneity varies during task execution at runtime.

We develop and implement A-BSP model in Spark, and also extend it onto Petuum. The implementation, which is on the basis of the existing BSP implementation, does not require significant modification in the two platforms since A-BSP is developed as a lightweight BSP-compatible model. In contrast, SSP and ASP require significant modification (e.g., adding parameter server architecture) in BSP so that SSP and ASP cannot be easily integrated into the general-purpose platform Spark. We conduct performance evaluations with various ML jobs. Experimental results show that: 1) compared to BSP, A-BSP speeds up the job execution by up to 2.36x and 1.87x in multi-tenant and physical heterogeneous clusters, respectively; 2) A-BSP outperforms SSP and ASP for gradient descent jobs in both clusters. It also performs better than SSP for all ML jobs in the physical heterogeneous cluster.

In a nutshell, we make the following major contributions: 1) We design and develop A-BSP model, and implement it as a light-weight BSP-compatible mechanism in Spark. We also provide theoretical algorithm convergence proof for gradient descent. A-BSP is able to effectively tackle sparse data or stragglers, and significantly improve the performance of iterative ML jobs on Spark in heterogeneous clusters; 2) We extend A-BSP and implement it on ML-specific platform Petuum. Compared to SSP and ASP models, A-BSP works better for jobs based on the gradient descent algorithm. A-BSP also outperforms SSP on heterogeneous physical clusters.

The rest of this paper is organized as follows. Section 2 gives background and motivation on aggressive synchronization. Section 3 describes the design of A-BSP and compares with SSP and ASP. Section 4 describes the implementation. Section 5 and Section 6 present the experimental setup and evaluation results. Section 7 reviews related work. Section 8 concludes the paper.

2 BACKGROUND AND MOTIVATION

We first present the background on iterative ML jobs and BSP model for parallelization. We then provide two cases to show the potential gain of an aggressive synchronization that uses updates from partial input data for synchronization.

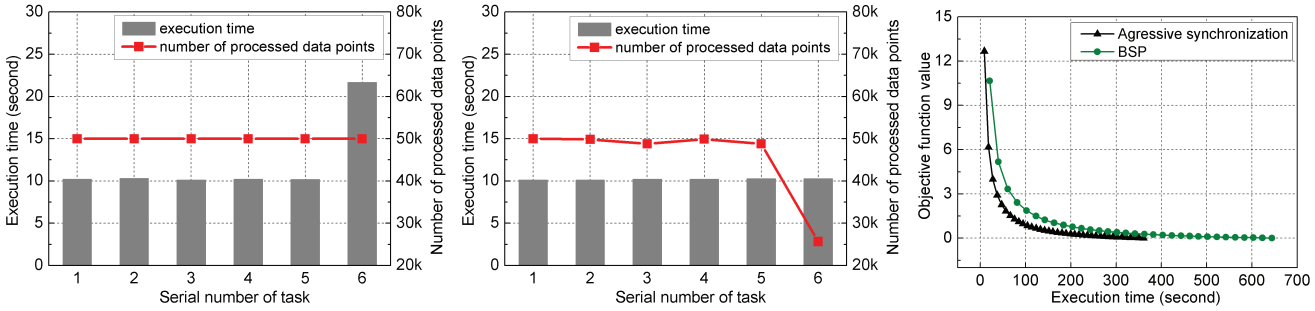
2.1 Iterative ML and BSP

Although iterative ML jobs come in many forms, such as Logistic Regression, Matrix Factorization, and K-means, they can all be presented as the model that seeks a set of parameters $\boldsymbol{V}$ to fit input data $\boldsymbol{D}$. Such a model is usually solved by an iterative algorithm that can be expressed in the following form:

$$V^{(t)} = V^{(t-1)} + \Delta(V^{(t-1)}, D) \quad (1)$$

where $V^{(t)}$ is the state of parameters in iteration t and update function $\Delta()$ trains parameters from the previous iteration t-1 on input data D. This operation repeats itself until state $V^{(t)}$ meets certain convergence criterion measured by an objective function (e.g., loss function and likelihood) with respect to the parameters for the entire input data. Methods such as Gradient Descent and Expectation Maximization are widely used in function $\Delta()$.

Executing iterative ML jobs on data-parallel clusters often distributes input data over parallel workers (e.g., machines) and trains



(a) One iteration execution with BSP. (b) One iteration execution with aggressive synchronization. (c) Convergence, each point represents one iteration.

Figure 2: Logistic Regression job execution with two synchronization models in a heterogeneous cluster.

parameters on the input split in parallel. To synchronize parallel updates $\Delta(V^{(t-1)}, D_i)$ in BSP model, all parallel tasks execute the same iteration at any given time, and synchronization is enforced by barriers (e.g., stages).

2.2 Case Studies

BSP model implicitly assumes that the parallel tasks of a job have the same execution time. This assumption might hold true if tasks are executed in a homogeneous cluster. However, it does not hold in the following cases. We further show how aggressive synchronization with partial processing could improve job performance in these cases.

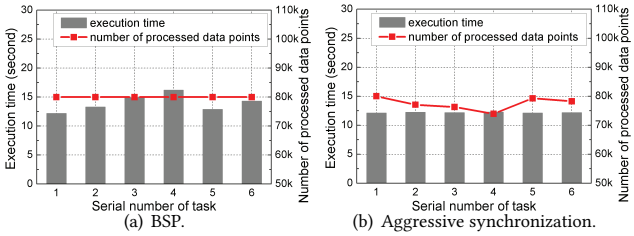
Case 1 (Stragglers). One primary performance issue for BSP is the straggler problem that often results from computational heterogeneity in physical heterogeneous clusters or multi-tenant cloud clusters. To illustrate it, we created a 7-node heterogeneous Spark cluster and ran Logistic Regression jobs. Logistic Regression jobs update parameters based on gradient descent that computes gradients of the objective function as follows:

$$V^{(t)} = V^{(t-1)} - \alpha \cdot \frac{\partial J(V^{t-1}, D)}{\partial V^{t-1}} \tag{2}$$

where α refers to the user-defined learning rate and the gradients means partial derivative with respect to parameters.

The cluster consists of one master node and six slave nodes (i.e., workers). The slave nodes contain five fast workers and a slow one (worker 6). The entire input data contains $300k$ (k=1000) data points from SparkBench [28]. Each worker has $50k$ points as task input and launches one task with the same serial number. In BSP, compared to the other tasks, task 6 takes significantly longer time in training parameters on all $50k$ points due to the computation heterogeneity, as shown in Figure 2(a). Thus, tasks 1 to 5 wait for the synchronization until task 6 is finished.

In aggressive synchronization, each task still has $50k$ points in its input. The key difference is that, when any one of tasks 1 to 5 is finished, other tasks that only train parameters on the partial input at the time are terminated, following which the current updates from the partial input are fetched to conduct synchronization. The result is shown in Figure 2(b). Task 6 takes a shorter time in training parameters on a smaller size of data points. Such time saving comes from the fact that input data in task 6 is partially processed, leaving almost half data points unprocessed.



(a) BSP. (b) Aggressive synchronization.

Figure 3: One iteration execution with sparse data.

We further set the convergence threshold to be 0.1 and ran the job with two synchronization models respectively. Aggressive synchronization requires 6 more iterations to reach the convergence, as shown in Figure 2(c). The average per-iteration reductions in objective function value are 0.396 and 0.333 by BSP and by aggressive synchronization, respectively. Although the average per-iteration convergence progress by aggressive synchronization is smaller than that by BSP, the execution time of one iteration is almost 2x faster. Specifically, aggressive synchronization and BSP spend 360 and 644 seconds reaching convergence, respectively. Therefore, the overall execution time is 82% faster by aggressive synchronization.

Case 2 (Sparse data). In homogeneous clusters, when input data contains sparse features that cause asymmetric workload, the execution time of parallel tasks varies. Although such various execution time is less severe than the straggler problem, the synchronization is delayed to some extent by tasks finished late. To illustrate this, we created a 7-node homogeneous Spark cluster with one master node and six slave nodes. We ran Logistic Regression job with sparse input data. The input on each worker contains $80k$ data points and has different sparsity degree that is set through `Vectors.sparse` in Spark MLlib. Note that although the inspector-executor API for Sparse BLAS is used in Intel Math Kernel Library, it does not support Spark MLlib.

In BSP, each task takes a different amount of time in training parameters of all $80k$ points as shown in Figure 3(a). Thus, the synchronization has to wait for the slowest task 4. In aggressive synchronization, after task 1 is finished, other tasks are terminated in the same way as in Case 1. The result is shown in Figure 3(b). To reach the convergence threshold (0.2), aggressive synchronization requires 3 more iterations, but the overall execution time is 31% faster than that of BSP.

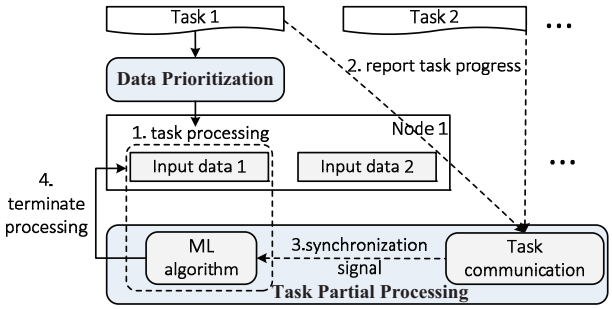


Figure 4: The architecture of A-BSP.

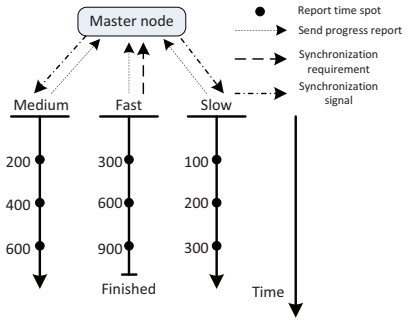


Figure 5: Task communication example. Three tasks are at different speeds. The number means how many data points are processed at the report time spot.

3 A-BSP DESIGN

3.1 Architecture

The key idea of A-BSP is to use updates from partial input data for synchronization. All tasks in a cluster start with the same size of input data, and partially process input data separately. Figure 4 shows the architecture of A-BSP. It centers on the design of two new mechanisms: task partial processing and data prioritization. We describe the functionality of each as follows:

- **Task Partial Processing** terminates the input data processing through the cooperation between task communication and ML algorithm augmentation. The task communication decides when to terminate the data processing. The augmentation enables the termination during the task execution.
- **Data Prioritization** prioritizes unprocessed data at two levels. At the task level, the mechanism ensures the unprocessed data points within the task input to be processed firstly in the next iteration. At the job level, when a certain input split has the least number of process times across multiple iterations, the mechanism moves the split to the fastest worker in order to avoid being partially processed again in later iterations.

3.2 Task Partial Processing

The goal of partial processing is to detect the first finished task and terminate others. When the first finished task is detected, the master node broadcasts the synchronization signal to parallel tasks running on slave nodes. Each task receives the signal and terminates its input data processing.

3.2.1 Task communication. During the execution, each task periodically sends a report to the master node. The report interval is set to one second that is the same as the default heartbeat interval in Hadoop and Spark. The report specifies the number of processed data points so that the master node can track the entire processed dataset in the cluster. One can reduce the report interval to get more fine-grained tracking of the number of processed data points. When the fastest task is finished, it sends the synchronization request to the master node. The master node makes the decision on when to broadcast the synchronization signal to the other tasks.

To avoid the network overhead due to frequent synchronizations, A-BSP does not send the signal immediately after the first task is finished. Instead, after receiving the request, synchronization decision is dependent on the number of processed data points in the cluster. Algorithm 1 shows the process, in which it triggers the signal after the ratio of processed data points reaches the synchronization ratio R_s. Users can define R_s according to the available network bandwidth. The decision process in Algorithm 1 supports various ML jobs since iterative ML jobs process the data points within input split sequentially. We leave job-specified decision making, which can leverage on algorithm characteristics such as the objective function value reduction between two successive iterations, to future work.

Algorithm 1 Synchronization decision in the master node.

1: **Input** N: the number of data points in a cluster;
2: **Input** R_s: synchronization ratio;
3: Receive synchronization request;
4: **do**
5: Get the number of current processed points in the cluster: N_p;
6: Processed ratio $R_p = N_p / N$;
7: **while** $R_p < R_s$
8: Send synchronization signal;

Figure 5 shows the example in which the number of data points in the cluster is 3,000 and the synchronization ratio is set to 0.5. Each task has 1,000 points in the input and sends its progress report to the master node at each time spot (i.e., every second). When the master node receives the synchronization request from the fastest task, the number of processed points is 1,800 and the processed ratio (0.6) is larger than the synchronization ratio. Thus, the master node sends the synchronization signal to the other two tasks to terminate their input processing.

3.2.2 ML algorithm augmentation. We describe how to modify the current ML algorithm implementation in order to add a new feature that receives the synchronization signal and terminates the input data processing. We do not implement the algorithm from scratch since the current computing platforms such as Spark already provide implementations of various ML algorithms.

Recall that an iterative ML algorithm trains parameters based on the update function Δ as shown in Equation 1. Widely used update methods such as Gradient Descent and Expectation Maximization support adding the receive-terminate feature. For example, in Spark MLlib, the core implementation of update function repeats a specified operation on each data point within each iteration through Scala language. We classify the repetition into two categories: `foreach` and `map`. `foreach` applies the operation on

each data point and the previous parameters but returns none. The operation in map returns updated parameters. Petuum uses for loop in C++ language to repeat specified operation on each data. Table 1 shows the update method of seven jobs used for experimental evaluation (refer to Section 6).

As for the algorithm modification, the foreach repetition in Spark is broken after receiving the synchronization signal, leaving the rest data points unprocessed. For the map repetition, we do not break the repetition after receiving the signal. Instead, the previous parameters are regarded as updated ones without the operation. In Petuum, the for loop is broken after receiving the synchronization signal. After termination, task execution is viewed as finished and the current updates are fetched for synchronization. The above modification is transparent to users and does not require any change to algorithm interface.

3.2.3 Convergence. For the five jobs based on gradient descent, the A-BSP implementations can be viewed as a variant of mini-batch gradient descent, in which mini-batch size is adaptively adjusted during task execution. The convergence proof is provided as follows.

THEOREM 1. *Gradient descent algorithms with A-BSP model converge as long as the algorithms with BSP model converge.*

PROOF. Previous studies have shown that for mini-batch gradient descent with mini-batch n under BSP model, the convergence rate is $O(1/\sqrt{K*n})$ [3, 17, 29, 30], that is:

$$J(V^K, D) - J(V^*, D) \leqslant \frac{C}{\sqrt{K*n}} \tag{3}$$

where K refers to the number of iterations and V^* denotes the optimal parameters. C is a constant determined by initial parameters V^0, V^*, and learning rate α in Equation 2.

We use N and ϵ to refer to the number of data points in the entire dataset and convergence threshold, respectively. Under BSP model, the algorithms can reach convergence with K_{bsp} iterations, in which K_{bsp} meets the equation $C/\sqrt{K_{bsp}*N} = \epsilon$.

With A-BSP model, we use n_i (i.e., mini-batch size) to refer to the number of data points processed in iteration i (i = 1, 2, 3, ...). Due to task partial processing, we have $R_s * N \leqslant n_i \leqslant N$, where R_s is the synchronization ratio in Algorithm 1. In the worst case, each iteration only processes $R_s * N$ data points. As a result, A-BSP needs to spend K_{worst} iterations reaching convergence, in which K_{worst} meets the equation $C/\sqrt{K_{worst}*(R_s*N)} = \epsilon$.

A-BSP model falls in the middle between BSP model and the worst case. Therefore, as long as BSP model can converge with K_{bsp} iterations, A-BSP model can converge within the range of $[K_{bsp}, K_{worst}]$ where $K_{worst} = K_{bsp}/R_s$. Theorem 1 is proved.

Note that the traditional mini-batch gradient descent randomly chooses n data points from the entire dataset to ensure that each point has the same possibility to be processed. Similarly, A-BSP employs data prioritization technique (in Section 3.3) to make sure that each point is processed roughly the same times (shown in Figure 15) in the entire job execution.

For K-means and Matrix Factorization jobs, A-BSP convergence is demonstrated through experimental evaluation.

3.3 Data Prioritization

In one iteration execution, a job runs parallel tasks in a cluster and partitions input data to the tasks. Each task is assigned with one input split and processes input data points in the split. In BSP, the tasks train parameters on every input data point one time. In another word, if we define the number of process times as the number of times a data point or an input split is processed, the number of process times of every data point in one iteration execution equals to one. Thus, in the entire job execution, the number of process times of every data point equals to the number of iterations. However, in A-BSP, the partial processing leaves some data points unprocessed within each iteration, leading to uneven process times as well as slowed per-iteration convergence progress. To remedy such weakness, we ensure that unprocessed data is prioritized for processing in subsequent iterations. Data prioritization serves as the complement to task partial processing, and is conducted before each iteration begins.

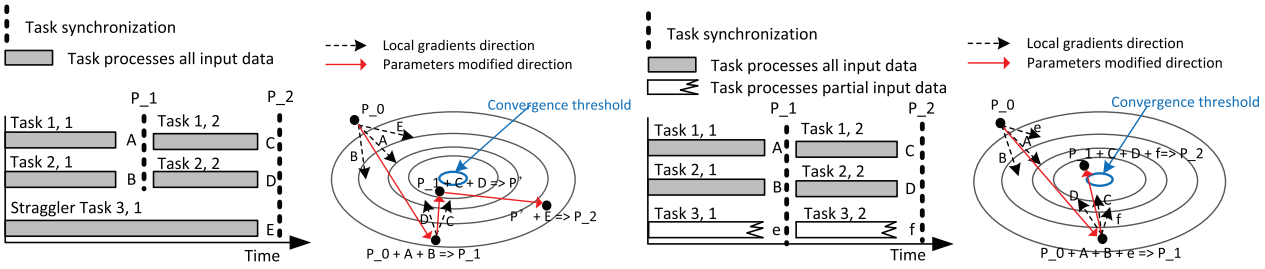
3.3.1 Task level. The goal of task level prioritization is to ensure that, for input data within the task, each data point has the same number of process times. To do so, unprocessed data in the current iteration will be processed firstly in the next iteration by changing the starting point of the iteration. For example, assume that the default processing sequence in each iteration is from data point 1 to n. If i is the last processed data point in the current iteration, data points from i+1 to n are unprocessed. The processing sequence in the next iteration is changed to: points i+1 to n, and points 1 to i.

3.3.2 Job level. Task level prioritization only balances the number of process times of data points within each input split. The job level prioritization further enables that each split has roughly the same number of process times. To this end, we first calculate the number of process times of each input split and then prioritize the split with the minimum number of process times as shown in Algorithm 2. Note that the job level prioritization that balances the number of process times is different from previous approaches [11, 25] that balance input data size of each task.

Algorithm 2 Job level data prioritization in the master node.

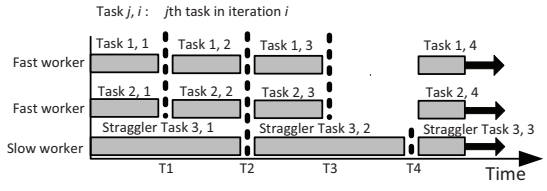
1: **Input** P_t: the prioritization threshold;
2: **Input** N_i^{iter}: the number of process times of input split i;
3: **Input** $N_{i,k}^{iter}$: the number of process times of data point k in split i;
4: N_i^{iter} = min ($N_{i,1}^{iter}$, $N_{i,2}^{iter}$, ..., $N_{i,n}^{iter}$);
5: Calculate the number of process times of each input split;
6: Get split *min* with the minimum number of process times N_{min}^{iter};
7: Get split *max* with the maximum number of process times N_{max}^{iter};
8: Calculate difference *diff* = N_{max}^{iter} - N_{min}^{iter};
9: **if** *diff* > P_t
10: Exchange the locations of split *max* and split *min*;
11: **end if**

For input split i, the number of process times N_i^{iter} that relies on the processing speed of the local worker is calculated as line 4. Thus, when input split *min* is partially processed in every iteration and has the smallest N_{min}^{iter} compared to others (line 6), we assume that its local worker is the slowest one. Meanwhile, when input split *max* is rarely partially processed and has the largest N_{max}^{iter}, we assume that its local worker is the fastest one (line 7). When the difference between N_{max}^{iter} and N_{min}^{iter} is larger than a threshold

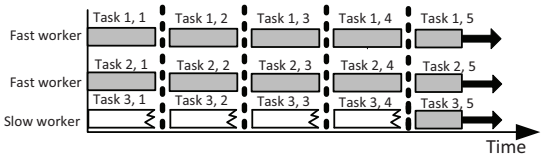


(a) Job execution in SSP and ASP. (b) Synchronization in SSP and ASP. (c) Job execution in A-BSP. (d) Synchronization in A-BSP.

Figure 6: A synchronization example of the gradient descent based job to illustrate why parameters are abnormally modified in SSP and ASP. A-E, e, and f represent the local gradients generated from each task. P_i means the parameters after synchronization. P' represents the intermediate parameters after adding C and D.



(a) SSP.



(b) A-BSP.

Figure 7: A job execution example in a physical heterogeneous cluster to illustrate why stragglers delay other tasks in SSP. At T3 in (a), fast workers cannot launch tasks since they can only be one iteration ahead of the slow worker.

(line 9), we move the split *min* to the fastest worker and the split *max* to the slowest worker in order to balance the load (line 10). Network overhead incurred by moving input split is evaluated in Section 6.3. Users can define the prioritization threshold based on the profiled cluster heterogeneity.

3.4 A-BSP, SSP, and ASP

A-BSP, SSP, and ASP models exploit the convergent nature of iterative ML algorithms. However, the advantage of A-BSP compared to SSP and ASP lies in the following two scenarios.

3.4.1 Jobs based on Gradient Descent. First, in SSP and ASP, stale updates from the straggler task could abnormally modify parameters, especially for jobs based on gradient descent [23]. In these jobs, local gradients (i.e., updates) from each task are directly added to parameters in the synchronization. However, stale gradients from the straggler task could modify parameters in the wrong descent direction, requiring extra iterations in order to reach convergence.

We use the example in Figure 6 to further explain the reason. The initial parameters are **P_0** and the circles represent the contour lines of the objective function. In SSP as shown in Figure 6(b), gradients **A** and **B** are added to **P_0** in the first synchronization so that parameters are modified to be **P_1**. After adding gradients **C** and **D** in the second synchronization, parameters become **P'** which is close to the convergence threshold. However, the stale gradients **E** push **P'** away from the threshold, leading to abnormally modified parameters **P_2**. The reason is that gradients **E** that are generated based on initial parameters are unaware of the state of the latest parameters **P_1**. In contrast, A-BSP enables that local gradients from each task are generated based on the latest parameters as shown in Figure 6(d), ensuring the right descent direction.

3.4.2 Physical heterogeneous cluster. Second, in physical heterogeneous clusters with static heterogeneity, synchronization is still delayed by straggler tasks in SSP [22]. SSP allows the fastest task to be a bounded number (i.e., staleness) of iterations ahead of the straggler task. However, the iteration gap between them increases over time in the physical heterogeneous cluster. Eventually, the straggler task delays the fastest one.

For example, Figure 7 shows the job execution in two models. We set the staleness to be 1. In SSP as shown in Figure 7(a), fast workers start $Task_{1,2}$ and $Task_{2,2}$ at time **T1** and then iteration gap becomes 1. They can also start two tasks at time **T2**. However, if they start $Task_{1,4}$ and $Task_{2,4}$ at time **T3**, the iteration gap increases to 2 that is larger than the staleness. As a result, the two tasks are delayed by $Task_{3,2}$ and started at time **T4**. Therefore, SSP only addresses the straggler problem at the beginning of job execution. A naive solution is to set a larger staleness. However, larger staleness increases the possibility that parameters are abnormally modified. In contrast, A-BSP addresses the straggler problem as shown in Figure 7(b). However, more frequent synchronizations bring extra network transmission cost, which is evaluated in Section 6.3.

4 IMPLEMENTATION

We have implemented A-BSP in Spark (version 2.1.1). Task communication is implemented in package `spark.core`. Algorithm augmentation and data prioritization are implemented in package `spark.mllib`. The augmentation of gradient descent jobs is also implemented in Scala (version 2.11.8) `scala.collection`. We also implemented A-BSP in Bösen [39] (version 1.1), a subsystem that supports iterative ML in Petuum.

Task communication. In Spark, an executor contains multiple tasks. The tasks communicate by sharing several global variables in

Table 1: ML algorithm update method and modification file in Spark and Petuum.

Algorithm	Update method	Core implementation category in Spark	Modification file in Spark	Modification file in Petuum
Logistic Regression (LR)	Gradient Descent	`foreach`	`TraverableOnce.scala`	`dnn.cpp`
SVM	Gradient Descent	`foreach`	`TraverableOnce.scala`	`dnn.cpp`
Lasso	Gradient Descent	`foreach`	`TraverableOnce.scala`	`dnn.cpp`
Ridge Regression (RR)	Gradient Descent	`foreach`	`TraverableOnce.scala`	
K-means	Lloyd Algorithm	`foreach`	`mllib.clustering.KMeans.scala`	`kmeans_worker.cpp`
Matrix Factorization (MF)	Expectation Maximization	`map`	`ml.recommendation.ALS.scala`	
Deep Neural Network (DNN)	Gradient Descent			`dnn.cpp`

```
Default implementation:
Input data.
foreach {
    operation(data point,
    previous parameters);
}

Input data.
map {
    updated parameters =
    operation(data point,
    previous parameters);
    return updated parameters;
}

Modified implementation:
Input data.
foreach {
    receiving signal;
    if (received) {
        break;
    } else {
        operation(data point,
        previous parameters);
    }
}

Input data.
map {
    receiving signal;
    if (received) {
        updated parameters =
        previous parameters;
    } else {
        updated parameters =
        operation(data point,
        previous parameters);
    }
    return updated parameters;
}
```

(a) Algorithm augmentation on Spark.

```
Default implementation:
for (int i=0; i<size_batch; i++) {
    operation(data points[i], previous parameters);
}

Modified implementation:
for (int i=0; i<size_batch; i++) {
    receiving signal;
    if (received){
        break;
    } else {
        operation(data points[i],previous parameters);
    }
}
```

(b) Algorithm augmentation on Petuum.

Figure 8: Algorithm augmentation in pseudo code.

`Executor.scala`. The task changes the variable `NumProcessed` to report its progress and changes the variable `SynRequire` to send the synchronization request when it is finished. The executor changes variable `SynSignal` to send synchronization signals to each task. Communication between the executor and master is implemented by built-in `RPC` in `CoarseGrainedExecutorBackend.scala` and `CoarseGrainedSchedulerBackend.scala`. In Petuum, we used one machine as a parameter server. Each task reports its progress to the server and the server broadcasts the synchronization signals to all tasks. The task communication is implemented based on `Remote Procedure Call` protocol using Python language. The communication between Python and C++ is implemented through `ctypes` model in Python language.

ML algorithm augmentation. We modified the source code of seven ML jobs used for performance evaluation. Table 1 shows the modified source files in Spark and in Petuum. For jobs based on gradient descent, their modified files are the same since they share the same update method. Figure 8 shows the default and modified implementations of ML algorithms in Spark and Petuum.

Table 2: The hardware configuration of the physical cluster.

Machine model	CPU model	Number
PowerEdge T630	Intel Sandy Bridge 2.3GHz	1
PowerEdge T110	Intel Nehalem 3.2GHz	3
OPTIPLEX 990	Intel Core i7 3.4GHz	9

Data prioritization. In Spark, we changed the processing sequence in collection `Iterator` so as to implement the task level prioritization. For the job level prioritization, we recorded the number of process times of each input split in the master node. To change the location of the input split, a task that processes the split is scheduled to the designated worker. This worker then automatically and remotely reads the split. Petuum uses `array` data structure to store data points for each task. Thus, we rearranged the sequence of data points in the array to implement the task level prioritization. For the job level prioritization, we start a new worker thread (i.e., task) that processes the input split in the designated worker. This worker can also automatically and remotely read the split through `Network File System` used in Petuum.

5 EVALUATION SETUP

5.1 Testbed

We set up two heterogeneous clusters to evaluate the performance of the proposed A-BSP model and implemented mechanism.

A multi-tenant cluster is a 37-node virtual cluster in a university multi-tenant cloud. The virtual cluster runs on 8 HP BL460c G6 blade servers interconnected with 10Gbps Ethernet. VMware vSphere 5.1 was used to provide the server virtualization. Each virtual node was configured with 2 vCPUs and 4GB memory. The cloud is shared by all faculty, staffs, and students in a multi-tenant manner. Thus, it exhibits dynamic heterogeneity from co-hosted VMs that contend for shared resources.

A physical cluster is a 13-node heterogeneous cluster dedicated to running ML jobs, which consists of three different types of machines. Table 2 lists the hardware configurations of the machines in the cluster. We use this cluster to assess different synchronization models in the cluster with static heterogeneity. Each cluster uses one node as the master and the rest nodes as slaves.

All nodes in the two clusters run Ubuntu Server 14.04 with Linux kernel 4.4.0-64, Java 1.8, Scala 2.11.8 and GCC 5.4.0.

5.2 Workloads

To estimate the effectiveness of A-BSP, we chose eight representative ML jobs. Seven of them are shown in Table 1 and they use dense

input data. We also run **Sparse LR** that is the logistic regression job with sparse input data.

DNN is a fully-connected deep neural network that contains an input layer, four hidden layers, and an output layer. It uses a dataset generated by script `gen_data.sh` in Petuum. The dataset contains $2m$ *(m=1,000,000)* data points for the multi-tenant cluster and $1m$ points for the physical cluster. The convergence threshold is set to 0.5 (default value) for each cluster.

Datasets for the rest jobs are generated through SparkBench [28] and convergence thresholds are set to the default values in SparkBench. Specifically, the dataset for **LR**, **Sparse LR**, and **SVM** contains $2.5m$ data points for the multi-tenant cluster and $1.25m$ points for the physical cluster. The convergence threshold is set to 0.01 for each cluster. Dataset for **Lasso** and **RR** contains $2m$ data points for the multi-tenant cluster and $1m$ points for the physical cluster. The convergence threshold is set to 0.02 for each cluster. The six jobs update parameters based on gradient descent and assign tasks with full-batch data points before each iteration begins.

The dataset for **K-means** contains $40m$ *num_of_samples* for the multi-tenant cluster and $20m$ *num_of_samples* for the physical cluster. Each sample has 200 *dimensions*. The convergence threshold is set to 10^{-10} for each cluster. **MF** job uses Expectation Maximization based alternating least squares algorithm [5]. Its dataset contains a $300k$*$30k$ matrix for the multi-tenant cluster and a $200k$*$20k$ matrix for the physical cluster. The convergence threshold is set to 10^8 for each cluster.

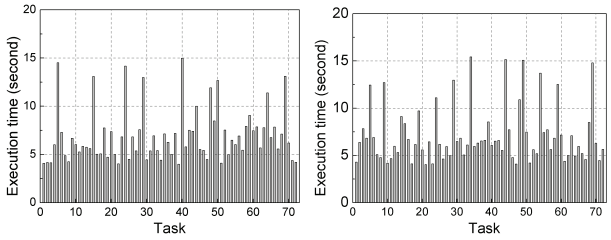
5.3 Approaches and Performance Metrics

We evaluate the performance of four synchronization models: A-BSP, BSP, SSP, and ASP using two computing platforms. A-BSP and BSP are evaluated in Spark since Spark only supports these two models. We also deploy A-BSP without data prioritization in Spark (A-BSP w/o DP) to evaluate the effect of data prioritization. Compared to the general-purpose computing platform Spark, Petuum is a platform specific for iterative ML, which takes advantage of the convergent property of ML algorithms. Petuum employs parameter server architecture to manage network communication and supports SSP and ASP models in parameter synchronization. Thus, A-BSP, SSP, and ASP are evaluated in Petuum. Also, SSP is evaluated with typical small and large staleness [22, 23]. ASP is evaluated with staleness = ∞. Synchronization ratio is set to 0.5 and the prioritization threshold is set to 5 in A-BSP.

The performance metrics include job execution time under the same convergence criterion, number of iterations, and averaged per-iteration time. For SSP and ASP models, each task sends its updates to parameter servers individually so that there is no explicit number of iterations. Therefore, the number of iterations is defined as $\frac{number\ of\ sending\ updates}{number\ of\ parallel\ tasks}$.

6 EXPERIMENTAL EVALUATION

This section evaluates the effectiveness of A-BSP. We compare A-BSP to BSP in Spark and compare A-BSP to SSP and ASP in Petuum. We also evaluate the overhead of A-BSP. The results show that: (1) A-BSP greatly outperforms BSP in the two clusters for all ML jobs (Section 5.2); (2) A-BSP outperforms SSP and ASP for gradient descent jobs in the two clusters. It also performs better than SSP for all ML jobs in the physical cluster. The reason is discussed in Section 3.4. (3) A-BSP brings extra network transmission overhead. However, the overhead is much smaller compared to the performance improvement.



(a) Execution time in iteration a. (b) Execution time in iteration b.

Figure 9: Dynamic heterogeneity of the multi-tenant cluster.

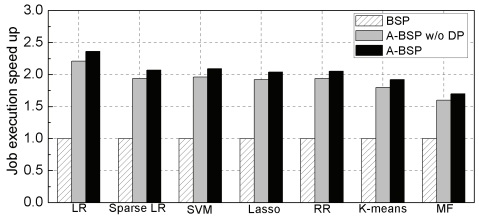


Figure 10: Job execution speedup by A-BSP compared to BSP in the multi-tenant Spark cluster.

6.1 A-BSP and BSP in Spark

6.1.1 In the Multi-tenant cluster. Multi-tenant cluster exhibits dynamic heterogeneity. To demonstrate it, we run LR job with BSP and present the task execution time in two random selected iterations (iterations a and b) in Figure 9. Each virtual node runs two tasks and the number of parallel tasks is 72. The result of both iterations shows that the heterogeneity changes dynamically and the fastest task can be almost 4x faster than the slowest one.

Figure 10 plots the job execution speedup in A-BSP compared to BSP, in which BSP is regarded as the baseline. A-BSP converges significantly faster for all ML jobs. It can be up to 2.36x faster than that of BSP. MF job obtains the least performance improvement. The reason is that the processing time and network transmission cost are two bottlenecks for MF job and A-BSP only optimizes the former. Moreover, A-BSP performs better than A-BSP w/o DP, showing that data prioritization can further improve job performance.

Table 3 shows the results of three performance metrics for the seven ML jobs. The speedup in Figure 10 is calculated based on the job execution time. Compared to BSP, A-BSP needs more iterations to reach convergence. However, the averaged per-iteration time in A-BSP is much shorter. A-BSP w/o DP has roughly the same averaged per-iteration time as A-BSP. The reason is that data prioritization only determines the priority of each data point. It has no impact on the number of processed data points in one iteration.

Figure11 shows the number of processed data points in three input splits across 20 iterations of LR job. Three splits are located on different machines. In BSP, three splits have the same number of processed points in the iterations. In A-BSP, three splits have fewer

Table 3: Job performance with A-BSP and BSP in the multi-tenant Spark cluster

Metric	Model	Iterative ML job						
		LR	Sparse LR	SVM	Lasso	RR	K-means	MF
Job execution time in seconds	BSP	781	825	708	411	361	496	1237
	A-BSP w/o DP	353	425	361	213	186	275	772
	A-BSP	331	398	338	201	176	258	728
# of iterations	BSP	78	108	82	66	62	32	70
	A-BSP w/o DP	93	141	119	86	79	41	79
	A-BSP	87	132	112	79	74	38	75
Averaged per-iteration time	BSP	10.01	7.64	8.63	6.23	5.82	15.50	17.67
	A-BSP w/o DP	3.79	3.01	3.03	2.47	2.35	6.70	9.77
	A-BSP	3.80	3.02	3.02	2.54	2.38	6.79	9.71

Table 4: Job performance with A-BSP and BSP in the physical Spark cluster.

Metric	Model	Iterative ML job						
		LR	Sparse LR	SVM	Lasso	RR	K-means	MF
Job execution time in seconds	BSP	630	660	571	479	463	369	990
	A-BSP w/o DP	418	447	381	310	299	261	779
	A-BSP	347	372	315	256	251	198	648

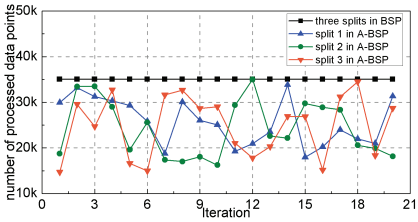
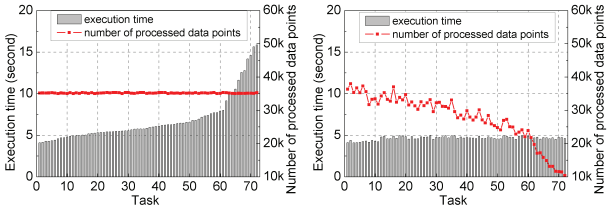
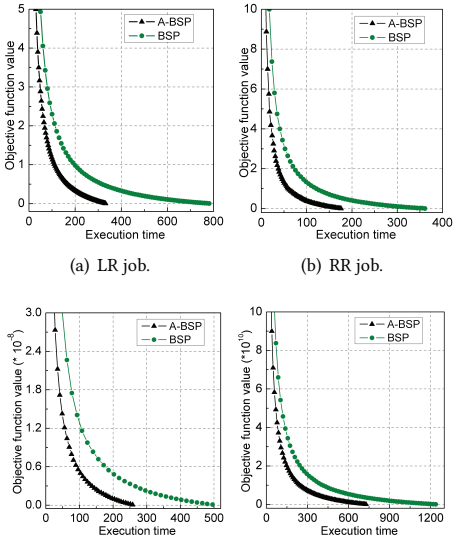


Figure 11: Number of processed data points in three input splits of LR job in the multi-tenant Spark cluster.



(a) BSP Spark. (b) A-BSP Spark.

Figure 12: One iteration execution of LR job in Spark.



(a) LR job. (b) RR job.

(c) K-means job. (d) MF job.

Figure 13: Convergence process of BSP and A-BSP in Spark.

data points processed due to task partial processing. Moreover, for each split, the number is dynamically changed across the iterations since the computation capacity of its local machine varies during job execution. In iteration 12, split 2 is fully processed since its local machine is the fastest at that time. Results demonstrate the adaptivity of A-BSP under dynamic heterogeneity in the multi-tenant cluster.

Figure 12 shows the task execution time in one iteration of LR job. Specifically, Figure 12(a) presents the ranked task execution time as well as the number of processed data points of each task in BSP. Similar to that in Figure 9, the fastest task is 4x faster than the slowest one. Figure 12(b) shows the counterpart in A-BSP. We observe that most tasks only process their partial input data so that the iteration execution time is significantly reduced and balanced. The parallel tasks do not have exactly the same execution time due to the latency in sending synchronization signals.

Figure 13 shows the convergence progress of four jobs by BSP and A-BSP in Spark. The A-BSP convergence curves are always beneath the BSP curves, demonstrating that A-BSP can achieve lower objective function value with the same execution time.

6.1.2 In the Physical cluster. The physical cluster exhibits static computation heterogeneity and the fastest task is 2.2x faster than the slowest one. Table 4 shows the result of job execution time of the seven iterative ML jobs by A-BSP and BSP. Figure 14 plots the execution speedup by A-BSP compared to BSP. In general, A-BSP outperforms BSP and is at most 1.87x faster. Moreover, A-BSP performs much better than A-BSP w/o DP, because data prioritization can greatly improve job performance in the physical cluster.

Figure 15 presents the number of process times of each input split in LR job (i.e., N_i^{iter} in Algorithm 2). Each worker has four splits so that the number of splits is 48. In BSP, LR job spends 72 iterations reaching convergence. Thus, the number of process

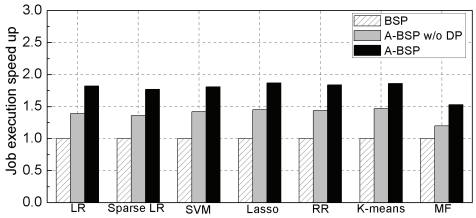
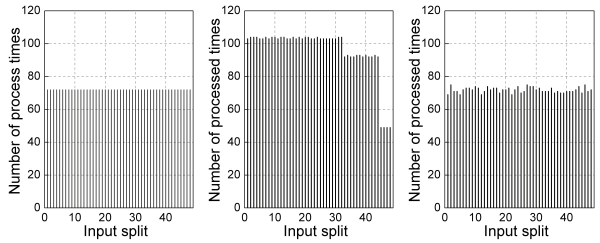


Figure 14: Job execution speedup by A-BSP compared to BSP in the physical Spark cluster.



(a) BSP (b) A-BSP w/o DP (c) A-BSP

Figure 15: Number of process times of input split in LR job in the physical Spark cluster.

times equals to the number of iterations since each split is fully processed in one iteration as shown in Figure 15(a). In A-BSP w/o DP, input splits in slow workers are partially processed in each iteration and have fewer number of process times as shown in Figure 15(b). Thus, they contribute fewer updates to parameters, and more iterations are needed to fit in parameters. In A-BSP, job level data prioritization enables that each split can achieve roughly the same number of process times, improving per-iteration convergence progress compared to A-BSP w/o DP.

6.2 A-BSP, SSP, and ASP in Petuum

6.2.1 In the Multi-tenant cluster. Table 5 shows the results of the seven iterative ML jobs by A-BSP, SSP, and ASP in Petuum. Variable *s* refers to the staleness in SSP model. Figure 16 plots the execution speedup by A-BSP compared to that of SSP and ASP. SSP with large staleness (s=10) is regarded as the baseline. The comparison result between A-BSP and BSP (i.e., SSP with s=0) is omitted since the performance improvement over BSP in the Petuum cluster is similar to that in the Spark cluster.

For jobs based on gradient descent, A-BSP outperforms SSP with small staleness (s=3). The reason is that the stable gradients from a straggler task could update parameters in the wrong descent direction as discussed in Section 3.4. The performance improvement is not substantial since SSP with s=3 can also address the straggler problem to some extent. The performance of SSP further degrades when using large staleness. Large staleness requires more iterations to reach convergence. ASP with unlimited staleness has the worst performance. Note that we have tested SSP with various staleness values from 1 to 10, and SSP with s=3 has the best performance among them. Figure 16 plots the performance of SSP with s=3 and s=10 since 3 and 10 are the typical values of staleness in previous studies [23, 37].

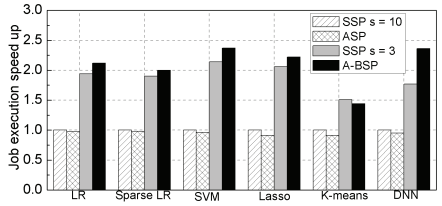
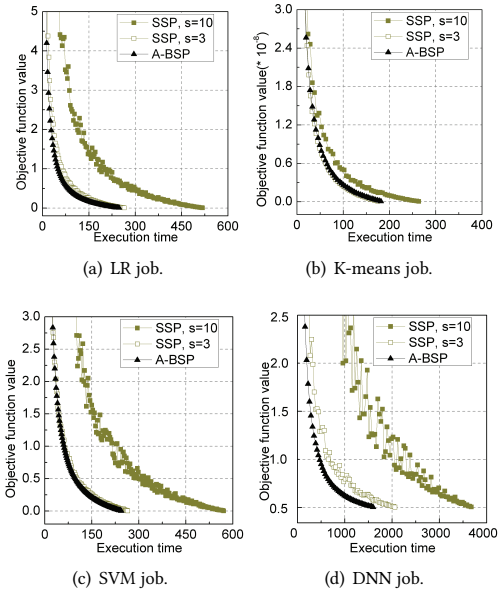


Figure 16: Job execution speedup with A-BSP compared to SSP and ASP in the multi-tenant Petuum cluster.



(a) LR job. (b) K-means job.

(c) SVM job. (d) DNN job.

Figure 17: Convergence process of A-BSP and SSP in Petuum (s denotes the staleness of SSP).

For K-means job, SSP performs slightly better than A-BSP, because parameter server architecture in Petuum is designed for SSP and it can optimize the network communication of SSP. Specifically, in A-BSP, all tasks communicate with a master node simultaneously during the synchronization, leading to bursty network traffic. In SSP, each task communicates with parameter servers immediately after the task finishes so that there is no bursty network traffic.

Figure 17 shows the convergence process of four jobs with A-BSP and SSP models in Petuum. A-BSP converges faster than SSP does with small staleness for LR, SVM, and DNN jobs. With large staleness, the convergence process of the three jobs is further fluctuated because of the wrong descent direction from stale gradients, leading to a greater number of iterations. DNN job exhibits the most severe fluctuation since it contains more parameters than other jobs do. SSP only converges faster for K-means job, because of parameter server architecture in Petuum.

6.2.2 In the Physical cluster. Table 6 shows the job execution time with A-BSP, SSP, and ASP in the physical cluster. Figure 18 plots the execution speedup with A-BSP compared to SSP and ASP. A-BSP performs much better than SSP with small staleness for all ML jobs. The reason is that SSP suffers from static heterogeneity as discussed in Section 3.4.

Table 5: Job performance with A-BSP, SSP, and ASP in the multi-tenant Petuum cluster.

Metric	Model	Iterative ML job					
		LR	Sparse LR	SVM	Lasso	K-means	DNN
Job execution time in seconds	ASP	531	601	593	321	288	3868
	SSP, s=10	518	588	571	293	263	3674
	SSP, s=3	265	309	265	141	175	2061
	A-BSP	248	297	244	133	181	1599
# of iterations	ASP	164	213	209	169	50	128
	SSP, s=10	153	198	194	153	54	115
	SSP, s=3	89	139	119	87	43	70
	A-BSP	88	131	113	77	45	51

Table 6: Job performance with A-BSP, SSP, and ASP in the physical Petuum cluster.

Metric	Model	Iterative ML job					
		LR	Sparse LR	SVM	Lasso	K-means	DNN
Job execution time in seconds	ASP	472	498	563	341	208	3098
	SSP, s=10	499	508	579	369	223	3285
	SSP, s=3	382	378	359	262	193	1912
	A-BSP	248	257	226	168	131	1198

Table 7: Network transmission cost of A-BSP in Spark.

Cluster	Model	Iterative ML job						
		LR	Sparse LR	SVM	Lasso	RR	K-means	MF
Multi-tenant	BSP	2.3%	1.9%	2.1%	2.4%	2.5%	0.9%	23%
	A-BSP	2.6%	2.2%	2.6%	2.9%	3.1%	1.3%	29%
Physical	BSP	2.1%	1.5%	1.7%	1.9%	2.1%	0.8%	19%
	A-BSP	3.7%	2.9%	3.6%	3.4%	3.8%	2.8%	23%

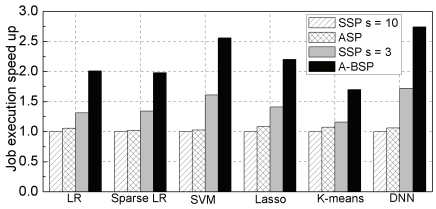


Figure 18: Job execution speedup with A-BSP compared to SSP and ASP in the physical Petuum cluster.

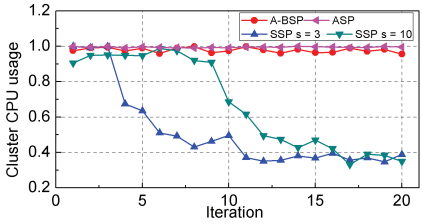


Figure 19: Cluster CPU usage across first 20 iterations of LR job in the physical Petuum cluster.

Figure 19 shows the cluster CPU usage across the first 20 iterations of LR job (SSP and ASP use the iteration of the slowest task). A-BSP keeps high usage since the synchronization is aggressively conducted without waiting for the slowest task. SSP has high usage during the first 3 or 10 iterations, depending on the value of staleness. However, the usage decreases in later iterations since fast workers have to wait for the slowest task. ASP can achieve high usage across all iterations since it allows fast workers to launch tasks without any constraint. However, the stale updates from the slowest task abnormally modify the global parameters, leading to worse performance compared to A-BSP.

6.3 A-BSP Communication Overhead

In each synchronization, updates from parallel tasks are transmitted from slave nodes to the master node, causing network transmission cost. A-BSP incurs extra transmission cost (i.e., overhead) because of aggressive synchronization. Table 7 measures the transmission cost percentage in the execution time of seven jobs in Spark. The overhead in the physical cluster is larger than that in the multi-tenant cluster due to the input split movement in job level data prioritization. Also, the overhead for MF is larger than that for the other jobs since MF has the largest transmission cost. Overall, the overhead is much smaller compared to the performance improvement by A-BSP. Similar results are observed in Petuum.

Note that if the size of the current two clusters increases, the communication overhead increases since more slave nodes communicate with the master node during the synchronization. However, the performance gain of A-BSP in larger clusters is also more significant than that in the current clusters since the straggler problem becomes more severe when the cluster size increases [22]. Thus, the communication overhead would still be smaller than the performance gain in larger clusters.

7 RELATED WORK

BSP is a simple and easy-to-use model adopted by many general-purpose distributed computing platforms. Solving performance issues, e.g., straggler problem, for BSP is the focus of many efforts.

BSP-based approaches: Speculative execution approaches are widely used to address the straggler problem in computing platforms like Hadoop and Spark [6, 7, 16]. The concept is to run stragglers, i.e., slow tasks, redundantly on multiple machines. However, Hadoop's scheduler starts speculative tasks based on a simple

heuristic comparing each task's progress to the average progress. Although this heuristic works well in homogeneous environments where stragglers are obvious, it can lead to severe performance degradation in heterogeneous clusters. LATE [44] uses a new algorithm called Longest Approximate Time to End (LATE) for speculative execution that is robust to heterogeneity and is highly effective in practice. It is based on three principles: prioritizing tasks to speculate, selecting fast nodes to run on, and capping speculative tasks to prevent thrashing.

In task size adjustment approaches, FlexMap [11] launches heterogeneous tasks with different sizes in the map phase to match the processing capability of machines. PIKACHU [19] proposes a novel partition scheme to launch heterogeneous tasks with different sizes in the reduce phase. DynamicShare [41] proposes new intermediate data partition algorithms to adaptively adjust the task size for MapReduce jobs. FlexSlot [21], a user-transparent task slot management scheme, automatically identifies map stragglers and resizes their slots accordingly to accelerate task execution.

Work stealing approaches [2, 18] wait for a worker to be idle and then move task loads from a busy worker to an idle worker. SkewTune [25] is an automatic skewness mitigation approach for user-defined MapReduce programs. When a node in the cluster becomes idle, SkewTune identifies the task with the greatest expected remaining processing time. The unprocessed input data of this straggler task is then repartitioned in a way that fully utilizes the nodes in the cluster. FlexRR [22] combines flexible consistency bounds with a new temporary work reassignment mechanism called RapidReassignment. RapidReassignment detects and temporarily shifts work from stragglers before they fall too far behind so that workers never have to wait for one another. The above approaches can be applied to BSP-based platforms, however, they either require redundant resources or become less efficient when encountered with dynamic heterogeneity in a multi-tenant cloud cluster running data-intensive parallel jobs [36].

Less strict synchronization approaches: BSP model exhibits the strict need for synchronization. Recent approaches explore alternative models with loose synchronization. While the models are able to address the straggler problem, they can only be implemented in ML-specified platforms such as Petuum that lack generality of BSP-based platforms. The approaches are described as follows.

YahooLDA [4] proposes a communication structure for asynchronously updating global variables. However, it is specialized for latent dirichlet allocation algorithm. Computing systems such as Petuum [40] and MXNet [10] improves YahooLDA with SSP or "Bounded Delay consistency" [26, 27] model to allow any task to be up to a bounded number (i.e., staleness) of iterations ahead of the slowest one. While SSP model has been shown to reduce the effects of stragglers on execution time, it suffers from static heterogeneity and could result in incorrect convergence under inappropriate staleness [23].

Hogwild [33] proposes an asynchronous stochastic gradient descent algorithm. It allows processors to run SGD in parallel without locks. Systems such as GraphLab [31] and TensorFlow [1, 15] improves Hogwild with ASP model in which tasks on the fast worker can be processed without waiting for others so as to reduce the need for synchronization. ASP model often requires a strict structure of communication patterns. For example, GraphLab programs structure computation as a graph, where data can exist on nodes and edges. Communication occurs along the edges of the graph. If two nodes on the graph are sufficiently far apart they may be updated without synchronization. This model can significantly reduce synchronization in some cases, though it requires the application programmer to specify the communication pattern explicitly.

Partial Gradient Exchange [38] removes the burden of parameter server deployment. It assumes that clusters are homogeneous. SSP-based DynSSP [23] introduces heterogeneity-aware dynamic learning rate for gradient descent based jobs. However, DynSSP cannot effectively address the straggler problem in physical heterogeneous clusters with static heterogeneity, which is similar to SSP. Probabilistic BSP [35] first randomly samples a subset of tasks and then conducts synchronization for these tasks. However, the synchronization could be delayed since the random sampling cannot guarantee that all tasks in the subset have the same execution time.

Recent work [37] proposes a flexible synchronous parallel (FSP) framework for expectation-maximization (EM) algorithm. The idea is to suspend local E-step computation and conduct global M-step. A-BSP differs from FSP mainly in two aspects. First, as YahooLDA and Hogwild, FSP is a specific parallel framework with the synchronization operation dedicated to EM algorithm. Instead, A-BSP offers an alternative synchronization model in general ML platforms (e.g., Spark and Petuum). It has been proved by the experimental evaluation that A-BSP works for different ML algorithms. Second, work [37] only compare FSP to pre-defined synchronous parallel design (a BSP-based strict synchronization design for EM algorithm) and ASP model, lacking the comparison between FSP and SSP models. Instead, our work compares A-BSP with various synchronization models. Experimental results show that A-BSP outperforms SSP for jobs based on gradient descent, or jobs in physical heterogeneous clusters.

8 CONCLUSION

This paper presents the design and development of A-BSP, a BSP-based aggressive synchronization model for iterative ML Jobs, which is implemented as a new BSP-compatible mechanism in Spark and Petuum. The key idea is that A-BSP allows ML algorithms to use updates from partial input data for synchronization. A-BSP can address performance issues in BSP model including the straggler problem in heterogeneous clusters and asymmetric workloads incurred by sparse data. We have performed comprehensive evaluations with various ML jobs in different clusters while providing convergence proof for gradient descent algorithm. Experimental results show that in Spark, the job execution with A-BSP is significantly faster than that with BSP by up to 2.36x. In Petuum, A-BSP performs better than SSP and ASP for gradient descent based jobs, and outperforms SSP for jobs in the physical heterogeneous cluster. In future work, we plan to extend A-BSP to other ML-specified platforms (e.g., TensorFlow).

ACKNOWLEDGMENTS

We thank our shepherd Luis Veiga and the anonymous reviewers for their feedback. This research was supported in part by U.S. NSF award CNS-1422119.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proc. of OSDI*.

[2] Umut A Acar, Arthur Charguéraud, and Mike Rainey. 2013. Scheduling parallel programs by work stealing with private deques. In *Proc. of ACM PPoPP*.

[3] Alekh Agarwal and John C Duchi. 2011. Distributed delayed stochastic optimization. In *Proc. of NIPS*.

[4] Amr Ahmed, Moahmed Aly, Joseph Gonzalez, Shravan Narayanamurthy, and Alexander J Smola. 2012. Scalable inference in latent variable models. In *Proc. of ACM WSDM*.

[5] Muzaffer Can Altinigneli, Claudia Plant, and Christian Böhm. 2013. Massively parallel expectation maximization using graphics processing units. In *Proc. of ACM SIGKDD*.

[6] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. 2013. Effective Straggler Mitigation: Attack of the Clones. In *Proc. of USENIX NSDI*.

[7] Ganesh Ananthanarayanan, Srikanth Kandula, Albert G Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. 2010. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *Proc. of OSDI*.

[8] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. 2015. Spark sql: Relational data processing in spark. In *Proc. of ACM SIGMOD*.

[9] Jaime G Carbonell, Ryszard S Michalski, and Tom M Mitchell. 1983. An overview of machine learning. In *Machine learning*. Springer, 3–23.

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

[11] Wei Chen, Jia Rao, and Xiaobo Zhou. 2017. Addressing Performance Heterogeneity in MapReduce Clusters with Elastic Tasks. In *Proc. of IEEE IPDPS*.

[12] Dazhao Cheng, Jia Rao, Yanfei Guo, and Xiaobo Zhou. 2014. Improving MapReduce performance in heterogeneous environments with adaptive task tuning. In *Proc. of Middleware*.

[13] James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Gregory R Ganger, Garth Gibson, Kimberly Keeton, and Eric P Xing. 2013. Solving the Straggler Problem with Bounded Staleness. In *Proc. of USENIX HotOS*.

[14] Henggang Cui, Alexey Tumanov, Jinliang Wei, Lianghong Xu, Wei Dai, Jesse Haber-Kucharsky, Qirong Ho, Gregory R Ganger, Phillip B Gibbons, Garth A Gibson, et al. 2014. Exploiting iterative-ness for parallel ML computations. In *Proc. of ACM SoCC*.

[15] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. 2012. Large scale distributed deep networks. In *Proc. of NIPS*.

[16] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* (2008).

[17] Ofer Dekel, Ran Gilad-Bachrach, Ohad Shamir, and Lin Xiao. 2012. Optimal distributed online prediction using mini-batches. *Journal of Machine Learning Research* 13, Jan (2012), 165–202.

[18] James Dinan, D Brian Larkins, Ponnuswamy Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. 2009. Scalable work stealing. In *Proc. of ACM SC*.

[19] Rohan Gandhi, Di Xie, and Y Charlie Hu. 2013. PIKACHU: How to Rebalance Load in Optimizing MapReduce On Heterogeneous Clusters. In *Proc. of USENIX ATC*.

[20] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proc. of OSDI*.

[21] Yanfei Guo, Jia Rao, Changjun Jiang, and Xiaobo Zhou. 2014. FlexSlot: Moving hadoop into the cloud with flexible slot management. In *Proc. of IEEE SC*.

[22] Aaron Harlap, Henggang Cui, Wei Dai, Jinliang Wei, Gregory R Ganger, Phillip B Gibbons, Garth A Gibson, and Eric P Xing. 2016. Addressing the straggler problem for iterative convergent parallel ML. In *Proc. of ACM SoCC*.

[23] Jiawei Jiang, Bin Cui, Ce Zhang, and Lele Yu. 2017. Heterogeneity-aware distributed parameter servers. In *Proc. of ACM SIGMOD*.

[24] Tim Kraska, Ameet Talwalkar, John C Duchi, Rean Griffith, Michael J Franklin, and Michael I Jordan. 2013. MLbase: A Distributed Machine-learning System. In *Proc. of CIDR*.

[25] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. 2012. Skewtune: mitigating skew in mapreduce applications. In *Proc. of ACM SIGMOD*.

[26] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server. In *Proc. of OSDI*.

[27] Mu Li, David G Andersen, Alexander J Smola, and Kai Yu. 2014. Communication efficient distributed machine learning with the parameter server. In *Proc. of NIPS*.

[28] Min Li, Jian Tan, Yandong Wang, Li Zhang, and Valentina Salapura. 2015. Sparkbench: a comprehensive benchmarking suite for in memory data analytic platform spark. In *Proc. of ACM CF*.

[29] Mu Li, Tong Zhang, Yuqiang Chen, and Alexander J Smola. 2014. Efficient mini-batch training for stochastic optimization. In *Proc. of ACM SIGKDD*.

[30] Xiangru Lian, Ce Zhang, Huan Zhang, Cho-Jui Hsieh, Wei Zhang, and Ji Liu. 2017. Can decentralized algorithms outperform centralized algorithms? a case study for decentralized parallel stochastic gradient descent. In *Proc. of NIPS*.

[31] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. 2012. Distributed GraphLab: a framework for machine learning and data mining in the cloud. In *Proc. of VLDB*.

[32] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. 2016. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research* 17, 1 (2016), 1235–1241.

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

[34] Sashank J Reddi, Ahmed Hefny, Suvrit Sra, Barnabas Poczos, and Alex Smola. 2016. Stochastic variance reduction for nonconvex optimization. In *Proc. of ICML*.

[35] Liang Wang, Ben Catterall, and Richard Mortier. 2017. Probabilistic Synchronous Parallel. *arXiv preprint arXiv:1709.07772* (2017).

[36] Shaoqi Wang, Wei Chen, Xiaobo Zhou, Liqiang Zhang, and Yin Wang. 2018. Dependency-aware Network Adaptive Scheduling of Data-Intensive Parallel Jobs. *IEEE Transactions on Parallel and Distributed Systems* (2018).

[37] Zhigang Wang, Lixin Gao, Yu Gu, Yubin Bao, and Ge Yu. 2017. FSP: towards flexible synchronous parallel framework for expectation-maximization based algorithms on cloud. In *Proc. of SoCC*.

[38] Pijika Watcharapichat, Victoria Lopez Morales, Raul Castro Fernandez, and Peter Pietzuch. 2016. Ako: Decentralised deep learning with partial gradient exchange. In *Proc. of SoCC*.

[39] Jinliang Wei, Wei Dai, Aurick Qiao, Qirong Ho, Henggang Cui, Gregory R Ganger, Phillip B Gibbons, Garth A Gibson, and Eric P Xing. 2015. Managed communication and consistency for fast data-parallel iterative analytics. In *Proc. of ACM SoCC*.

[40] Eric P Xing, Qirong Ho, Wei Dai, Jin Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. 2015. Petuum: A new platform for distributed machine learning on big data. *IEEE Transactions on Big Data* 1, 2 (2015), 49–67.

[41] Nikos Zacheilas and Vana Kalogeraki. 2014. Real-Time Scheduling of Skewed MapReduce Jobs in Heterogeneous Environments. In *Proc. of USENIX ICAC*.

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

[43] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized streams: Fault-tolerant streaming computation at scale. In *Proc. of ACM SOSP*.

[44] Matei Zaharia, Andy Konwinski, Anthony D Joseph, Randy H Katz, and Ion Stoica. 2008. Improving MapReduce performance in heterogeneous environments. In *Proc. of OSDI*.