Elastic Distributed Bayesian Collaborative Filtering

Alex Beutel
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
abeutel@cs.cmu.edu

Markus Weimer
Microsoft
Redmond, WA
mweimer@microsoft.com

Tom Minka
Microsoft Research
Cambridge, UK
minka@microsoft.com

Yordan Zaykov
Microsoft Research
Cambridge, UK
yordanz@microsoft.com

Vijay Narayanan
Microsoft
Mountain View, CA
vkn@microsoft.com

Abstract

In this paper, we consider learning a Bayesian collaborative filtering model on a shared cluster of commodity machines. Two main challenges arise: (1) How can we parallelize and distribute Bayesian collaborative filtering? (2) How can our distributed inference system handle elasticity events common in a shared, resource managed cluster, including resource ramp-up, preemption, and stragglers? To parallelize Bayesian inference, we adapt ideas from both matrix factorization partitioning schemes used with stochastic gradient descent and stale synchronous programming used with parameter servers. To handle elasticity events we offer a generalization of previous partitioning schemes that gives increased flexibility during system disruptions. We additionally describe two new scheduling algorithms to dynamically route work at runtime. In our experiments, we compare the effectiveness of both scheduling algorithms and demonstrate their robustness to system failure.

1 Introduction

How can we efficiently learn a Bayesian collaborative filtering model in a commodity, shared cluster? Collaborative filtering models for recommendation have found many applications across industry and academia, with Bayesian collaborative filtering models recently showing improved performance. However, in order to make such models useful, we need to be able to scale the inference of the models to Big Data and to be able to run our inference algorithms in the shared, commodity clusters used in practice.

Shared clusters, either in the form of a public or private cloud, are increasingly common. They enable true multi-tenancy: not only are the resources shared between users, they are also utilized by many different applications such as relational processing, graph analytics or, relevant here, machine learning. By running many applications on the same infrastructure, cluster administrators aim to maximize the utilization of cluster resources, and thus maximize the value of the cluster. Resource Managers are tasked with this challenge; systems like Apache YARN [5], Apache Mesos [8], Facebook Corona or Google Omega facilitate job admission and resource allocation. Individual applications lease *containers* from the Resource Managers, a fraction of memory and CPU cores of a machine.

In order to achieve the high utilization of the cluster as a whole, individual applications are faced with resource elasticity, the two main forms of which are *preemption* and *ramp-up*. When the cluster is under resource pressure, the resource manager may choose to revoke access to individual containers in order to re-assign the containers to higher-priority applications; this is called preemption. Depending on the sophistication of the preempted application, this can be treated just like a container failure or in more intelligent ways. On a well utilized cluster, a large resource request is unlikely to be satisfied immediately.

Instead, the resource manager satisfies it incrementally as resources become available, called ramp-up. If the application can't make use of these partial allocations, the utilization of the cluster suffers.

When building machine learning systems for use on resource managed clusters, these issues need to be addressed. Obviously, they can be dealt with by the underlying distributed runtime like Spark [18] or Pregel [12]. By design, these systems *hide* these elasticity events from the machine learning code by dealing with them in a generic way, often incuring a high cost in the form of increased use of stable storage. As we shall show below, such a fiction isn't always necessary if we can derive algorithms that accept the underlying challenges.

While much of the research in distributed machine learning has assumed a constant number of machines or relied on the underlying system to hide the elasticity events, such approaches are less efficient when run in real-world shared clusters. In this paper, we explore another approach by devising a machine learning algorithm that reacts to these elasticity events directly, without requiring the underlying runtime to hide them. In taking this approach to Bayesian collaborative filtering, we must solve a number of new challenges. In this paper, we offer the following contributions:

1. A generalization of previous partitioning schemes that enables flexible scheduling of blocks under different system constraints, such as elasticity, stragglers, and network latency. It also enables pipelining of the inference in containers and dynamic scheduling of work.
2. Two new scheduling algorithms for work allocation under variable system resources. Both scheduling algorithms gracefully handle variable numbers of machines and load balance work allocation in the face of slower machines.
3. We use this partitioning scheme and scheduling algorithm for Bayesian collaborative filtering.

2 Related Work

2.1 Big Data Processing

Distributed file systems like the Hadoop Distributed Filesystem (HDFS) [5] have made it economical to store large quantities of data for extended periods of time. Variants of MapReduce [4, 16] are often used to analyze this data, and the value derived from it accelerated the overall trend of keeping ever increasing amounts of data with the expectation of eventual value extraction. But MapReduce is not well suited to iterative computations. In response, a number of alternative frameworks have been developed [18], some with fundamentally different programming models [11, 12].

These frameworks are popular in large part because they hide the true chaotic nature of the distributed system. In order to create this fiction, they incur various overheads. For example, they anticipate stragglers and mitigate their impact on execution times via speculative execution: some work is done several times on different containers, which allows the framework to use the results from the first container to complete them. Similarly, elasticity in all its forms (preemption, ramp-up, failures) is anticipated by frequently storing intermediate results to stable storage and rolling back computations to ensure sequential consistency. We show that such effort isn't always necessary for machine learning computations.

2.2 Distributed machine learning

Machine learning computations often involve numerical operations that iterate until convergence. These algorithms are naturally tolerant of faults and communication delays, so we do not need to hide these characteristics of the distributed system. Common algorithms fall into three types: fixed-point algorithms, stochastic approximators (including SGD), and Markov chain Monte Carlo (MCMC). A fixed-point algorithm is defined as having a unique deterministic update for each variable, which are executed in a round-robin fashion. Fixed-point algorithms include belief propagation, expectation propagation, batch gradient descent, and coordinate descent (which itself includes k-means, expectation maximization, alternating least squares, and variational Bayes). When implementing such algorithms, it is not enough to maximize the number of updates processed per CPU hour—you also have to consider the impact on convergence rate and accuracy of the algorithm.

In all three types of algorithm, we have a choice in how to sequentially order the updates. Some of these orderings will have faster convergence rate than others. Some orderings will also have more natural parallelism arising from consecutive updates that have no dependencies on each other. Running such updates in parallel has no effect on the convergence rate or accuracy of the algorithm. Sometimes the orderings with

high convergence rate are also parallel, but sometimes they are not. For example, the update order used by the alternating least squares algorithm for matrix factorization has high parallelism and fast convergence rate. Similarly, the DSGD algorithm [6] orders the SGD updates to provide high parallelism. However, this ordering slows down the convergence rate, so DSGD only beats a sequential implementation when there are sufficiently many threads. GraphLab [11] automatically constructs a sequentially consistent schedule to maximize parallelism.

In cases where there isn't enough natural parallelism, or the orderings with natural parallelism have too low convergence rate, we can run updates in parallel even when they have dependencies [1, 9]. This means that some updates will be using stale values of the variables, so the algorithm will not be equivalent to any sequential ordering. This does not affect the accuracy of a fixed-point algorithm (it doesn't change the set of fixed points) or stochastic approximation (as long as the step size is changed appropriately [10]), but it does affect the accuracy of MCMC, since the stationary distribution will change. Stale values can significantly slow convergence if we are not careful. For example, consider the fixed-point system $(x = f(y), y = g(x))$. If we run both updates in parallel, we can do twice as many updates per second as a sequential implementation, but the convergence rate will be exactly halved, so in the end we achieve nothing.

Another approach to increase the amount of parallelism, as well as achieve load balancing, is to use an unfair schedule, where some updates are done more often than others. This is explicitly prevented in DSGD, but allowed in NOMAD [17] and FPSGD [19]. Fairness does not affect the accuracy of a fixed-point algorithm or MCMC, but it does affect the accuracy of stochastic approximation, since it changes the distribution of data seen by the algorithm. Since NOMAD uses SGD with an unfair schedule, the answer it gives depends on how many updates happen to be done by each worker. Unfairness can also negatively affect convergence rate. A simple example is if we update a variable (without a self-loop) in a fixed-point system, and then immediately update that variable again. Because the updates are deterministic, the same values are computed both times. A good schedule should avoid this sort of wheel spinning.

The distributed algorithm presented in this paper is based on a fixed-point algorithm (expectation propagation) and combines all of these techniques to achieve maximum parallelism.

3 Parallel Bayesian Collaborative Filtering

3.1 The Matchbox model

In this paper we set out to perform fast, scalable, distributed inference of collaborative filtering models. Unlike most of the previous work in scalable collaborative filtering, we focus on Bayesian collaborative filtering models, such as Matchbox [15], TrueSkill [7], Bayesian Probabilistic Matrix Factorization [14], CoBaFi [2]. All of the above graphical models attempt to decompose a matrix into its latent factors.

We begin by giving a high-level description of the Matchbox model, which we will use for our explanation and testing; a more detail description of the model can be found in [15]. The model considers a dataset of N users, M movies, and R ratings with discrete values ranging from 1 to L; we can consider this dataset to be a sparse $N \times M$ matrix $\mathbf{X}$ where $x_{i,j}$ is the rating by user i of movie j. In the simplest version of the model, each user i has a vector $\mathbf{u}_i$ of latent traits and a vector $\mathbf{t}_i$ of thresholds, each movie j has a vector of traits $\mathbf{v}_j$, and all of these latent variables have Gaussian priors. A rating is assumed to have come from a real-valued affinity $a_{i,j} = \langle \mathbf{u}_i, \mathbf{v}_j \rangle + noise$ that was turned into a rating via the smallest index x such that $t_{i,x} > a_{i,j}$.

This defines a probabilistic model of user ratings, but not how to do infer the latent traits from data. To do that, we apply the expectation propagation (EP) algorithm. In this algorithm, each observed rating sends a message (in the form of a Gaussian distribution) to each parameter involved in generating that rating. These messages form a fixed-point system that needs to be iterated until convergence.

3.2 Partitioned Bayesian inference

The fixed-point updates in EP exhibit natural parallelism, since the updates involving an observed rating $x_{i,j}$ have no interaction with the updates for $x_{i',j'}$ where $i \neq i'$ and $j \neq j'$. To exploit this, we partition our N users and M movies into K sets, inducing a partition of the ratings matrix into $K \times K$ blocks. As in DSGD [6], the blocks can be grouped into strata of size K, where no two blocks share the same users or movies. This allows us to run updates on each block within a stratum simultaneously without interference.

When running inference on a block, we need the data for that block, the current posterior distributions for the corresponding latent variables, and the most recent messages going into those variables. This information is

enough to continue the algorithm just as though we were following a serial schedule. Therefore, the general process for each block is to (1) load the data and distributions relevant to the block; (2) run an iteration of Bayesian inference on the block; (3) save the updated distributions for use later.

3.3 Asynchronously merging shared parameters

The above algorithm can handle models where the entire latent parameter space can be partitioned. However, in many Bayesian models, there is a set of parameters that cannot be partitioned. In collaborative filtering, this arises when we want to make use of metadata associated with users and items. In the Matchbox model, such metadata is incorporated into the affinity via an additional set of parameters, shared across all users and items. (Suppose each item also has a vector of metadata $\mathbf{f}_j$. This is incorporated into the rating model by adding $\langle \mathbf{u}_i, \mathbf{W}\mathbf{f}_j \rangle$ to the affinity, where $\mathbf{W}$ is a matrix of shared parameters.) Since these parameters are involved in generating every rating, updates performed on any block of the data matrix affect all other blocks.

To achieve parallelism in this situation, we must allow some updates to use stale values of the shared parameters. In EP, these updates correspond to multiplication of distributions, which is an associative and commutative operation. Thus we follow the approach of [1, 9] and allow workers to update their own local copy of the shared parameters. When a block is completed, these updates are batched up and sent to a central parameter server, where they can be applied in any order. Because our shared parameter space is small relative to the size of the data and total parameter space, we store the shared parameters in a single machine and we do not impose a bound on staleness. Whenever a new block is to be processed, the most recent version of the shared parameters is obtained from the server.

4 System Design

The target environment is a resource managed, shared cluster (or: cloud). It exposes computational resources in the form of *containers* which are fractions of machines. In this environment, it is natural to execute the system by assigning blocks to a set of worker containers according to some scheduling policy. The computing environment provides the following constraints when defining the partitioning and scheduling policies:

Scale: The system is composed of multiple containers that do not share memory. Hence, we need to pay special attention to the cost and latency of communication between containers.

Fault tolerance: Containers may fail without warning due to a local hardware failure or a communications failure.

Elasticity: The resource manager may request the system to vacate some of the containers. This is different from a failure since there is prior warning.

Ramp-Up: Requested resources may become available a few containers at a time. The time for all requested containers to become available may be a significant fraction of the effective compute time of the job.

Stragglers: Some containers may be temporarily or even permanently slower than other containers. Like failure, this behavior is often induced by local hardware or software issues of the machine that hosts the container.

This section describes how to address these challenges within the machine learning algorithm, via an appropriate policy for partitioning and scheduling.

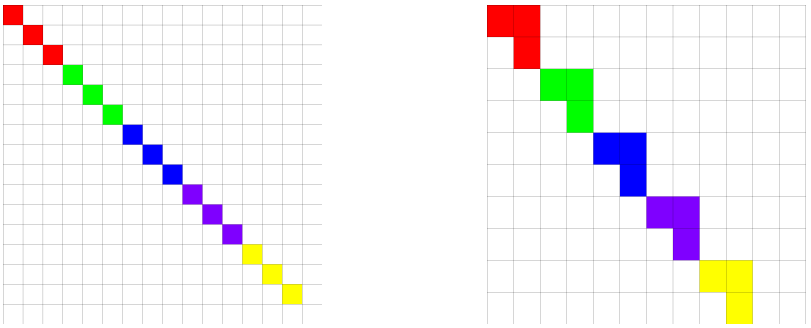
4.1 Little blocks for asynchronous, pipelined processing

Partitioning For our partitioning, we assume that during computation we will use *at most* p containers during inference. We start with a data matrix X of size N by M, and we will partition the rows into k_N clusters and the columns into k_M clusters resulting in $k_N k_M$ blocks. (For simplicity, we assume k_N and k_M evenly divide N and M respectively.) Setting $k_N = k_M = p$ results in the partitioning scheme of DSGD [6]; $k_N = k_M = p+1$ results in the partitioning scheme of FPSGD [19]; and $k_N = p$ and $k_M = qp$ for some large integer q results in the partitioning scheme of NOMAD [17]. The additional blocks allow asynchronous processing.

Here, we set $k_N \geq \ell p + 1$, where ℓ is an integer representing the average number of rows of blocks each container would be assigned if all p containers were being used. Additionally, we set $k_M \geq qp + 1$ where

Input: *blockID* to process
Load data for *blockID* from distributed disk: load data and relevant parameter space;
Run one EP iteration on *blockID*;
Save updated parameters to distributed disk and then notify the master that *blockID* was processed;

Algorithm 1: High level algorithm run by worker containers when receiving a block to process.



(a) Simple scheduling (b) Locality-aware scheduling

Figure 1: Above we give examples of $B = 3$ block allocations to $p = 5$ workers (designated by each color) under both the simple scheduler and locality-aware scheduler. (a) For the simple scheduler we see that all blocks allocated are independent and there is a free row and column for when a container finishes one of its blocks. (b) For the locality-aware scheduler we see each container receives more than one block per row and per column, allowing for larger blocks for the same values of B and p.

q is an integer loosely representing the queue length such that each container can pipeline their blocks. In practice we will always set $q > 1$ and $\ell > 1$ so that we can achieve all of the necessary features.

Block processing Each worker container is continually allocated blocks to process. The algorithm run by each worker container is shown in Algorithm 1. We use Infer.NET [13], a probabilistic programming language that compiles to an expectation propagation (EP) algorithm, to run updates on our model. The compiled algorithm from Infer.NET offers an API to load data and parameters into the model, run a single iteration of EP, and save the updated parameters, enabling us to implement Algorithm 1.

Pipelining For collaborative filtering models, and especially for complex graphical models, there is a sizable parameter space along with the data that must be loaded and subsequently saved when processing each block. By setting $q > 1$ or $\ell > 1$ we are able to schedule multiple blocks at a time to each container.

Because each container can lock multiple blocks at once, they can pipeline their work. In particular, we create a thread in the container for each of the three steps in Algorithm 1. Each thread has a queue of blocks to process, runs its step and passes the block to the appropriate queue for the next thread. The result is that we have two threads primarily performing network communication and a third performing CPU intensive updates. Our goal is that each container is always running updates on a block and to make this possible we need to carefully tune the pipeline.

4.2 Centralized dynamic block scheduling

The partitioning scheme above offers great flexibility but does not specify how to efficiently take advantage of the parallelism. Unlike previous work, we do not assume there is a fixed schedule. Rather, we have a centralized master container that keeps track of all locks and dynamically allocates blocks to containers as it feels appropriate. As we show below, there are numerous design decisions in making a dynamic block schedule for collaborative filtering, and we believe this opens the door to further research toward improved efficiency. Here, we describe a simple dynamic block scheduler, similar to that offered in [19], and a second locality-aware scheduling heuristic. In both algorithms, we would like there to be at most B blocks allocated to each container at a given time. The master keeps a queue of blocks that need to be processed, ordered following the DSGD [6] schedule; we allocate blocks from the first f blocks at the front of the queue following the algorithms below.

Input: CID: ID of container needing a new block
initialize *blockID* to NULL;
Set *blockID* to a free block from a row that has not yet been assigned to another CID; if none keep NULL;
if (*blockID* = NULL) Set *blockID* to any available block from the rows and columns already locked by CID; if none keep NULL;
if (*blockID* = NULL) Set *blockID* to any available block from the rows already locked by CID, prioritizing rows that have had fewer blocks processed; if none keep NULL;
if (*blockID* = NULL) *Soft steal* row from a container that have multiple rows that are significantly behind the rows allocated to CID; if none keep NULL;
if *blockID* is not NULL **then**
  Lock the row and column of the block.
end
Output: *blockID*

Algorithm 2: Locality-aware block scheduler

Simple scheduling Following [19], we take advantage of the fact that as long as there is at least one additional unlocked row and column at any given moment we can allocate a new block to an available container. Therefore, we set $q = \ell = B$ and set $k_N = \ell p + 1$ and $k_M = qp + 1$. Doing this, as seen in Figure 1(a), we can allocate B blocks to each container and still always have an unlocked row and column. As a result, whenever a container finishes processing a block, the scheduler will unlock the row and column from the previous block and the container will be allocated a new block from the previously unlocked row or column.

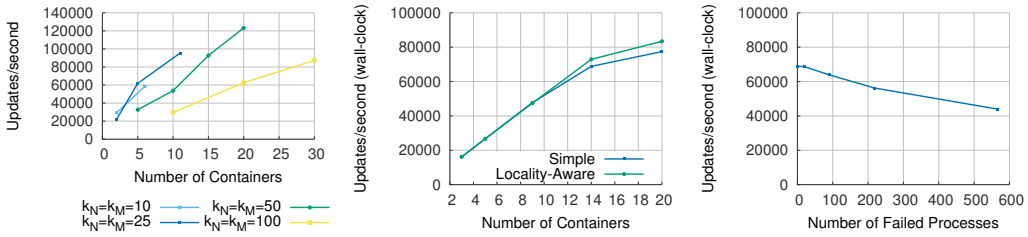
Locality-aware scheduling While the above scheduling algorithm works well, it suffers from two key issues. First, all blocks in a container's pipeline are independent and containers often do not get blocks from the same row or column as they had recently used, resulting in increased time necessary to read remote parameters. Second, the schedule wastes available parallelism, resulting in the size of blocks dropping quadratically when the number of blocks allocated to each container, B, increases linearly.

To address these issues, we observe that while a container has a row and column locked, other blocks in that row or column could be added to that container's pipeline without breaking the lock, as seen in Figure 1(b). For the sake of simplicity, we consider that each container is allocated a set of row partitions and are primarily allocating blocks from those rows. As such, we set ℓ to the number of rows we want each container to be allocated and $q \geq B/\ell$. We then follow the algorithm listed in Algorithm 2 to select which block should be allocated next to a particular container. There we refer to "soft-stealing" a row; this is a mechanism for load balancing. If there are no blocks that can be allocated to a given container CID_{fast}, since we only look ahead f blocks in the scheduling queue, and there is another container $CID_{\text{straggler}}$ straggling significantly behind the free container, then we set one of the rows from the $CID_{\text{straggler}}$ to be locked by CID_{fast}. As a result, $CID_{\text{straggler}}$ will in the future be responsible for processing blocks from one less row and CID_{fast} can take on blocks from an additional row once $CID_{\text{straggler}}$ has finished those blocks it is already processing.

In both scheduling algorithms above, we are able to achieve pipelined, asynchronous processing of blocks. While the second, dynamic algorithm offers some improvements over the first, it comes at the cost of added complexity. Given it is generally governed by heuristics, we believe additional research can unearth improved dynamic scheduling algorithms for the given partition scheme. We hope the proposed partitioning perspective opens the door for other researchers to explore this space.

4.3 Elasticity and fault tolerance

The above partitioning scheme and scheduling algorithm inherently handle elastic resource availability. Because no worker container knows of the others' activities, they can merely operate on the blocks assigned to them. When a container becomes available, the master allocates the next available block to it, "stealing" a row if necessary. If a container fails, the previous values of parameters are still stored on the distributed filesystem, blocks that were previously being processed by the container are inserted at the front of the scheduling queue, and any locks associated with that container are released. This enables the implementation to run inference with a variable number of containers and smoothly handle resource availability and machine failure.



(a) Block size (b) Scheduling (c) Elasticity

Figure 2: We test the performance of our system across a variety of settings. (a) We observe that larger blocks are faster to process but smaller blocks offer more parallelism. (b) We observe that both scheduling algorithms scale linearly to additional containers, with the locality-aware scheduler being able to more efficiently use higher numbers of containers. (c) Our system is highly fault tolerant, with many container failures causing only a slight decrease in performance.

5 Experiments

5.1 Experimental Setup

We begin by explaining how we implemented the above algorithms and how we have tested them.

Implementation Our system is built on top of REEF, the retainable evaluator execution framework [3]. REEF allocates a master container, called the driver, which is notified of all elasticity events including newly allocated containers, preempted containers, and failed processes in containers. Within each worker container we run Algorithm 1 and communicate progress directly to the master container. As mentioned before, the expectation propagation algorithm for the Matchbox model is generated by Infer.NET.

Cluster environment All experiments, unless specified otherwise, are run on an HDInsight[1] cluster with 3GB of main memory per container. We use a distributed file system, in this case Azure Storage, to save and communicate data and parameters.

Data Generation To provide a flexible testing framework, all of our data is generated by Infer.NET using the Matchbox generative model with randomly initialized parameters. For each test we specify the number of users N, number of movies M, number of ratings R, and number of partitions k_N, k_M. This enables us to test the success of our system under different constraints and work loads. Unless noted otherwise, we set $N = 500000$, $M = 20000$, $R = 1000000000$, $B = 3$ and use the simple scheduler. To test performance of our system under different constraints, we track the number of updates completed per second (wall-clock time) for two iterations (two complete passes over the data).

5.2 Machine scalability

The most important aspect we wanted to test was scalability. To do this, we first look at how well we can scale by adding additional containers to a problem of constant size. Even for a constant size problem, there are two free variables that we test: block size and scheduling algorithm.

To test the effect of block size, we vary k_N and k_M from 10 to 100 and observe the performance for different numbers of containers using the simple scheduler. (Note, this experiment was run on a private cloud at Microsoft running Hadoop YARN, HDFS and faster containers than HDInsight used in other experiments.) As can be seen in Figure 2(a), we find that for a set block size, our system scales linearly in the resources available. Additionally, we find that large blocks are faster to process, although of course they offer less parallelism than many small blocks. As a result, like all partitioning-based parallel matrix factorization methods, this trade-off results in an optimal block size given the number of available containers.

As noted previously, while the partitioning scheme offers the parallelism, it is up to the scheduling algorithm to exploit it. As a result, to maximally use the resources available, we compare the ability of each scheduling algorithm to use additional containers. In this experiment, we set $k_N = k_M = 30$ and $B = 3$. As a result,

[1] We used HDInsight version 3.1, `http://azure.microsoft.com/en-us/services/hdinsight/`

the simple scheduler works optimally (non-blocking and fully pipelined) for up to 9 containers. However, the locality-aware scheduler can maintain the full pipelining for up to 14 containers. As we see in Figure 2(b), the locality-aware scheduler continues to scale linearly to 14 containers, using additional containers just as efficiently as when there is less contention for blocks. For even greater numbers of containers, we observe that the locality-aware scheduler continues to scale better than the simple scheduler, which cannot as effectively make use of additional resources. This experiment is also reassuring that although the locality-aware scheduling is more complex, the complexity does not result in any overhead or loss of performance when applied to simpler situations, i.e. far fewer containers and many blocks.

5.3 Elasticity Events

To test the ability of our system to handle elasticity events, we cause our program in each worker container to fail with probability P whenever a block is allocated to it; when the program fails, REEF notifies the master container and the system quickly starts a new instance of the runtime in the container. We vary $P \in \{0, 0.01, 0.05, 0.1, 0.2\}$ and observe the change in performance of the system. Note that, because upon program failure the blocks that were previously being processed have to be reallocated, we see that the number of container failures grows faster than P, the probability of failure with each block allocation.

As can be seen in Figure 2(c), the system performance degrades at a much slower rate than the increased rate of failures. This demonstrates that not only can the system handle preemption and failures, but such events, even in large quantities, do not significantly hurt performance. As explained before, this result is in contrast to systems that either have to start the entire job over or duplicate work across multiple machines.

Discussion These preliminary experiments offer a number of conclusions and provide further questions that we plan to explore. From Figure 2(c), it is clear that our system successfully handles elasticity events with marginal impact on performance. Our test comparing the two scheduling algorithms in Figure 2(b) shows that the dynamic, locality-aware scheduling can improve the scalability of partitioning based systems Last, Figure 2(a) demonstrates that the question of how to set the block size appropriately is valuable for optimizing performance. We plan to investigate these questions further and run additional experiments in the coming months.

6 Conclusion

Above, we consider the problem of learning Bayesian Matrix Factorization models on shared, resource managed clusters. Different from the literature on distributed machine learning, we accept the challenge of that system environment as one of the machine learning algorithm, not the underlying runtime. Namely, we pay attention to resource elasticity events such as stragglers, preemption and ramp-up.

In the experiments we have seen that our system scales well to additional resources, and the locality-aware scheduling helps to push the scalability further. Additionally, our efforts to handle elasticity events directly lets our system quickly recover from significant failures without significantly hurting performance.

In the future, we plan on expanding this work in several dimensions. For one, we will perform more experiments on much larger clusters and datasets. Further, we plan to perform a more in-depth analysis of the proposed scheduling algorithm as well as partitioning scheme for convergence rates. Lastly, we will expand the approach presented to additional models.

References

[1] A. Ahmed, Mohamed Aly, Joseph Gonzalez, Shravan Narayanamurthy, and A. J. Smola. Scalable inference in latent variable models. In *Proceedings of The 5th ACM International Conference on Web Search and Data Mining (WSDM)*, 2012.

[2] Alex Beutel, Kenton Murray, Christos Faloutsos, and Alexander J Smola. CoBaFi: collaborative bayesian filtering. In *Proceedings of the 23rd international conference on World wide web*, pages 97–108. International World Wide Web Conferences Steering Committee, 2014.

[3] Byung-Gon Chun, Tyson Condie, Carlo Curino, Chris Douglas, Sergiy Matusevych, Brandon Myers, Shravan Narayanamurthy, Raghu Ramakrishnan, Sriram Rao, Josh Rosen, et al. Reef: Retainable evaluator execution framework. *Proceedings of the VLDB Endowment*, 6(12):1370–1373, 2013.

[4] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *CACM*, 51(1):107–113, 2008.

[5] The Apache Software Foundation. Apache Hadoop, 2009. http://hadoop.apache.org/core/.

[6] R. Gemulla, E. Nijkamp, P.J. Haas, and Y. Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 69–77. ACM, 2011.

[7] Ralf Herbrich, Tom Minka, and Thore Graepel. Trueskill^{TM}: A Bayesian skill ranking system. In *NIPS*, 2007.

[8] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, pages 22–22, 2011.

[9] Q. Ho, J. Cipar, H. Cui, S. Lee, J. Kim, P. Gibbons, G. Gibson, G. Ganger, and E. Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *NIPS*, 2013.

[10] J. Langford, A. J. Smola, and M. Zinkevich. Slow learners are fast. In *Neural Information Processing Systems*, 2009.

[11] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. GraphLab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence*, 2010.

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

[13] Tom Minka, John Winn, John Guiver, and David Knowles. Infer.NET 2.5, Microsoft Research Cambridge, 2012.

[14] Ruslan Salakhutdinov and Andriy Mnih. Bayesian probabilistic matrix factorization using markov chain monte carlo. In *Proceedings of the 25th international conference on Machine learning*, pages 880–887. ACM, 2008.

[15] David H Stern, Ralf Herbrich, and Thore Graepel. Matchbox: large scale online bayesian recommendations. In *Proceedings of the 18th international conference on World wide web*, pages 111–120. ACM, 2009.

[16] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 1–14, 2008.

[17] Hyokun Yun, Hsiang-Fu Yu, Cho-Jui Hsieh, SVN Vishwanathan, and Inderjit Dhillon. Nomad: Non-locking, stochastic multi-machine algorithm for asynchronous and decentralized matrix completion. *arXiv preprint arXiv:1312.0193*, 2013.

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

[19] Yong Zhuang, Wei-Sheng Chin, Yu-Chin Juan, and Chih-Jen Lin. A fast parallel sgd for matrix factorization in shared memory systems. In *Proceedings of the 7th ACM Conference on Recommender Systems*, pages 249–256. ACM, 2013.