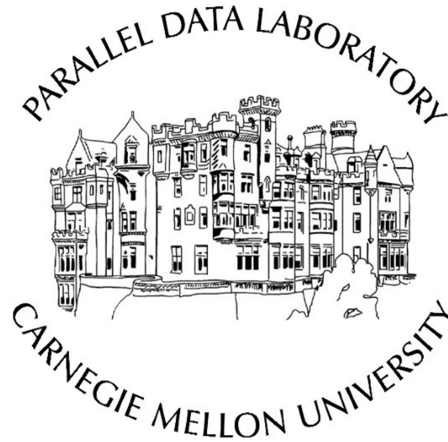


Benchmarking Apache Spark with Machine Learning Applications

Jinliang Wei, Jin Kyu Kim, Garth A. Gibson

CMU-PDL-16-107
October 2016

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

We benchmarked Apache Spark with a popular parallel machine learning training application, Distributed Stochastic Gradient Descent for Matrix Factorization [5] and compared the Spark implementation with alternative approaches for communicating model parameters, such as scheduled pipelining using POSIX socket or MPI, and distributed shared memory (e.g. parameter server [13]). We found that Spark performance suffers substantial overhead with only modest model size (rank of a few hundreds). For example, the PySpark implementation using one single-core executor was about $3\times$ *slower than a serial out-of-core Python implemenation and* $226\times$ *slower than a serial C++ implementation. With a modest dataset (Netflix dataset containing 100 million ratings), the PySpark implementation showed* $5.5\times$ *speedup from* 1 *to* 8 *machines, using* 1 *core per machine. But it failed to achieve further speedup with more machines or gain speedup from using more cores per machine. While it's still ongoing investigation, we believed that shuffling as Spark's only scalable mechanism of propagating model updates is responsible for much of the overhead and more efficient approaches for communication could lead to much better performance.*

Acknowledgements: We thank the member companies of the PDL Consortium (Broadcom, Citadel, Dell EMC, Facebook, Google, HewlettPackard Labs, Hitachi, Intel, Microsoft Research, MongoDB, NetApp, Oracle, Samsung, Seagate Technology, Tintri, Two Sigma, Uber, Veritas, Western Digital) for their interest, insights, feedback, and support.

Keywords: distributed systems, machine learning, Apache Spark

1 Introduction

Apache Spark [14, 2] is a distributed computing system that implements the map-reduce [4] programming model and is attracting wide attention for Big Data processing. Spark achieves high throughput compared to the previous dominating open-source map-reduce implementation Apache Hadoop [1] by retaining data in main memory whenever possible and possibly through better implementation of its operations (such as reusing JVM across jobs, etc). Moreover, it allows application developers to program in a declarative language by employing a DAG scheduler that executes the inferred execution plan. It is said that the DAG scheduler greatly improves the programmer productivity as it relieves programmers from maintaining the complex dependencies among different tasks and scheduling them.

Machine Learning (ML) training is becoming an increasingly popular kind of applications for distributed computing. Such applications are featured by iterativeness and bounded error tolerance, where the algorithm repeatedly processes training data to refine the model parameters until an acceptable solution is found given properly bounded errors. The distrubuted training process typically involves distributed workers computing and applying delta updates to model parameters and reading updated parameter values, and thus may incur high volume of network communication. A number of techniques have been proposed for the parameter server system to minimize network communication overhead [7, 9] or make better use of the limited network bandwdith [13] by taking advantage of the applications' ML-specific properties. It has also been proposed to carefully schedule the update computation with respect to the dependences in parameter accesses so frequent, random parameter accesses are turned into pipelined bulk communication without violating any dependence [8].

Spark has been perceived as a suitable choice for iterative algorithms, including ML training, as it avoids the costly disk I/O between iterations employed by previous map-reduce systems. Moreover, its declarative programming language and DAG scheduler may allow faster application development. However, as Spark does not apply any ML-specific optimizations mentioned above, it remains a question whether Spark may achieve comparable throughput compared to specialized distributed algorithm implementations or ML-oriented systems. In order to explore this question, we implemented a well-known parallel ML algorithm – Distributed Stochastic Gradient Descent for Matrix Factorization [5] on Spark and compared it with alternative implementations. We found that our PySpark implementation suffers significant runtime penalty (226× slower than our specialized implementation in C++) and scales poorly to larger number of machines or larger datasets.

This report presents an exploratory comparison. It is intended to set forth a problem for furture work to improve.

2 Background

2.1 Apache Spark

Spark organizes data into *Resilient Distributed Datasets* (RDD). An RDD is a read-only, partitioned collection of records. RDDs can only be created ("written") through deterministic, coarse-grained operations on either 1) data in persistent storage or 2) other RDDs [15]. Such operations are referred to as *transformations*, which include *map*, *filter* and *join*. Spark maintains the lineage of each RDD, i.e. how the RDD is created from data in persistent storage, so Spark doesn't need to materialize the RDDs immediately and any missing RDDs may be created through the deterministic operations according to its lineage. Another kind of RDD operations are referred to as *actions*, which trigger computation that returns results to the Spark program. Examples include *count* and *reduce*.

Spark application developers write a *driver program* that defines RDDs and invokes operations on them. The Spark runtime consists of a *master* and many *workers* (i.e. *executors*). The driver program is executed by

the master, which commands the workers to execute the RDD operations accordingly. RDD transformations are lazily evaluated, i.e. they are not executed when they are invoked. Instead, the RDDs' lineage graphs are recorded and the transformations are executed only when they are needed for computing return values to the application program (i.e. actions).

There may be two types of RDD dependencies: *narrow dependencies*, where each parent RDD partition is needed by at most one child RDD partition; and *wide dependencies*, where multiple child partitions may depend on it. Narrow dependecies (e.g. map) allow pipelined execution on one cluster node; while wide dependencies (e.g. join) depend on all parent partitions and thus require shuffling. A spark scheduler groups as many pipelined transformations as possible into one *stage*. The boundary of each stage is shuffle operation. When an action is run, the scheduler builds a DAG of stages to execute from the RDD's lineage graph.

Spark supports two mechanisms for communicating values to workers, which may be used when executing RDD operations. Firstly, the driver program may broadcast the value of a local variable to workers as a read-only copy. Such variables are referred to as *broadcast variables*. Additionally, the *collect* operation on an RDD creates a local variable in the driver program that consists of all of its records. Together *collect* and *broadcast* allow the driver to intervene and define a communication mechanism. Secondly, two RDDs may be joined by a field in their records as key. Joined RDDs communicate information between workers without driver intervention by joining records from other RDDs that are in different machines, then partitioning the new RDD to cause information that used to be in different machines to land in the same machine.

2.2 Machine Learning Traning

In Machine Learning, training is the process of finding the parameters to an expert-suggested model that best represents some dataset, which is typically referred to as the *training data*. The training process typically starts with a set of randomly initialized parameters and optimizes them using an algorithm that iteratively processes the training data to generate refinements to the model parameters, until an acceptable solution is found according to some *stopping criteria*. In this work, we focus on batch training where the training dataset remains unchanged during training.

Despite the large variety of models and training algorithms, the iterative step can usually be summarized by the following equation:

$$A^{(t)} = A^{(t-1)} + \Delta(A^{(t-1)}, \mathscr{D}), \tag{1}$$

where $A^{(t)}$ is the state of the model parameters at iteration t, and $\mathscr{D}$ is all training data, and the update function $\Delta()$ computes the model updates from data, which are added to form the model state of the next iteration.

Due to the complexity of a training algorithm, programmers may resort to a distributed system for its computing power even if the training data and model parameters are not particularly large (relative to the memory size of a machine). As computing the delta updates is usually the most computation-intensive step, it needs to be efficiently distributed allowing each cluster node compute a fraction of the updates. The major challenge in distributing the update computation comes from data dependence - computing the update for a parameter requires reading values of other parameters from the last iteration. In a distributed cluster, it implies that the updates computed by one node need to be communicated to potentially many other nodes before they can proceed to compute the next iteration. As the size of model grows, frequent communication of updates quickly becomes the major bottleneck. We describe how such communication can be handled with a concrete example of distributed stochastic gradient descent for Matrix Factorization.

3 Distributed Stochastic Gradient Descent for Matrix Factorization

3.1 Parallelizing Stochastic Gradient Descent for Matrix Factorization

Recommender systems predict users' interests based on their existing feedback, often represented as an incomplete *rating matrix*. For example, in the "Netflix problem" [3], the matrix entries are user movie ratings, whose rows correspond to users and columns to movies. The matrix factorization model factorizes the rating matrix into a *user matrix* and an *item matrix* based on the known entries. Then the missing entries can be predicted by multiplying the two factor matrices.

The matrix factorization problem can be stated as follows. Given a $m \times n$ matrix $\boldsymbol{V}$ and a rank k, find an $m \times k$ matrix $\boldsymbol{W}$ and a $k \times n$ matrix $\boldsymbol{H}$ such that $\boldsymbol{V} \approx \boldsymbol{W} \times \boldsymbol{H}$, where the quality of the approximation is defined by a loss function L [5]. In this work, we used the commonly used nonzero squared loss $L = \sum_{i,j:V_{ij} \neq 0} (V_{ij} - [WH]_{ij})^2$. Our goal is thus to find the matrices $\boldsymbol{W}$ and $\boldsymbol{H}$ that minimize the loss function. There exists serveral optimization algorithms that can solve this problem, among which a popular choice is *stochastic gradient descent* due to its relatively low computation cost.

Algorithm 1 Stochastic Gradient Descent for Matrix Factorization

1: Require: A training set $\boldsymbol{V}$, initial values W_0 and H_0
2: **while** not converged **do**
3: select an entry $V_{ij} \in \boldsymbol{V}$ uniformly at random
4: $W'_{i*} \to W_{i*} - \varepsilon_n \frac{\varphi}{\varphi W_{i*}} l(V_{ij}, W_{i*}, H_{*j})$
5: $H_{*j} \to H_{*j} - \varepsilon_n \frac{\varphi}{\varphi H_{*j}} l(V_{ij}, W_{i*}, H_{*j})$
6: $W_{i*} \to W'_{i*}$
7: **end while**

The serial stochastic gradient descent algorithm is described in Algorithm 1. The two factor matrices are randomly initialized. Then for a randomly selected entry V_{ij}, the algorithm reads the corresponding row W_{i*} and column H_{*j} to compute the updates that are then applied to them.

Based on the observation that two data points $V_{i_1 j_1}$, $V_{i_2 j_2}$ are interchangeable (i.e. they can be processed in any serial order) if they share either row or column, i.e. $i_1 \neq i_2$ and $j_1 \neq j_2$, Gemulla et. al. proposed a parallel algorithm that can be efficiently distributed [5]. The algorithm firstly partitions the rating matrix by both row and column. For example, a 3×3 partitioned rating matrix can be seen in Fig. 1. A full pass over the training data for this example takes 3 *sub-epochs*, and in each sub-epoch a stratum is processed. Since entries from different blocks in a stratum do not share neither row nor column, those blocks can be processed in parallel while entries from the same block are processed in serial. Thus the above partitioning creates a program with parallelism of degree 3. Same as the serial algorithm, the algorithm makes passes over the training data repeatedly until it converges.

3.2 Communicating Model Parameters

Given a cluster of P workers, the rating matrix can be partitioned into $P \times P$ blocks. In each sub-epoch, P blocks can be processed in parallel by the P workers. The W and H matrices are both P-way partitioned. Processesing rating matrix block (i, j) reads and writes the i-th and j-th blocks of matrix W and H respectively. After a rating matrix block is processed, at least one of the updated parameter blocks (preferrably the smaller one) need to be communicated to another worker before it proceeds to the next sub-epoch. Here we discuss four strategies of communicating the updated model parameters, two of which can be realized in Spark. Without losing generality, we assume H is smaller in size and thus preferrable for communication.

1,1	1,2	1,3
2,1	2,2	2,3
3,1	3,2	3,3

(a) sub-epoch 1

1,1	1,2	1,3
2,1	2,2	2,3
3,1	3,2	3,3

(b) sub-epoch 2

1,1	1,2	1,3
2,1	2,2	2,3
3,1	3,2	3,3

(c) sub-epoch 3

Figure 1: A 3×3 partitioned rating matrix. In each sub-epoch, the algorithm processes one stratum (the colored blocks). The blocks in the same stratum can be processed in parallel while entries in each block are processed in serial.

Initially the p-th worker is assigned the rating blocks $(p,1)$, $(p,2)$, ..., (p,P), as well as the p-th block of W. In our Spark implementation, this is achieved by joining the RDD of the partitioned rating matrix V and the RDD of matrix W.

Broadcasting. We can let the Spark driver program construct the H matrix as a local variable and then broadcast it to workers. At the end of a sub-epoch, the driver program may retrieve the updated values of H by invoking the *collect* operation on the RDD that contains those variables. They can then be broadcasted for the next sub-epoch. While simple and straightforward, this approach fails to scale to large number of parameters or workers. The driver node needs to be able to hold at least 1 copy of the H matrix, which may contain billions of parameters in a modest problem. Moreover, the driver program sends a full copy of the H matrix to each worker even though they only need $\frac{1}{P}$ of H, wasting a substantial amount of network bandwidth. Since we seek a solution that can potentially scale to large number of model parameters without having to be restricted by a single machine, broadcasting is rejected.

RDD Join. The H matrix can be stored as an RDD so it's not restricted by the size of any single machine. The H RDD can be joined with the RDD that contains the partitioned V and W matrices to have the H values available for processing corresponding V blocks. Since a worker p uses a different partition of H in each sub-epoch, one join operation is needed per sub-epoch. Recall that creating a P-way parallelism requires partitioning the rating matrix into $P \times P$ blocks and takes P sub-epochs for a full pass of the training data. Thus higher degree of parallelism (utilizing more workers) causes more joins. Joining two RDDs typically involves shuffling.

Pipelining. Since processing block (i, j) of the rating matrix V always uses the i-th block of W and the j-th block of H, with a proper static schedule, i.e. static mappings from rating matrix blocks to workers, the computation and communication can be pipelined. For example, with the strata computation schedule shown in Fig. 1, the execution can be pipelined as in Fig. 2.

Distributed Shared Memory. The matrix H could be stored and served by distributed shared memory, such as used in parameter servers [13, 9]. While this approach is similar to broadcasting, it does not suffer the scalability and redundant communication issues as experienced by Spark broadcasting. Firstly, the server could be distributed so it is not restricted by the capability of a single node. Secondly, each worker only has to fetch the parameters needed for the next sub-epoch instead of the whole H matrix.

4 Evaluation and Results

In order to understand Spark's performance with proper context, we compare the following implementations:

Python-Serial or simply **Serial**: a serial implementation in Python (v3.4.0) of the Stochastic Gradient

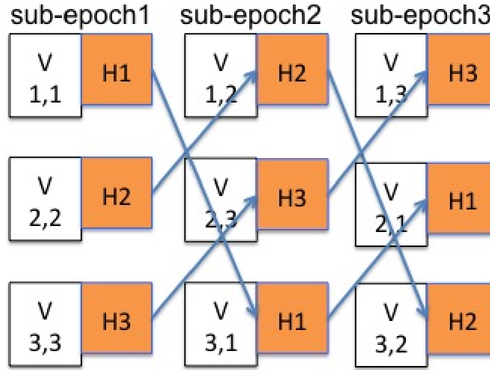


Figure 2: The pipelined execution of the distributed SGD algorithm. Arrows represent the direction of communication. The W matrix is omitted as it does not affect scheduling and is not communicated.

Descent algorithm with Gemulla's rating matrix partitioning strategy [5]. It stores the rating matrix and parameter matrices on disk, loads blocks into memory when they are needed and writes them back to disk before loading data for the next sub-epoch.

Python-Spark or simply **Spark** is implemented on PySpark (v1.6.0) and uses RDD joins for communicating the model parameters.

Pipelined-Standalone or simply **Standalone** implements the scheduled pipelining communication strategy in C++ based on Gemulla's algorithm [5] using the POSIX socket interface for communication.

Pipelined-MPI or simply **MPI** implements the scheduled pipelining communication strategy in Python using MPI4Python (v1.3 and MPI version 1.4.3) for communication.

Bösen is an parameter server that provides a distributed shared memory implementation for storing model parameters. Note that this is not an exact implementation of the distributed shared memory strategy mentioned in Section 3.2 as 1) it partitions the rating matrix by row rather than by row and column (so different workers may concurrently read and update the same parameters); 2) the parameter updates are not propagated to other workers until the entire local data partition has been swept through (i.e. no notion of sub-epochs).

STRADS is a framework that supports a Scheduled Model Parallel programming pradigm [8]. The static model-parallel SGD-MF implementation of STRADS is effectively the same partitioning and scheduling strategy as Gemulla's algorithm [5].

As we are mostly interested in the processing throughput of various implementations, we report **time per data pass** as our performance metric for comparison, which measures the time taken to process all data samples in the training set once. It should be noted the Bösen and STRADS implementations use the adaptive gradient algorithm [11] for step size tuning which significantly improves the per-data-pass convergence rate with slightly higher computational cost. Moreover, the Bösen implementation allows conflicting writes and employs staleness, which slightly harms the per-data-pass convergence rate. The Spark, Python-Serial, Standalone and MPI implementations have the same per-data-pass convergence rate.

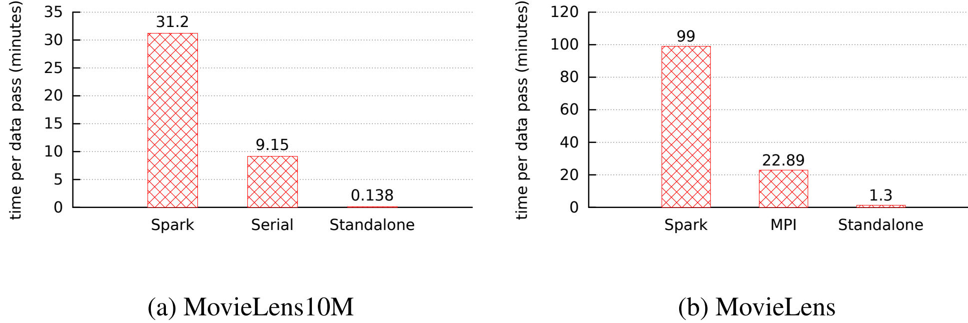
Our experiments used the datasets and configurations as shown in Table 1. Duplicated Netflix datasets are used for weak scaling experiments. Expriments are conducted using the PRObE Nome cluster [6], where each node contains 16 physical cores and 32GB of memory and nodes are connected via 10Gbps Ethernet.

4.1 Single-Threaded Baseline

Fig. 3a and 3b show the time per data pass with various implementations running on a single execution thread on two smaller datasets: MovieLens10M and MovieLens. While all are implemented in Python, the serial and

Dataset	Size	# Ratings	# Users	# Movies	Rank
MovieLens10M	134MB	10M	71.5K	10.7K	100
MovieLens	335MB	22M	240K	33K	500
Netflix	1.3GB	100M	480K	18K	500
Netflix4	5.2GB	400M	1.92M	72K	500
Netflix16	20.8GB	1.6B	7.68M	288K	500
Netflix64	83.2GB	6.4B	30.7M	1.15M	500
Netflix256	332.8GB	25.6B	122.9M	4.6M	500

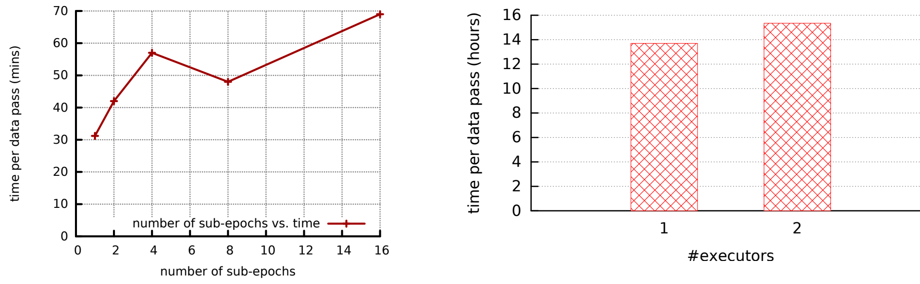
Table 1: Datasets used for the experiments.



(a) MovieLens10M

(b) MovieLens

Figure 3: Single-threaded baseline



(a) The overhead with increasing number of strata (i.e number of sub-epochs)

(b) Spark SGD-MF fails to gain speedup using more cores on the same machine

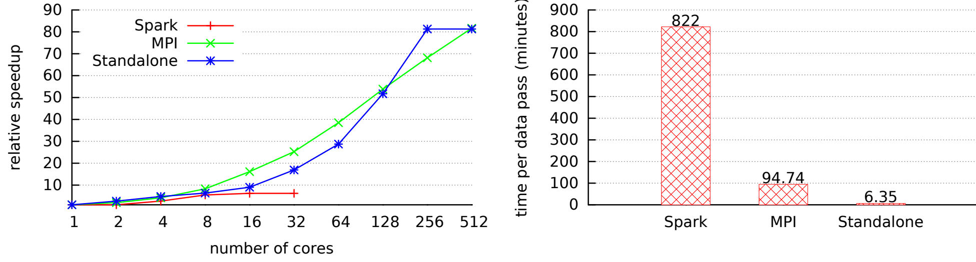
Figure 4: Spart running on a single machine

MPI implementations are $3-4\times$ faster than the Spark implementation. The standalone C++ implementation is near or more than 2 orders of magnitude faster than the PySpark implementation.

4.2 Overhead with More Strata

Gemulla's algorithm partitions the ratings matrix into P strata, which each consist of P blocks that can be processed in parallel. Thus P represents the number of processors that can be effectively used. Ideally we expect the time per data pass to be inversely propotional to the number of strata. However, since having more strata (i.e. more sub-epochs) incurs overhead, such as more synchronization barriers, the time per data pass may not decrease linearly as higher parallelism is introduced.

We demonstrate how such overhead affects Spark's per-data-pass processing time by processing the same ratings matrix (MovieLens10M) with increasing number of strata (i.e. number of sub-epochs) using only a single core. As shown in Fig 4a, the per-data-pass processing time almost doubled when the number



(a) Speedup relative to a single core

(b) Time per data pass on a single core

Figure 5: Strong scaling with respect to number of cores

of strata is increased from 1 to 4, though the overhead increases much more slowly after that.

4.3 Strong Scaling

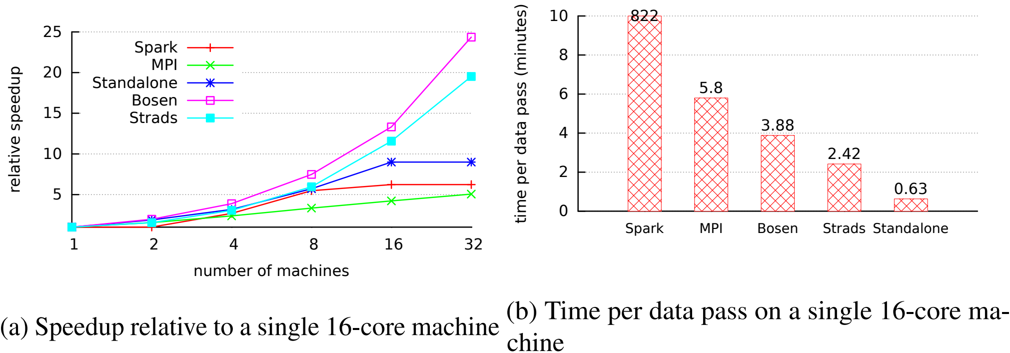
We evaluated the strong scalability of different implementations using the Netflix dataset. The time per data pass of three implementations (Spark, MPI and Standalone) using a single core is shown in Fig. 5b. Generally we scale up the number of cores employed on a single shared-memory machine first before scaling out to multiple machines. In this case, the number of sub-epochs is the same as the number of cores. However, with Spark, we used 4 strata with 1 core and 12 strata with 2 cores on the same machine as these are the minimum number of strata that don't cause Spark to crash due to memory failure[1]. Surprisingly, the Spark implementation fails to gain speedup from using more cores on a single machine, as shown in Fig. 4b. Thus we scale the Spark implementation using one core on each of multiple machines. Since a higher number of strata introduces additional overhead, we used the minimum number of strata that can effectively utilize the given number of cores and not cause Spark to crash. It should be noted that the standalone implementation is about 130× faster and 14.9× faster than the Spark and MPI implementation, respectively, when running on a single core.

As shown in Fig. 5a, the Spark implementation gains a 5.5× speedup using 8 cores, 1 core on each of 8 machines, but gains no speedup from more cores. The standalone implementation gains a 6.3× speedup with 8 cores on the same machine, but only 9× with 16 cores. The limited scalability of the standalone code on shared memory is largely due to higher number of cores incuring 10× more cache misses, as shown in 7b. The standalone implementation gains a 80× speedup with 256 cores spread over 16 machines, with each data pass taking 4 seconds. There's no further speedup from using more cores and most of the time is now spent on communication. The MPI implementation scales similarly to the standalone implementation.

Strong scalability with respect to the number of machines and a comparison with two general-purpose ML systems Bösen and STRADS is shown in Fig. 6a, and the time per data pass with different implementations running on a single machine (Spark uses only 1 core) is shown in Fig. 6b.

The Bösen implementation runs slower than the standalone implementation on a shared-memory multi-core machine as the Bösen client library employs locking for concurrency control and uses the adaptive gradient method [12] while the standalone implementation doesn't. While STRADS (also using adaptive gradient) scales better than the standalone implementation, the standalone implementation achieves about 4 seconds per data pass with 16 machines and STRADS achieves 6 seconds per data pass with 64 machines (no further speedup with more machines for both implementations).

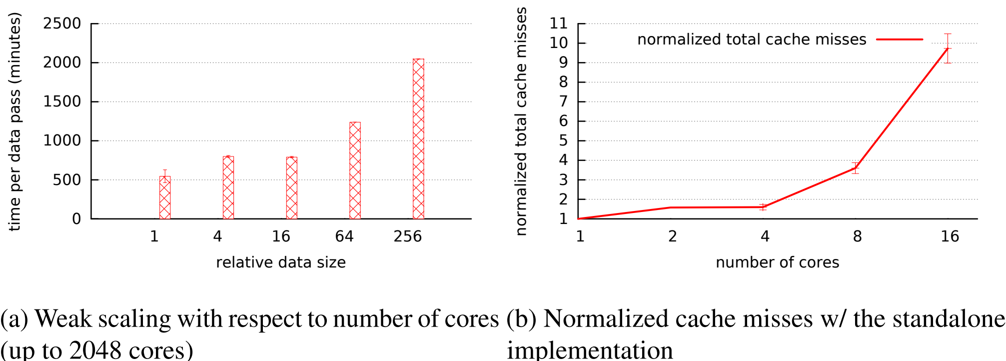
[1] Cause unknown.



(a) Speedup relative to a single 16-core machine

(b) Time per data pass on a single 16-core machine

Figure 6: Strong scaling with respect to number of machines



(a) Weak scaling with respect to number of cores (up to 2048 cores)

(b) Normalized cache misses w/ the standalone implementation

Figure 7: Weak scaling and cache misses

4.4 Weak Scaling

We evaluated the weak scalability of the standalone implementation using the Netflix dataset duplicated a number of times propotional to the number of cores. 8 cores were assigned for the original Netflix data; 32 cores (2 Nome nodes) were assigned for 4× Netflix data; etc. While the time per data pass remains roughly unchanged up to 16× duplicated data, with the number of cores proportionally increasing, it increased considerably for the 64× and 256× duplicated datasets.

We failed to run the Spark implementation on even 4× duplicated Netflix data due to consistent executor-lost failure during the first data pass.

4.5 Discussion

Based on our development experience, Spark indeed greatly simplified the application development compared to the standalone implementation (a few days vs. two weeks). But the experimental evaluation suggested that Spark suffers major performance penalty.

Our goal is not to develop the most efficient SGD matrix factorization program. While recently proposed techniques such as [10] may be used to reduce cache misses and improve the standalone implementation's performance, they are not employed here; and room for improving the scalability of the standalone implementation may exist. Nevertheless, our results are convincing that there is an opportunity for significant performance improvements of Spark for machine learning applications, possibly by using other mechanisms for communicating updates.

5 Conclusion

We implemented a popular, parallel machine learning training program - Stochastic Gradient Descent for Matrix Factorization on Apache Spark as well as other on other frameworks. Our experimental comparison showed that the Spark implementation is considerably slower than alternatives and is slower than a standalone implementation by two orders of magnitude. The Spark implementation also had limited scalability. The results suggested that RDD joins via shuffling incurs huge performance penalty compared to other approaches such as scheduled pipelining. Implementation of alternative communication schemes as well as a more sophisticated query planner are needed for Spark programmers to enjoy performant execution along with its declarative programming interface.

References

[1] Apache Hadoop. https://hadoop.apache.org/.

[2] Apache Spark. http://spark.apache.org/.

[3] J. Bennett and S. Lanning. The netflix prize. *In KDD Cup and Workshop*, 2007.

[4] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[5] 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*, KDD '11, pages 69–77, New York, NY, USA, 2011. ACM.

[6] G. Gibson, G. Grider, A. Jacobson, and W. Lloyd. Probe: A thousand-node experimental cluster for computer systems research. volume 38, June 2013.

[7] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing. More effective distributed ml via a stale synchronous parallel parameter server. In C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 1223–1231. Curran Associates, Inc., 2013.

[8] J. K. Kim, Q. Ho, S. Lee, X. Zheng, W. Dai, G. A. Gibson, and E. P. Xing. Strads: A distributed framework for scheduled model parallel machine learning. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 5:1–5:16, New York, NY, USA, 2016. ACM.

[9] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 583–598, Broomfield, CO, Oct. 2014. USENIX Association.

[10] Z. Liu, Y.-X. Wang, and A. Smola. Fast differentially private matrix factorization. In *Proceedings of the 9th ACM Conference on Recommender Systems*, RecSys '15, pages 171–178, New York, NY, USA, 2015. ACM.

[11] H. B. McMahan and M. Streeter. Delay-tolerant algorithms for asynchronous distributed online learning. *Advances in Neural Information Processing Systems (NIPS)*, 2014.

[12] H. B. McMahan and M. Streeter. Delay-tolerant algorithms for asynchronous distributed online learning. *Advances in Neural Information Processing Systems (NIPS)*, 2014.

[13] J. Wei, W. Dai, A. Qiao, Q. Ho, H. Cui, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Managed communication and consistency for fast data-parallel iterative analytics. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 381–394, New York, NY, USA, 2015. ACM.

[14] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, San Jose, CA, 2012. USENIX.

[15] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, San Jose, CA, 2012. USENIX.