Performance and Scalability of Broadcast in Spark

Mosharaf Chowdhury*
University of California, Berkeley

ABSTRACT

Although the MapReduce programming model has so far been highly successful, not all applications are well suited to this model. Spark bridges this gap by providing seamless support for iterative and interactive jobs that are hard to express using the acyclic data flow model pioneered by MapReduce. While benchmarking Spark, we identified that the default broadcast mechanism implemented in the Spark prototype is a hindrance toward its scalability.

In this report, we implement, evaluate, and compare four different broadcast mechanisms (including the default one) for Spark. We outline the basic requirements of a broadcast mechanism for Spark and analyze each of the compared broadcast mechanisms under that guideline. Our experiments in high-speed, low-latency, and cooperative data center environments also shed light on characteristics of multicast and broadcast mechanisms in data centers in general.

1. INTRODUCTION

With the advent of MapReduce [12] and similar frameworks as well as of cloud services like Amazon's EC2 [1], a new model of cluster computing has become mainstream in recent years. In this model, data-parallel computations are executed on commodity clusters by systems that automatically provide locality-aware scheduling, fault tolerance, and load balancing. MapReduce [12] pioneered this model, while systems like Dryad [13] and Map-Reduce-Merge [16] generalized the types of data flows supported. These systems achieve their scalability and fault tolerance by providing a programming model where the user creates acyclic data flow graphs to pass input data through a set of operators. This allows the underlying system to manage scheduling and to react to faults without user intervention.

Although this programming model has been highly successful, some applications cannot be expressed efficiently as acyclic data flows. Spark [17] is a framework optimized for one such type of applications - *iterative jobs*, where a dataset is shared across multiple parallel operations. Spark provides a functional programming model similar to MapReduce, but also lets users ask for data to be cached between iterations, leading to up to 10x better performance than Hadoop on some jobs. This is achieved without sacrificing MapReduce's fault tolerance. Moreover, the ability of Spark to load a dataset into memory and query it repeatedly makes it especially suitable for interactive analysis of big datasets.

While benchmarking Spark against Hadoop, we observed that the default broadcast mechanism (CHB[1]) in its prototype implementation is an impediment toward its scalability. Specifically, the default broadcast implementation takes a centralized approach where the broadcast variable is serialized and written to the HDFS by the sender and all the receivers read and deserialize it to reconstruct the variable - essentially creating a bottleneck at the HDFS. Further pondering revealed that in addition to scalability, performance, fault tolerance, and adaptability to unstructured clusters are three more major requirements for a viable broadcast mechanism for Spark and similar frameworks running on commodity clusters.

In this report, we consider three more broadcast mechanisms: Chained Streaming Broadcast (CSB), BitTorrent Broadcast (BTB), and SplitStream Broadcast (SSB), each with diverse characteristics. We have implemented, evaluated, and compared them to identify the better choice for broadcast in data center environments. While multicast and broadcast are well understood topics in computer networks, to the best of our knowledge, all of them assume that nodes are distributed across the Internet; whereas, we are concerned about high-speed, low-latency, and cooperative data center environments. From that perspective, in addition to making Spark more scalable, we expect our work to enrich the networking literature as well.

The remainder of the report is organized as follows. Section 2 provides an overview of the Spark framework along with several working examples of iterative jobs written in Spark. In Section 3, we examine performance and scalability characteristics of Spark and identify broadcast as a potential candidate for improving Spark scalability. Section 4 lists the requirements of an efficient and scalable broadcast mechanism for Spark and similar frameworks that run in a shared data center environment. In Section 5, we describe four different broadcast mechanisms accompanied by their evaluation results and analyze how each of them fulfill the requirements. Section 6 presents more experimental results on the performance and scalability of CSB, which we have found to be the best of the four compared mechanisms. We conclude with a discussion of future work in Section 7.

2. AN OVERVIEW OF SPARK

In this section, we provide an overview of Spark objectives, job execution model, and its programming model. We also highlight different aspects Spark using three examples. Of them, the alternating least squares method example (Sec-

*This is a joint work with Matei Zaharia and Ion Stoica.

[1]CHB stands for Centralized HDFS Broadcast.

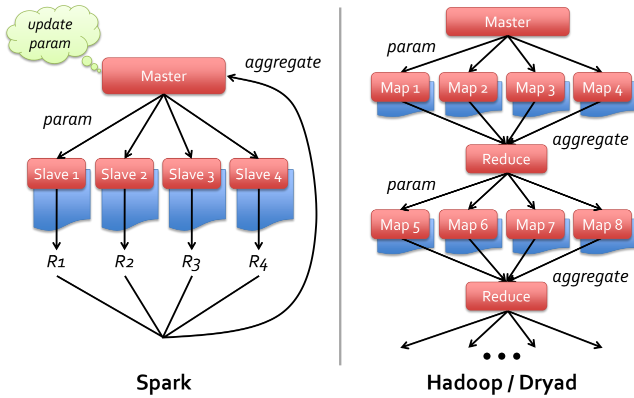


Figure 1: Comparison of job execution in Spark and Hadoop/Dryad models. In Spark, the same worker processes are reused across iterations and reads only happen once.

tion 2.4.3) is used for benchmarking purposes throughout the rest of this report.

2.1 Objectives

Spark is designed with two specific types of jobs in mind that the existing acyclic data flow-based programming models are not good at. These are:

1. **Iterative jobs:** Many common machine learning algorithms apply a function repeatedly to the same dataset to optimize a parameter (e.g., through gradient descent). While each iteration can be expressed as a MapReduce/Dryad job, each job must reload the data from disk, incurring a significant performance penalty.

2. **Interactive analysis:** Hadoop is often used to perform ad-hoc exploratory queries on big datasets using SQL interfaces such as Pig [14] and Hive [2]. Ideally, a user would be able to load a dataset of interest into memory across a number of machines and query it repeatedly. However, with Hadoop, each query incurs significant latency (tens of seconds) because it runs as a separate MapReduce job and reads data from disk.

While providing better support for these jobs, Spark wants to leverage the functional programming interface of Scala to provide a clean, integrated programming experience to end users.

Finally, we want to retain the fine grained fault tolerance model of MapReduce in Spark.

2.2 Job Execution Model

To use Spark, developers write a *driver program* that implements the high-level control flow of a Spark job. This driver program runs in the master node of a Spark cluster and launches various operations in parallel when programmers invoke operations like *map*, *filter* and *reduce* (explained later) by passing closures (functions) to the slaves (also referred to as workers).

As is typical in functional programming, these closures can refer to variables in the scope where they are created. Normally, when Spark runs a closure on a worker node, these variables are copied to the worker. In addition, users can manually broadcast variables (detailed later) to worker nodes that are not contained within the closure and instruct the workers to cache them.

What sets Spark apart from MapReduce-like frameworks is that in Spark the same worker nodes are reused across iterations. The workers hold onto the cached variables and reuse them without reading from the file system or re-receiving from the master. In comparison, standard MapReduce (consequently, Hadoop) does not support multi-stage jobs, so the only way of supporting iterative jobs is to store results in the file system at the end of every iteration and read stored results in the next iteration. Dryad, on the other hand, supports the notion of multi-stage jobs, but there is no support for cross-iteration data persistence and broadcast variables.

2.3 Programming Model

Spark provides two main abstractions for parallel programming: *resilient distributed datasets* and *parallel operations* on these datasets (invoked by passing a function to apply on a dataset). In addition, Spark supports two restricted types of *shared variables* that can be used in functions running on the cluster.

2.3.1 Resilient Distributed Datasets (RDDs)

A resilient distributed dataset (RDD) is a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. The elements of an RDD need not exist in physical storage; instead, a handle to an RDD contains enough information to *compute* the RDD starting from data in reliable storage. This means that RDDs can always be reconstructed if nodes fail.

In Spark, each RDD is represented by a Scala object. Spark lets programmers construct RDDs in four ways:

- From a *file* in a shared file system, such as the Hadoop Distributed File System (HDFS).

- By *"parallelizing"* a Scala collection (e.g., an array) in the driver program, which means dividing it into a number of slices that will be sent to multiple nodes.

- By *transforming* an existing RDD. A dataset with elements of type A can be transformed into a dataset with elements of type B using an operation called *flatMap*, which passes each element through a user-provided function of type $A \Rightarrow List[B]$.[2] Other transformations can be expressed using *flatMap*, including *map* (pass elements through a function of type $A \Rightarrow B$) and *filter* (pick elements matching a predicate).

- By changing the *persistence* of an existing RDD. By default, RDDs are lazy and ephemeral. That is, partitions of a dataset are materialized on demand when they are used in a parallel operation (e.g., by passing a block of a file through a map function), and are discarded from memory after use.[3] However, a user can alter the persistence of an RDD through two actions:
  - The *cache* action leaves the dataset lazy, but hints that it should be kept in memory after the first time it is computed, because it will be reused.

[2] *flatMap* has the same semantics as the *map* in MapReduce, but *map* is usually used to refer to a one-to-one function of type $A \Rightarrow B$ in Scala.

[3] Similar to "distributed collections" function in DryadLINQ.

– The *save* action evaluates the dataset and writes it to a distributed filesystem such as HDFS. The saved version is used in future operations on it.

2.3.2 Parallel Operations

Several parallel operations can be performed on RDDs:

- *reduce*: Combines dataset elements using an associative function to produce a result at the driver program.
- *collect*: Sends all elements of the dataset to the driver program. For example, an easy way to update an array in parallel is to parallelize, map and collect the array.
- *foreach*: Passes each element through a user provided function. This is only done for the side effects of the function (which might be to copy data to another system or to update a shared variable as explained below).

Spark does not currently support a grouped reduce operation as in MapReduce; reduce results are only collected at one process (the driver)[4]. We plan to support grouped reductions in the future using a "shuffle" transformation on distributed datasets. However, even using a single reducer is enough to express a variety of useful algorithms. For example, a recent paper on MapReduce for machine learning on multi-core systems [11] implemented ten learning algorithms without supporting parallel reduction.

2.3.3 Shared Variables

Spark allows programmers create two restricted types of *shared variables* to support two simple but common usage patterns:

- *Broadcast variables*: If a large read-only piece of data (e.g., a lookup table) is used in multiple parallel operations, it is preferable to distribute it to the workers only once instead of packaging it with every closure. Spark lets the programmer create a "broadcast variable" object that wraps the value and ensures that it is only copied to each worker once.
- *Accumulators*: These are variables that workers can only "add" to using an associative operation, and that only the driver can read. They can be used to implement counters as in MapReduce and to provide a more imperative syntax for parallel sums. Accumulators can be defined for any type that has an "add" operation and a "zero" value. Due to their "add-only" semantics, they are easy to make fault-tolerant.

2.4 Examples

We now show some sample Spark programs. Note that we omit variable types because Scala uses type inference, but Scala is statically typed and performs comparably to Java.

2.4.1 Text Search

Suppose that we wish to count the lines containing errors in a large log file stored in HDFS. This can be implemented by starting with a file dataset object as follows:

```
val file = spark.textFile("hdfs://...")
val errs = file.filter(_.contains("ERROR"))
val ones = errs.map(_ => 1)
val count = ones.reduce(_+_)
```

[4]Local reductions are first performed at each node, however.

We first create a distributed dataset called `file` that represents the HDFS file as a collection of lines. We transform this dataset to create the set of lines containing "ERROR" (`errs`), and then map each line to a 1 and add up these ones using *reduce*. The arguments to *filter*, *map* and *reduce* are Scala syntax for function literals.

Note that `errs` and `ones` are lazy RDDs that are never materialized. Instead, when *reduce* is called, each worker node scans input blocks in a streaming manner to evaluate `ones`, adds these to perform a local reduce, and sends its local count to the driver. When used with lazy datasets in this manner, Spark closely emulates MapReduce.

Where Spark differs from other frameworks is that it can make some of the intermediate datasets persist across operations. For example, if wanted to reuse the `errs` dataset, we could create a cached RDD from it as follows:

```
val cachedErrs = errs.cache()
```

We would now be able to invoke parallel operations on `cachedErrs` or on datasets derived from it as usual, but nodes would cache partitions of `cachedErrs` in memory after the first time they compute them, greatly speeding up subsequent operations on it.

2.4.2 Logistic Regression

The following program implements logistic regression [7], an iterative classification algorithm that attempts to find a hyperplane w that best separates two sets of points. The algorithm performs gradient descent: it starts w at a random value, and on each iteration, it sums a function of w over the data to move w in a direction that improves it. It thus benefits greatly from caching the data in memory across iterations. We do not explain logistic regression in detail, but we use it to show a few new Spark features.

```
// Read points from a text file and cache them
val points = spark.textFile(...)
                 .map(parsePoint).cache()
// Initialize w to a random D-dimensional vector
var w = Vector.random(D)
// Run multiple iterations to update w
for (i <- 1 to ITERATIONS) {
  val gradient = spark.accumulator(new Vector(D))
  for (p <- points) { // Runs in parallel
    val s = (1/(1+exp(-p.y*(w dot p.x)))-1)*p.y
    gradient += s * p.x
  }
  w -= gradient.value
}
```

First, although we create an RDD called `points`, we process it by running a `for` loop over it. The `for` keyword in Scala is syntactic sugar for invoking the `foreach` method of a collection with the loop body as a closure. That is, the code `for(p <- points){body}` in this case is equivalent to `points.foreach(p => {body})`. Therefore, we are invoking Spark's parallel `foreach` operation.

Second, to sum up the gradient, we use an accumulator variable called `gradient` (with a value of type $Vector$). Note that the loop adds to `gradient` using an overloaded `+=` operator. The combination of accumulators and `for` syntax allows Spark programs to look much like imperative serial programs. Indeed, this example differs from a serial version of logistic regression in only three lines.

2.4.3 Alternating Least Squares

Our final example is an algorithm called alternating least squares (ALS). ALS is used for collaborative filtering problems, such as predicting users' ratings for movies that they have not seen based on their movie rating history (as in the Netflix Challenge). Unlike our previous examples, ALS is CPU-intensive rather than data-intensive.

We briefly sketch ALS and refer the reader to [18] for details. Suppose that we wanted to predict the ratings of u users for m movies, and that we had a partially filled matrix R containing the known ratings for some user-movie pairs. ALS models R as the product of two matrices M and U of dimensions $m \times k$ and $k \times u$ respectively; that is, each user and each movie has a k-dimensional "feature vector" describing its characteristics, and a user's rating for a movie is the dot product of its feature vector and the movie's. ALS solves for M and U using the known ratings and then computes $M \times U$ to predict the unknown ones. This is done using the following iterative process:

1. Initialize M to a random value.
2. Optimize U given M to minimize error on R.
3. Optimize M given U to minimize error on R.
4. Repeat steps 2 and 3 until convergence.

ALS can be parallelized by updating different users / movies on each node in steps 2 and 3. However, because all of the steps use R, it is helpful to make R a broadcast variable so that it does not get re-sent to each node on each step. A Spark implementation of ALS that does is shown below. Note that we *parallelize* the collection `0 until u` (a Scala range object) and *collect* it to update each array:

```
val Rb = spark.broadcast(R)
for (i <- 1 to ITERATIONS) {
  U = spark.parallelize(0 until u)
           .map(j => updateUser(j, Rb, M))
           .collect()
  M = spark.parallelize(0 until m)
           .map(j => updateUser(j, Rb, U))
           .collect()
}
```

3. MEASUREMENTS & IMPLICATIONS

In this section, we present a comparative performance measurement of Spark against Hadoop on a large logistic regression job and a detailed analysis of Spark scalability characteristics on an ALS job using the Netflix dataset [5]. Our work in this report is motivated by our findings from the breakdown of the later job.

3.1 Logistic Regression

We compared the performance of the logistic regression job in Section 2.4.2 to an implementation of logistic regression for Hadoop, using a 29 GB dataset on m1.xlarge EC2 nodes with 4 cores each. The results are shown in Figure 2. With Hadoop, each iteration takes 127s, because it runs as an independent MapReduce job. With Spark, the first iteration takes 174s (likely due to using Scala instead of Java), but subsequent iterations take only 6s, each because they reuse cached data. This allows the job to run up to 10× faster.

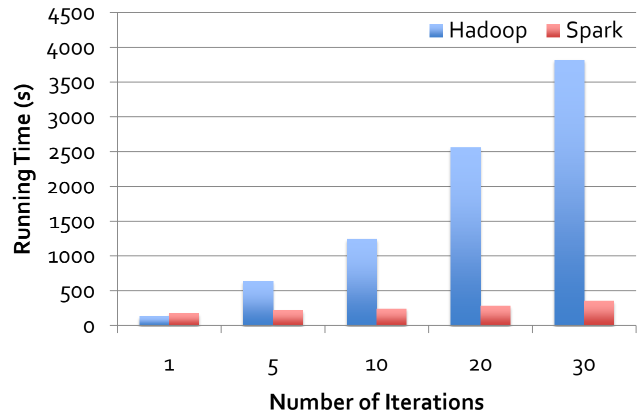


Figure 2: Logistic regression performance in Hadoop and Spark (29GB dataset. All nodes: m1.xlarge EC2 instances).

3.2 Alternating Least Squares

We have implemented the alternating least squares job in Section 2.4.3 to measure the benefit of broadcast variables for iterative jobs that copy a shared dataset to multiple worker nodes. We found that without using broadcast variables, the time to resend the ratings matrix R on each iteration dominated the job's running time.

Figure 3 shows the performance of the ALS job using the default file system-based broadcast implementation (detailed later) in Spark. In the first iteration, we have to broadcast large static matrices that are used across different iterations. In each of the later iterations, matrices of relatively smaller size ($\approx$ 230MB) are broadcast. We noticed that the time for completion for the first iteration decreases up to a certain number of nodes and then increases, whereas the time for completion for later iterations keep decreasing for larger number of nodes.

A breakdown of the computation and communication costs in each iteration (shown in Figure 4) unveiled that while the computation time was predictably decreasing with increasing number of workers, the broadcast times grew linearly with the number of nodes, limiting the scalability of the job. It became apparent that the default broadcast implementation was the biggest impediment against large-scale Spark jobs that use broadcast variables.

4. BROADCAST REQUIREMENTS

Based on our experience with Spark, we conclude that any broadcast mechanism for Spark should at least satisfy the following requirements:

- **Performance:** It must be able to minimize the maximum time to broadcast a particular variable to each individual worker. The objective is to bring that minimum value as close as possible to the line speed inside the data center (theoretically, 1Gbps shared, but on the average 550-600Mbps point-to-point speed is observed in EC2).
- **Scalability:** A broadcast mechanism must scale up to 1000s of nodes (if not 10000s), where each node has multi- to many-cores with minimal overhead. Ideally,

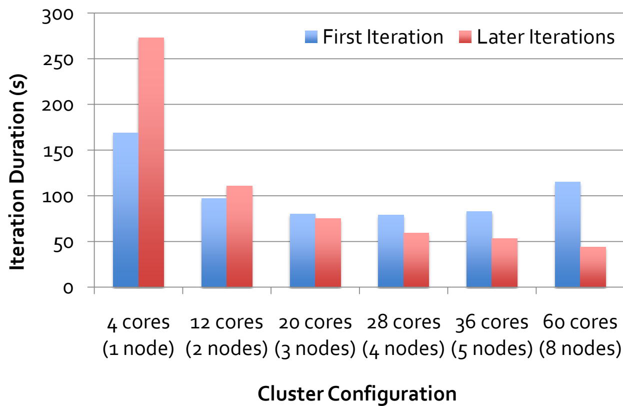


Figure 3: ALS performance in Spark using the Netflix dataset (k=60; Master: m2.xlarge and slaves: c1.xlarge EC2 instances).

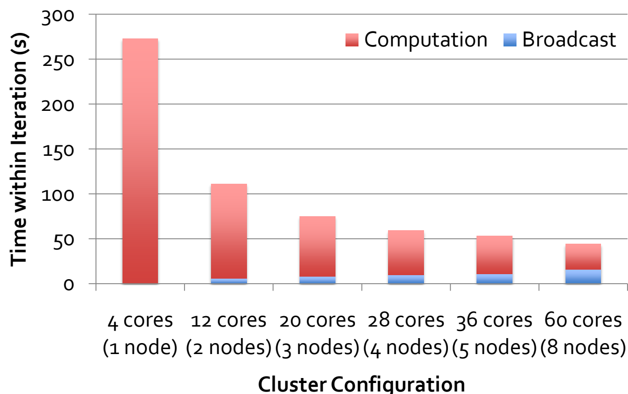


Figure 4: Breakdown of computation and communication times in later iterations of the ALS job shown in Figure 3.

broadcast time vs number of nodes graph should be a line parallel to the x-axis (as oppose to the linear increase observed in Figures 3 and 4).

- **Fault tolerance:** It must be able to withstand and gracefully handle faults in worker nodes. It would be great to be able to handle faults in master as well; however, most data-parallel systems (e.g., Hadoop, Dryad, and Spark) with "stateful" masters are known to be not-so-good at handling and recovering from master failures.

- **Topology independence:** Unlike high performance computing systems, clusters for data-parallel jobs are allocated in an ad hoc, per-job basis. As a result, there is no structured physical topology that a broadcast mechanism can take advantage of. However, a broadcast mechanism can create logical overlays over the unstructured cluster to manage itself.

5. BROADCAST MECHANISMS CONSIDERED FOR SPARK

In this section, we present an overview of four different broadcast mechanisms that we have considered and evaluated for Spark in this report. These include the default centralized implementation, referred to as the Centralized HDFS Broadcast (CSB), a straightforward but very effective Chained Streaming Broadcast (CSB), and two experimental mechanisms: BitTorrent Broadcast (BTB) and SplitStream Broadcast (SSB). In general, none of these mechanisms make any assumptions about the physical topology or structure of the cluster.

5.1 Centralized HDFS Broadcast (CHB)

5.1.1 Overview

Broadcast variables are implemented using classes with custom serialization formats. When one creates a broadcast variable b with a value v, v is saved to a file in a shared file system, which can be HDFS or NFS. The serialized form of b is a path to this file. When b's value is queried on a worker node, Spark first checks whether v is in a local cache; if it is not, then Spark reads it from HDFS. HDFS, by default, keeps three replicas for any file for fault tolerance as well as for performance. Workers read b from their corresponding closest replicas.

5.1.2 Analysis

Performance.
Since each worker reads the broadcast variable from its closest replica, observed performance of CHB is pretty good (up to 160 Mbps) as long as it can scale. After that, HDFS becomes the bottleneck.

Scalability.
We have seen in Figures 3 and 4, in CHB, broadcast time increases linearly with the number of worker nodes that are trying to simultaneously read the variable from the shared storage system. Figure 5 presents the scalability characteristics of CHB in an even larger scale. After 40 nodes, even with multiple distributed replicas, CHB reaches a breakdown point and broadcast time increases super-linearly.

Fault tolerance.
CHB can handle faults very easily and robustly using the multiple replicas stored for each broadcast variables.

5.2 Chained Streaming Broadcast (CSB)

5.2.1 Overview

For each broadcast variable b, the master serializes it using a custom serialization mechanism, divides it into blocks, and waits for slaves to connect. Initially, the master is the only node with the complete copy of b. It puts itself in a priority queue of *source* nodes that have some part of b (these are called *leechers*) or the complete variable (these are *seeds*).

Whenever a slave requests for b, the master selects a source from the priority queue, sends it to the slave that requested b, and puts the slave in the priority queue as a source for future requests. Effectively, a tree structure is formed with the master at the root; the degree of this tree can be controlled by system parameters and can possibly be changed by the programmer. However, most of our experiments at this point were limited to degrees 1 or 2.

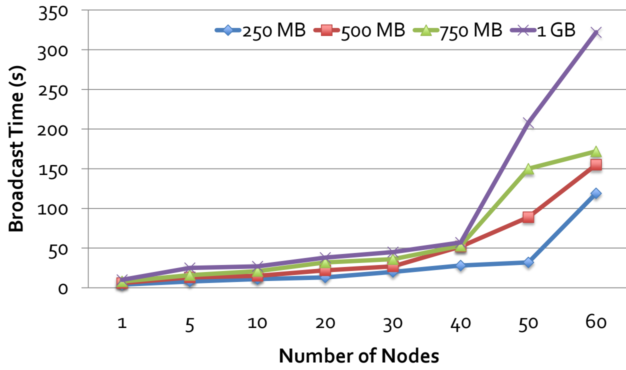


Figure 5: Scalability and performance of CHB (All nodes: m1.large EC2 instances).

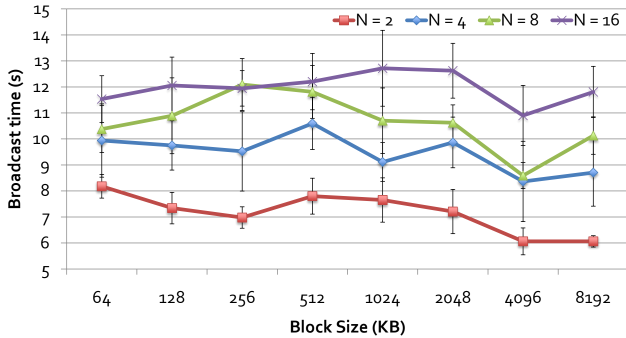


Figure 6: Search for an appropriate block size for CSB (All nodes: m1.xlarge EC2 instances).

Once a slave completes receiving a broadcast, it becomes a seed itself. This results in creation of multiple broadcast trees (a forest) during the lifetime of a single broadcast.

The master node maintains a *tracker* for handling multiple simultaneous broadcast variables. In effect, slaves request for broadcast variables to the tracker, and the tracker hands it off to appropriate handler for that variable in the master, which then maintains the broadcast forest as described earlier.

Selection of block size.

In order to decide the appropriate block size, we performed an exhaustive search over the parameter space in a spirit similar to auto-tuners. Figure 6 shows that 4MB was the best choice for the EC2 cluster we were using. Note that, this value can vary based on cluster type, cluster size, and temporal as well as spatial conditions (since we are in a shared cluster).

Policy for the prioritization.

We employed two different policies to prioritize sources in the priority queue maintained by the master when forwarding a broadcast request:

1. **Lowest leecher count first (LLCF):** Using LLCF, the master selects the source that are serving the least number of slaves. One potential problem with this policy is that a slow source somewhere in the middle of the broadcast tree can slow down everyone else underneath it.

2. **Highest observed speed first (HOSF):** In this case, each slave sends back a feedback regarding the observed speed for a particular source it received a broadcast variable from. The master keeps a running average of such observed speeds for each source and prioritizes faster sources over their slower counterparts. This essentially sends slower sources toward the leaves of the broadcast tree and minimizes their slowdown impact.

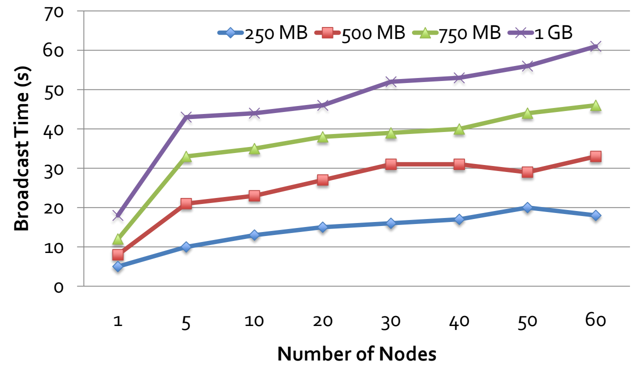


Figure 7: Scalability and performance of CSB (All nodes: m1.large EC2 instances).

We mostly used LLCF in the experiments presented in this report, because we came up with the HOSF policy much later in the process while exploring the root cause of some erratic behavior. However, barring some rare cases, nodes in EC2 and data centers in general have more or less uniform bandwidths. So using LLCF did not drastically affect CSB performance.

5.2.2 Analysis

Performance.

Performance of CSB is comparable to CHB at up to 40 nodes and orders of magnitude better after that (since CHB does not scale after that and CSB does). For a single node broadcast, CHB can attain almost the full speed (close to 400Mbps for 1GB variable). But it rises significantly for more than one node because, in such cases, a slave has to receive a block, keep a copy for itself, and then forward that block to its leechers. Since we perform these steps in Scala in the application layer, there is a significant overhead.

Scalability.

It is evident from Figure 7 that CSB scales much better than CHB due to the fact that broadcast load is distributed across all the sources. However, there is still a sub-linear rise of broadcast time as the number of slaves increases. A close scrutiny revealed that this was due to a couple of slightly slower receiving nodes. We intend to perform experiments in even larger scale in the future.

Fault tolerance.

CSB is not free from the Achilles' heel of any tree-based distribution scheme: if an inner-node fails, every one else

in its sub-tree will be affected. In our case, whenever a leecher notices that its source has unceremoniously closed the connection, it tries to reconnect multiple times before contacting the master for another source. Then it resumes from where the previous source left off.

In case there is no source currently broadcasting that variable (e.g., even the master has flushed that variable from its memory), the master keeps a persistent of each broadcast variable in the HDFS and refers to that. This scenario happens when a slave itself fails, loses its states, and is restarted later. Unless there is a large-scale failure, this last resort does not create a bottleneck.

5.3 BitTorrent Broadcast (BTB)

5.3.1 Overview

A BitTorrent [3] session distributes a file from the *seed* to multiple receivers (*leechers*), which can join and leave the session dynamically. When a receiver participates in a BitTorrent session, it is forced to donate some of its upload bandwidth to the session (using tit-for-tat strategy). The file being distributed is typically divided into 256KB pieces. For each piece, the seed computes an SHA1 hash and distributes them along with corresponding pieces for verifying their integrity. Each piece is further divided into blocks, typically of size 16KBytes. A receiver can download blocks within the same piece from multiple peers simultaneously, but it can relay blocks of a piece to other nodes only after it receives and verifies the integrity of the entire piece.

A centralized tracker keeps track of all the peers who have the file (both partially and completely) and lookup peers to connect with one another for downloading and uploading. In recent implementations, in addition to the centralized tracker, DHT-based lookup mechanisms allow distributed peer identification.

BitTorrent implements a choking (and unchoking) mechanism to discourage free-riders, to better match with TCP congestion control, and to control the number of simultaneous downloads for better TCP performance.

While BitTorrent is widely popular for file and content distribution in the Internet, it is not well-studied in the data center environment. The only known usage of BitTorrent inside a data center is for software and server upgrade distributions inside Twitter data centers [8].

In our implementation (based on BitTornado [4]), the master acts as the initial seed and the tracker and the slaves act as the leechers. The variable to broadcast is serialized into a file, turned into a *.torrent* file, and then normal BitTorrent mechanism is used to transfer it to different slaves.

5.3.2 Analysis of Default BitTorrent

Performance.

Since BitTorrent is highly optimized for the Internet, the performance of BTB inside high-speed, low-latency data center is not comparable to that of CHB or CSB. Specially, the default overhead of BitTorrent (for 1 node) is too high.

Scalability.

While in theory, BitTorrent (even the default one) should be scalable inside data center, we observed sudden increase in broadcast time after 30 nodes (Figure 8). We are not completely sure exactly what caused this increase; it can be

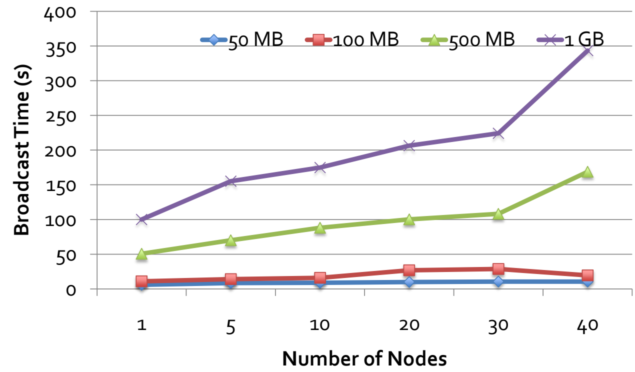


Figure 8: Scalability and performance of BitTorrent with default settings (All nodes: m1.xlarge EC2 instances).

an outcome of interactions between several BitTorrent parameters. We discuss the most important parameters later.

Fault tolerance.

BitTorrent is highly robust to failures as long as the tracker (initial seed) is alive. A persistent copy similar to CSB can be stored in HDFS to avoid tracker problems.

5.3.3 Modifications for the Data Center Environment

The following characteristics set data center networking inside a cluster apart from networking in the Internet: (i) low latency, (ii) high speed, (iii) absence of selfish and malicious peers, and (iv) no data corruptions (normal corruptions will be handled by link and network layers, and there is no malicious corruptions).

Based on these observations, we identified the following parameters that must be changed for better BTB performance inside a data center:

1. Choking should be removed, because there are no free-riders in Spark. If at all, there can be a reverse-choking mechanism that provides bandwidth to slower nodes as much as they can absorve.

2. Piece size and block size should be varied. Our micro-experiments showed 4MB and 8MB to be the better piece sizes (note that 4MB was found better for CSB as well); but increased block size increases broadcast time.

3. Extensive hashing is not required in our trusted environment.

4. Unlike the default BitTorrent, there should not be any manual upload or download limits. Every participating peer should be using its maximum capacities.

5. Number of simultaneous uploads should be increased, because available bandwidth is higher; but it should not be too high, because that will add TCP overhead.

In our current BTB implementation, piece size is set to 4MB, number of simultaneous uploads is 8, there is no download or upload limits. But we have not removed hashing or choking yet. Now, choking is not a big problem for us, because unless there is a really slow node (which is rare) we

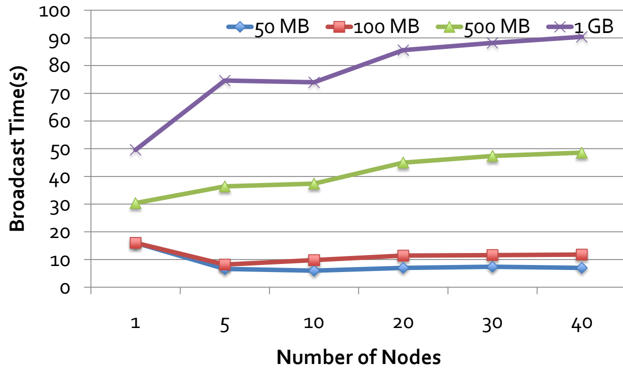


Figure 9: Scalability and performance of modified BitTorrent (All nodes: m1.xlarge EC2 instances).

would not see it in experiments. Hashing does add some overhead, but removing it would require changing the BitTornado library. We want to explore it in the future.

5.3.4 *Analysis of Modified BitTorrent*

Performance.

From Figure 9 we can see that the simple modifications significantly changed the performance of BTB. While it is still not as good as CHB or CSB, it is within the range and with further study we should be able to find the best settings for data centers.

Scalability.

The main reason behind our excitement about BitTorrent is its excellent scalability characteristics. Figure 9 shows that broadcast using BTB remains constant for increasing number of nodes for smaller variables, and it increases slower than CSB for larger variables.

Fault tolerance.

Fault tolerance characteristics for modified BTB is similar to that of the default version.

5.4 SplitStream Broadcast (SSB)

5.4.1 *Overview*

SplitStream [9] splits the content into k stripes and multicast each stripe using a separate tree. Peers join as many trees as there are stripes they wish to receive and they specify an upper bound on the number of stripes that they are willing to forward. SplitStream constructs this forest of multicast trees in a way so that an interior node in one tree is a leaf node in all the remaining trees and the bandwidth constraints specified by the nodes are satisfied. This ensures that the forwarding load can be spread across all participating peers. SplitStream uses the Pastry [15] DHT and Scribe [10] anycast system.

In case of SSB, all nodes wish to receive k stripes and they are willing to forward k stripes. SplitStream constructs a forest such that the forwarding load is evenly balanced across all nodes while achieving low delay and link stress across the system. Our implementation is based on FreePastry [6] codebase, an open source implementation of Pastry, Scribe, and SplitStream.

5.4.2 *Analysis*

Performance and scalability.

Theoretically, SplitStream has provably high utilization of available upload bandwidth and is highly scalable. However, during experimentation, we found that it is highly optimized for streaming (as in audio/video streaming), and even if some blocks are dropped/missed (they actually do due to some synchronization issues in distributed message passing), the receivers can gracefully handle that. Unfortunately, SSB cannot tolerate such failures, because a single dropped block will result in a corrupted variable. To address this we are currently designing an SSB protocol that is suited to our needs.

Fault tolerance.

Striping across multiple trees increases SplitStream's resilience to node failures. On the average, the failure of a single node causes the temporary loss of at most one of the stripes (in rest of the trees that node is a leaf and does not affect anything). This can be masked or mitigated using proper data encodings while that particular tree is repaired [9]. Such encoding schemes often achieve partial or full recovery by adding redundant information at the expense of bandwidth.

6. MORE CSB EVALUATION

In this section, we present additional CSB-based experimental results demonstrating its performance on ALS jobs and how EC2 attributes affect the performance of a broadcast strategy. We chose CSB for these experiments, because it came out as the best of the four choices we have experimented with so far.

6.1 ALS Performance

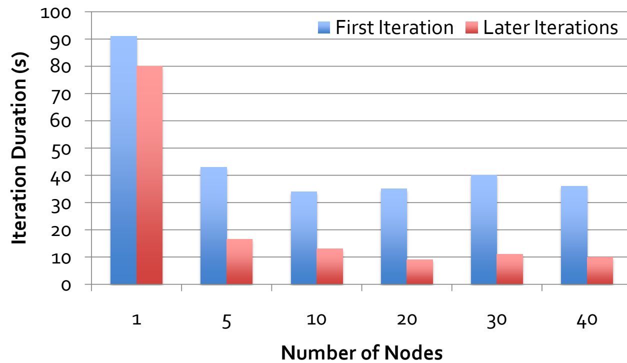
Figure 10 shows the performance of the ALS algorithm using CSB as the broadcast mechanism. Note that, these experiments do not use the Netflix data set, rather they use synthetic data so that we can vary different parameters. The values of m and u are set to 5000 and 15000 respectively in this figure, but we have performed experiments by varying them resulting in similar behavior. The biggest matrix to broadcast at a time using m=5000 and u=15000 is approximately 600MB in size (in the first iteration). The increasing value of k, super-linearly increases the amount of computation in each iteration.

Except for Figure 10(a) (with k=25), in all other cases, job duration decreases with the increased number of slaves (as expected). Table 2 summarizes the breakdown of computation and broadcast times in this case. While there are slight variations in broadcast times (which is not unexpected in a shared environment), the sudden increase of computation time for 30 nodes is quite unexpected (each EC2 instance runs inside virtual containers with strong isolation guarantees). Right now we do not have any explanation for why this happened other than attributing it to a possible measurement error.

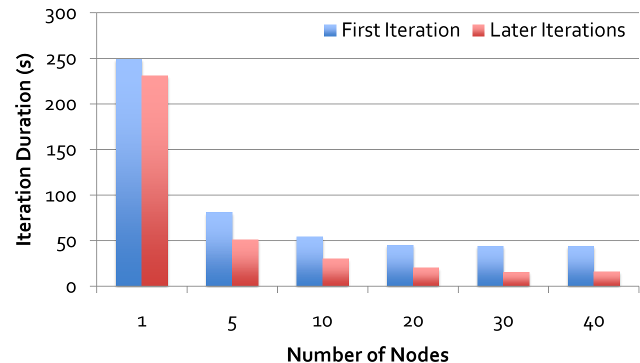
Figure 11 presents broadcast times in the first iterations for each ALS job in Figure 10. We can see that broadcast times mostly remain the same with increasing number

Table 1: **Summary of Different Broadcast Mechanisms**

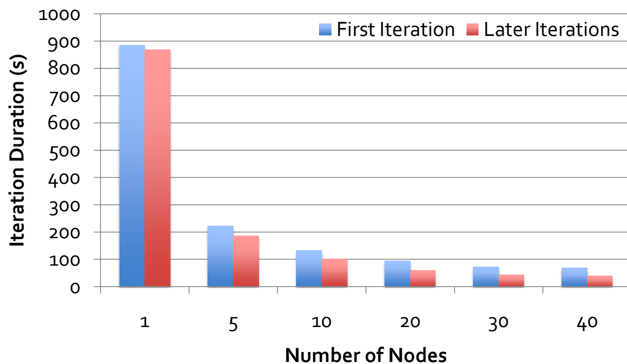
Broadcast Mechanism	Structure	Performance (1GB to 40 nodes)	Scalability (# nodes)	Fault Tolerance	Topology Independence	Implementation
CHB	Centralized	57s	40	Yes	Yes	100 lines Scala + Hadoop libraries
CSB	Tree-based	53s	60+	Yes	Yes	800 lines of Scala
BTB	Unstructured	90s	40+	Yes	Yes	100 lines of Shell Script + BitTornado libraries
SSB	Tree-based	TBD	TBD	No	Yes	400 lines Scala + FreePastry libraries



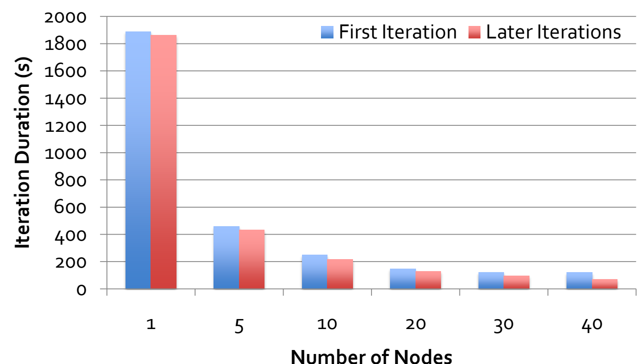
(a) k=25



(b) k=50



(c) k=100



(d) k=150

Figure 10: ALS performance using Chained Streaming Broadcast for varying feature size (i.e., varying k) (m=5000; u=15000; All nodes: c1.xlarge EC2 instances).

of slaves (as we have already seen for individual broadcast experiments in Section 5.2).

6.2 Impact of Cluster Configuration

Figure 12 shows the difference between broadcast times for the same workload running on clusters with different configurations. Each of the m1.xlarge instances has 15GB of memory and 8 EC2 compute units, whereas each c1.xlarge ones has 7GB of memory and 20EC2 compute nodes.

We can see that the cluster with the higher processing capacity can complete broadcast faster than its slower counterpart. As the number of nodes increase, the gap between the two lines keeps increasing. The reason is that with increased fire power, streaming nodes can minimize the overhead of copying blocks from the input interface to the output interface through the application layer.

We also believe that the spike at 10 nodes for the c1.xlarge cluster can be due to some experimental or measurement error; however, it is more likely to be caused by transient congestion due to the shared nature of the EC2 cluster. The fact that it is the highest value for any number of receivers for the c1.xlarge cluster, makes it a bigger suspect. In any case, we will rerun the experiment to investigate if we are missing any underlying phenomenon.

7. CONCLUSIONS AND FUTURE WORK

We presented an in-depth overview of the Spark framework along with multiple working examples. Based on our

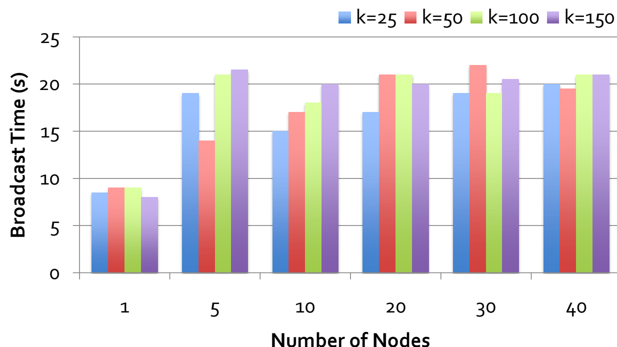


Figure 11: Broadcast times in the first iteration of the ALS jobs in Figure 10.

Table 2: Breakdown of computation and broadcast times for the first iteration of Figure 10(a)

# Nodes	Computation (s)	Broadcast (s)
1	82	9
5	24	19
10	19	15
20	18	17
30	21	19
40	16	20

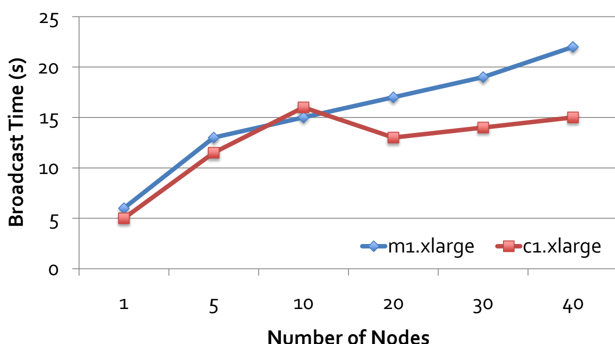


Figure 12: Impact on CSB broadcast time for different EC2 instance types (Variable size: 400MB).

experiments with Spark, we identified the default broadcast mechanism (CHB) to have potential room for improvement. We outlined the requirements of a broadcast mechanism for Spark and similar frameworks running on commodity clusters, and implemented, evaluated on EC2, and finally, compared the default mechanism with three other broadcast mechanisms (CSB, BTB, and SSB) using the requirements guideline. Our experiments so far have pointed CSB to be the better of the four choices.

There are several phenomena in this report that we could not explain to our satisfaction. We will continue investigating the root causes behind such unexplained behaviors. In addition, we are working on creating a reliable SSB that will be able to fully utilize the available bandwidth without incurring additional coding overhead to provide robustness against node failures. We expect this to outperform its counterparts.

Acknowledgements

We would like to thank Sameer Agarwal for the initial version of the profiler, Beth Trushkowsky for EC2-related tips throughout the project, and Neil Conway, Lester Mackey, Bill Marczak, Sara Alspaugh, and everyone else who provided valuable feedback and suggestions. Finally, a special shout out to the Nexus team for creating Nexus, the preferred substrate for Spark.

8. REFERENCES

[1] Amazon Elastic Compute Cloud (Amazon EC2). `http://aws.amazon.com/ec2/`.
[2] Apache Hive. `http://hadoop.apache.org/hive`.
[3] BitTorrent. `http://www.bittorrent.com/`.
[4] BitTorrent. `http://www.bittornado.com/`.
[5] Dissecting the Netflix Dataset. `http://www.igvita.com/2006/10/29/dissecting-the-netflix-dataset/`.
[6] FreePastry. `http://www.freepastry.org/`.
[7] Logistic regression – Wikipedia. `http://en.wikipedia.org/wiki/Logistic_regression`.
[8] Murder. `http://github.com/lg/murder/`.
[9] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: high-bandwidth multicast in cooperative environments. In *SOSP '03*, pages 298–313, 2003.
[10] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. Scalable application-level anycast for highly dynamic groups. In *Networked Group Communications*, 2003.
[11] C. T. Chu, S. K. Kim, Y. A. Lin, Y. Yu, G. R. Bradski, A. Y. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In *NIPS '06*, pages 281–288. MIT Press, 2006.
[12] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
[13] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72, 2007.
[14] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08*, 2008.
[15] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware '01*, pages 329–350, 2001.
[16] H.-c. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *SIGMOD'07*, pages 1029–1040, 2007.
[17] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets, 2010.
[18] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan. Large-scale parallel collaborative filtering for the Netflix prize. In *AAIM '08*, pages 337–348, 2008.