Bridging the Gap Between HPC and Big Data Frameworks

Michael Anderson[1], Shaden Smith[2], Narayanan Sundaram[1], Mihai Capotă[1], Zheguang Zhao[3], Subramanya Dulloor[4], Nadathur Satish[1], and Theodore L. Willke[1]

[1]Parallel Computing Lab, [4]Infrastructure Research Lab, Intel Corporation; [2]University of Minnesota; [3]Brown University

michael.j.anderson@intel.com, shaden@cs.umn.edu, narayanan.sundaram@intel.com, mihai.capota@intel.com, sam@cs.brown.edu, subramanya.r.dulloor@intel.com, nadathur.rajagopalan.satish@intel.com, ted.willke@intel.com

ABSTRACT

Apache Spark is a popular framework for data analytics with attractive features such as fault tolerance and interoperability with the Hadoop ecosystem. Unfortunately, many analytics operations in Spark are an order of magnitude or more slower compared to native implementations written with high performance computing tools such as MPI. There is a need to bridge the performance gap while retaining the benefits of the Spark ecosystem such as availability, productivity, and fault tolerance. In this paper, we propose a system for integrating MPI with Spark and analyze the costs and benefits of doing so for four distributed graph and machine learning applications. We show that offloading computation to an MPI environment from within Spark provides $3.1-17.7\times$ speedups on the four sparse applications, including all of the overheads. This opens up an avenue to reuse existing MPI libraries in Spark with little effort.

1. INTRODUCTION

As dataset sizes increase, there is a need for distributed tools that allow for exploration and analysis of data that does not fit on a single machine. Apache Hadoop provides a fault-tolerant distributed filesystem as well as a map-reduce processing framework that allows for analytics on large datasets. More recently, Apache Spark introduced the resilient distributed dataset (RDD) which can be cached in memory, thereby accelerating iterative workloads often found in machine learning [34]. In addition to map-reduce-style computation, Spark includes support for joins, as well as extensive library support for graph computations, machine learning, and SQL queries. These applications generally achieve more than an order of magnitude better performance in Spark compared to Hadoop for workloads that can leverage in-memory data reuse.

Despite large speedups compared to Hadoop, Spark's performance still falls well short of the performance achievable with high performance computing (HPC) frameworks that use native code and optimized communication primitives, such as MPI [9]. For example, a recent case study found C with MPI is 4.6–10.2$\times$ faster than Spark on large matrix factorizations on an HPC cluster with 100 compute nodes [11]. Another case study on distributed graph analytics found that for weakly-connected components, one can outperform Spark's GraphX library by up to two orders of magnitude on a cluster with 16 compute nodes, with roughly 2000 lines of C++ using OpenMP for shared-memory parallelism and MPI for distributed communication [28].

The Spark environment provides many attractive features such as fault-tolerance, as RDDs can be regenerated through lineage when compute nodes are lost, as well as programmer productivity through the use of Scala and high-level abstractions. We would ideally like to be able to use libraries written for MPI and use data that originates in distributed Spark RDDs, allowing users to temporarily trade the benefits of the Spark environment in exchange for high performance. However, this creates several problems. First, RDD data lives in JVM processes that are not launched within an MPI environment, and it is not immediately obvious how one would configure an MPI environment on the fly starting with these processes. Second, even if we could configure such an environment on the fly, communicating between Spark executors would potentially violate the assumptions that Spark makes about RDDs. An RDD depending on an MPI collective operation, for example, would not have this dependence encoded in Spark's own bookkeeping and therefore may not be recoverable in the event of a failure. Due to these challenges, tight integration of MPI and Spark is generally not practiced today.

In this paper, we propose Spark+MPI, a system for integrating MPI-based programs with Spark. Our approach is to serialize data from Spark RDDs and transfer the data from Spark to inter-process shared memory for MPI processing. Using information from the Spark driver, we execute plain MPI binaries on the Spark workers with input and output paths in shared memory. The results of the MPI processing are copied back to persistent storage (HDFS), and then into Spark for further processing. This approach requires no changes to Spark itself, and it can adapt to changes in the cluster, such as adding or removing compute nodes.

We contrast this with an another approach to integrating native code with Spark, which is to accelerate user-defined functions (UDFs) in Spark by calling native C++ code through the Java Native Interface (JNI), while retain-

Proceedings of the VLDB Endowment, Vol. 10, No. 8
Copyright 2017 VLDB Endowment 2150-8097/17/04.

ing the use of Spark for distributed communication and scheduling. This optimization technique is commonly employed. For example, many operations in Spark's machine learning and graph analysis libraries offload computation to the *Breeze* library, which contains optimized numerical routines implemented in either Java or JNI-wrapped C.

To test our Spark+MPI system, we implement four distributed graph and machine-learning applications: Latent Dirichlet Allocation (LDA), PageRank (PR), Single Source Shortest Path (SSSP), and Canonical Polyadic Decomposition (CPD). We find that offloading computation from Spark to MPI using our system can provide $3.1 - 17.7\times$ speedups compared to Spark-based implementations on real-world large-scale datasets on a 12-node cluster. This includes time spent transferring data to and from an MPI environment, and any time required for MPI applications to construct any special data structures.

To summarize, our contributions are the following:

1. We propose Spark+MPI, a system demonstrating tight integration of Spark and MPI, which enables use of existing MPI-based libraries in the Spark environment.

2. Our optimization strategies result in $3.1-17.7\times$ speedups on sparse graph and machine learning algorithms compared to Spark-based implementations.

3. We quantify the overheads of Spark and show the limits of alternative optimization strategies for augmenting Spark with native code.

The rest of the paper is organized as follows. Section 2 describes related work. Section 3 describes the design and implementation details of our Spark+MPI system. Section 4 describes the four graph and machine learning applications considered for optimization. Section 5 shows the application implementations and alternative means of increasing Spark performance. Section 6 describes our experimental setup, including the hardware and software configuration, profiling tools, and datasets. Section 7 presents performance and profiling results and analysis, as well as insights gained from our experiments. Section 8 offers conclusions from our work.

2. RELATED WORK

It has been widely noted that MPI-based HPC frameworks outperform Spark or Hadoop-based big data frameworks by an order of magnitude or more for a variety of different application domains, e.g., support vector machines & k-nearest neighbors [24], k-means [14], graph analytics [25, 28], and large-scale matrix factorizations [11]. A recent performance analysis of Spark showed that compute load was the primary bottleneck in a number of Spark applications, specifically serialization and deserialization time [22]. Our performance results are consistent with this research.

Other work has tried to bridge the HPC-big data gap by using MPI-based communication primitives to improve performance. For example, Lu et al. [19] show how replacing map-reduce communicators in Hadoop (Jetty) with an MPI derivative (DataMPI) can lead to better performance; the drawback of this approach is that it is not a drop-in replacement for Hadoop and existing modules need to be re-coded to use DataMPI.

It has been shown that it is possible to extend pure MPI-based applications to be elastic in the number of nodes [23] through periodic data redistribution among required MPI ranks. However, this assumes that we are still using MPI as the programming framework, hence we do not get the other benefits of processing in the cloud with Spark or Hadoop, such as high productivity or fault tolerance.

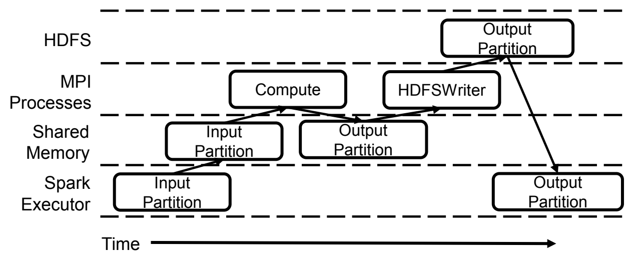

Figure 1: Spark+MPI implementation overview: data is transferred from Spark to shared memory for MPI processing, then back to Spark through HDFS.

Efforts to add fault tolerance to MPI have been ongoing since at least 2000, when Fagg and Dongarra proposed FT-MPI [8]. Although the MPI standard has still not integrated any fault tolerance mechanism, proposed solutions continue to be put forth, e.g., Fenix [10]. However, there is a large productivity gap between the APIs of Fenix and Spark.

Several machine learning libraries have support for interoperating with Spark, such as H2O.ai [13] and deeplearning4j [7], and include their own communication primitives. However, these are Java-based approaches and do not provide for direct integration of existing native-code MPI-based libraries.

SWAT [12], which stands for Spark With Accelerated Tasks, creates OpenCL code from JVM code at runtime to improve Spark performance. As opposed to our work, SWAT is limited to single-node optimizations; it does not have access to the communication improvements available through MPI.

Thrill [4] is a project building a Spark-like data processing system in C++ and using MPI for communication. Unlike Thrill, our goal is not to build an entirely new system, but rather to augment the existing successful Spark ecosystem with available performant MPI-based libraries.

3. SPARK+MPI SYSTEM DESIGN AND IMPLEMENTATION

Spark+MPI bridges the gap between big data and HPC frameworks by taking the best of both worlds: Spark fault tolerance and productivity, MPI performance. We achieve this by basing our design on RDDs, using the Spark resource management infrastructure to launch MPI processes, and providing a high-level API that integrates seamlessly in Spark.

We designed Spark+MPI as an extension to Spark which enables access to standard MPI implementations, therefore users are not required to install a special distribution of Spark or a custom MPI implementation.

3.1 Implementation

We implemented Spark+MPI using the Linux shared memory file system `/dev/shm` for exchanging data efficiently between Spark and MPI. Figure 1 depicts a high-level overview of our implementation.

We begin with data stored in a distributed RDD within Spark with the goal of using MPI-based libraries to process the data. The first step is to make the data accessible to the MPI processes. We serialize and write the data to a memory region mapped to `/dev/shm` using a `mapPartitions` transformation on the RDD that we would like to copy. The `mapPartitions` operation produces a partition ID and a host name associated with each partition, which are collected by the Spark driver. This collect operation forces Spark to perform the action and copy the data. Next, the driver uses the partition information to launch an MPI execution using the standard `mpiexec` command. The MPI execution receives a configurable amount of RAM, which is not available to Spark. The MPI execution performs the application-specific computation by consuming the binary data prepared by Spark; it produce new binary data, which is also stored in `/dev/shm`. When we are done computing using MPI, we write the results to HDFS. Then, the results are read from HDFS into Spark and stored as a new RDD. Since HDFS is persistent and fault-tolerant, storing the results of the MPI computation in HDFS guarantees that they will be preserved regardless of future failures.

3.2 API

The API of Spark+MPI is composed of Scala and C++ functions. The following Scala functions allow for a traditional Spark program to call an MPI program:

rddToNative(rdd, serializeFn)->inputHandle The first function in the API, `rddToNative`, prepares an RDD for MPI processing. `rddToNative` requires a serialization function that produces output in the binary format expected by the MPI program.

emptyNative(inputHandle)->outputHandle Creates a placeholder for the output of the MPI binary using information about the compute nodes from the RDD pre-processed by `rddToNative`.

runMPI(inputHandle, outputHandle, binary, args) Calls the MPI binary specified as a file system path using the input and output data handles and additional application-specific arguments.

nativeToHDFS(outputHandle)->hdfsPath Saves the output of the MPI binary to persistent HDFS storage.

nativeToRDD(deserializeFn, hdfsPath)->outputRDD Makes the MPI output available to further Spark processing by transforming it to an RDD. `nativeToRDD` requires a deserialization function which builds JVM objects from the application-specific binary data.

The last two functions in the API, `nativeToHDFS` and `nativeToRDD`, can be omitted when building a pipeline that repeatedly calls MPI binaries, thus amortizing the cost of HDFS I/O. Furthermore, these functions can be used as a checkpoint-restore mechanism to balance the advantages of avoiding HDFS I/O with the risk of compute node failures.

While Spark+MPI works with standard MPI implementations, applications must use our C++ API to access input and output data:

partition Structure that stores data along with metadata about the compute nodes used for processing.

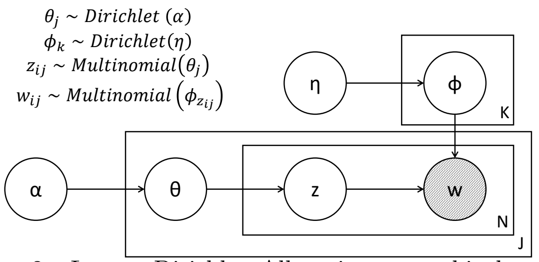
$\theta_j \sim Dirichlet\ (\alpha)$
$\phi_k \sim Dirichlet(\eta)$
$z_{ij} \sim Multinomial(\theta_j)$
$w_{ij} \sim Multinomial\left(\phi_{z_{ij}}\right)$
η
ϕ
K
α
θ
z
w
N
J

Figure 2: Latent Dirichlet Allocation - graphical model. Only the words are observed. α and η are hyperparameters. θ_j and ϕ_k are probability distributions over topics and words respectively.

binaryFiles(inputHandle)->partitionsIn Prepares C++ input data structures based on the input handle received from the Scala API.

savePartitions(partitionsOut, outputHandle) Prepares serialized output in the format required by the Scala API based on the `partition`s outputted from application-specific C++ processing.

4. APPLICATIONS

We use the following graph and machine learning algorithms as driving applications throughout the paper: Latent Dirichlet Allocation, PageRank, Single Source Shortest Path, and Canonical Polyadic Decomposition.

4.1 Latent Dirichlet Allocation (LDA)

Latent Dirichlet Allocation (LDA) tries to group documents (which are modeled as bags of words) into "topics" (which are defined as a probability distributions over words). Figure 2 shows the graphical model used in LDA. For details on the LDA model, we refer the readers to Blei et al. [5]. We use Maximum a Posteriori (MAP) estimation using an Expectation–Maximization (EM) algorithm in order to learn the topic and word distributions [3]. Assuming we have J documents and a W word vocabulary, each document j has a word count vector N_{wj} for word w. This is modeled as a bipartite graph of size $(J+W)\times(J+W)$ with edges (w,j) with weights N_{wj}. α and η are the priors to the Dirichlet distributions θ_j and ϕ_k for k topics, as shown in Figure 2.

The EM algorithm works iteratively. At each iteration, the following updates are performed.

$$\gamma_{wjk} = \frac{(N_{wk}+\eta-1)(N_{kj}+\alpha-1)}{N_k+W\eta-W} \quad (1)$$

$$\hat{\gamma}_{wjk} = \frac{\gamma_{wjk}}{\sum_k \gamma_{wjk}} \quad (2)$$

$$N_{wk} = \sum_j N_{wj}\hat{\gamma}_{wjk} \quad (3)$$

$$N_{kj} = \sum_w N_{wj}\hat{\gamma}_{wjk} \quad (4)$$

$$N_k = \sum_w N_{wk} = \sum_j N_{kj} \quad (5)$$

Once the iterations are complete, the MAP estimates of θ and ϕ are obtained as

$$\hat{\theta}_{kj} = \frac{N_{kj}+\alpha-1}{N_j+K\alpha-K} \quad \hat{\phi}_{wk} = \frac{N_{wk}+\eta-1}{N_k+W\eta-W} \quad (6)$$

4.2 PageRank (PR)

PageRank (PR) is an algorithm used to rank vertices on the basis of how probable it is for them to be encountered on a random walk along the edges of a directed unweighted graph. It was originally used to rank web pages with the pages themselves modeled as graph vertices and hyperlinks as edges. The algorithm iteratively updates the rank of each vertex according to the following equation:

$$PR^{t+1}(v) = r + (1-r) \times \sum_{u|(u,v)\in E} \frac{PR^t(u)}{\text{degree}(u)} \tag{7}$$

where $PR^t(v)$ denotes the page rank of vertex v at iteration t, E is the set of edges in a directed graph, and r is the probability of random surfing. The algorithm is run until the page ranks converge.

4.3 Single Source Shortest Path (SSSP)

Single Source Shortest Path (SSSP) is an algorithm used to find the shortest path from a given vertex to all the other vertices in a directed, weighted graph. The algorithm is used in many applications such as finding driving directions in maps or computing the min-delay path in telecommunication networks. The algorithm starts with a given vertex (called *source*) and iteratively explores all the vertices in the graph to assign a distance value to each of them, which is the minimum sum of edge weights needed to reach the vertex from the source. At each iteration t, each vertex performs the following:

$$Distance(v) = \min(Distance(v), \min_{u|(u,v)\in E}\{Distance(u) + w(u,v)\}) \tag{8}$$

where $w(u,v)$ represents the weight of the edge (u,v). Initially the *Distance* for each vertex is set to ∞ except the source with *Distance* value set to 0. We use a slight variation on the Bellman-Ford shortest path algorithm where we only update the distance of those vertices that are adjacent to those that changed their distance in the previous iteration.

4.4 Canonical Polyadic Decomposition (CPD)

Tensors are the generalization of matrices to higher dimensions (called *modes*). The CPD is one generalization of the singular value decomposition to tensors and has uses in many applications such as web search [16], collaborative filtering [26], and others [27]. The rank-F CPD models a tensor $\underline{\mathbf{X}} \in \mathbb{R}^{I_1\times\cdots\times I_N}$ with factor matrices $\mathbf{A}^{(1)} \in \mathbb{R}^{I_1\times F}$, $\ldots$, $\mathbf{A}^{(N)} \in \mathbb{R}^{I_N\times F}$. The CPD, shown in Figure 3, is most intuitively formulated as the sum of outer products.

The CPD is a non-convex optimization problem and most commonly computed using alternating least squares (ALS). ALS cyclically updates one factor matrix at a time while holding all others constant. The *matricized tensor times Khatri-Rao product* (MTTKRP) is the bottleneck of ALS and is a generalization of sparse matrix-vector multiplication (SpMV) [17]. When updating the nth factor matrix, the MTTKRP computes the matrix $\mathbf{K} \in \mathbb{R}^{I_n\times F}$ and uses the remaining $(N-1)$ factor matrices. Suppose the non-zero $\boldsymbol{v}$

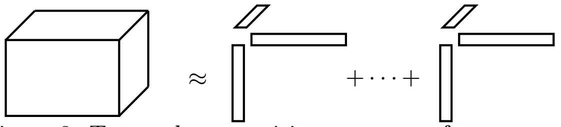

Figure 3: Tensor decomposition as a sum of outer products.

has coordinate $(i_1,\ldots,i_N)$. The MTTKRP updates $\mathbf{K}$ via:

$$\mathbf{K}_{i_n} \leftarrow \mathbf{K}_{i_n} + \boldsymbol{v}\left(\mathbf{A}^{(1)}_{i_1} * \cdots * \mathbf{A}^{(n-1)}_{i_{n-1}} * \mathbf{A}^{(n+1)}_{i_{n+1}} * \cdots * \mathbf{A}^{(N)}_{i_N}\right), \tag{9}$$

where $\mathbf{M}_i$ denotes the *ith* row of a matrix $\mathbf{M}$ and $*$ denotes the *Hadamard* (element-wise) product. Like SpMV, distributed-memory MTTKRP requires up to two communication stages: (i) aggregating partial computations and (ii) exchanging the new elements of the factor matrix [15].

5. APPLICATION IMPLEMENTATIONS

In this section we describe our strategies for implementing and optimizing the aforementioned applications in Spark. First, we consider Spark-based libraries as a baseline. Second, we consider offloading key computations into optimized C++ routines and calling these using JNI, while retaining the use of Spark for distributed communication and scheduling. Third, we offload the entire computations from Spark into the MPI environment using our Spark+MPI system, which allows us to leverage existing high-performance MPI-based libraries.

5.1 Spark libraries

Three of the four applications we consider are readily available in existing Spark libraries. We use the built-in PR provided by GraphX [33]. For SSSP, we use the Pregel framework in GraphX. For LDA, we use the MLlib [21] implementation which is based on Expectation–Maximization (EM), and also uses GraphX data structures and computational primitives. These implementations store both vertex properties and edges as distributed RDDs, and use shuffles to communicate this data between machines during each iteration.

We implemented CPD ourselves, as no Spark implementations exist, to the best of our knowledge. Our version uses Spark's parallel primitives for the distributed MTTKRP operation in Equation 9. The results of the MTTKRP operations are collected by the driver. Operations on the factors are done on the driver node using dense linear algebra routines provided in the *Breeze* library.

5.2 Optimized Spark

Recently, MLlib added support for optimized linear algebra operations on sparse and dense matrices [6]. One suggested usage model for sparse matrix operations in Spark is to do matrix operations in a distributed fashion and vector operations on the driver. This means that vectors are first broadcast to the executors, operated on, and then collected back to the driver during each iteration of an algorithm. If the vectors do not fit on the driver, then a different set of matrix-vector routines can be used which distribute the vectors as well as the matrices. We implemented our applications following the same communication and computation model as Spark's sparse matrix-vector (SpMV) multiplications [6]. We refer to our implementations as OptSpark.

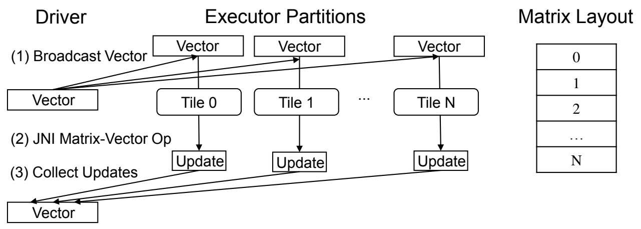

Figure 4: OptSpark: iterative sparse algorithms with JNI, modeled after MLlib's sparse matrix operations. Matrix data is stored in 1D partitions on executors where it is process in a distributed fashion, while vector operations take place on the driver.

For our applications and hardware configuration, the vectors do fit on the driver and so we implemented the applications based on the MLlib 1D SpMV communication pattern, shown in Figure 4. For example, for PR, the graph dataset becomes a sparse matrix with a nonzero for every edge. In the case of a Twitter follower-followee graph, the vector stores a single ranking for each Twitter user. The matrix rows (i.e., the Twitter users) are distributed across executors. At every iteration, the vector is first broadcast from the driver to the executors. For PR and SSSP, a custom matrix-vector operation is implemented using C++ routines made available through the Java Native Interface (JNI). This custom matrix-vector operation is applied to the input vector and returned to the driver using `collect`, where it is used to update the vector for the next iteration.

We implemented OptSpark PR and SSSP using the aforementioned approach, with the following implementation details. In order to maximize the granularity of our calls through JNI, we merge all rows within a partition of the sparse matrix into a single Compressed Sparse Row (CSR) representation. Each array that is needed inside the native function is accessed using `GetPrimitiveArrayCritical` to avoid unnecessary copies between Java and native arrays. We assume that the partition ID does not change during execution, so we can collect the matrix row indices on the driver ahead of time which reduces the total size of the collect. The driver also uses OpenMP-parallelized JNI functions to update the vectors before and after each iteration.

For LDA and CPD, we provide a best-case per-iteration performance estimate using the following micro-benchmark. We simulate algorithm iterations with the following steps: (1) broadcast random data with type `Array[Double]` to the workers, (2) perform a map over each partition of the dataset, where each task copies a portion of the broadcast variable into a result array, and (3) collect partitions of the result array, packaged as `(Int, Array[Double])`, to the driver. The broadcast and collected data is sized according to the algorithm being simulated. This micro-benchmark gives us a lower-bound estimate of runtime per iteration, by assuming that the matrix-vector operation compute time is negligible. For the LDA micro-benchmark we broadcast a dense double-precision matrix of size $(J+W)\times K$ and collect a matrix of the same size back to the driver. For the CPD micro-benchmark, we do one broadcast/collect for each factor. Broadcasting all CPD factors at once caused caused problems in Spark due to the large total size of the factor matrices.

5.3 Spark+MPI

For the Spark+MPI implementation of the applications, we use two readily available MPI-based libraries.

For PR, SSSP, and LDA, we use GraphMat [32], a framework for graph analytics. It provides high performance by leveraging optimized OpenMP-parallel sparse matrix primitives. GraphMat was recently extended to distributed systems through the integration of MPI-based distributed primitives [2]. Graph data is distributed across nodes in a 2D block cyclic format and vertex properties are distributed across nodes as well. Messages derived from vertex properties are compressed into sparse vectors and sent to other nodes using MPI. Local sparse matrix-vector multiplications are parallelized with OpenMP.

For CPD, we use the Surprisingly ParalleL spArse Tensor Toolkit (SPLATT), an MPI+OpenMP library for factoring large, sparse tensors [30]. It uses a *medium-grained* decomposition which distributes an N-mode tensor over an N-dimensional grid of MPI processes [31]. The medium-grained decomposition requires two communication stages, but limits communication in the ith mode to processes that share a coordinate in the ith dimension of the process grid.

6. EXPERIMENTAL SETUP

To test the performance of Spark+MPI, we perform experiments on a cluster with up-to-date hardware and software using datasets representative for the applications we consider.[1]

6.1 Configuration

Each machine in our cluster has 2 Intel® Xeon® E5-2699 v4 processors (22 cores, 44 threads each), 128 GiB DDR4 RAM, an SSD, a 1 Gbit/s Ethernet interface, and a 100 Gbit/s Intel® Omni-Path interface. We use CentOS 7.2 (with Linux 3.10), Intel® Compiler 16.0.3, Intel® MKL 11.3.3, Intel® MPI 5.1.3, Oracle Java 1.8.0, HDFS 2.7.2, and Spark 2.0.0.

Our cluster has 13 machines. We run the HDFS and Spark master on one machine, the Spark driver on another machine, and 11 Spark workers on the remaining machines in the cluster. When we run MPI on the cluster, it uses the 12 compute nodes (the driver + 11 workers).

Spark is configured to use the G1GC garbage collector and the Kryo serializer. We considered (1) allowing Spark to choose the default number of partitions based on the number of HDFS blocks, (2) setting the number of partitions to the total number of cores, and (3) setting the number of partitions to twice the number of cores. We found that (1) provided the best overall performance for all Spark implementations except for CPD which excelled with (3). We use this setting for our results. We also use the default partitioning strategy for GraphX data structures. When Spark is running alone, we give 96 GiB of memory for both the driver and the executors. When we run MPI alongside Spark, we give the Spark driver and executors 80 GiB of memory. We

[1]Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information go to http://www.intel.com/performance

set the number of driver and executor cores to 88, which is equal to the number of threads the CPUs can execute concurrently, meaning that one executor runs per worker. When we launch MPI, we set the number of OpenMP threads to 44, equal to the number of cores in each machine. We run one MPI process per machine.

We disregard the time to initially load input datasets from HDFS into Spark. Each benchmark begins timing only after the input dataset has been loaded from HDFS, stored in an RDD which is cached using `cache`, and then counted which ensures that the previous tasks complete before moving on to the rest of the application. Because Spark uses lazy execution, we need to take actions such as these to ensure that all computations are actually executed at the expected time during each benchmark. PR, SSSP, and LDA all produce their outputs in RDDs. We call `cache` and `count` on these outputs to do this, and we include the runtime for this overhead as part of the total algorithm runtime. For CPD, both the Spark implementation and Spark+MPI implementation produce outputs on the driver. The overhead time to transfer these outputs from RDDs to the driver for the Spark+MPI implementation is included as part of the total algorithm runtime. We also call `cache` and `count` on the Spark data structures after their construction and include this as part of Spark data structure construction time.

PR, LDA, and CPD all use double-precision arithmetic for all implementations. For SSSP, we store the depths as Int for all implementations. GraphX represents vertex IDs as Long, while GraphMat and OptSpark store these as Int.

We run PR for 54 iterations, which is the total number of iterations required for convergence in GraphMat for our dataset (for Spark and SparkOpt, we set the number of iterations to this number). We solve LDA with 20-dimensional factors, α and η equal to the Spark defaults (3.5 and 1.1, respectively), and run for 10 iterations. We run CPD with 10-dimensional factors for 22 iterations, which is the total number of iterations needed for convergence in SPLATT. SSSP naturally runs the same number of iterations in each implementation. Micro-benchmarks run for 10 iterations total and the runtime of the first iteration is discarded.

For system-level performance analysis, we use the `sar` tool from the Sysstat package executed with the Intel Performance Analysis Tool[2]. We collect samples every 1 s for CPU usage and network transfer.

To analyze Spark implementations, we developed an event log parser that computes aggregate metrics from the Spark event log for the following components: scheduler delay, task deserialization and result serialization, data shuffling, and algorithm computation. To identify bottlenecks, we calculate the *observed* time of each of these components while accounting for task-level parallelism. For example, to calculate the observed data shuffling time for a stage, the parser reconstructs the task events from the timestamps, and then subtracts the time for data shuffling in each task to compute the stage runtime as if without the data shuffling. The observed data shuffling time is the difference between this hypothetical stage runtime and the actual stage runtime.

6.2 Datasets

Table 1 summarizes the datasets used in our performance evaluation. `Twitter` is a graph of the "follower-followee" relationships from 41.7 million user profiles in 2010. We use `Twitter` for both PR and SSSP. For LDA, we use `Wikipedia`, a sparse term-frequency matrix formed from 3.3 million Wikipedia articles. For CPD, we use `Amazon`, which is a sparse tensor of *user-product-word* triplets taken from product reviews. Each non-zero $\underline{\mathbf{X}}_{uiw}$ in `Amazon` is the frequency of word w appearing in user u's review of product i.

[2]https://github.com/intel-hadoop/PAT

Table 1: Dataset Summary

Dataset	Records	Record Type
`Wikipedia` [1]	0.7 billion	(DocID, WordID, Count)
`Twitter` [18]	1.5 billion	(SrcID, DstID, Weight)
`Amazon` [29, 20]	1.7 billion	(UserID, ItemID, WordID, Count)

7. RESULTS

In this section, we provide performance evaluation results, as well as insight into the bottlenecks that are behind the results.

7.1 Overall performance

Table 2 shows the runtime for all implementations on each of the four algorithms. These runtimes are broken down further into categories:

- **Build Time:** Time to construct a data structures needed for computing the algorithm result, for example going from a collection of records to a graph or sparse matrix data structure. This step can potentially be amortized by running several similar algorithms on the same data.
- **Compute Time:** Time which is specific to a given algorithm, where the inputs and outputs exist in the most easily accessible formats for a given implementation. This includes both computation and any internode communication required for the algorithm.
- **Overhead:** The total time minus the build time and compute time. For Spark+MPI this includes the time to transfer data between the Spark and MPI environments. It also includes the time to construct the correct output format (e.g., cached RDD or on driver) in Spark.
- **Total Time:** The time to compute the result, beginning with a cached RDD and ending with the application result (cached RDD for PR, SSSP, and LDA ; on driver for CPD)

We also present the following speedup metric, which is normalized to the runtime of the Spark implementation for each algorithm.

- **Overall Speedup:** The ratio of one implementation's total time to another implementation's total time. This considers all overheads, including building the data structures, computing the result, and any overheads we cannot attribute more specifically.

The OptSpark implementations are able to achieve a 1.8× and 1.9× overall speedup on PR and SSSP, respectively. The overall speedup for OptSpark SSSP, compared to Spark SSSP, is substantially less than its improvement in compute time. This is due to the relatively large cost of building

Table 2: Runtime Summary

Application	Implementation	Build Time (s)	Compute Time (s)	Overhead (s)	Total Time (s)	Overall Speedup
PR	Spark	28.8	413.9	0.6	443.3	-
	OptSpark	67.5	214.1	5.3	286.8	1.9
	Spark+MPI	8.8	9.1	17.6	35.5	12.5
SSSP	Spark	27.8	150.6	0.6	178.9	-
	OptSpark	62.7	34.2	4.5	101.4	1.8
	Spark+MPI	9.0	2.1	21.6	32.6	5.5
LDA	Spark	20.9	449.0	0.6	470.5	-
	Spark+MPI	4.1	19.4	18.7	42.2	11.1
CPD	Spark	0.0	1883.1	0.0	1883.1	-
	Spark+MPI	38.7	33.8	33.8	106.3	17.7

the sparse matrices used in OptSpark SSSP. This cost could potentially be amortized across multiple algorithms.

The build time varies across implementations because the details of the data structures needed for each is different. In PR, for example, Spark+MPI does a global shuffle and builds a compressed-sparse-row matrix which is used for further computation. The OptSpark implementation also builds a sparse matrix, but it does it in Spark using the *GroupByKey* primitive. The Spark implementation builds a GraphX graph which consists of a vertex RDD and edge RDD.

The Spark+MPI implementations achieve 5.5−17.7× overall speedup compared to Spark, and a 3.1× and 8.1× overall speedup compared to OptSpark on SSSP and PR, respectively. The overall speedup includes all overheads for transferring data to and from Spark. Spark+MPI spends the majority of its time in build time and overhead. Therefore, the Spark+MPI approach could become more efficient if we amortized these overheads, for example by running more iterations of each algorithm, or by running multiple MPI analyses on the same data.

We compare the per-iteration runtime of the PR micro-benchmark to the real per-iteration runtime of OptSpark PR. The micro-benchmark shows an average per-iteration runtime of 2.9 s, and a minimum of 2.5 s, compared to an average per-iteration runtime of 3.5 s for OptSpark. The LDA micro-benchmark shows a per-iteration runtime average of 14.9 s, and a minimum of 12.6 s. The CPD micro-benchmark shows a per-iteration runtime average of 19.2 s, and a minimum of 14.6 s. These per-iteration best-case runtime estimates are well above the measured runtimes of Spark+MPI for LDA and CPD. No Spark implementation of these algorithms, using this communication pattern, can reduce the cost of this broadcast. Therefore, this is a fundamental limitation of Spark implementations which use the aforementioned broadcast/reduce pattern, compared to our Spark+MPI implementation.

7.2 Performance vs. number of iterations

Even for the relatively costly algorithms considered in this paper, the best approach to take depends on the number of iterations executed and whether or not data structure construction can be amortized. Figure 5 shows the total runtime for PR, including build time, for different runs from 1 to 20 iterations. In this situation, the Spark implementation has roughly the same performance as Spark+MPI for small numbers of iterations. Spark is also faster than OptSpark in these cases, since the OptSpark build time and degree calculation are more costly than Spark's. After about four iterations, Spark+MPI begins to be more efficient. Though not shown on the graph, OptSpark eventually overtakes Spark due to its faster per-iteration runtime.

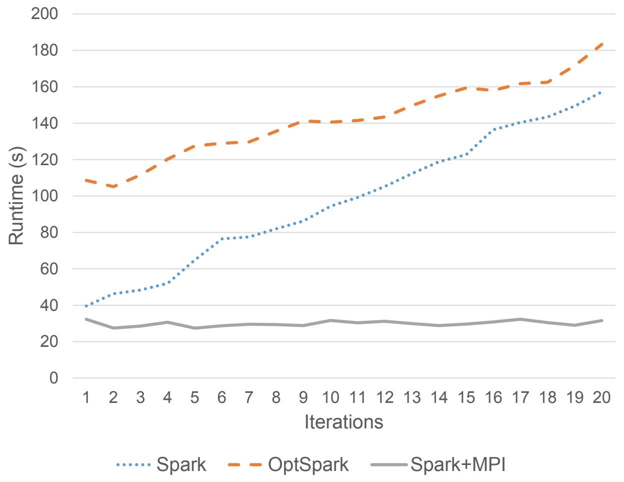

Figure 5: Total runtime for PR, for different runs between 1 and 20 iterations.

7.3 System-level profiling

We investigate the resource usage of each application to understand the performance differences between the systems. Figures 7 and 8 shows the mean CPU usage averaged across workers during the compute portion of the runtime for PR and LDA respectively. Recall that we sample CPU usage every second. For Spark implementations, we note a cyclic pattern of high and low CPU usage periods corresponding to the iterations in the applications, especially for LDA. Overall, for Spark+MPI, CPU usage is significantly higher and more consistent. MPI communication is faster and does not keep the CPU idle. Finally, OptSpark exhibits a pattern similar to Spark, but has a lower CPU usage, indicating that the more efficient native code is hindered by Spark communication primitives.

Table 3 shows the total CPU and network utilization averaged across workers during the compute portions of the applications. The CPU usage difference between Spark and Spark+MPI is approximately in proportion to the application compute time difference. Spark workers perform much more computation to achieve the same results as Spark+MPI workers. On the other hand, OptSpark workers perform ap-

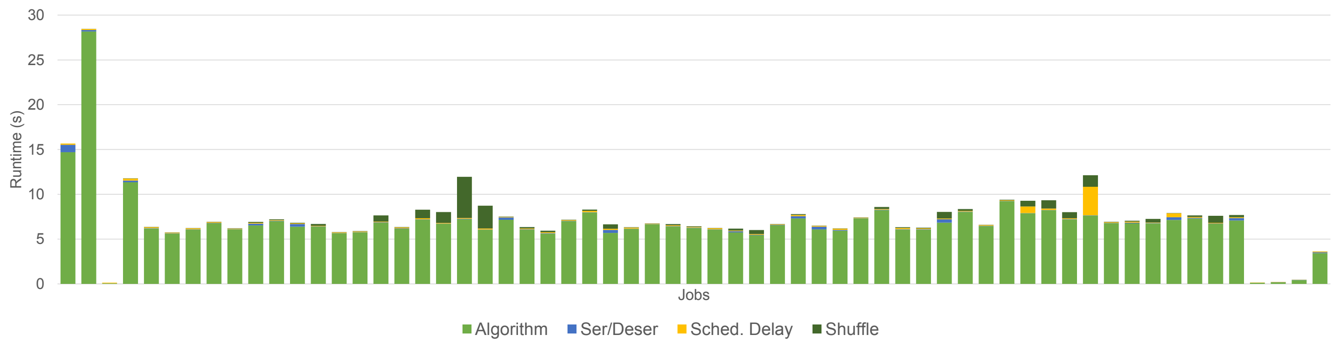

Figure 6: Spark PR: Job profiling from Spark logs.

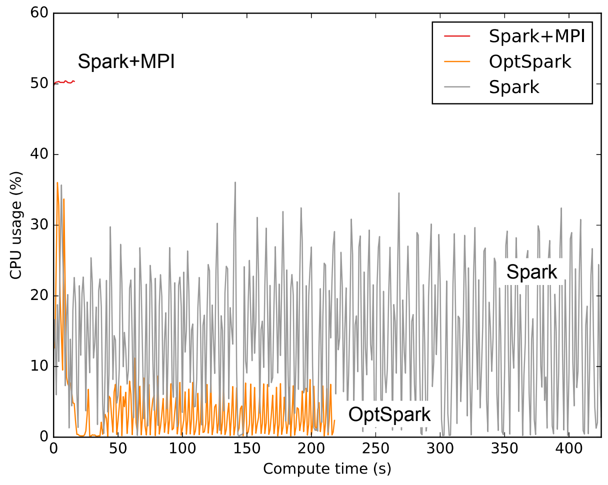

Figure 7: PR: CPU utilization of compute portion only.

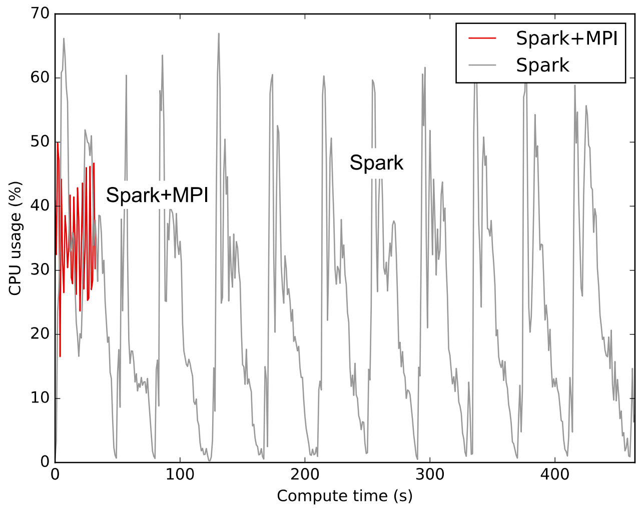

Figure 8: LDA: CPU utilization of compute portion only.

Table 3: CPU and network total utilization during compute.

App	System	CPU (s)	Network (Gbit)
LDA	Spark	111	284
	Spark+MPI	11	431
PR	Spark	61	481
	OptSpark	9	217
	Spark+MPI	9	282
SSSP	Spark	21	70
	OptSpark	4	14
	Spark+MPI	2	33
CPD	Spark	547	598
	Spark+MPI	23	105

proximately the same amount of computation as Spark+MPI workers. Therefore, the compute time performance gap between OptSpark and Spark+MPI is attributable to other factors, which we explore in Section 7.5.

7.4 Spark profile

Figure 6 shows a breakdown of each job for PR in Spark, computed with our log parser using data from Spark's logs. There is one job per iteration, and the algorithm runs for 54 iterations. The large runtimes for early jobs are likely because of data structure construction. For a typical iteration, the vast majority of the runtime is classified as *algorithm time*, which means time that is spent executing the task. Algorithm time is distinct from scheduler overhead, serialization and deserialization time for the task, as well as shuffle time.

Though not shown here, the other applications have the same property with respect to algorithm time. For PR, SSSP, LDA, and CPD, algorithm time accounts respectively for 92%, 95%, 95%, and 91% of the total job time. We can relate this with the results of system-level profiling in Section 7.3. These results show that during the compute portion, Spark implementations used significantly more overall CPU resources than either OptSpark or Spark+MPI implementations. The logs suggest that, for Spark implementations, these CPU cycles are spent inside of the Spark tasks.

Figure 9 is a screenshot from the Java Flight Recorder profiler, showing the state of six threads (vertical axis) over time (horizontal axis) running several iterations of PR on one cluster node. We observe load imbalance across threads; furthermore, we have noticed load imbalance across cluster nodes from the logs (not depicted). We attribute the load

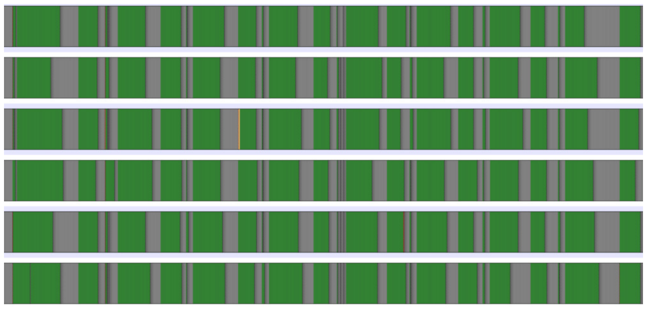
Figure 9: Spark PR Java Flight Recorder profile for 6 executor threads running several iterations. Green indicates a running thread, gray indicates a parked thread (in this case, waiting for a task).

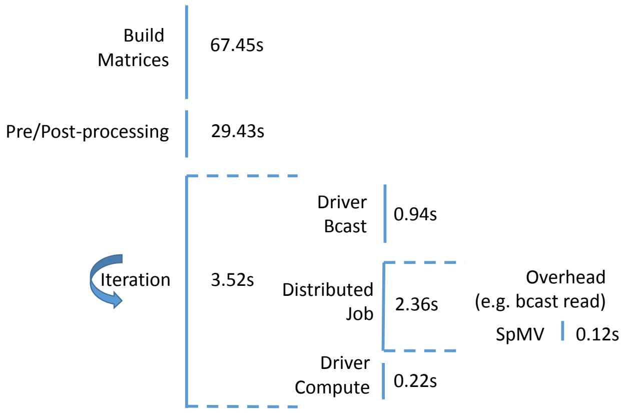

Figure 10: OptSpark PR Profile

imbalance to the irregular graph computation, combined with Spark's barrier-based execution. Using the profiler, we also note that fewer threads are being used than are available in the CPUs, because Spark is using fewer partitions than available cores, for best performance (see Section 6.1. Overall, the Java Flight Recorder analysis explains the jigsaw aspect of the CPU usage in Figure 7.

7.5 OptSpark profile

As shown in Table 3, the compute portion of the OptSpark implementation uses roughly an order of magnitude fewer CPU resources than the Spark implementations, and roughly the same total CPU resources as the Spark+MPI implementation. However, the performance of OptSpark is still well below the performance of Spark+MPI. We use the Spark logs, as well as our own custom application timestamp profiling to elucidate OptSpark's performance profile.

We performed application-level breakdowns of the PR and SSSP OptSpark implementations, which are shown in Figures 10 and 11. The applications begin with a large amount of time devoted to building data structures (matrices). We also delineate pre/post processing time, such as computing vertex degrees for PR and constructing and caching the output RDDs. Within the main iteration loop of each algorithm, we separate the runtime into (1) the average time spent by the driver to prepare the broadcast variable (`sc.broadcast()`), (2) the average time spent in the distributed job, including the `collect()` operation, and (3) average time the driver spends computing the vector for the

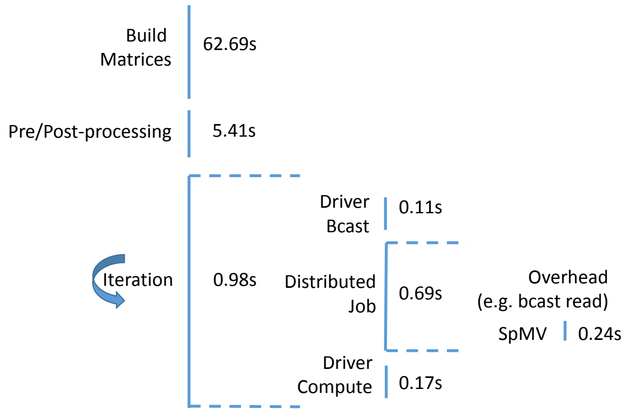

Figure 11: OptSpark SSSP Profile

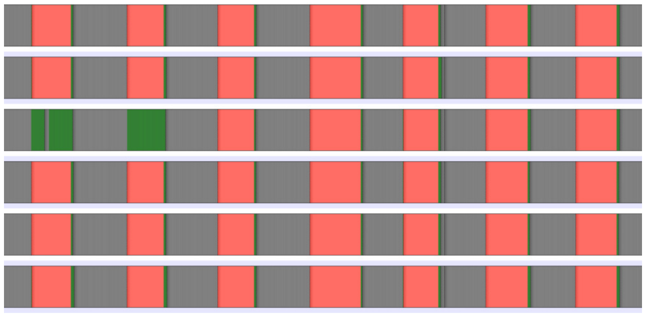
Figure 12: OptSpark PR Java Flight Recorder profile for several executor threads. Green indicates a running thread, gray indicates a parked thread (in this case, waiting for a task), and pink indicates a blocked thread (in this case, waiting for the broadcast variable to be read).

next iteration. Within the distributed job, we also calculate the time spent doing the SpMV (single JNI call per task). For this, we compute the maximum JNI runtime for each job across all executors, which determines the actual duration of the job, and average this across all iterations.

A significant portion of the OptSpark PR runtime is not spent actually computing the SpMV. Instead, it is spent either broadcasting the variable on the driver, reading the broadcast variable on the executor, or collecting the result on the driver. In fact, the average time per iteration we measured in the JNI SpMV is roughly comparable to the average time per iteration in our Spark+MPI native SpMV implementation. This indicates that the bottleneck for OptSpark PR lies in the broadcast/reduce overheads. This additionally helps to explain why the total CPU resource usage is lower for OptSpark. Operations, such as reading the broadcast variable, may create a dependency which can delay all running tasks.

For OptSpark SSSP, the time spent in the JNI SpMV portion is higher than in OptSpark PR. This is because SSSP transfers less data from driver to executors per iteration than OptSpark PR, due to sparsity inherent in SSSP. However, the JNI SpMV portion still only accounts for 35% of the average distributed job runtime, which suggests that broadcast and collect are also a significant performance bottleneck for OptSpark SSSP.

Figure 12, a Java Flight Simulator PR screenshot, shows very little compute time in each thread, which confirms our

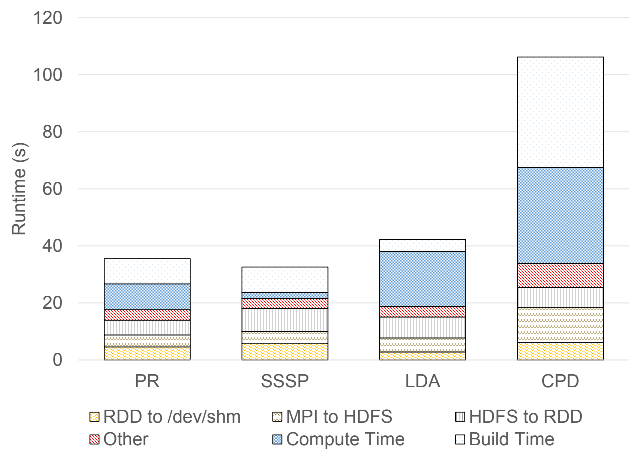

Figure 13: Spark+MPI: Breakdown of the overheads incurred when transferring data between Spark and MPI.

application profiling results in Figure 10. In Spark applications, the broadcast variable is read by only one thread and the other threads are blocked during this time; this is represented in the pink in the figure. Note the short SpMV computation in each thread (green) after the broadcast variable is read. Gray indicates the results are being collected by the driver, the driver is computing alone, or the driver is preparing the next broadcast variable.

7.6 Spark+MPI profile

Figure 13 shows a breakdown of total Spark+MPI runtime, including overheads incurred when transferring data between the Spark and MPI environments. PR and SSSP overheads are almost identical because the sizes of the input graph and output data are roughly the same. However, the SSSP computation is much smaller so this means that the overheads are a much larger percentage of total SSSP runtime compared to PR. The time to transfer data from Spark to shared memory, and from HDFS back into Spark, is roughly proportional to the size of the input and output data. The portion of time spent in compute ranges from a minimum of 6% (SSSP) to a maximum of 46% (LDA). This show great potential for gains through repeated MPI computations in a pipeline, which would amortize the overheads.

7.7 Fault tolerance

One of the benefits of Spark is its ability to detect and recover from faults. When one or more RDD partitions are discovered to be missing, if the missing RDD partitions cannot be found in persistent storage, Spark is able to regenerate them by examining the record of RDD transformations which generated the missing partitions and re-executing operations if necessary.

Since Spark+MPI performs operations outside of the Spark environment, we can not rely entirely on Spark's recovery mechanisms. Instead, we restart the entire Spark+MPI computation in response to a fault. We perform experiments to quantify the performance cost of our approach compared to Spark's fault recovery for the same workload. We simulate faults by killing the HDFS and Spark processes of one of the cluster nodes after the last iteration of the algorithm commences. This represents the worst-case fault timing.

Table 4: Fault recovery runtime for 10 iterations of PR on `Twitter`.

	No fault (s)	Restart (s)	Recovery (s)
Spark	90.5	235.4	2461.6
Spark+MPI	32.1	85.5	-

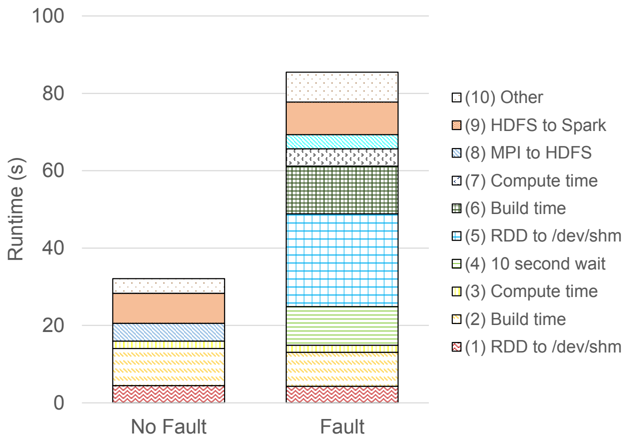

Figure 14: Spark+MPI PR runtime with restart after fault.

Table 4 shows the performance of Spark and Spark+MPI in the presence of the aforementioned fault for 10 iterations of PR. *No fault* is the regular runtime without any fault. *Restart* is the runtime with a fault simulated by killing the HDFS and Spark processes and then restarting the computation in the user code. *Recovery* is the time it takes Spark to regenerate the output using lineage information, which means we simulate the fault by killing processes, then call *count* on the RDD which contained the output of PR. There is a 2.6× overall slowdown in Spark if you completely restart the computation from the beginning, and a 27× slowdown for relying on recovery from lineage. We therefore conclude that Spark libraries could benefit from application-level fault recovery.

Figure 14 shows the runtime breakdown for Spark+MPI when there is no fault, and when there is a complete restart after fault detection. The overall runtime is 2.7× higher in the case of a fault compared to no fault. The runtime is broken down into ten segments which occur chronologically: (1) Copying the RDD to shared memory for the first time, (2) building the graph data structure, (3) computing PR iterations, (4) 10 seconds of wait time while killing process, (5) Spark regenerating the input RDD after restart and copying it to shared memory for the second time, (6) building the graph data structure for the second time, (7) computing PR iterations for the second time, (8) writing the results to HDFS from MPI, (9) reading the results from HDFS into Spark, and (10) time otherwise unaccounted for.

It takes much longer to copy the RDD after the fault, shown in segment (5), than before the fault. This is due to Spark needing to regenerate the inputs from HDFS. It also takes longer to (6) build data structures in MPI, and (7) compute in MPI during the second run. Since there are only 11 compute nodes, a prime number, GraphMat is forced to use a 1D layout instead of the usual 2D layout, which hurts performance.

7.8 Summary

Spark+MPI has runtime and memory overheads, which depend on the size of input and the output of the operation. Among our four application experiments, it takes between $3 - 5$ seconds to transfer inputs to MPI through shared memory, and between $9 - 19$ seconds to transfer outputs back into Spark through HDFS, plus between $4 - 8$ seconds of otherwise unaccounted time which may be related to data transfer. By comparison, simple operations in Spark, such as *count*, can complete in one second or less on the same datasets. Using Spark+MPI only makes sense for operations which run slower in Spark than these fixed overheads. Simple operations, such as scaling a vector stored in an RDD, should be done in Spark.

As long as there is enough work to amortize these fixed overheads, applications can benefit from Spark+MPI. However, some other applications may run with similar efficiency in Spark, namely those which have less communication volume and frequency. Since the main benefit of MPI is optimized communication primitives, applications which have more communication and more complex communication patterns will benefit relatively more from Spark+MPI.

8. CONCLUSIONS

There is a large gap between the efficiency of Spark and MPI. This gap is exemplified by the variance in compute time we observed for Spark, OptSpark, and Spark+MPI. We sought to explain the nature of the gap by optimizing and profiling Spark to better understand its performance. The system we introduce in this paper, Spark+MPI, is a solution to bridging the gap, as proven quantitatively in our evaluation on real applications and datasets.

In profiling Spark, we observed that the Spark implementations consume a large amount of CPU resources compared to OptSpark and Spark+MPI implementations. Furthermore, the Spark logs tell us that, for the Spark implementations, the time is being spent in algorithm time, running on the executors. The OptSpark and Spark+MPI implementations of PageRank show that the same algorithm can be implemented using much fewer CPU cycles. Therefore, we conclude that the Spark implementations are making inefficient use of CPU resources. At the same time, considering OptSpark, we conclude that the performance of implementations based on Spark's communication primitives is greatly impeded by Spark's broadcast and collect operations.

Despite its overheads, our Spark+MPI system provides better performance than the alternatives for the applications and datasets considered in this paper. For algorithms with fewer iterations, less work per iteration, or opportunities to amortize data structure build times in Spark, it may be more efficient to stay in Spark to avoid the overheads of Spark+MPI. We demonstrated this by measuring PR runtimes with different numbers of iterations. On the other hand, finding larger chunks of work to do in MPI would make it is even more advantageous to use our proposed system because the Spark+MPI overheads could be amortized further.

In addition to overhead, another potential drawback of the Spark+MPI approach is that it must share memory with Spark. In this paper, we statically partition the memory between Spark and MPI, which is sufficient for these applications. However, to accommodate MPI and Spark applications with more diverse memory requirements, we are considering for future work a mechanism to allow MPI to borrow memory from Spark.

We will release Spark+MPI as open-source software.

9. REFERENCES

[1] D. C. Anastasiu and G. Karypis. L2knng: Fast exact k-nearest neighbor graph construction with l2-norm pruning. In *Proceedings of the 24th ACM International Conference on Information and Knowledge Management*, CIKM 2015, pages 791–800, New York, NY, USA, 2015. ACM.

[2] M. J. Anderson, N. Sundaram, N. Satish, M. M. A. Patwary, T. L. Willke, and P. Dubey. GraphPad: Optimized graph primitives for parallel and distributed platforms. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 313–322, May 2016.

[3] A. Asuncion, M. Welling, P. Smyth, and Y. W. Teh. On smoothing and inference for topic models. In *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence*, UAI 2009, pages 27–34, Arlington, Virginia, United States, 2009. AUAI Press.

[4] M. Axtmann, T. Bingmann, E. Jöbstl, S. Lamm, H. C. Nguyen, A. Noe, M. Stumpp, P. Sanders, S. Schlag, and T. Sturm. Thrill - distributed big data batch processing framework in C++. `http://project-thrill.org/`, 2016.

[5] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent Dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022, Mar. 2003.

[6] R. Bosagh Zadeh, X. Meng, A. Ulanov, B. Yavuz, L. Pu, S. Venkataraman, E. Sparks, A. Staple, and M. Zaharia. Matrix computations and optimization in Apache Spark. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD 2016, pages 31–38, New York, NY, USA, 2016. ACM.

[7] deeplearning4j. Deep Learning for Java. Open-Source, Distributed, Deep Learning Library for the JVM. `http://deeplearning4j.org/`, 2016.

[8] G. E. Fagg and J. J. Dongarra. FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface (EuroPVM/MPI)*, pages 346–353. Springer Nature, 2000.

[9] M. P. I. Forum. Mpi: A message-passing interface standard version 3.1. Technical report, 2015.

[10] M. Gamell, D. S. Katz, H. Kolla, J. Chen, S. Klasky, and M. Parashar. Exploring automatic, online failure recovery for scientific applications at extreme scales. In *International Conference for High Performance Computing, Networking, Storage and Analysis.* IEEE, 2014.

[11] A. Gittens, A. Devarakonda, E. Racah, M. Ringenburg, L. Gerhardt, J. Kottalam, J. Liu, K. Maschhoff, S. Canon, J. Chhugani, P. Sharma, J. Yang, J. Demmel, J. Harrell, V. Krishnamurthy, M. W. Mahoney, and Prabhat. Matrix factorizations at scale: A comparison of scientific data analytics in

Spark and C+MPI using three case studies. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 204–213, Dec 2016.

[12] M. Grossman and V. Sarkar. SWAT: A programmable, in-memory, distributed, high-performance computing platform. In *International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pages 81–92, New York, New York, USA, 2016. ACM Press.

[13] H2O.ai. Sparkling Water. `https://github.com/h2oai/sparkling-water`, 2016.

[14] S. Jha, J. Qiu, A. Luckow, P. Mantha, and G. C. Fox. A tale of two data-intensive paradigms: Applications, abstractions, and architectures. In *IEEE International Congress on Big Data*. IEEE, 2014.

[15] O. Kaya and B. Uçar. Scalable sparse tensor decompositions in distributed memory systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 77. ACM, 2015.

[16] T. G. Kolda and B. Bader. The TOPHITS model for higher-order web link analysis. In *Proceedings of Link Analysis, Counterterrorism and Security 2006*, 2006.

[17] T. G. Kolda and B. W. Bader. Tensor decompositions and applications. *SIAM review*, 51(3):455–500, 2009.

[18] H. Kwak, C. Lee, H. Park, and S. B. Moon. What is Twitter, a social network or a news media? In *WWW*, pages 591–600, 2010.

[19] X. Lu, F. Liang, B. Wang, L. Zha, and Z. Xu. Datampi: Extending MPI to Hadoop-like big data computing. In *28th IEEE International Parallel and Distributed Processing Symposium*, pages 829–838, May 2014.

[20] J. McAuley and J. Leskovec. Hidden factors and hidden topics: understanding rating dimensions with review text. In *Proceedings of the 7th ACM conference on recommender systems*, pages 165–172. ACM, 2013.

[21] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar. MLlib: Machine learning in Apache Spark. *J. Mach. Learn. Res.*, 17(1):1235–1241, Jan. 2016.

[22] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun. Making sense of performance in data analytics frameworks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI 2015, pages 293–307, Berkeley, CA, USA, 2015. USENIX Association.

[23] A. Raveendran, T. Bicer, and G. Agrawal. A framework for elastic execution of existing MPI programs. In *IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum (IPDPSW)*, pages 940–947, May 2011.

[24] J. L. Reyes-Ortiz, L. Oneto, and D. Anguita. Big data analytics in the cloud: Spark on Hadoop vs MPI/OpenMP on Beowulf. *Procedia Computer Science*, 53:121 – 130, 2015.

[25] N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey. Navigating the maze of graph analytics frameworks using massive graph datasets. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD 2014, pages 979–990, New York, NY, USA, 2014. ACM.

[26] Y. Shi, A. Karatzoglou, L. Baltrunas, M. Larson, A. Hanjalic, and N. Oliver. TFMAP: optimizing map for top-n context-aware recommendation. In *Proceedings of the 35th International ACM SIGIR conference on Research and development in information retrieval*, pages 155–164. ACM, 2012.

[27] N. D. Sidiropoulos, L. De Lathauwer, X. Fu, K. Huang, E. E. Papalexakis, and C. Faloutsos. Tensor decomposition for signal processing and machine learning. *arXiv preprint arXiv:1607.01668*, 2016.

[28] G. M. Slota, S. Rajamanickam, and K. Madduri. A case study of complex graph analysis in distributed memory: Implementation and optimization. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 293–302, May 2016.

[29] S. Smith, J. W. Choi, J. Li, R. Vuduc, J. Park, X. Liu, and G. Karypis. FROSTT: The formidable repository of open sparse tensors and tools, 2017.

[30] S. Smith and G. Karypis. SPLATT: The Surprisingly ParalleL spArse Tensor Toolkit. `http://cs.umn.edu/~splatt/`.

[31] S. Smith and G. Karypis. A medium-grained algorithm for distributed sparse tensor factorization. In *30th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2016)*, 2016.

[32] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey. GraphMat: High performance graph analytics made productive. *Proc. VLDB Endow.*, 8(11):1214–1225, July 2015.

[33] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. GraphX: A resilient distributed graph system on Spark. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES 2013, pages 2:1–2:6, New York, NY, USA, 2013. ACM.

[34] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI 2012, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.