Anatomy of Machine Learning Algorithm Implementations in MPI, Spark, and Flink

Supun Kamburugamuve*, Pulasthi Wickramasinghe*, Saliya Ekanayake†, Geoffrey C. Fox*
*School of Informatics and Computing
Indiana University, Bloomington
{skamburu, pswickra, gcf}@indiana.edu
† Network Dynamics and Simulation Science Laboratory
Biocomplexity Institute, Virginia Tech, Blacksburg, VA
{esaliya}@vt.edu

***Abstract*—With the ever-increasing need to analyze large amounts of data to get useful insights, it is essential to develop complex parallel machine learning algorithms that can scale with data and number of parallel processes. These algorithms need to run on large data sets as well as they need to be executed with minimal time in order to extract useful information in a time constrained environment. MPI is a widely used model for developing such algorithms in high-performance computing paradigm while Apache Spark and Apache Flink are emerging as big data platforms for large-scale parallel machine learning. Even though these big data frameworks are designed differently, they follow the data flow model for execution and user APIs. Data flow model offers fundamentally different capabilities than the MPI execution model, but the same type of parallelism can be used in applications developed in both models. This paper presents three distinct machine learning algorithms implemented in MPI, Spark, and Flink and compares their performance and identifies strengths and weaknesses in each platform.**

***Keywords*—Machine Learning, Big data, HPC, MDS, MPI, Spark, Flink, K-Means, Terasort**

I. Introduction

Machine learning algorithms are increasingly popular for handling large-scale data analytics. With the prevailing era of tremendous data sizes and complex algorithms, it is necessary to use parallel computing to make these algorithms compute in a reasonable amount of time. There are many frameworks available to run such algorithms in parallel from the High performance computing (HPC) and Big Data communities. Message passing interface (MPI) is the most widely used and dominant technology in HPC for parallel data analytics. Numerous frameworks exist in the Big Data community for doing large-scale parallel computations for machine learning algorithms, including Hadoop White (2009), Spark Zaharia et al. (2010), Flink *Apache Flink: Scalable Batch and Stream Data Processing* (n.d.), Carbone et al. (2015), Tez Saha et al. (2015) and Google Dataflow Akidau et al. (2015), and it is worthwhile to note there is no single technology that stands out among others as the best.

Hadoop is an early big data system for analyzing very large data sets. It employed a limited data flow API with map and reduce functions to move data and do computations. Later on it became clear that Hadoop was not efficient enough for the majority of Big Data problems, especially machine learning applications involving iterative computations Ekanayake et al. (2010). Spark, Flink and Tez are some of the later systems that overcame these bottlenecks. They expose richer data flow APIs along with in-memory computations and iterative computing support to be much more efficient than Hadoop.

MPI programming API utilises an in-memory in-place execution model where the computations and communications both happen in the same process under the same programming scope. On the other hand, Big Data systems like Spark and Flink have adopted a data flow style programming model and execution model for processing large amounts of data. In Data-flow programming model, operators are applied on distributed data sets which produce other distributed data sets. This abstraction provides a simple yet very powerful programming API. These APIs are usually written in accordance with the functional programming principles, making them less error prone and easy to program. A data-flow execution model separates the communications and computations by allowing computing to happen in self-contained tasks and not permitting communication within the task execution. The tasks do stateless computations on the data.

In contrast to these high level programming APIs, MPI provides bare communication primitives necessary for parallel computing. The MPI primitives are highly optimized for HPC environments with primary focus on in-memory stateful computing and support for advanced hardware such as high performance interconnects. With recent trends in the Big Data community, we see Big Data frameworks are inclined to use some of the hardware and algorithmic advances applied in traditional HPC frameworks. Even though MPI is a generic programming API, it is harder to use than Spark or Flink style programs without using substantial programming.

Among the big data frameworks we have mentioned earlier, Apache Flink and Spark are popular and efficient examples. These two have been used heavily in machine learning applications owing to having personalize machine learning libraries called FlinkML Carbone et al. (2015) and MLib Meng et al. (2016), respectively. Because of the in-memory nature of the computations, we can argue that MPI, Flink and Spark provide a comparable feature set for machine learning applications. Even though these frameworks have been positioned as enablers of large-scale machine learning, it is hard to find complex machine learning algorithms with heavy computational and communication demands implemented in these platforms.

This paper examines three distinctively different algorithms implemented in Spark, Flink and MPI and studies their performance characteristics. The algorithms discussed here are Multidimensional Scaling Kruskal (1964), K-Means and Terabyte sort. We compare the two contrasting styles of programming and execution models offered by Big Data frameworks and MPI to point out the advantages and disadvantages of both approaches for parallel machine learning applications. The main contribution of the paper is to analyze different algorithms on two big data frameworks and MPI to draws a clear distinction between the strengths and weaknesses of big data and HPC frameworks in the context of machine learning.

The rest of the paper is organized as follows. Section II describes the data flow and MPI models of computation along with background information about Spark and Flink. Section III IV V explains the MDS, K-Means and Terasort algorithms respectively and explains their implementations in Spark, Flink and MPI. The next Section VI describes the experiments conducted while Section VII explains the results observed. Related work is discussed in Section VIII and the paper concludes in Section IX.

II. Data flow and MPI

A. Execution models

Data flow frameworks model computation as a graph where nodes represent the operators that are applied to data and edges represent the communication channels between the operators. The input and output of an operator is sent through the edges. Operators are user defined code that execute on the data and produce other data. The graph is created using data flow functions provided by the framework. These data flow functions act upon the data to create the graph. An example function is a partition function, often called a map in data flow engines. A map function works on partitions of a data set and presents the partitioned data to the user defined operators at the nodes. The output of a user defined operator is connected to another user defined operator using a data flow function. Map and reduce are such two widely used data flow functions.

The data flow model consist of a logical graph and an execution graph. The logical graph is defined by the user with data, data flow functions and user operators. The execution graph is created by the framework to deploy the logical graph on the cluster. For example, some user functions may run in larger numbers of nodes depending on the user defined parallelism and size of input. Also when creating the execution graph the framework can apply optimizations to make some data flow operations more efficient. An example logical graph is shown in Fig. 1 where it displays a map and reduce data flow functions. Fig. 3 shows an execution graph for this logical graph where it runs multiple map operations and reduce operations in parallel. It is worth noting that each user defined function runs on its own program scope without access to any state about previous tasks. The only way to communicate between tasks is using messaging, as tasks can run in different places.

For task execution, frameworks use a thread model with fewer task managers executing the parallel tasks for a node. This leads to a smaller number of TCP connections between

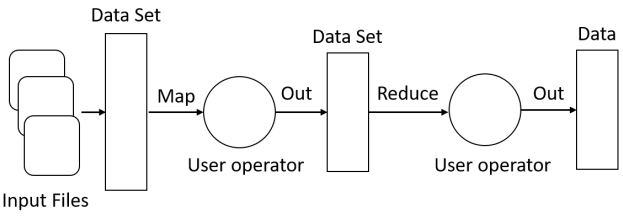

Fig. 1. Logical graph of a data flow

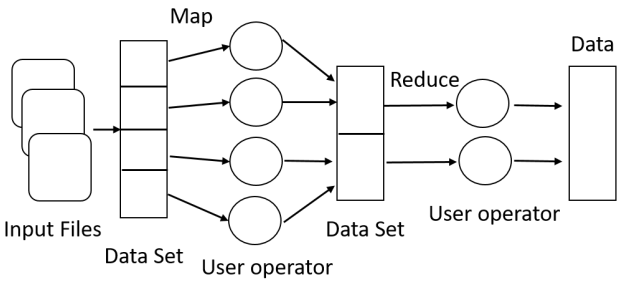

Fig. 2. Execution graph of a data flow

the nodes, but the individual tasks are subjected to interference by other task executions.

Message passing interface (MPI) is a generic model for doing parallel computations. It is used most widely in the single program multiple data (SPMD) style of parallel programs. The parallel processes can employ MPI functions for both message passing and synchronization between the processes. Fig. 3 shows the execution model of MPI. Unlike in data flow, the same process does the communion and computations throughout the execution. The processes are normal Unix versions spawned by MPI.

B. Programming APIs

Popular data flow engines such as Spark, Flink and Google Data Flow all enable a functional programming API. The data is abstracted as high level language objects. A large distributed data set is represented as a programming construct. For example, in Flink the data structure is called a DataSet, while in Spark it is a RDD. These data sets are considered immutable, meaning they cannot be changed once created.

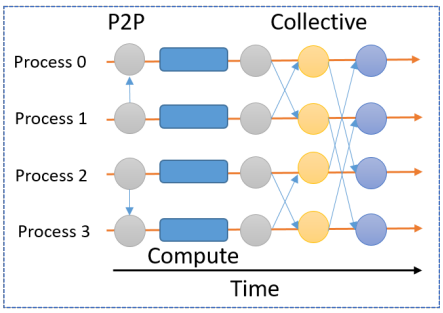

Fig. 3. Execution flow of MPI program with compute and communication regions

The user defined operators are implemented according to the functional programming paradigm with no side effects, meaning a user defined code cannot modify out-of-scope data sets or passed-in data sets. The only way an operator can produce a new data set is by outputting it, thus creating a different version every time. This clean programming paradigm makes it easier for an average programmer to write data-flow programs faster and error-free. But it comes with the cost of creating new structures every time a data set has to be modified. In MPI style programs, we almost always modify the existing memory structures without producing new data sets every time a small operation is applied to the data. The data flow functions are always applied to these data objects. Usually a data flow function implements a user defined function to act upon the data it receives, although there are functions that don't accept user defined programs.

The data loading and data saving is abstracted out in terms of data sources and sinks. Data source and sinks are responsible for reading and writing the data while hiding the details of different file systems such as local file systems and distributed file systems like HDFS. The default data sources are sufficient for large sets of applications but for more advanced use cases users often end up writing their own readers and writers.

C. Apache Flink

Apache Flink is a batch and stream processing engine that models every computation as a data flow graph which is then submitted to the Flink cluster. The nodes in this graph are the computations and the edges are the communication links. Flink closely resembles the both the data flow execution model and API. The user graph is transformed into an execution graph by Flink before it is executed on the distributed nodes. While undergoing this transformation, Flink optimizes the user graph, taking into account the data locality. Flink uses and thread based worker model for executing the data flow graphs. It can chain consecutive tasks in the work flow in a single node to make the run more efficient by reducing data serializations and communications.

Even though Flink has a nice theoretically sound data flow abstraction for programming, we found that it is difficult to program in a strictly data-flow fashion for complex programs. This primarily due to the fact that control flow operations such as if conditions and iterations are harder code in data flow paradigm.

D. Apache Spark

Apache Spark is a distributed in-memory data processing engine. The data model in Spark is based around RDDs Zaharia et al. (2012), and the execution model is based on RDDs and lineage graphs. The lineage graph captures dependencies between RDDs and their transformations. The logical execution model is expressed through a chain of transformations on RDDs by the user. This lineage graph is also essential in supporting fault tolerance in Apache Spark.

RDD's can be read in from a file system such as HDFS, and transformations are applied to the RDDs. Spark transformations are lazy operations and actual work is only done when an action operation such as count, reduce are invoked. By default, intermediate RDDs created through transformations are not cached and will be recomputed when needed. The user has the ability to cache or persist intermediate RDDs by specifying this explicitly. This is very important for iterative computation where same data sets are being used over and over again.

Spark primarily uses an thread based worker model for executing the tasks. Unlike in Flink where user submits the execution graph to the engine, Spark programs are controlled by a driver program. This driver program usually runs in a separate master node and the parallel regions in this driver program are shipped to the cluster to be executed. With this model complex control flow operations that needs to run serially such as iterations and if conditions run in master while data flow operators are executed in worker nodes. While this model makes it easier to write complex programs, it is harder to do complex optimizations on the data flow graph as it needs to be executed on the fly.

III. Multidimensional Scaling

Multidimensional scaling is a popular, well established machine learning technique for projecting high dimensional data into a lower dimension so that they can be analyzed. It has been extensively used to visualize high dimensional data by projecting into 3D or 2D models. MDS is a computationally expensive algorithm. The best algorithms are in the range of $O(N^2)$, where N is the number of data points. When applied to a larger data set, the computation time increases exponentially. The algorithm can be made to run efficiently in parallel to reduce the computation time requirements.

Parallel version of MDS requires multiple parallel regions and nested iterations for its computations. We will not go through the details of the algorithm as it is already described in several previous works. The initial work Ruan & Fox (2013) illustrates how the original MDS algorithms complexity is being lowered from $O(N^3)$ to $O(N^2)$. The proceeding papers Ekanayake et al. (2016), (n.d.) describe techniques for improving the implementation efficiency using Java and MPI. For the purpose of this study we only investigate the algorithm in terms of parallel operations it requires, their complexity and how they can be run in parallel as we try to analyze this algorithm on three execution platforms.

Given a high dimensional distance matrix, the goal of MDS algorithm is to produce a point file with the target dimension. Optionally the algorithm can take a weight matrix. The flow of the algorithm is shown in flow diagram 4. MDS algorithm begins by reading two files which contain the pair of matrices. If the data set has N high dimensional data points, the size of these matrices will be $N \times N$ with each cell containing 2 bytes of data in the form of a Short Integer. One matrix file contains the distances and the other contains the weights. Each worker in the computation only reads part of the $N \times N$ matrix. Aside from these two matrices, the algorithm requires a $N \times M$ matrix where M is the target projection dimension. In most cases M will be 3 or 2.

The pseudo code of the algorithm is shown in 1 emphasizing on the important parallel operations. Since this particular implementation of MDS uses deterministic annealing(DA) as an optimization technique, the algorithm has an outer loop

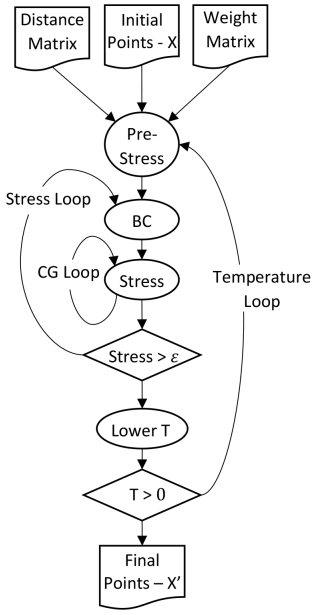

Fig. 4. Multidimensional Scaling algorithm with Deterministic Annealing control flow

involving temperature. The temperature is lowered after each loop until it reaches a configured low value. For each degree reading, the solution to the MDS equation is found as an optimization. For each iteration, the stress is lowered until the difference between consecutive stresses are minimal enough. This involves the last two loops, where the middle loop is over stress and the inner loop is for the optimization computation.

The core of the computation involves several matrix multiplications. The remaining computations are mainly serial in nature. These matrix multiplications are present in the two inner loops and largely involve $N \times N$ into $N \times M$ matrices. The algorithm is made parallel around these matrix multiplications.

Apart from the 3 matrices already described, three other matrix of size $N \times M$ are required, called BC,MMr and MMa and one array V of size N. For this algorithm we can achieve a balanced load across the parallel workers. The two NN matrices read are partitioned row-wise. The $N \times M$ matrix, which contains the projected points, is maintained as a whole in all the workers along with along with BC,MMr and MMa.

A. MPI Implementation [1]

The MPI implementation is a BSP style program written in Java using the OpenMPI Java bindings. Computations in DA-MDS grow $O(N^2)$ and communications $O(N)$. The algorithm involves gather/reduce of large matrices. Because of this the communication between the workers is very intensive. The MPI implementation uses shared memory explicitly to reduce the cost of collective operations by doing local operation to a node first before doing the global collective operation.

[1] https://github.com/DSC-SPIDAL/damds

Algorithm 1 MDS Parallel operations

d, w_i, P, n, m ▷ Where d - partition of D, w - partition of W, p - Initial points, N - number of points, M - target dimension, K - number of parallel tasks
$P = [N, M]$ point matrix
$b = \frac{N}{K}$
$d = [b, N]$ partition of D matrix
$w = [b, N]$ partition of W matrix
$BC = [N, M]$ NxM matrix
$V = [b]$ Array
T DA temperature
while $T > 0$ **do**
  $preStress = calculateStress(d, w, P, T)$
  while $stressDiff > delta$ **do**
    $BC = calculateBC(d, w, P, T)$
    $P = conjugateGradient(d, w, P, T, BC)$
    $stress = calculateStress(d, w, P, T)$
    $stressDiff = preStress - stress$
    $preStress = stress$
  end while
  $T = \alpha T$
end while
function CONJUGATEGRADIENT(d, w, P, T, BC)
  $MMr = [N, M]$ NxM matrix
  $MMr = calculateMM(w, V, P)$
  $BC = BC - MMr$
  $MMr = BC$
  $rTr = innerProduct(MMr, MMr)$
  $testEnd = rTr * cgThreshold$
  while $itr < cgCount$ **do**
    $MMAp = calculateMM(w, V, BC)$
    $\alpha = rTr / innerProduct(BC, MMAp)$
    $P = P + \alpha * BC$
    if $rTr < testEnd$ **then**
      break
    end if
    $MMr = MMr - \alpha * MMAp$
    $temp = innerProduct(MMr, MMr)$
    $\beta = temp / rTr$
    $rTr = temp$
    $BC = MMr + \beta * BC$
  end while
  return P
end function
function CALCULATESTRESS(d, w, P, T) ▷ parallel operation
  calculate partial value of Stress (Double value)
  All Reduce partial stress (sum)
  return $stress$
end function
function CALCULATEBC(d, w, P, T) ▷ parallel operation
  calculate partial value of BC for $[b, M]$ matrix
  gather the other parts from peers (AllGather)
  return BC
end function
function CALCULATEMM(w, V, A) ▷ parallel operation
  calculate wA using V for diagnol values of w
  AllGather $[b, M]$ parts from peers
  return the collected $[N, M]$ matrix
end function
function INNERPRODUCT(A, B)
  return sum of vector dot products
end function

B. Apache Flink Implementation [2]

Flink implementation starts with custom input formats for reading the binary matrix files and text based point files required by MDS. Compared to MPI implementation it was much easier to deal with the high level APIs of the input format for handling data partitioning and reading. The parallel operations in the MDS algorithm are mostly implemented using Map, Reduce and GroupReduce operations in Flink API. Also the broadcast functionality is used heavily throughout the program.

As of this work, Flink doesn't support nested iterations and only supports single level iterations. Because of this limitation, only the last loop is implemented as a data-flow. To handle the two outer loops, the implementation used separate jobs. As such each job executes the innermost loop and saves the computation to disk. Then the next outer loop starts as a new job with the saved state. This is a very inefficient way of implementing the iterations because Flink has to schedule the job, load data and save intermediate data for each of the two outer loops. We noticed that data loading doesn't add much overhead, but scheduling the tasks does.

Flink does not support outputting variables created inside an iteration. This leads to the creation of unified data sets with both the loop variable and output variable in a single data set. Because of this we had to pass these two data sets to distributed operations that only handle one of them at a time. Flink has no mechanism to load the same data set over all the workers and maintain such structures across all the nodes throughout different data flow operations. So the point matrix is created only in a single node. Every time there is a parallel operation requiring the point matrix, it has to be broadcast to the workers. This applies to other matrices such as BC and MMr as well.

The data flow operators are scheduled by Flink without much control from the user about where to place the data and operators in a cluster. For a complex data flow application such as MDS, it is important to have some control over where the data and operators are placed in the cluster while doing the computation to apply many application-specific optimizations. For simple data flow application, this gives a perfect abstraction and an easy API for users to write efficient programs. But for complex applications it may not translate well as in the easy case. Also it is worth noting that, Flink has to model the serial operation in the algorithm according to the data flow style.

Even though Flink programming model can become complex and non-intuitive for complex algorithms, we found that the programs were surprisingly free of programming errors compared to MPI programs. This is primarily due to the functional style of programming API where user cannot change state.

C. Spark Implementation [3]

The programming model of Spark does not allow tasks to communicate with one another, therefore all-to-all collectives are not supported in Spark. As a result, at each iteration the data needs to be collected at the master process and then a new set of tasks has to be created with the new data for the next iteration. This adds two additional overheads to the algorithm. The first is task creation and distribution at each iteration. The other is caused by the additional I/O that needs to be done at each step to perform reduce and broadcast instead of an AllReduce operation. Because of these limitations the main loops of the MDS algorithm are executed in the driver program and large tasks such as calculateBC, calculateMM and calculateStress are performed as distributed map-reduce tasks. This results in a large number of map reduce phases. The resulting values are then broadcast from the driver program to the cluster before the next iteration is executed. Several RDD's that contain distance data and weight data are cached to improve performance.

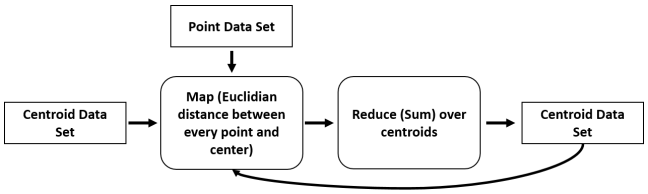

Fig. 5. K-Means data flow graph for Flink and Spark

Because Spark does not allow in-place changes on RDDs, the algorithm generates intermediate data sets that are not required in the MPI implementation. These intermediate data sets increase the memory usage of the algorithm, which is problematic because memory is a limiting factor when running MDS on very large matrices.

IV. K-Means algorithm

K-Means is an efficient and practical classical machine learning algorithm for data clustering. The algorithm maintains K cluster points called centroids. There are many variations of K-means available, but for these experiments we use the simplest form. As described in the pseudo code 2, parallel algorithm works as follows: the input to the system is N points and K initial centroids generated randomly. The N points are partitioned among the parallel tasks and each parallel processes read the K initial centroids. After this step, every task calculates its nearest centroid for each point. The local average of these points for each centroid are used in a global average to get the new centroids position. This is essentially an AllReduce operation with sum.

1) MPI Implementation [4]*:* Each parallel MPI implementation reads its partition of the point data set and calculates the nearest centroid for each point. The average of these local values are summed over all the ranks using the MPI AllReduce operation to find the new centroids.

2) Spark [5] *& Flink Implementations* [6]*:* Spark and Flink K-Means data flow graph is shown in Fig 5. At each iteration, a new set of centroids are calculated and fed back to the beginning of the iteration. The algorithm partitions the points into multiple map tasks and uses the full set of centroids in

[2]https://github.com/DSC-SPIDAL/flink-apps
[3]https://github.com/DSC-SPIDAL/damds.spark
[4]https://github.com/DSC-SPIDAL/KMeans
[5]https://github.com/DSC-SPIDAL/SparkKmeans
[6]https://github.com/DSC-SPIDAL/flink-apps

every one. Each map task assigns its points to their nearest centroid. The average of points is reduced (sum) for each centroid to get the new set of centroids, which are broadcast to the next iteration. Spark MLib provides a implementation of K-Means, which is used for evaluations.

In MPI after the AllReduce operation, the centroids are updated within the program in-place. On the other hand, for Spark and Flink these centroids need to be broadcast back to all the tasks that do the nearest neighbor calculation in the next iteration. We can argue that all reduce operation in MPI is equivalent to reduce + broadcast, which is the mechanism used in Flink and Spark. But it is worth noting that AllReduce can be optimized for greater efficiency than running reduce and broadcast separately by using algorithms such as recursive doubling.

V. TERASORT

Sorting terabytes of data is a utility algorithm used by larger machine learning applications. Spark, Flink and MPI implementations of the algorithm use the strategy shown in Fig 6 to sort 1 terabyte of data in parallel. At the initial stage, the data is partitioned into equal size chunks among the processes. The processes load these chunks into memory and uses a sample set of data to find an ordered partitioning of the global data set. It does this by sorting the samples gathered from a configurable number of processes. Given that the data is well balanced and the number of sample partitions are reasonably high, this step normally generates a balanced partitioning of the data. In the next step, this global partitioning is used by all the processes to send the data to the correct parallel task. We call this the shuffling phase. The parallel version of the algorithm is described in code 3. After a process receives the data it requires from other processes, it sorts and writes them to a file. This creates a globally sorted data set across the parallel tasks.

Algorithm 2 Parallel K-Means algorithm

```
W number of parallel processes
P point partition
C initial centers
C_1 = C
for i = 1 to iterations do
    C_{i+1} = 0 next set of centers
    for p = in P do
        Calculate the nearest center i for p in C_i
        add p to the i center in C_{i+1}
    end for
    All reduce sum on C_{i+1} and take average (no of assigned points)
end for
```

The input to the system is according to the format defined for Indy sort by sortbenchmark.org [7]. Each point is 100 bytes long with a 10 bytes key and 90 bytes of random data. The data is sorted using only the key part of the data but final output contains the full 100 bytes for each point.

A. MPI Implementation [8]

In the MPI implementation the initial sampling is done in memory by choosing a subset of ranks. The samples are gathered to a single process to sort and create the partitioning. This partitioning is then broadcast to all the parallel tasks. The algorithm used send/receives along with iprobe to send the data to correct ranks. A chained send/receive topology in a ring is used, as shown in Fig 7. Unlike in Spark and Flink case lot of functionality had to be written in-order to get the algorithm working in a memory limited environment with file based sorting. Also to send the data efficiently the algorithm gathered large enough data set before sending it to the correct process.

[7] http://sortbenchmark.org/

[8] https://github.com/DSC-SPIDAL/mpi-terasort

Algorithm 3 Terabyte sort parallel algorithm

```
W number of parallel processes
P point partition
R Final points of this process
s ⊆ P take samples from P
S = ⋃ s_i Gather the samples from parallel processes
S = sort(S) Sort the samples gathered
T ⊆ S T is a partition that divides the samples across the parallel processes
for p = in P do
    Calculate the partition i which p belongs to using T
    Send p to i
end for
for j = 1 to W do
    Receive points from ith process and gather them to R
end for
sort R
save R
```

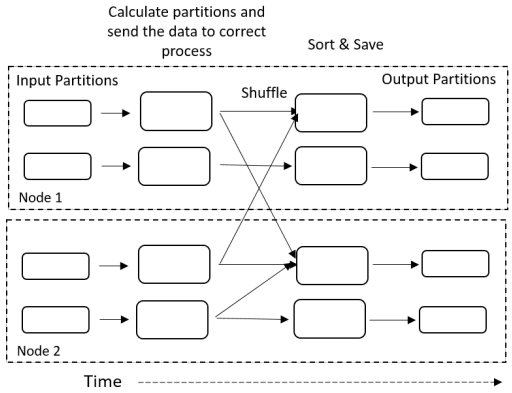

Fig. 6. Parallel sorting flow. Large boxes show the processes/tasks and small boxes are input files and output files.

B. Spark and Flink Implementations [9]

Terasort can be implemented in Spark and Flink using the built-in capabilities of the platform. The data is loaded from HDFS cluster running in the same nodes. The data partitioning

[9] https://github.com/DSC-SPIDAL/terasort

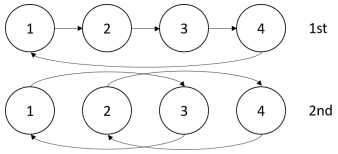

Fig. 7. MPI's data shuffling algorithm. Processes send and receive in a ring topology.

is done using the HDFS file system by reading the sample chunks in a single node; sorting them and writing the partitions back to HDFS. The sorting algorithm uses this partitioning to send the data to correct maps. The data is sorted using the sorting functions of Flink and Spark.

For Terasort, there was no apparent difference in the communication or the computation used by MPI, Flink or Spark. The MPI algorithm used send/receive operations which is essentially the mechanism used by Spark and Flink. Flink and Spark buffer the data internally while we buffered the data manually for sending using MPI.

VI. Evaluation

The experiments were run on Juliet, which is an Intel Haswell HPC cluster. Up to 64 nodes were used for the experiments. These nodes have 24 cores (2 sockets x 12 cores each) per node with 128GB of main memory and 56Gbps Infiniband interconnect and 1Gbps dedicated Ethernet connections. MPI experiments were conducted using TCP instead of Infiniband to make it comparable with Spark and Flink. For Flink and Spark experiments, the data was loaded from a HDFS cluster running on the same nodes. For MPI experiments data was copied to local storage in the nodes. We only used up to 20 cores in each node to reduce interference from other processes such as HDFS data nodes, Yarn, Flink TaskManager that run on these. MPI can utilize all the cores as it doesn't include additional processes.

For the experiments we ran MDS with a limited number of iterations with 5 temperature loops, 2 stress loops and 8 conjugate gradient iterations to limit the time required for experiments. Fig. 8 shows time (displayed in log scale) for running MDS on 16 nodes with 20 parallel tasks in each node. The points are varied from 4000 to 64000. Flink performed very poorly as expected because it doesn't support nested iterations. Spark did considerably well compared to Flink but still proved much slower than MPI. This figure also shows the compute time of the MPI algorithm. It is evident that in MPI the communication overhead and other overheads when running the algorithm in parallel is minimal. Since the same parallel algorithm is implemented in Spark and Flink, this computing time provides a baseline for them as well. The large increase in time in Flink is caused by overheads introduced by the frameworks.

Fig. 9 Shown running the MDS algorithm with 32000 points on different number of nodes each having 20 parallel tasks. In MPI the running time decreases as expected while increasing the parallelism, and in Spark and Flink the running time increases with more nodes.

We conducted two sets of experiments for the K-Means algorithm. In one experiment we used 10 million points each having 100 attributes (100 dimensions) and 10 iterations to calculate centers. In the next set we used 1 million points with 2 attributes and 100 iterations. The first test requires higher computation and lower communication time compared to the later experiment due to large number of points and small number of iterations.

Fig. 10 shows the results of the 10 million point experiment in 16 nodes with varying number of centers. Flink worked comparatively better than Spark in this case and even performed closer to MPI performance. Fig. 11 shows the results of having 10 million points with 16000 centers and varying the number of nodes. Unlike in the MDS case, we saw decreases in time in all three frameworks when the parallelism increased. Fig. 12 shows the results of the 1 million point experiment in 8 nodes. The performance gap between MPI and Flink is wider in this case. Fig. 12 shows the same experiment with 64000 centers and varying number of nodes. As the parallelism increased again, MPI performed better but Flink did not scale well.

Fig. 14 shows the run time of sorting 1 Terabyte of data in 64 nodes with all three frameworks. Because we are mainly comparing the in-memory performance of the algorithms, we used sufficient nodes so that we can do the sorting in-memory without using the disks. As such the largest time of the algorithm was spent on shuffling the data across multiple processes. In this case all the frameworks performed reasonably well and produced results closer to each other, although the MPI Infiniband results are significantly faster than other approaches. For 32 node test the memory was insufficient and the program had to use the disk to perform the sorting. The MPI-IB test shows that MPI with Infiniband performed best in transferring the data quickly. For 32 node MPI case we noticed that Java garbage collection was affecting the performance. This is an initial investigation, which we will extend later.

A micro benchmark was conducted to measure the reduce operation communication time in the three frameworks and its results are shown in Fig. 15. The experiment was conducted in 32 nodes with 20 parallel tasks in each node having 640 parallel tasks. Integer array of varying sizes is used for the reduction operation, which was conducted several times in an iteration to calculate the average time. It is clear from the graph that there is a large difference in time between Flink and MPI.

VII. Discussion

MDS is the most complex algorithm among the three algorithms considered here. We have encountered many inefficiencies of Spark and Flink while implementing the algorithm. For Flink the biggest inefficiency was its inability to support nested loops. This leads to a very laborious implementation where we save the intermediate data to file system at each iteration in the outer two loops. Also the way Flink is designed means it needs to read the input files each time it does the iterations, adding to the overhead. The main inefficiency in the Spark MDS implementation was caused due to the lack of all-to-all collective operations. Using a reduce operations followed by a broadcast operation added couple of overheads to the algorithm.

K-Means showed some interesting characteristics with the three frameworks we used. In parallel K-Means, the communication cost is a direct function of the number of centroids involved and it doesn't depend on the number of points. With increased number of points, the computation time increases, but the communication time remains the same. When using the 10 million data set it was evident that Flink performed close to MPI, and when using the 1 million data set with 100 iterations, the performance gap widened. We concluded that Flink-like frameworks need improvements in communication algorithms

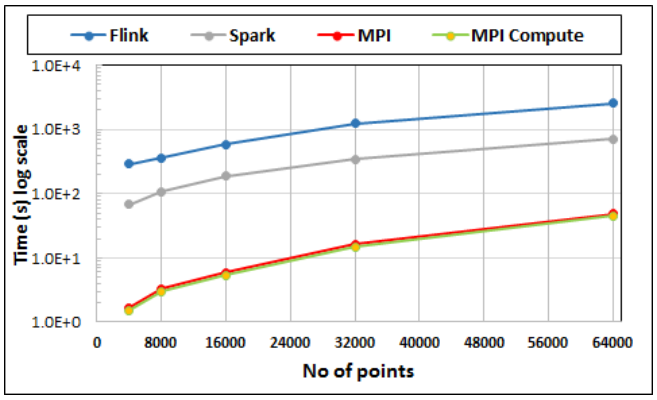

Fig. 8. MDS execution time on 16 nodes with 20 processes in each node with varying number of points

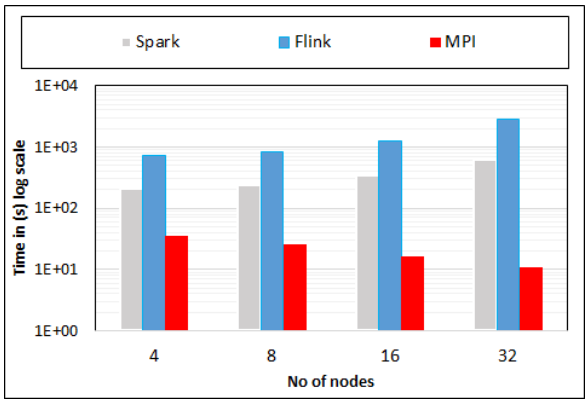

Fig. 9. MDS execution time with 32000 points on varying number of nodes. Each node runs 20 parallel tasks.

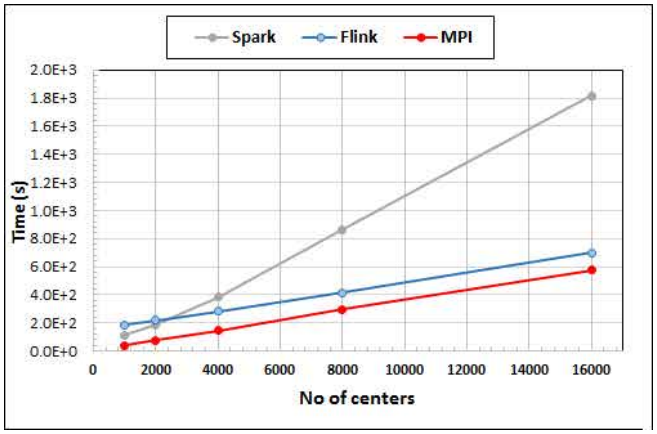

Fig. 10. K-Means execution time on 16 nodes with 20 parallel tasks in each node with 10 million points and varying number of centroids. Each point has 100 attributes.

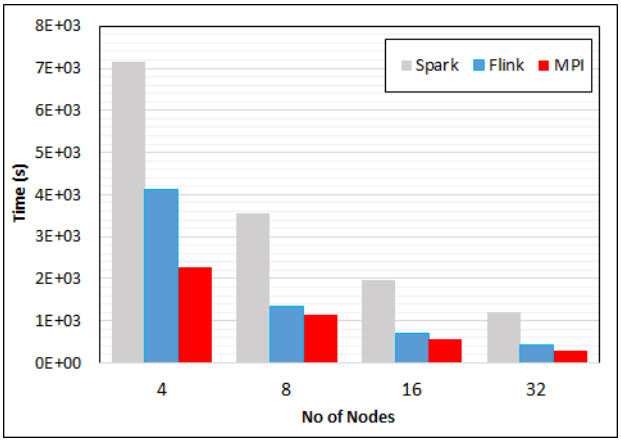

Fig. 11. K-Means execution time on varying number of nodes with 20 processes in each node with 10 million points and 16000 centroids. Each point has 100 attributes.

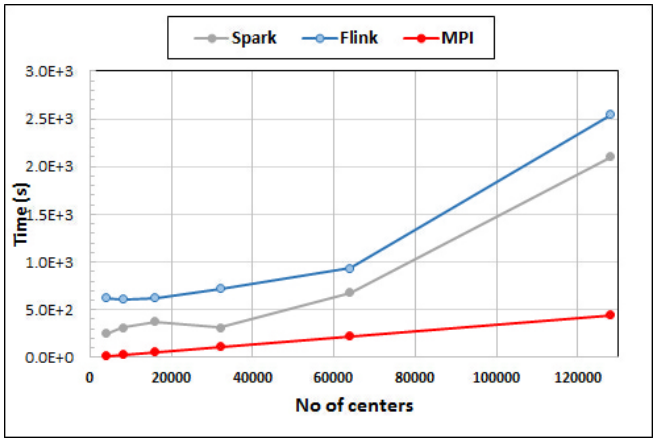

Fig. 12. K-Means execution time on 8 nodes with 20 processes in each node with 1 million points and varying number of centroids. Each point has 2 attributes.

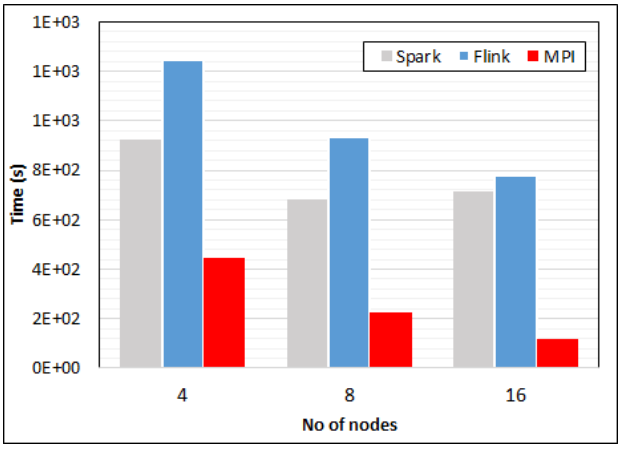

Fig. 13. K-Means execution time on varying number of nodes with 20 processes in each node with 1 million points and 64000 centroids. Each point has 2 attributes.

used for transferring data to scale to larger nodes when communication requirements are high. Most of the practical clustering problems do not have hundreds of thousands of clusters. This means K-Means can perform equally better in Flink or spark for practical data analytics tasks.

For Terasort, both Flink and MPI displayed comparable performance results. The communication in Terasort involves transferring large amounts of data among the nodes. Ring-like topologies produce the best results for throughput in such bulk transfers since they effectively use the networks in all the nodes

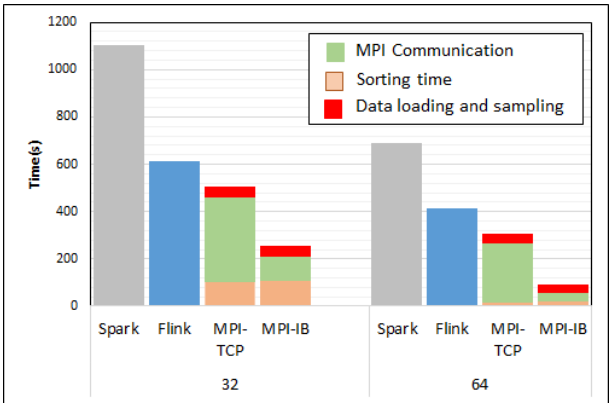

Fig. 14. Terasort execution time in 64 and 32 nodes. Only MPI shows the sorting time and communication time as other two frameworks doesn't provide a viable method to accurately measure them. Sorting time includes data save time. MPI-IB - MPI with Infiniband

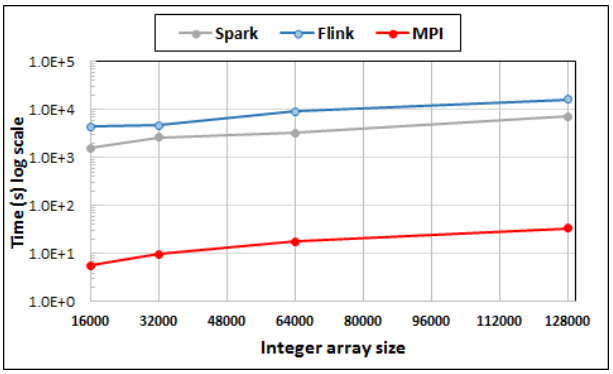

Fig. 15. Reduce operation time with 640 parallel tasks in 32 nodes. Integer array with different sizes is used for the reduce operation.

at the same time as evident by the good performance in MPI algorithm. Since Flink and Spark do the asynchronous point to point communications, they both saw the similar performance to MPI. Writing the MPI algorithm required time and effort and for such tasks involving transferring large amount of data Flink and Spark can be better choices given the ease of use.

The experiments showed an interesting observation where big data frameworks doesn't scale well for algorithms with high frequency of communicate and compute regions. Also it was evident that Spark performed better than Flink when there was high frequency of communication and computation regions. Algorithms with longer computation and communication times performed well in Flink compared to Spark.

The micro-benchmarks show there is a big difference in communication times for collective communications between MPI and the big data platforms. The primary reason is that, MPI has implemented very efficient collective communication algorithms while Flink and Spark rely on point to point connections.

VIII. Related Work

Owing to the increasing popularity of machine learning algorithms, most Big Data frameworks provide machine learning libraries. Mlib Meng et al. (2016) is built on top of Spark and offers a wide variety of machine learning algorithms. FlinkML Carbone et al. (2015) is the requisite library for Flink. Intel Data Analytics Acceleration Library (DAAL) *Intel Data Analytics Acceleration Library (Intel DAAL)* (n.d.) from Intel has been tuned for Intel architecture; it provides functions for deep learning, classical machine learning, etc. Tensorflow Abadi et al. (2016) is a library developed at Google. Algorithms developed using TensorFlow can be executed in a wide variety of heterogeneous systems that range from mobile devices to supercomputers. h2o *H2O* (2017) is a machine learning framework that supports Spark and Hadoop with simplified APIs for ease of use. Apache Mahout Owen et al. (2012) remains another popular framework originally developed to support machine learning on top of Hadoop and later expanded to support other frameworks. Understanding the anatomy of these frameworks and comparing them with HPC frameworks such as MPI as the authors have done, will allow both the Big Data and HPC communities to incorporate the best of both technologies.

An immense amount of research is being done on improving performance and resource utilization in both the Big Data and HPC communities. Research on improving performance of Spark is being done in Project Tungsten *Project Tungsten: Bringing Apache Spark Closer to Bare Metal* (2015) where they aim to improve its memory management and caching. Improving memory management in Spark is very important, because as pointed in the text, Spark has higher memory usage than MPI. Even though we mainly talked about Spark and Flink, there are a large number of frameworks utilized in the Big Data community. Pregal Malewicz et al. (2010), Apache Hama Seo et al. (2010) and Apache Giraph *Apache Giraph* (n.d.) are frameworks that were developed around the BSP model. There are also numerous distributed graph processing frameworks, including GraphX Gonzalez et al. (2014). Doekemeijer et al. Doekemeijer & Varbanescu (2014) compare and contrast many distributed graph processing frameworks. Much research has also been done on integrating HPC technology into big data frameworks to improve performance. Lu et al. Lu et al. (2013) integrate RDMA technology to improve performance of Hadoop. Hpc-abds Fox et al. (2015) discusses and summarizes HPC and Big data convergence.

Lu et al. Lu et al. (2014) use Terasort to compare DataMPI to Hadoop and Marcu et al. Marcu et al. (2016) use Terasort to compare Spark and Flink. MDS is a popular method used for dimension reduction, DA-MDS Bae et al. (2010) used in this paper is a Deterministic Annealing approach to MDS which can outperform other MDS implementations.

There is a lot of focus today on bridging HPC technology with Big Data technology in order to take advantages of both approaches. The authors have studied about HPC/Big Data convergence Fox et al. (1858) in more detail. Reyes-Ortiz et al. Reyes-Ortiz et al. (2015) compare Spark and MPI/OpenMP on Google Cloud platform to compare and contrast performance, data management, etc. between the two frameworks. Liang et al. Liang et al. (2014) compare DataMPI Lu et al. (2014), which is an extension of MPI developed to execute Hadoop-like Big Data jobs, with Spark and Hadoop using BigDataBench Wang et al. (2014). The authors previously studied factors that affect performance of Java machine learning applications in Multicore HPC Clus-

ters (n.d.). This paper contributes towards the HPC/Big Data convergence by identifying areas that can be improved in each technology domain in the context of machine learning.

IX. Conclusion & Future Work

The three algorithms presented here show different performance across the three frameworks. MPI performed the best for all three algorithms, but it was the hardest to program among the three. A reasonable person can pick Spark or Flink over MPI simply due to the trade off of performance vs ease of use. Flink and Spark performed well on the Terasort algorithm and K-Means with minimal coding efforts while performing poorly on the more sophisticated MDS algorithm. There are large number of machine learning algorithms that are in the range of K-Means and Terabytesort complexity and they can be efficiently implemented in these platforms. For more complex algorithms these frameworks needs to be improved to support the algorithm requirements. For example, Flink and Spark require efficient communication algorithms to scale complex machine learning algorithms that require tight synchronizations and collective communications. The authors are working on to improve the performance of Flink and Spark by improving their communication algorithms.

Acknowledgment

This work was partially supported by NSF CIF21 DIBBS 1443054 and NSF RaPyDLI 1415459. We thank Intel for their support of the Juliet system, and extend our gratitude to the FutureSystems team for their support with the infrastructure.

References

(n.d.), Technical report.

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M. et al. (2016), 'Tensorflow: Large-scale machine learning on heterogeneous distributed systems', *arXiv preprint arXiv:1603.04467* .

Akidau, T., Bradshaw, R., Chambers, C., Chernyak, S., Fernández-Moctezuma, R. J., Lax, R., McVeety, S., Mills, D., Perry, F., Schmidt, E. et al. (2015), 'The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing', *Proceedings of the VLDB Endowment* **8**(12), 1792–1803.

Apache Flink: Scalable Batch and Stream Data Processing (n.d.).
URL: *https://flink.apache.org/*

Apache Giraph (n.d.). Accessed: December 14 2016.
URL: *http://giraph.apache.org/*

Bae, S.-H., Qiu, J. & Fox, G. C. (2010), Multidimensional scaling by deterministic annealing with iterative majorization algorithm, *in* 'e-Science (e-Science), 2010 IEEE Sixth International Conference on', IEEE, pp. 222–229.

Carbone, P., Fóra, G., Ewen, S., Haridi, S. & Tzoumas, K. (2015), 'Lightweight asynchronous snapshots for distributed dataflows', *arXiv preprint arXiv:1506.08603* .

Doekemeijer, N. & Varbanescu, A. L. (2014), 'A survey of parallel graph processing frameworks', *Delft University of Technology* .

Ekanayake, J., Li, H., Zhang, B., Gunarathne, T., Bae, S.-H., Qiu, J. & Fox, G. (2010), Twister: A Runtime for Iterative MapReduce, *in* 'Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing', HPDC '10, ACM, New York, NY, USA, pp. 810–818.
URL: *http://doi.acm.org/10.1145/1851476.1851593*

Ekanayake, S., Kamburugamuve, S. & Fox, G. (2016), Spidal: High performance data analytics with java and mpi on large multicore hpc clusters, *in* 'Proceedings of the 2016 Spring Simulation Multi-Conference (SPRINGSIM)', Pasadena, CA, USA.

Fox, G., Qiu, J., Jha, S., Ekanayake, S. & Kamburugamuve, S. (1858), Big data, simulations and hpc convergence, Technical report, Technical Report· January 2016, DOI: 10.13140/RG. 2.1.

Fox, G., Qiu, J., Kamburugamuve, S., Jha, S. & Luckow, A. (2015), Hpc-abds high performance computing enhanced apache big data stack, *in* 'Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on', pp. 1057–1066.

Gonzalez, J. E., Xin, R. S., Dave, A., Crankshaw, D., Franklin, M. J. & Stoica, I. (2014), Graphx: Graph processing in a distributed dataflow framework, *in* '11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)', pp. 599–613.

H2O (2017).

Intel Data Analytics Acceleration Library (Intel DAAL) (n.d.). Accessed: January 02 2017.
URL: *https://software.intel.com/en-us/intel-daal*

Kruskal, J. B. (1964), 'Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis', *Psychometrika* **29**(1), 1–27.

Liang, F., Feng, C., Lu, X. & Xu, Z. (2014), Performance benefits of DataMPI: a case study with BigDataBench, *in* 'Workshop on Big Data Benchmarks, Performance Optimization, and Emerging Hardware', Springer, pp. 111–123.

Lu, X., Islam, N. S., Wasi-Ur-Rahman, M., Jose, J., Subramoni, H., Wang, H. & Panda, D. K. (2013), High-performance design of hadoop rpc with rdma over infiniband, *in* '2013 42nd International Conference on Parallel Processing', IEEE, pp. 641–650.

Lu, X., Liang, F., Wang, B., Zha, L. & Xu, Z. (2014), DataMPI: extending MPI to hadoop-like big data computing, *in* 'Parallel and Distributed Processing Symposium, 2014 IEEE 28th International', IEEE, pp. 829–838.

Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N. & Czajkowski, G. (2010), Pregel: a system for large-scale graph processing, *in* 'Proceedings of the 2010 ACM SIGMOD International Conference on Management of data', ACM, pp. 135–146.

Marcu, O.-C., Costan, A., Antoniu, G. & Pérez-Hernández, M. S. (2016), Spark versus flink: Understanding performance in big data analytics frameworks, *in* 'Cluster Computing (CLUSTER), 2016 IEEE International Conference on', IEEE, pp. 433–442.

Meng, X., Bradley, J., Yuvaz, B., Sparks, E., Venkataraman, S., Liu, D., Freeman, J., Tsai, D., Amde, M., Owen, S. et al. (2016), 'Mllib: Machine learning in apache spark', *JMLR* **17**(34), 1–7.

Owen, S., Anil, R., Dunning, T. & Friedman, E. (2012), 'Mahout in action'.

Project Tungsten: Bringing Apache Spark Closer to Bare Metal (2015).

URL: *https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html*

Reyes-Ortiz, J. L., Oneto, L. & Anguita, D. (2015), 'Big data analytics in the cloud: Spark on hadoop vs mpi/openmp on beowulf', *Procedia Computer Science* **53**, 121–130.

Ruan, Y. & Fox, G. (2013), A robust and scalable solution for interpolative multidimensional scaling with weighting, *in* '9th IEEE International Conference on eScience, eScience 2013, Beijing, China, October 22-25, 2013', pp. 61–69.
URL: *http://dx.doi.org/10.1109/eScience.2013.30*

Saha, B., Shah, H., Seth, S., Vijayaraghavan, G., Murthy, A. & Curino, C. (2015), Apache tez: A unifying framework for modeling and building data processing applications, *in* 'Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data', ACM, pp. 1357–1369.

Seo, S., Yoon, E. J., Kim, J., Jin, S., Kim, J.-S. & Maeng, S. (2010), Hama: An efficient matrix computation with the mapreduce framework, *in* 'Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on', IEEE, pp. 721–726.

Wang, L., Zhan, J., Luo, C., Zhu, Y., Yang, Q., He, Y., Gao, W., Jia, Z., Shi, Y., Zhang, S. et al. (2014), Bigdatabench: A big data benchmark suite from internet services, *in* '2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)', IEEE, pp. 488–499.

White, T. (2009), *Hadoop: The Definitive Guide*, 1st edn, O'Reilly Media, Inc.

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S. & Stoica, I. (2012), 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', USENIX Association, pp. 2–2.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S. & Stoica, I. (2010), Spark: Cluster computing with working sets, *in* 'Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing', HotCloud'10, USENIX Association, Berkeley, CA, USA, pp. 10–10.
URL: *http://dl.acm.org/citation.cfm?id=1863103.1863113*