© 2020 IJRTI | Volume 5, Issue 6 | ISSN: 2456-3315

# Performance Comparison of a Parallel Recommender Algorithm across three Hadoop-based Frameworks

**Aishwarya J[1], K Manasa[2], Manikya K[3], Pallavi sutar[4], Prof. Hemanth Kumar N P[5]**

Department of Computer Science and Engineering,
AIET, Moodabidri

***Abstract*: One of the challenges our society faces is the ever increasing amount of data. Among existing platforms that address the system requirements, Hadoop is a framework widely used to store and analyze "big data". On the human side, one of the aids to finding the things people really want is recommendation systems. This paper evaluates highly scalable parallel algorithms for recommendation systems with application to very large data sets. A particular goal is to evaluate an open source Java message passing library for parallel computing called MPJ Express, which has been integrated with Hadoop. As a demonstration we use MPJ Express to implement collaborative filtering on various data sets using the algorithm ALSWR (Alternating-Least-Squares with Weighted- $\lambda$-Regularization). We benchmark the performance and demonstrate parallel speedup on Movie lens and Yahoo Music data sets, comparing our results with two other frameworks: Mahout and Spark. Our results indicate that MPJ Express implementation of ALSWR has very competitive performance and scalability in comparison with the two other frameworks.**

***Index Terms*: HPC, MPJ Express, Hadoop, Map Reduce, YARN, Spark, Mahout**

## I. INTRODUCTION

Over the last decade Apache Hadoop has established itself as a pillar in the ecosystem of software frameworks for "big data" processing. As an open source, mostly Java-based Apache project with many industrial contributors, it retains a commanding position in its field.

When first released Hadoop was a platform primarily supporting the Map Reduce programming model, and other projects built on top of Map Reduce. Around 2014 with the release of Hadoop 2.0 the platform was re-factored into a separate YARN (Yet another Resource Negotiator) resource allocation manager, with Map Reduce now just one of multiple possible distributed computation frame- works that could be supported on top of YARN. Several other major big data projects rapidly mi- grated to allow execution on the Hadoop YARN platform (for example APR pache Spark [24], Apache Graph [1], Apache Tez [15], and Microsoft Dryad [9]). Around the same time the present authors envisaged adding our existing MPJ Express frame- work for MPI-like computation in Java to that distinguished group, and developed a version of our software that could also run under Hadoop YARN [22].

MPJ Express is a relatively conservative port of the standard MPI 1.2 parallel programming interface to Java, and is provided with both "pure Java" implementations (based on Java sockets and threads) and "native" implementations exploiting specific interconnect interfaces, or implementations on top of standard MPI. The vision was thus to support MPJ as one computational framework among many largely Java-based or JVM-based frameworks that could be mixed and matched for different stages of complex big data processing, with Hadoop and HDFS (the Hadoop Distributed File System) as the "glue" between stages.

The main goal of the present paper is to pro- vide evidence that such a scenario can be realized and that it may be advantageous. We concentrate on one particular computationally intensive "big data" problem - generating product recommendation through the collaborative filtering algorithm ALSWR (Alternating Least Squares with Lambda Regularization). A version of this algorithm was developed and evaluated using MPJ running under Hadoop. We then go on to compare our implementation with two existing implementations of AL- SWR that can run under Hadoop—one taken from the Apache Mahout project using Map Reduce, and one using Apache Spark. Results suggest the MPJ approach can provide useful performance gains over these other established Big Data frameworks on suitable compute-intensive kernels.

The rest of the paper is organized as follows. Section I-A reviews selected related work. Back- ground materials in Section II review Hadoop, YARN and HDFS; outline the architecture of MPJ Express and its integration in YARN; and give an overview of the collaborative filtering technique. Section III describes how we implement the collaborative filtering with ALSWR in MPJ. The Section IV evaluates and compares our results with Mahout and Spark. Section V concludes the paper and discusses future works.

### *A. Related Work*

Recommender systems have been a subject of tremendous interest lately. However, our interested is limited to collaborative filtering applied to very large datasets with millions of records, leading to a need for parallel processing.

For the Netflix Prize [25] proposed an approach called ALSWR. In order to implement their algorithm in parallel authors used Mat lab [7]. The result of the experiments showed a better performance of the ALSWR as the number of features and iterations increased. The experiments were made on the Netflix dataset consisting of 100 million ratings.

The current state-of-the-art dataset comprises of billions of ratings [10] and processing this requires scalable methods. The authors in [10] explained how ALS and SGD algorithms are used with Apache Graph to process an average of 100 billion ratings from Facebook. Yu, Hsieh, Si and Dhillon in [21] compare ALS methods to Stochastic Gradient Descent (SGD) methods and Coordinate Descent methods. A comparison of SGD with various algorithms was done in [13]. More studies have been realized in [11] with a different approach called Co-clustering Dataflow Bergman as well as another interesting research geared towards very large datasets in [16] that used ALS as algorithm on Hadoop Map Reduce and JBlas as framework.

In the context of MPJ Express, previous work [22] focused on integrating the software with YARN allowing end-users to execute Java MPI programs on Hadoop clusters. As part of this effort, a new YARN-based runtime system was added to the MPJ Express library. The paper demonstrated reasonable comparative performance of YARN- based runtime against the existing runtime. This study did not compare performance of YARN- based MPJ Express library against some of the newer technologies including Apache Spark.

## II. BACKGROUND

### A. Hadoop Overview

Hadoop is a framework that stores and processes voluminous amount of data in a reliable, fault- tolerant manner [19]. Since Hadoop 2, YARN (Yet Another Resource Negotiator) has been integrated in the infrastructure as the resource manager, enabling many other distributed frameworks besides Map Reduce to process their data on Hadoop clus- ter. YARN depends on three main components to complete a task: a Resource Manager (RM), Node Managers (NMs), and an Application Master (AM). The RM is responsible for managing and allocating the resources across the cluster. NMs run on all nodes available in a cluster and report all the tasks to the RM such as the number of cores and memory space. Each job that is started has an AM specific to the processing framework that manages operation within containers and ensures there are sufficient containers for the task. The communication be- tween the master nodes and slave nodes is achieved through the Heart Beat Mechanism [6].

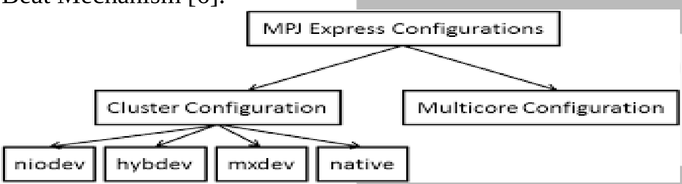


Fig. 1: MPJ Express Configuration

### B. MPJ Express

1) niodev - uses Java New I/O (NIO) Sockets
2) mxdev - uses Myrinet eXpress (MX) library for Myrinet networks
3) hybdev - for clusters of multicore processors
4) native - uses a native MPI library (like MPICH, MVAPICH, Open MPI)

MPJ Express [14] is an open source Java MPI- like library that allows application developers to write and execute parallel applications on multicore processors and compute clusters. The MPJ Express software can be configured in various ways as depicted in Figure 1. Under the cluster configuration, the MPJ Express software provides different communication devices that are suitable for the underlying interconnect. Currently, there are four communication devices available:

Since 2015, the MPJ Express software provides a YARN-based runtime that exploits the niodev communication device to execute parallel Java code on Hadoop clusters. Under this setting, HDFS is used as the distributed file system where application datasets, MPJ Express libraries, and application programs are loaded to allow all processes to access the material.

Figure 2 presents the implementation of the MPJ Express library on YARN. In this setting, the Hadoop cluster consists of a client node, where Re- source Manager (RM) executes, and two compute nodes, where a Node Manager (NM) executes. The NM process operates on each compute node and is responsible for executing assigned tasks. The main phases of the implementation of YARN are explained in [22].

### C. Collaborative Filtering Techniques

Recommender systems are software tools and techniques that provide suggestions to users to help them find and evaluate items likely to match their

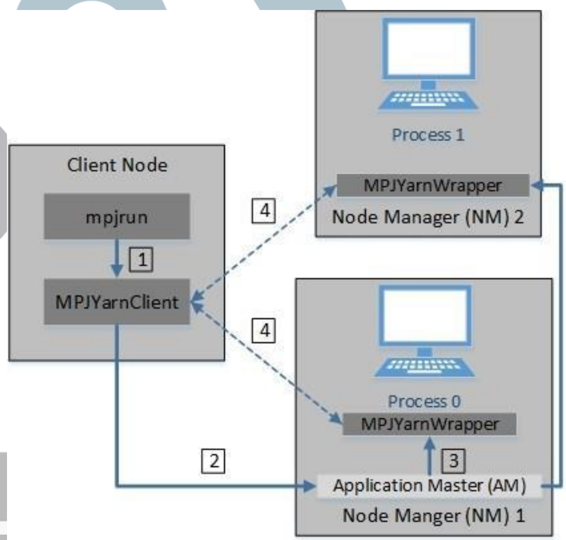


Fig. 2: MPJ Express Integrated in YARN- 1) Submit YARN application- 2) Request container allocation for AM- 3) AM generates a CLC and allocates container to each node- 4) Each mpj- yarn-wrapper send outputs and error streams of the program to the MPJ Yarn Client requirements. We use the term item to describe a product or service recommended by a system. Collaborative filtering systems are based on users' purchases or decisions histories. Assuming two individuals share the same opinion on an item, they are also more likely to have similar taste on another item. In our experiments we have opted for a model based approach and we specifically use Alternating-Least-Squares with Weighted-$\lambda$- Regularization (ALSWR) algorithm.

In this section, following [25], we will often refer to items as "movies". Assume we have *nu* users and *nm* movies, and *R* is the *nu nm* matrix of input ratings. Usually each user can rate only few movies. Therefore the matrix *R* will initially have many missing values or loosely speaking it will be sparse. The problem is to predict the unknown elements of *R* from the known elements.

We model the preferences of users by assuming they have simple numeric level of preference for each of a number *nf* of features to be found in movies; thus the behaviour of user *i* is modelled by a vector **u***i* of length *nf* . Similarly each movie is assumed to have each these features to a simple numeric degree so each movie $j$ is modelled by a vector **m***j* of the same size. The predicted prefer- ence of user $i$ for movie $j$ is the dot product **u***i* **m***j*. The vectors are conveniently collected together in matrices $U$ and $M$ of size *nu nf* and *nm nf* respectively.

To fit the model to the known elements of $R$ we use a least squares approach, adding a regu- larization term parameter $\lambda$ to the sum of square deviations to prevent the model from overfitting the data. The penalty function ALSWR strives to minimize is:

$$f(U, M) = (rij - ui \cdot mj)2$$
$$i,j$$
$$+ \lambda\, \Sigma\ nu\ u2 + \Sigma\ nm\ m2 \qquad (1)$$

where the first sum goes over $i$, $j$ values where the element *rij* of $R$ is known in advance, *nui* is the number of items rated by a user $i$, and *nmj* is the number of users who have rated a given movie $j$. ALSWR is an iterative algorithm. It shifts be- tween fixing two different matrices. While one is fixed, the other one is updated hence solving a matrix factorization problem. The same process goes through a certain number of iterations until a convergence is reached which implies that there is little or no more change on either users and movies matrices. The ALSWR algorithm as explained by Zhou et al [25] is as follows:

- Step 1: Initialize matrix $M$ in a pseudorandom way.
- Step 2: Fix $M$ , Solve $U$ by minimizing the objective function (the sum of squared errors);
- Step 3: Fix $U$ , Solve $M$ by minimizing the objective function similarly;

Steps 2 and 3 are repeated until a stopping criterion is satisfied. Step 2 is implemented by Equation 2 where *MIi* is the sub matrix of $M$ , representing the selection of any column j in the set of movies rated by a user $i$, $H$ is a unit matrix of rank equal to *nf* and $R(i, Ii)$ is the row vector where columns $j$ are chosen.

$$u = (M\ M\ T + \lambda n\ H)-\mathbf{1}M\ RT\ (i, I\ ) \qquad (2)$$

Step 3 is implemented by a similar formula ex- changing the roles of $U$ and $M$ .

III. MPJ IMPLEMENTATION OF ALSWR The basic strategy for distributing the ALSWR algorithm to run in parallel was already described by the original proposers in [25]. All nodes of a cluster contain a certain subset of the large,

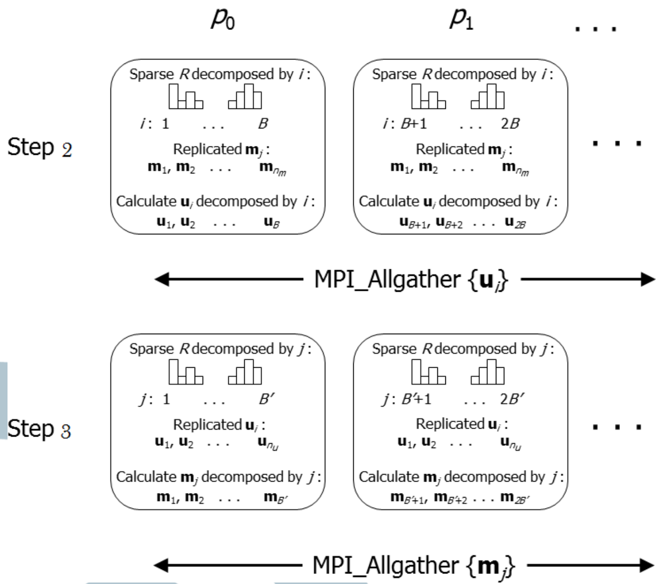


Fig. 3: Visualization of an iteration of distributed ALSWR algorithm. “Processor space” runs across the page, processes are labelled $p\mathbf{0}$, $p\mathbf{1}$, . . . and so on. Time runs down the pages with distributed computational steps labelled as on page 4. Between computational stages there are collective synchro- nizations in the form of “allgather” operations.

sparse, recommendations array, $R$. In particular it is convenient for the $R$ array to be stored in two ways across the cluster as a whole—divided across nodes by columns and also by rows. This is illustrated in figure 3, where $i$ is the subscript identifying users and $j$ is the subscript identifying items, and the two different forms of decomposition of $R$ are used in the two different steps. Step 2, as defined in equation 2, conveniently uses locally held $R$ decomposed by $i$ to update locally owned elements *ui* of the user model. $B$ is a block size for the locally held subset of elements, approximately constant across the cluster for good load balancing.

Because update of *ui* potentially involves *any* element of the item model **m**, to simplify this

step all elements of **m** should be stored locally,

in globally replicated fashion.

Step 3 has a complementary structure, but now update of *mj* may require access to any element of **u**. So between steps 1 and 2 all the locally computed elements of **u** must be gathered together and broadcast to processing nodes. Similarly be- tween step 2 and step 3 in the *next* iteration of the algorithm, the locally computed elements of **m** must be gathered and broadcast.

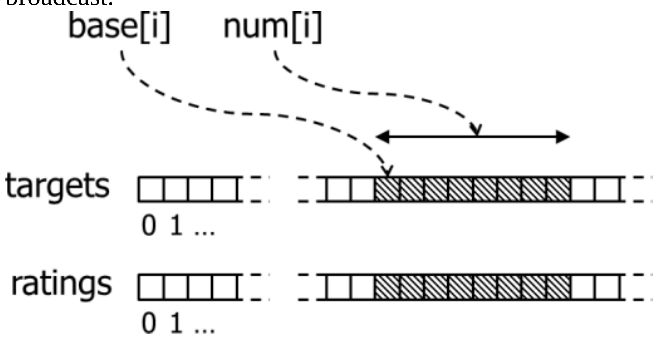

Fig. 4: Sparse data structure to represent locally held ratings. This whole structure is duplicated, once for ratings distributed by user and once for ratings distributed by items. In the “by user” case the size of the base and num arrays is the total number of locally held users, with num[*i*] being the number of ratings by user *i*; targets elements hold a *global* index of the rated item (index in the gathered array of item models). In the “by item” case the size of the top arrays is the number of locally held items, with num holding the number of ratings per item; a target element now holds the global index of the user who *made* the rating.

A great benefit of the MPI style of programming is the use of *collective communication*. This is embodied here in the use of MPI Allgather, that allows data to be gathered from each process then to be distributed to all processes.

In our program the data that we used for the implementation of the ALSWR code consists of a sparse matrix of ratings, partitioned by user or by item. Figure 4 illustrates the organization of the data.

In order to solve the symmetric positive definite matrix we use Cholesky decomposition from the Intel Data Analytics Acceleration Library (DAAL) [8].

The code assumes each node holds numLocal elements of the distributed user model. Within a node we run NUM_THREADS long lived threads (they are started at the beginning of the program), where the NUM_THREADS parameter will be related to the number of cores on the node. The variable me identifies a thread within the local node (not to be confused with the MPI *rank* which identifies a node). Threads will be synchronized before MPI collective operations using barriers implemented by java.util.concurrent.CyclicBarrier. The MPI operations themselves are only executed by the me = 0 thread.

The ratings data for our MPJ code are read from the same HDFS text files as used by the third party implementations of ALS discussed below. We use HDFS API to determine the blocks that have replicas on nodes running MPJ processes. A heuristic is used to choose a load balanced set of local replicas to read. The locally read ratings are then partitioned to destination nodes using a variant of the CARI communication schedules introduced in [18].

## IV. PERFORMANCE EVALUATION AND COMPARISON OF MPJ EXPRESS, MAHOUT, AND SPARK

This section details our experiments focusing on the comparative performance evaluation of MPJ Express against well-known platforms including Hadoop, Mahout and Spark. The performance eval- uation compares their parallel speedup.

Apache Mahout is a distributed linear algebra framework [2], widely used for its distributed im- plementation on Apache Hadoop. This essentially means that datasets are stored on the HDFS and various machine learning algorithms such as col- laborative filtering can be applied to the data. The ALSWR implementation with Apache Mahout is done through its machine learning library and more specifically the map-reduce implementation of ALS. This last consists of two stages: a parallel matrix factorization phase followed up by some recommendations. Both phases are detailed in [12]. Apache Spark is an open-source cluster- computing framework suitable for large scale data processing. Since Hadoop 2, Spark has been inte- grated with Hadoop allowing its programs to run on YARN. Spark can use memory and disk processing through its Resilient Distributed Datasets (RDD). As explained in [23], the default is to keep the RDD in memory; when there is no more space in the RAM, Spark stores the rest on disk. Shared variables and parallel operations available in Spark are detailed in [24] and [3]. We have implemented ALS on Spark through its standard machine learn- ing library (MLlib).

For the purpose of performance evaluation, we acquired our datasets from public domains. These consist of anonymous user ratings from two different sources: MovieLens and Yahoo Music. The dataset obtained from MovieLens contains
1.8 million users [20]. The data from Yahoo has been separated in training and test datasets. Our test environment includes a Linux cluster composed of 2 nodes having 6 cores each and 2 other nodes with 4 cores each; giving us in total 20 cores. Nevertheless in the experiments we limit ourselves to 16 cores in order to get the best results. Using too many cores could lead to a degradation of the performance. The software used for the tests consist of:

- Java 1.7
- Apache ant 1.6.2
- Hadoop-2.7.3
- MPJ Express (version 0.44), Mahout (version
- Intel Data Analytics Acceleration Library (DAAL) 2017

### *A. MovieLens 20M Ratings Experiments*

Our ALSWR code is tested with 50 features, 10 iterations, 0.01 for the regularization parameter lambda $\lambda$ and 0.01 for the parameter epsilon *s* that is used in the initial guess for the item model. Figure 5a compares the performances between MPJ Express, Spark and Mahout on different number of processes. MPJ Express and Spark have both a good performance and parallel speed up: as the number of cores increases the time decreases; Ma- hout does not show much variances from four cores and above. Figure 5b focus on MPJ and Spark. MPJ Express has the best performance amongst the 3 frameworks. It is, on average, 13.19 times faster than Mahout and on average 1.4 faster than Spark. Figure 6 represents the parallel speedup of MPJ Express and Spark. With sixteen cores MPJ Express is almost 10 times faster than when it is run in sequence while Spark is just about 4.5 times faster than its result with one process.

### *B. Yahoo Webscope 700M Ratings Experiments*

Mahout was unable to cope with the large Yahoo dataset. For this reason, we have evaluated only MPJ Express and Spark versions of the code for this dataset. Figure 7 shows a pattern quite similar to figure 5b although this time our dataset is about 35 times bigger. Table I displays the time measurement in minutes of the assessed frameworks. A closer

| # of pros | MPJ Express | Spark |
|---|---|---|
| 1 | 298 | 417 |
| 2 | 142 | 217 |
| 4 | 84.4 | 136 |
| 8 | 45.56 | 65 |
| 12 | 33.15 | 54 |
| 16 | 28.35 | 55 |

TABLE I: Performance MPJ Express vs Spark in minutes

Look at figure 8 demonstrates a significant parallel speedup improvement of MPJ Express which now runs more than 10.5 times faster on 16 cores than its sequential time. The parallel speedup of Spark has also improved. It implements the ALS on Yahoo dataset 7.5 times faster with 16 cores than when it is run in sequence. However from 12 cores onwards, the performance of the Spark version starts decreasing.

*C. Analysis of the results*

The Mahout implementation of ALS—not neces- sarily representative of the wider Mahout project— is based on MapReduce. The performance limi- tations of MapReduce on iterative algorithms are well documented, see for example [4]. According to pseudocode given in [24], the Spark implemen- tation uses a combination of its *parallelize* and *collect* operations to reproduce the communication operation called *MPI Allgather* here. We assume that the MPI collective algorithms can implement this pattern more efficiently. There is a discussion of efficient implementations of Allgather in [17] for example. Additionally there may be some degrada- tion of the performance of Spark when there is not enough memory (RAM) as the storage has to be on disk when the program is running out of space.

## V. CONCLUSION

On our future work we need to evaluate alter- native parallel organizations of the recommender code, like the rotational hybrid approach described in [10]. Preliminary analysis suggests that imple- mentation of similar schemes in MPI style may benefit from extensions to the standard set of MPI collectives, currently embodied in MPJ Express. Again such an extended library could form part of a future data centric version of MPJ Express that builds on experiences of MPI processing in the Hadoop environment

## REFERENCES

[1] Apache Giraph. http://giraph.apache.org/, 2014. [accessed 19-January-2018].
[2] Apache Mahout. https://mahout.apache.org/, 2017. [ac- cessed 30-January-2018].
[3] Spark rdd operations-transformation & action with exam- ple. https://data-flair.training/blogs/spark-rdd-operations-transformations-actions/. [accessed 11-June-2018].
[4] Rui Maximo Esteves, Rui Pais, and Chunming Rong. K-means clustering in the cloud–a mahout test. In *Advanced Information Networking and Applications (WAINA), 2011 IEEE Workshops of International Conference on*, pages 514–519. IEEE, 2011.
[5] Datasets — GroupLens. http://grouplens.org/datasets/, 2015. [accessed 14-December-2016].
[6] Rong Gu, Xiaoliang Yang, Jinshuang Yan, Yuanhao Sun, Bing Wang, Chunfeng Yuan, and Yihua Huang. Shadoop: Improving mapreduce performance by optimizing job ex- ecution mechanism in hadoop clusters. *Journal of parallel and distributed computing*, 74(3):2166–2179, 2014.
[7] Duane Hanselman and BC Littlefield. Mastering MAT- LAB 5: A comprehensive tutorial and reference prentice hall. *Upper Saddle River, NJ, USA*, 1997.
[8] Data analytics acceleration library. https://software.intel.com/en-us/Intel-daal, 2017. [accessed 21-October-2017].
[9] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.
[10] Maja Kabiljo and Aleksandar Ilic. Recommending items to more than a billion people. https://code.facebook.com, 2015. [accessed 30-December-2017].
[11] Efthalia Karydi and Konstantinos Margaritis. Parallel and distributed collaborative filtering: A survey. *ACM Computing Surveys (CSUR)*, 49(2):37, 2016.
[12] Introduction to als recommendations with hadoop. https://mahout.apache.org/users/recommender/intro-als-hadoop.html. [accessed 22-June-2018].
[13] Faraz Makari, Christina Teflioudi, Rainer Gemulla, Peter Haas, and Yannis Sismanis. Shared-memory and shared-nothing stochastic gradient descent algorithms for ma- trix completion. *Knowledge and Information Systems*, 42(3):493–523, 2015.
[14] MPJ Express. http://mpjexpress.org, 2015. [accessed 18-January-2018].
[15] Bikas Saha, Hitesh Shah, Siddharth Seth, Gopal Vija- yaraghavan, Arun Murthy, and Carlo Curino. 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*, pages 1357–1369. ACM, 2015.
[16] Sebastian Schelter, Christoph Boden, Martin Schenck, Alexander Alexandrov, and Volker Markl. Distributed matrix factorization with mapreduce using a series of broadcast-joins. In *Proceedings of the 7th ACMCon- ference on Recommender Systems*, pages 281–284. ACM, 2013.
MPJE Computation Time Spark Parallel
[17] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in mpich. *The International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.
[18] Bryan Carpenter Wakeel Ahmad and Aamir Shafi. Collec- tive asynchronous remote invocation (cari): A high-level and efficient communication api for irregular applications. *Procedia Computer Science*, 4:26 – 35, 2011. International Conference On Computational Science, ICCS 2011.
[19] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 4th edition, 2015.
[20] Webscope. https://research.yahoo.com/, 2006. [accessed 14-December-2016].
[21] Hsiang-Fu Yu, Cho-Jui Hsieh, Si Si, and Inderjit Dhillon. Scalable coordinate descent approaches to parallel matrix factorization for recommender systems. In *Data Mining (ICDM), 2012 IEEE 12th International Conference on*, pages 765–774. IEEE, 2012.
[22] Hamza Zafar, Farrukh Aftab Khan, Bryan Carpenter, Aamir Shafi, and Asad Waqar Malik. Mpj express meets yarn: Towards java hpc on hadoop systems. *Procedia Computer Science*, 51:2678 – 2682, 2015. International Conference On Computational Science, ICCS 2015.

[23] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in- memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[24] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster com- puting with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[25] Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. Large-scale parallel collaborative filtering for the netflix prize. *Lecture Notes in Computer Science*, 5034:337–348, 2008.