

# Comparison of Platforms for Recommender Algorithm on Large Datasets

**Christina Diedhiou**

School of Computing, University of Portsmouth, United Kingdom  
christina.diedhiou@port.ac.uk

**Bryan Carpenter**

School of Computing, University of Portsmouth, United Kingdom  
bryan.carpenter@port.ac.uk

**Ramazan Esmeli**

School of Computing, University of Portsmouth, United Kingdom  
Ramazan.Esmeli@myport.ac.uk

---

## 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 MovieLens 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.

**2012 ACM Subject Classification** Information systems → Database management system engines

**Keywords and phrases** HPC, MPJ Express, Hadoop, Spark, Mahout

**Digital Object Identifier** 10.4230/OASISs.ICCSW.2018.4

**Category** Main Track

## 1 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 MapReduce programming model, and other projects built on top of MapReduce. 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 MapReduce now just one of multiple possible distributed computation frameworks that could be supported on top of YARN. Several other major big data projects rapidly migrated to allow execution on the Hadoop YARN platform (for example Apache Spark [19], Apache Giraph [2], Apache Tez [13], and Microsoft Dryad [9]). Around the same time the present authors envisaged adding our existing MPJ Express



© Christina Diedhiou, Bryan Carpenter, and Ramazan Esmeli;  
licensed under Creative Commons License CC-BY

2018 Imperial College Computing Student Workshop (ICCSW 2018).

Editors: Edoardo Pirovano and Eva Graversen; Article No. 4; pp. 4:1–4:10

OpenAccess Series in Informatics



OASIS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

framework for MPI-like computation in Java to that distinguished group, and developed a version of our software that could also run under Hadoop YARN [17].

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. More details on MPJ Express and Hadoop can be found in appendix A.

The main goal of the present paper is to provide 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 recommendations through the collaborative filtering algorithm ALSWR (Alternating Least Squares with Lambda Regularization). A version of this algorithm is developed and evaluated using MPJ running under Hadoop. We then go on to compare our implementation with two existing implementations of ALSWR that can run under Hadoop – one taken from the Apache Mahout project using MapReduce, 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 2 gives an overview of the collaborative filtering technique. Section 3 describes how we implement the collaborative filtering with ALSWR in MPJ. The Section 4 evaluates and compare our results with Mahout and Spark. Section 5 concludes the paper and discusses future work.

## 2 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 requirements. 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, we will often refer to items as “movies” due to the fact that one of our datasets consist of ratings on movies. Assuming we have  $n_u$  users and  $n_m$  movies, and  $R$  is the  $n_u \times n_m$  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  $n_f$  of features to be found in movies; thus the behaviour of user  $i$  is modelled by a vector  $\mathbf{u}_i$  of length  $n_f$ . Similarly each movie is assumed to have each these features to a simple numeric degree so each movie  $j$  is modelled by a vector  $\mathbf{m}_j$  of the same size. The predicted preference of user  $i$  for movie  $j$  is the dot product  $\mathbf{u}_i \cdot \mathbf{m}_j$ . The vectors are conveniently collected together in matrices  $U$  and  $M$  of size  $n_u \times n_f$  and  $n_m \times n_f$  respectively. To fit the model to the known elements of  $R$  we use a least squares approach, adding a regularization 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:

$$f(U, M) = \sum_{i,j} (r_{ij} - \mathbf{u}_i \cdot \mathbf{m}_j)^2 + \lambda \left( \sum_i n_{u_i} \mathbf{u}_i^2 + \sum_j n_{m_j} \mathbf{m}_j^2 \right) \quad (1)$$

where the first sum goes over  $i, j$  values where the element  $r_{ij}$  of  $R$  is known in advance,  $n_{u_i}$  is the number of items rated by a user  $i$ , and  $n_{m_j}$  is the number of users who have rated a given movie  $j$ .

ALSWR is an iterative algorithm. It shifts between 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 [20] 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  $M_{I_i}$  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  $n_f$  and  $R(i, I_i)$  is the row vector where columns  $j$  are chosen

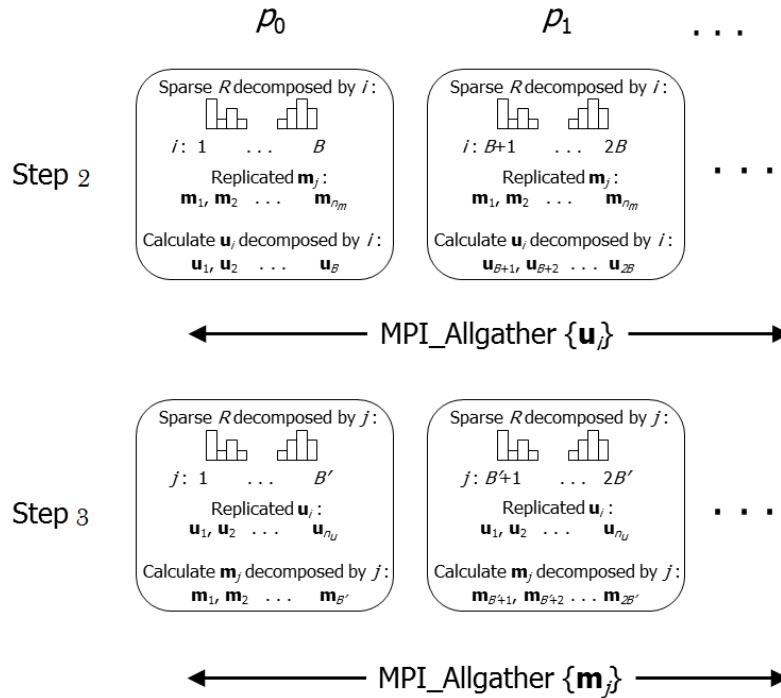
$$\mathbf{u}_i = (M_{I_i} M_{I_i}^T + \lambda n_{u_i} H)^{-1} M_{I_i} R^T(i, I_i) \quad (2)$$

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

### 3 MPJ Implementation of ALSWR

The basic strategy for distributing the ALSWR algorithm to run in parallel was already described by the original proposers in [20]. All nodes of a cluster contain a certain subset of the large, 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 1, 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  $u_i$  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  $u_i$  potentially involves *any* element of the item model  $\mathbf{m}$ , to simplify this step all elements of  $\mathbf{m}$  should be stored locally, in globally replicated fashion. Step 2 has a complementary structure, but now update of  $m_j$  may require access to any element of  $\mathbf{u}$ . So between steps 1 and 2 all the locally computed elements of  $\mathbf{u}$  must be gathered together and broadcast to processing nodes. Similarly between step 2 and step 3 in the *next* iteration of the algorithm, the locally computed elements of  $\mathbf{u}$  must be gathered and broadcast. 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



■ **Figure 1** Visualization of an iteration of distributed ALSWR algorithm. “Processor space” runs across the page, processes are labelled  $p_0, p_1, \dots$  and so on. Time runs down the pages with distributed computational steps labelled as on page 3. Between computational stages there are collective synchronizations in the form of “allgather” operations.

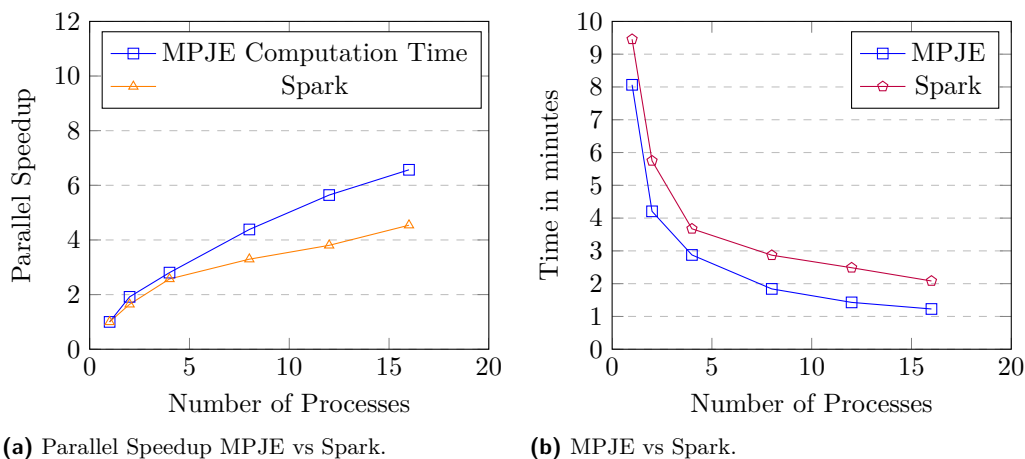
user or by item. Figure 6 in the appendices section illustrates the organization of the data. In order to solve the symmetric positive definite matrix we use Cholesky decomposition from the Intel Data Analytics Acceleration Library (DAAL) [8].

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 [1].

#### 4 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 evaluation compares their parallel speedup. More information on Apache



■ **Figure 2** Frameworks Performance Comparison MPJ Express with MovieLens dataset.

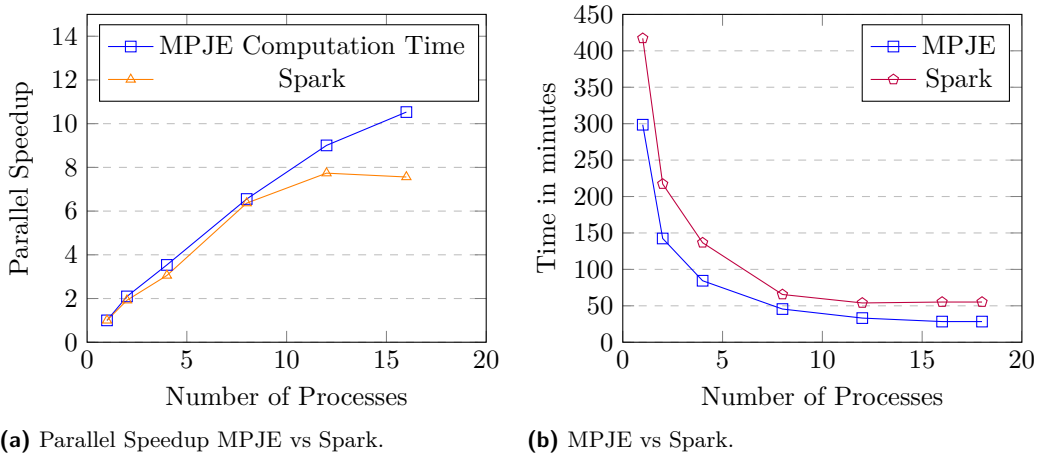
Mahout and Spark is provided in appendix B. For the purpose of performance evaluation, we acquired our datasets from public domains. These consist of anonymous user ratings from two different sources: MovieLens and Yahoo Music. Details on our datasets and test environment are provided in Appendix C.

#### 4.1 MovieLens 20M Ratings Experiments

Our ALSWR code is tested with 50 features, 10 iterations, 0.01 for the regularization parameter  $\lambda$  and 0.01 for the parameter  $\epsilon$  that is used in the initial guess for the item model. MPJ Express and Spark have both a good performance and parallel speed up: as the number of cores increases the time decreases; Mahout does not show much variances from four cores and above. Refer to Figure 5 in the Appendices section which compares the performances between MPJ Express, Spark and Mahout on different number of processes. Figure 2b 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 2a 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.

#### 4.2 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 3b shows a pattern quite similar to figure 2b although this time our dataset is about 35 times bigger. A closer look at figure 3a demonstrates a significant improvement regarding the parallel speedup 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 16 cores onwards, the performance of the Spark version starts decreasing.



■ **Figure 3** Frameworks Performance Comparison MPJ Express with Yahoo dataset.

### 4.3 Analysis of the results

The Mahout implementation of ALS – not necessarily representative of the wider Mahout project – is based on MapReduce. The performance limitations of MapReduce on iterative algorithms are well documented, see for example [5]. According to pseudocode given in [19], the Spark implementation 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 [14] for example. Additionally there may be some degradation 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.

## 5 Conclusion

Various computational frameworks have been adopted over the last few years for running compute-intensive kernels of recommender algorithms on Hadoop platforms. These include Apache Mahout, Apache Spark and Apache Giraph. In this paper we have added our MPJ Express framework to this list, and provided evidence that it can outperform other implementations of the central optimization algorithm. This additional performance certainly comes at some cost in terms of programming complexity. For example the MPJ programmer has to spend more time orchestrating communication between Hadoop nodes. Nevertheless we argue that for some intensive and often used kernels, the extra investment in programming may be justified by the potential performance gains. We see MPI-based processing stages as one more resource in the armoury of big data frameworks that may be used in processing pipelines run on Hadoop clusters. We also suggest that in this setting MPJ Express may be a natural choice of MPI-like platform, given that many other such processing stages will be coded in Java or JVM-based languages. On our future work we need to evaluate alternative parallel organizations of the recommender code, like the rotational hybrid approach described in [10]. Preliminary analysis suggests that implementation 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 Wakeel Ahmad, Bryan Carpenter, and Aamir Shafi. Collective 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.
- 2 Apache Giraph. <http://giraph.apache.org/>, 2014. [accessed 19-January-2018].
- 3 Apache Mahout. <https://mahout.apache.org/>, 2017. [accessed 30-January-2018].
- 4 Spark RDD Operations-Transformation & Action with Example. <https://dataflair.training/blogs/spark-rdd-operations-transformations-actions/>. [accessed 11-June-2018].
- 5 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.
- 6 Datasets | GroupLens. <http://grouplens.org/datasets/>, 2015. [accessed 14-December-2016].
- 7 Rong Gu, Xiaoliang Yang, Jinshuang Yan, Yuanhao Sun, Bing Wang, Chunfeng Yuan, and Yihua Huang. SHadoop: Improving MapReduce performance by optimizing job execution mechanism in Hadoop clusters. *Journal of parallel and distributed computing*, 74(3):2166–2179, 2014.
- 8 Data Analytics Acceleration Library. <https://software.intel.com/en-us/Intel-daal>, 2017. [accessed 21-October-2017].
- 9 Michael Isard, Mihai Budei, 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 Introduction to ALS Recommendations with Hadoop. <https://mahout.apache.org/users/recom-mender/intro-als-hadoop.html>. [accessed 22-June-2018].
- 12 MPJ Express. <http://mpjexpress.org>, 2015. [accessed 18-January-2018].
- 13 Bikas Saha, Hitesh Shah, Siddharth Seth, Gopal Vijayaraghavan, 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.
- 14 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.
- 15 Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 4th edition, 2015.
- 16 Webscope. <https://research.yahoo.com/>, 2006. [accessed 14-December-2016].
- 17 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. doi:10.1016/j.procs.2015.05.379.
- 18 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.



- 19 Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing 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. URL: <http://dl.acm.org/citation.cfm?id=1863103>. 1863113.
- 20 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.

## **A** Description of Hadoop and MPJ Express

### **A.1** Hadoop Overview

Hadoop is a framework that stores and processes voluminous amounts of data in a reliable, fault-tolerant manner [15].

Since Hadoop 2, YARN (Yet Another Resource Negotiator) has been integrated in the infrastructure as the resource manager, enabling many other distributed frameworks besides MapReduce to process their data on Hadoop cluster. 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 between the master nodes and slave nodes is achieved through the Heart Beat Mechanism [7].

### **A.2** MPJ Express

MPJ Express [12] 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 cluster or multicore. 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:

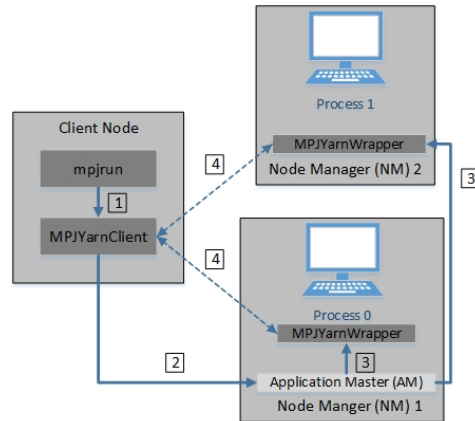
1. niodev - uses Java New I/O (NIO) Sockets
2. mxdev - uses Myrinet eXpress (MX) library for Myrinet networks
3. hybdev - for clusters of multicore processors
4. native - uses a native MPI library (like MPICH, MVAPICH, OpenMPI)

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.

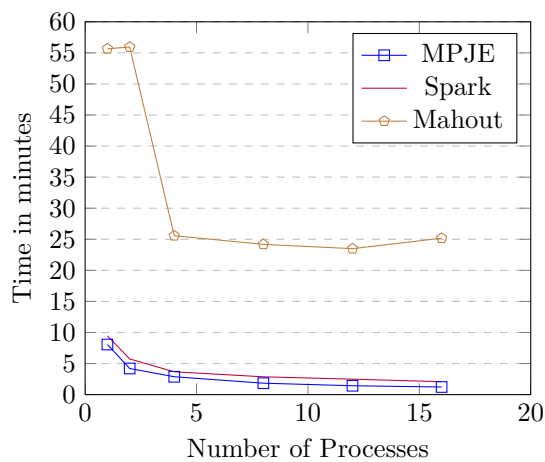
## **B** Description of Apache Mahout and Spark

Apache Mahout is a distributed linear algebra framework [3], widely used for its distributed implementation on Apache Hadoop. This essentially means that datasets are stored on the HDFS and various machine learning algorithms such as collaborative filtering can be

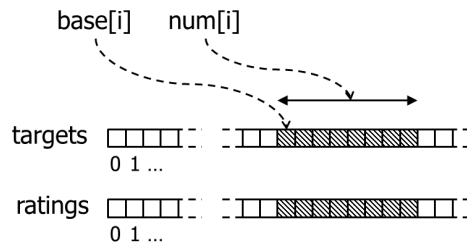




■ **Figure 4** MPJ Express Integrated in YARN- 1) Submit YARN application- 2) Request container allocation for AM- 3) AM generates a CLC and allocates container to each node- 4) Each mpj-yarn-wrapper send outputs and error streams of the program to the MPJYarnClient.



■ **Figure 5** MPJ Express, Spark and Mahout on MovieLens Dataset.



■ **Figure 6** 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.

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 [11].

Apache Spark is an open-source cluster-computing framework suitable for large scale data processing. Since Hadoop 2, Spark has been integrated 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 [18], 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 [19] and [4]. We have implemented ALS on Spark through its standard machine learning library (MLlib).

## C Description of Datasets and testing environment

The dataset obtained from MovieLens contains 20,000,263 ratings for 27,278 movies, created by 138,493 users [6]. The dataset from Yahoo Music – that is much larger – contains over 717 millions ratings for 136 thousands songs rated by 1.8 million users [16]. The data from Yahoo has been separated in training and test datasets. Our test environment includes a Linux cluster composed of 4 nodes with 20 cores in total. 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 0.12.2), and Spark (version 2.2.0)
- Intel Data Analytics Acceleration Library (DAAL) 2017