2015 IEEE 39th Annual International Computers, Software & Applications Conference

Unified Programming Model and Software Framework for Big Data Machine Learning and Data Analytics

Rong Gu, Yun Tang, Qianhao Dong, Zhaokang Wang, Zhiqiang Liu, Shuai Wang, Chunfeng Yuan, Yihua Huang
National Key Laboratory for Novel Software Technology
Collaborative Innovation Center of Novel Software Technology and Industrialization
Nanjing University, Nanjing, China 210093
{*gurong,tangyun,dongqianhao,wangzhaokang,liuzhiqiang*}*@smail.nju.edu.cn,* {*swang,cfyuan,yhuang*}*@nju.edu.cn*

***Abstract*—In a new era of Big Data, the rapid growth of the applications, such as social media and web-search, requires efficient and scalable machine learning and statistical analytical algorithms. However, there lacks easy-to-use and efficient software frameworks or systems that can support fast development of such big data analytical algorithms. To solve these problems, we propose Octopus, an easy-to-use and efficient analytical system for big data. Octopus allows data analysts conduct complex data analytics for big data with traditional programming languages and methods in an easy and efficient way. To achieve the goal of ease-to-use, we propose a matrix-based unified programming model, which is the core of many data-intensive statistical applications such as numerical analysis and data mining. Further, in order to improve the performance, the Octopus software framework adopts various distributed computing platforms, including Hadoop MapReduce, Spark and MPI. On these computing platforms, we design several parallel matrix computation algorithms, which are suitable for various scenarios. Finally, the features of Octopus are encapsulated into a library with matrix-based APIs and exposed to users as an R package. R is a widely-used statistical programming language and supports diversified data analysis tasks through extension packages. Experimental results show that Octopus achieves efficient performance and near linear scalability.**

***Keywords*-big data analysis; ease-to-use; matrix computation; parallel algorithm**

I. Introduction

As the scale of data grows, the traditional single-node computation systems can hardly cope with the large-scale data and computation. In a new era of Big Data, many real-world applications, such as social media analytics, web-search, computational advertising and recommendation systems, have created an increasing demand for scalable machine learning and statistical analytical systems on massive datasets [1][2][3]. There have emerged many distributed processing systems, such Hadoop and Spark, aiming to efficiently processing large-scale data sets. However, these systems offer low-level primitives for development, e.g. MapReduce, Spark RDD [4] . Programming with these systems requires advanced distributed system knowledge background, which is challenging to data scientists and analysts. Moreover, machine learning and statistical algorithms cannot always be naturally expressed by the exposed low-level primitives. It is cumbersome to write these algorithms directly in the data-parallel models [2].

In fact, many machine learning and scientific applications routinely involve matrix computations such as multiplication, Cholesky factorization, singular value decomposition (SVD) or LU factorization [5]. Due to the key role in these applications, matrix computation is considered as an important part of constructing computational platforms for a variety of problems [5]. On one hand, the existing distributed matrix computation libraries are inefficient or have poor fault tolerance and usability [6][7]. On the other hand, the widely-used data analysis tools, such as R, are limited by the data scalability. R is a widely-used statistical programming language and supports a variety of data analysis tasks through extension packages. A recent survey[1]of data scientists showed that R is the most frequently used tool other than SQL databases. However, data analysis in R is limited as the runtime is single threaded and can only process data sets that fit in a single machine.

In this paper, we propose Octopus, a high-level and unified programming model and platform for big data analytics and mining. Octopus works transparently on top of various distributed computing frameworks, allowing data analysts and big data application programmers to easily design and implement machine learning algorithms for big data analytics. Octopus allows users to run R scripts or shell commands interactively on a cluster without the distributed programming knowledge such as Hadoop MapReduce or Spark RDD. In the rest of the paper, we introduce the principles of the Octopus framework. Then, we evaluate the matrix computation performance of Octopus in experiments. We also demonstrate a scalable machine learning algorithm implemented on Octopus in the appendix.

II. Related Work

Previously, many research efforts were devoted to designing and developing efficient parallel matrix multiplication algorithms on distributed computing frameworks e.g. MPI

[1]In data scientist survey, R is the most-used tool http://www.r-bloggers.com/in-data-scientist-survey-r-is-the-most-used-tool-other-than-databases/

0730-3157/15 $31.00 © 2015 IEEE
DOI 10.1109/COMPSAC.2015.275

[6]. Among them, SUMMA has been widely used in the large-scale matrix multiplication applications [6]. However, this algorithm is only optimal in network transmission under the certain matrix dimension cases. Therefore, the '2.5D' algorithms [8], such as CARMA, as well as BFS/DFS-based algorithms are proposed.

In recent years, systems like MapReduce gain great success due to their good usability, scalability and fault-tolerance features. In the case of parallel matrix computation, HAMA [7], built on Hadoop with representing and storing matrices on HBase, provides distributed matrix computations. However, the execution performance of HAMA is not quite efficient due to the overhead and disk operations of the MapReduce jobs. On the other side, Microsoft develops MadLINQ [5] which translates LINQ programs into a DAG set of parallel operators that can be executed on the Microsoft Dryad platform. However, this approach does not exploit efficient memory sharing in the cloud. To solve the low programmability of traditional distributed computing environments, pbdR [9] tightly couples R with the MPI libraries, which enables developing high-level distributed data parallelism in R and also utilizing HPC platforms, but suffers the fault tolerance problems. SparkR [10] is an R package that provides a lightweight frontend to use Apache Spark from R. It exposes the low-level Spark API through the RDD class and allows users to interactively run jobs from the R shell on a cluster. In this paper, we design a big data analytic framework that can adopt various distributed computing platforms as plugin, not only MPI or Spark.

III. Software Framework and System Design

In this section, we first elaborate the software framework of Octopus and the interaction relationship of the components. Then, we introduce the typical features of Octopus.

A. Software Framework Overview

As illustrated in Figure 1, Octopus is a hierarchical system with multiple layers. At the bottom, the distributed file systems, such as HDFS and Tachyon, are adopted to store and index the large-scale matrix data. Up on the storage layer, Octopus can use various big data computing engines and the single-node R engine to execute matrix operations in different scale. The programming APIs provided to the users of Octopus are high-level matrix computation interfaces in R language. Users can easily develop big data machine learning applications without any distributed system knowledge.

B. Features

1) Easy-to-use High-level Programming APIs: Octopus provides users with a set of R-based APIs, called $OctMatrix$. The APIs are distributed matrix-computing operations, but looks similar to the Matrix/Vector operation APIs in the standard R language. Besides the large-scale matrix multiplication, Octopus also has some other matrix

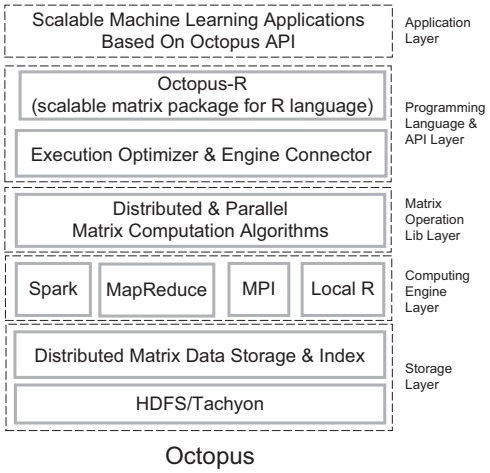


Figure 1. The system framework of Octopus

operations, such as Matrix-Matrix addition and subtraction, element wise multiplication and division, get submatrix and so on. The APIs are high-level matrix operators and operations, so that a user with basic R knowledge can program with it and implement a variety of analytical algorithms for big data. It does not require low-level knowledge or the programming skills of the distributed systems.

2) Write Once, Runs Everywhere: The programs written with Octopus can run on different computing engines. To implement algorithms with $OctMatrix$ APIs, users can run on a single-node R engine with small data for test, then run on a distributed engine with large scale data without modifying the codes. Users only need to switch to the underlying computing engines such as Spark, Hadoop MapReduce, or MPI. In addition, Octopus has a generic interface to easily implement plugins for different underlying file systems. We currently support a number of I/O sources including Tachyon, HDFS, and single-node local file systems.

3) Seamlessly Integration R Ecosystem: Octopus offers its features in an R package. Therefore, it naturally takes advantage of the rich resources of the R ecosystem, such as various third-party R packages. In addition,it can be used at many friendly R IDEs. Besides the conventional Matrix/Vector functions, Octopus also offers the apply function on $OctMatrix$. The parameter functions passed to apply can be any R functions include the UDFs. When the data of the $OctMatrix$ is located in a distributed environment, the parameter function will be executed on each element/row/column of the $OctMatrix$ on the cluster. In more detail, they are executed on the local RServer daemon process on each node of the cluster in parallel.

IV. Distributed Matrix Computation in Octopus

After introducing the framework stack of Octopus along with the features, we describe the distributed matrix com-

putation algorithms adopted in Octopus in the following subsections. We focus on the distributed matrix multiplication, which is a vital matrix operation required by many machine learning algorithms such as Page Rank, Logistic Regression and Deep Neural Networks. Also, in fact many matrix factorization operations can be approximated by matrix multiplication. We take Spark, a widely-used distributed computing engine, as the underlying processing platform to elaborate the distributed matrix computation algorithms. The algorithms can be implemented on other computing platforms in a similar way.

A. Approach 1 & 2: Block-splitting/CARMA matrix multiplication

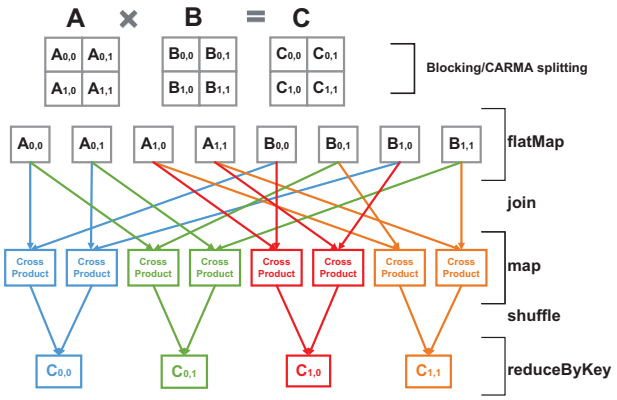


Figure 2. The workflow of block-splitting approach matrix multiplication on Spark programming model

These two approach are blocking-based matrix multiplication, which split two original matrices into blocked submatrices and execute the multiplication of submatrices in parallel. As described in Figure 2, the workflow of this algorithm is straightforward, and we can divide the whole procedure into 6 steps.

1) In *splitting* step, each original matrix has been preceded to $b \times b$ blocks.
2) In *flatMap* step each block then emits b times, which need write to disk b times in Spark programming model.
3) In *join* step, b^3 blocks of matrix B, the output of previous *flatMap* step, are transported through the network in order to get related blocks together for the next *map* step.
4) In *map* step, two related submatrices execute the cross product only once locally.
5) After the above four steps, there still exists another *shuffle* step to gather related cross product and then sum these results together in *reduceByKey* step.
6) Finally, in *reduceByKey* step, the results are grouped together and written back to the output system.

Here, the way to split the two input matrix is of great importance to the performance of the algorithm. The first approach is relatively easy and aims to deal with matrices which are near square. Each matrix can follow the near equal way by row and column. Also, the block number is a parameter for users to control the block's granularity for tuning the parallelism. However, when two input matrices are not square, the above dimension-splitting method is no longer very suitable. To solve this problem, we refer to the equal representation of dimension splitting in BFS steps of CARMA and design a dimension splitting method similar to CARMA. CARMA [11] has been proved to be communication-optimal. We also adopt the CARMA algorithm for block splitting as our second approach.

B. Approach 3: Broadcast matrix multiplication

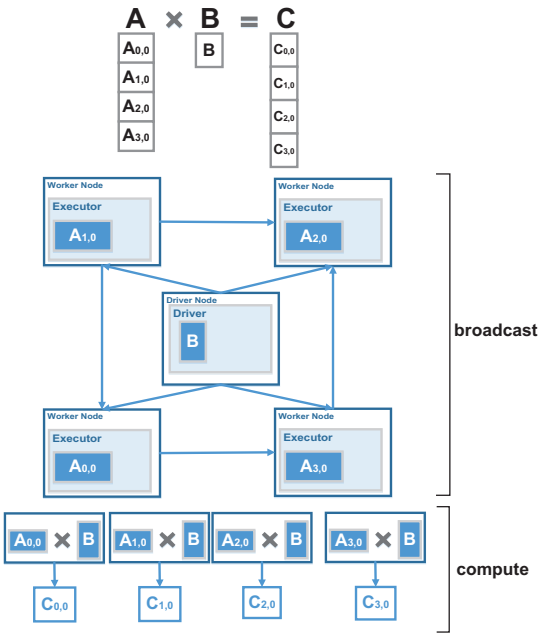


Figure 3. The workflow of the broadcast matrix multiplication approach on Spark programming model

If we want to improve the overall performance, we usually need to avoid the shuffle phases as less as possible. Based on this inspiration, we propose the broadcast approach, which reduces the shuffle cost by adopting broadcasting variables. If matrix B is quite small, it will be broadcast to each executor to avoid shuffling the large-scale matrix A across network, which can gain great performance improvement. This approach is illustrated in Figure 3. The whole procedure in only be divided into two steps, first broadcasting the variable, and then executing computation in each node's local memory in parallel. The level of parallelism is a tunable parameter r, which is the number of blocks split in

matrix A. This approach is particular suitable for the case when one input matrix is not large.

C. Native Linear Algebra Library Acceleration

Matrix computation is a typical computation-intensive task and there exist many single-node high performance linear algebra libraries, such as BLSA, Lapack and MKL. Basically, Octopus takes the divide-and-conquer strategy to deal with the large-scale matrix computation. Each computation of the divided submatrix is executed on a single node. In Fig. 4, instead of performing linear algebra computations on JVM with low performance, Octopus offloads the CPU-intensive matrix computation from JVM to the native linear algebra library (e.g. BLAS, Lapack) by JNILoader. It speeds up the performance significantly.

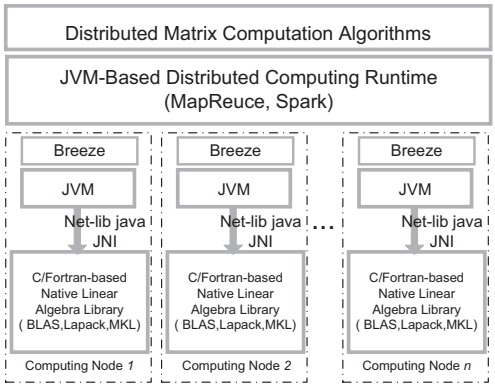


Figure 4. Adopting the single-node native matrix library to improve efficiency

V. Evaluation

In this section, we evaluate the performance of Octopus in efficiency and scalability aspects. First, we conduct a series of experiments to evaluate the effects of the proposed optimizations and approaches. Second, for comparison, we also evaluate the performance of Octopus on Spark with the SparkR [10] over various cases. SparkR is a lightweight frontend to use Apache Spark from R, which enables users to run R analytical tasks on a cluster with the support of Spark. It is the officially suggested and widely used solution to using Spark in R environment. Third, we evaluate the data scalability and machine scalability of Octopus.

A. Experimental Setup

All the experiments are conducted on a local cluster with 17 nodes connected with 1 Gb/s Ethernet. Among them, one is reserved to be the master, and the left nodes are used for computing. Each node has two Xeon Quad 2.4 GHz processors altogether 16 logical cores, 64 GB memory. All the nodes run on RHEL6 operating system with Ext3 file system. The version of the underlying Spark is 1.0.1. We adopt ATLAS version BLAS 3.2.1 as the native linear algebra library. For description simplicity, we define some terms in our following analysis. The Octopus contains three matrix multiplication approaches. The simple block-splitting approach is denoted as *Octopus-Simple Blocking*, the CARMA blocking approach is denoted as *Octopus-CARMA Blocking*, and the broadcast approach is denoted as *Octopus-Broadcast*. We perform the experiments over different sizes of matrices which are called different cases. As two input matrices, denoted as A and B, are needed for matrix multiplication, we represent a test case as $m \times k \times n$, where the size of matrix A is $m \times k$ and the size of matrix B is $k \times n$. In our comparison experiments, some systems failed to return results in a reasonable time (1 hour here), its execution time denoted as NA in the figures and tables.

B. Efficiency Performance Analysis

1) Effects of Adopting Native Linear Algebra Library: Firstly, we evaluate the effect of taking advantage of the native linear algebra library. Experimental results are shown in Figure 5. We find that both the execution time of enabling BLAS and the disabling BLAS increases as the input matrices' size scales up. The difference between these two ways is not significant for small input matrices. However, as the scale of the input matrices goes up, the enabling BLAS way is significantly faster than the disabling BLAS way. The reason is that when the matrices getting large, the computation becomes the bottleneck. And, BLAS - Fortran-implemented is native linear algebra library that is much faster than the libraries based on JVM for CPU-intensive tasks, such as matrix computation here.

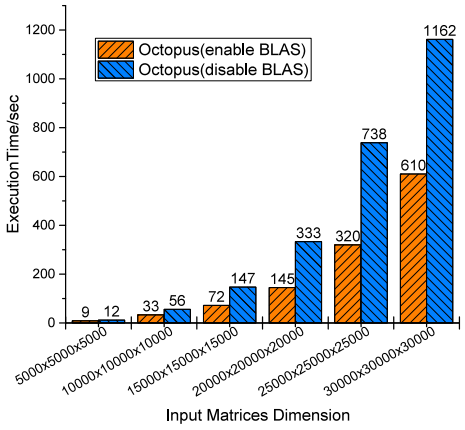


Figure 5. Performance comparison between Octopus enabling BLAS and disabling BLAS

2) Performance of Distributed Matrix Multiplication Approaches: In this subsection, we evaluate the performance of the three approaches on various cases. The experimental results on the representative cases are shown in Figure

6. We can see that the cases fall into three groups with different characteristics. Case 1 to 3 demonstrate that one of the input matrices is not so large, Case 4 to 6 stand for situations where both input matrices are large and square, and Case 7 to 9 represent situations where both two matrices are large but only with one large dimension. From the performance results, we can find that *Octopus-Broadcast* is around 10x faster than other two approaches for Case 1 to 3. This is because *Octopus-Broadcast* does not involve any shuffle process when the input matrix is small enough to be broadcast to the worker nodes. However, this method does not work when the sizes of both matrices become large. For Case 4 to 6, the *Octopus-Simple Blocking* can gain better performance than the others, because its tuning block number feature is suitable for large square matrices. From Case 7 to 9, we can find that when one dimension of the two matrices is extremely large, *Octopus-CARMA Blocking* approach is significantly faster than the others, because it avoids the shuffle process of input matrix splitting.

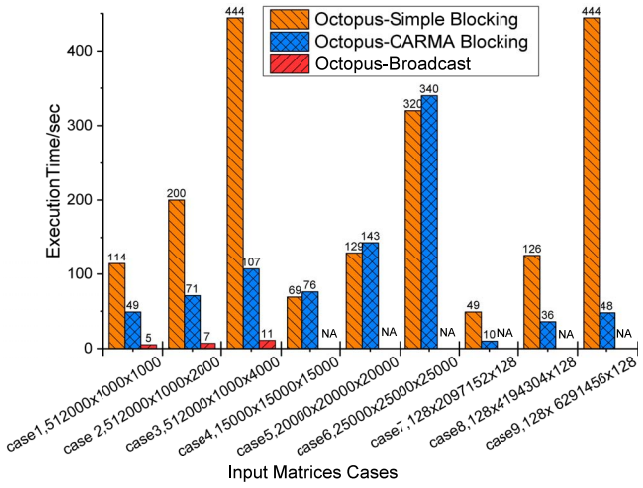


Figure 6. Performance comparison of three multiplication approaches in Octopus

3) Performance Comparison with Other Systems: In this section, we compare the performance of Octopus with the similar and representative system SparkR, which is an R package that provides a light-weight frontend to use Apache Spark from R. SparkR exposes low-level Spark primitives to users. We also implement the matrix computation algorithms on SparkR. The single-node R on both systems adopts the BLAS for acceleration. We use three groups of experiments to cover various the situations.

Group 1 represents the cases where one of dimension of the matrices keeps growing up. Group 2 stands for the cases where the common dimension of the two matrices keeps increasing. Group 3 is the mixture situations of the Group 1 and Group 2, namely multiple dimensions of the two matrices go up. In each test case, both Octopus and SparkR adopts the same distributed matrix multiplication approach. As the experimental results are shown in Table I, Octopus is always faster than SparkR, can achieve around 2 to 7 times faster. The main reason is that Octopus can directly calls the functions provided by the matrix library built on Spark without data movement between R and Spark. However, SparkR transmits the data stream between the Spark executor and GNU-R process through pipes. It brings overhead in both data transmission and serialization, which slow down the computation. Moreover, the extra data copies increase the memory footprint on the Spark worker nodes.

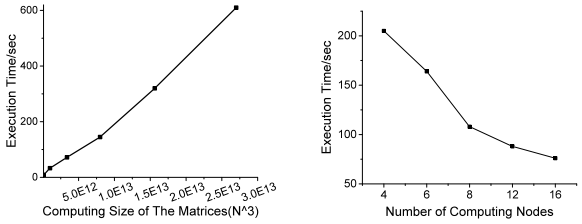
Table I
EXECUTION EFFICIENCY PERFORMANCE COMPARISON BETWEEN OCTOPUS AND SPARKR.

Matrix dimension	Octopus	SparkR	Speedup
100x100x1000	6.97 s	21.80 s	3.13
100x100x10000	8.75 s	23.08 s	2.72
100x100x100000	12.79 s	31.60 s	2.47
100x100x1000000	20.08 s	58.78 s	2.93
100x1000x100	7.07 s	21.04 s	2.98
100x10000x100	14.55 s	28.19 s	1.93
100x100000x100	16.09 s	45.65 s	2.84
100x1000000x100	18.73 s	132.42 s	7.70
1000x1000x1000	8.56 s	22.45 s	2.62
5000x5000x5000	10.19 s	55.04 s	5.43
10000x10000x10000	35.21 s	183.16 s	5.20
100000x10000x100000	624.05 s	NA	NA

C. Scalability Performance Analysis

We evaluated the scalability of Octopus by carrying out two experiments (1) scaling the size of input matrix while fixing the number of machines, and (2) scaling the number of machines while fixing the size of the input data.

1) Data Scalability: Experimental results are shown in Figure 7(a), and we observe that execution time of Octopus grows close to linearly with the increase of data size. It indicates that Octopus achieves good data scalability.



(a) Data scalability (b) Node scalability

Figure 7. Scalability

2) Node Scalability: We also conduct a group of experiments to evaluate the machine scalability performance of Octopus. The dimensions of the input matrices are fixed

at $15000 \times 15000 \times 15000$ in these experiments. It can be seen from Figure 7(b) that the execution time of Octopus decreases about 3 times as the number of cluster nodes increases. This means Octopus has good machine scalability.

VI. Conclusion and Future Work

Many real-world applications require large-scale data analytics. Matrix computation is a fundamental kernel of many machine learning and statistical analysis applications. In this paper, we propose Octopus, a high-level and unified programming model and platform for big data analytics and mining. It offers an R package that provides easy-to-use scalable matrix operations from R and seamlessly executes computation on the single-node R and distributed computing frameworks such as Spark, Hadoop, MPI, etc. Experimental results show that Octopus achieves good efficiency and near linear scalability. To the best of our knowledge, Octopus is first software framework that can transparently work on top of various distributed computing frameworks.

In the future work, we plan to design more efficient distributed matrix computation algorithms, such as matrix factorization and sparse matrix computation, into Octopus. We also want to explore adopting more underlying computing engines, including GPU and FGPA devices.

VII. Appendix

The script below is an implementation of the logistic regression training algorithm, which is based on the $OctMatrix$ programming API. The program only requires around 20 lines of straightforward code.

Script: Logistic Regression on Octopus

```
#Train logistic Regression Model
#@param x, training dataset,an OctMatrix
#@param y, labels, an OctMatrix
#@param iters, the iterative rounds
#@param step, the step size for gradient
#@return model(weights)
train.lr<-function(x, y, iters, step) {
  dims <- dim(x)
  m <- dims[1]
  n <- dims[2]
  x <- cbind2(ones(m, 1), x)
  theta <- zeros(n+1, 1)

  g <- function(z) {
    1.0 / (1.0 + exp(-z))
  }
  for (i in 1:iters) {
    z <- x %*% theta
    h <- apply(z, c(1,2), g)

    #update model
    grad <- t(x) %*% (h - y)
    theta <- theta - step/sqrt(i)*grad
  }
  theta
}
```

VIII. Acknowledgement

This work is funded in part by Jiangsu Province Industry Support Program (BE2014131) and China NSF Grants (No.61223003).

References

[1] E. R. Sparks, A. Talwalkar, V. Smith, J. Kottalam, X. Pan, J. E. Gonzalez, M. J. Franklin, M. I. Jordan, and T. Kraska, "MLI: an API for distributed machine learning," in *2013 IEEE 13th International Conference on Data Mining (ICDM), Dallas, TX, USA, December 7-10, 2013*, 2013, pp. 1187–1192.

[2] S. Venkataraman, E. Bodzsar, I. Roy, A. AuYoung, and R. S. Schreiber, "Presto: distributed machine learning and graph processing with sparse matrices," in *2013 European Conference on Computer Systems (EuroSys), Prague, Czech Republic, April 14-17, 2013*, 2013, pp. 197–210.

[3] M. Boehm, S. Tatikonda, B. Reinwald, P. Sen, Y. Tian, D. Burdick, and S. Vaithyanathan, "Hybrid parallelization strategies for large-scale machine learning in systemml," *PVLDB*, vol. 7, no. 7, pp. 553–564, 2014.

[4] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI*. USENIX Association, 2012.

[5] Z. Qian, X. Chen, N. Kang, M. Chen, Y. Yu, T. Moscibroda, and Z. Zhang, "Madlinq: large-scale distributed matrix computation for the cloud," in *2012 European Conference on Computer Systems (EuroSys), Bern, Switzerland, April 10-13, 2012*, 2012, pp. 197–210.

[6] R. A. Van De Geijn and J. Watts, "Summa: Scalable universal matrix multiplication algorithm," *Concurrency-Practice and Experience*, vol. 9, no. 4, pp. 255–274, 1997.

[7] S. Seo, E. J. Yoon, J. Kim, S. Jin, J.-S. Kim, and S. Maeng, "Hama: An efficient matrix computation with the mapreduce framework," in *CloudCom*. IEEE, 2010, pp. 721–726.

[8] E. Solomonik and J. Demmel, "Communication-optimal parallel 2.5 d matrix multiplication and lu factorization algorithms," in *Euro-Par*. Springer, 2011, pp. 90–109.

[9] D. Schmidt, G. Ostrouchov, W.-C. Chen, and P. B. Patel, "Tight coupling of r and distributed linear algebra for high-level programming with big data." in *SC Companion*, 2012, pp. 811–815.

[10] "Sparkr: R frontend for spark," 2013. [Online]. Available: http://www.r-project.org/

[11] J. Demmel, D. Eliahu, A. Fox, S. Kamil, B. Lipshitz, O. Schwartz, and O. Spillinger, "Communication-optimal parallel recursive rectangular matrix multiplication," in *IPDPS*. IEEE, 2013, pp. 261–272.