International Journal of Computer Science and Applications,
©Technomathematics Research Foundation
Vol. 13, No. 1, pp. 93 – 104, 2016

EFFICIENT MATRIX MULTIPLICATION IN HADOOP

SONG DENG

School of Software & Communication Engineering, Jiangxi University of Finance and Economics,

nanchang 330013, china
daonicool@sina.com

WENHUA WU

School of Software & Communication Engineering, Jiangxi University of Finance and Economics,

nanchang 330013, china
wuwenhua2012@gmail.com

In a typical MapReduce job, each map task processing one piece of the input file. If two input matrices are stored in separate HDFS files, one map task would not be able to access the two input matrices at the same time. To deal with this problem, we propose a efficient matrix multiplication in Hadoop. For dense matrices, we use plain row major order to store the matrices on HDFS; For sparse matrices, we use the row-major-like strategy. So, a mapper can get the rows and columns by only scannig through a consecutive part of a file. We modify the Hadoop MapReduce input format, add two file paths to the two input matrices and store the input matrices in row major order. With the new file split structure, all data are distributed properly to the mappers. Finally, we propose a user feedback method to avoid the overheads of starting multiple map waves. A number of comparative experiments are conducted, the result show that our method observably improve the performance of dense matrix multiplication in MapReduce.

Keywords: Matrix multiplication; Hadoop; MapReduce; Matrix Layout; Input Format

1. Introduction

Efficient statistical analysis of large datasets is important in both industry and academia. Huge amount of data are being generated and collected everyday. Companies need to extract useful business information out of the raw data by performing machine learning algorithms or statistical analysis. Research scientists in various fields also have large demand to analyze large data gathered from experiments.

The MapReduce framework, which was put forward by Google, has become to be a popular model for processing a mass of data in parallel. On the other hand, there are massive amount of computation being involved in the matrix multiplication which is common and important algebraic operation in many applications and can be parallelized. In this project, the matrix multiplication problem is studied in MapReduce. On the basis

of the conventional approach which runs in $\Theta(n^3)$ time in the non-parallel setting, designing, implementing and analyzing algorithms would be the focus of attention.

In a common MapReduce job, each map task takes the responsibility of processing one piece of the input file(s) called a split. By default in Hadoop, a piece of data from a single input file is contained in a file split. This is an indication that one map task would be able to access the two input matrices at the same time, if the two input matrices are stored in the same HDFS file. There are two approaches in previous works that deals with this problem.

At the reduce side, starting the multiplication would solve this problem. In this way, to finish the whole task, two MapReduce jobs would be needed. In the first job, the map phase reads in the two matrices to be multiplied and distributes them to the reducers as required. The reduce phase does block-level multiplication, and flushes the results into temporary HDFS file. Then in the second job, use identity mappers to read the blocks back from the temporary files and pass them on to the reducers, in which blocks are added up to produce the final result.

In this approach, to make distribution of data properly for the reducers, map phases in both jobs would be used. The block multiplications and additions are done in the reduce phases. Therefore, it copies the data repeatedly throughout this process, not only in the two shuffle phases, but also during HDFS I/O to/from the temporary files between the two jobs.

In order to finish the matrix multiplication in one MapReduce job, researchers could use a preprocessing phase to reorganize the input file in line accordance with some strategy. More specifically, it is because that one input file which contains replicated block data from both matrices are generated in an interlacing manner, so each file split contains the exact data a map task needs for the block-level multiplication. After that, the mappers do block multiplication and the reducers take the responsibility of adding the blocks up to acquire the output matrix. In this approach, the preprocessing phase basically copying data and replicating blocks of the two input matrices incurs massive amount of extra file system I/O.

After evaluating the pros and cons of both approaches, we made some modification about the Hadoop MapReduce input format and made a mapper have the ability to get the proper block data from both input matrices without the processing phase. Consequently, we have the ability to multiply two matrices in one MapReduce job as well as avoid the preprocessing cost at the same time.

2. Related Work

Packages like R, LAPACK[Bosilca (2014)] and Matlab start with the assumption that all data can fit in the memory of a single machine. RIOT[Zhang (2010)] and SOLAR[Konda (2013)] extends these in-memory packages to process data stored on disk, which made them capable to deal with larger data sizes, yet they are still limited to a single machine. As the data size is growing at an unprecedented speed, the limited computation resources of a single machine cannot suffice the data processing needs.

Over the years, the high performance community has built various packages like ScaLAPACK[Hasanov (2014)] aiming at parallelized computation in a distributed memory setting. The algorithm is often parallelized at a rather low level and thus requires a lot of inter-processor communication and/or coordination at each step. In cases where some nodes have lower network bandwidth or less powerful CPU, this node would slow down the whole cluster. Yet they have zero tolerance for machine failure. As a result, these packages are difficult to use in larger, distributed, and often heterogeneous, cluster settings. Furthermore, the data size is still bounded by the assumption that the data is already partitioned and resides in the memory of each machine. Consequently, these packages are more suitable to tightly-coupled and high-end clusters.

The cost of acquiring and maintaining a dedicated high-end cluster is prohibiting for most scientists and organizations. Cloud computing has made it possible to rent machines on demand and pay as we go. They also comes with more machine choices. Some commodity machine types might not be as powerful as high-end clusters with dedicated interconnect. But as the size of the cluster could potentially scale up to thousands of nodes, we can handle most jobs satisfactorily with reasonable prices.

When looking at this kind of clusters with the assumption that the data is too large to fit in the collective memory of all machines, the MapReduce framework[Hassan (2014)] becomes promising because of its superior scaling ability to leverage large clusters with fault tolerance, and because it is built based on a distributed file system as data source.

At the same time, there is also a great interest in building statistical engine in MapReduce. MapReduce has already shown its advantage in large scale text processing and analytics like Pig Latin[Gates (2009)]. Clusters of Hadoop[White (2010)], the open source implementation of MapReduce, are already widely deployed and in use. It will be nice if statistical workloads could also be carried out using the same platform with no or little modification.

There are many MapReduce-based works regarding statistical workloads. Mahout implemented a collection of machine learning algorithms in MapReduce, but they are algorithm specific and do not offer a general solution[Ekanayake (2010)]. MLlib[Meng (2015)] is a more recent ML library built on top of Spark[Zaharia (2012)]. Ingeneral, these library-based approaches are algorithm specific and do not offer a general solution. pR[Li (2011)] dispatches single line of R program from an R runtime into a Hadoop cluster for parallel execution, but it is limited and not a systematic and integrated parallelization solution. Rhipe[Oancea (2014)] and Ricardo[Das (2010)] provided program interface to integrate R and MapReduce. However, users need to know about MapReduce, reason and hand code the program. Decisions like algorithm choice, task partitioning and data management are all left to users. SystemML[Ghoting (2011)] [Boehm (2014)] provided a solution to automatically convert simplified R workflow into executable MapReduce jobs with limited optimizations.

However, we observed that the MapReduce programming model is not a natural fit for matrix workloads. In MapReduce, each piece of input data is accessible to only one mapper, while statistical computing operation often have more complex and specific data access requirements. One traditional approach to work around this is use the map phase

simply to read in data from file system, replicate and shuffle them to the reducers according to the semantics. The actually computation then happens at the reduce phase. We observed that in this approach essentially no effective computation is done in the map phase. Each piece of data get transfered up to twice before it is actually processed. This is a waste of both system resources and time.

3. Strategies

In this part, three strategies for multiplying two dense matrices and one strategy for multiplying sparse matrices in MapReduce will be discussed conceptually. To be simple and clear, we make an assumption that both of the inpit are square matrices.

In order to make quantitative analysis on different strategies easier, we use M to refer to the size of the input, n to the number of blocks partitioned along each side of the input matrix, and N to the number of physical map/reduce slots which are limited by the number of nodes in the cluster.

While existing works that shuffle all data from mappers to reducers and make reducers do the whole block multiplications, we are in a position to do the block multiplications, which decides the total computing time, at the map phase, using the techniques depicted in Section 4, thus accelerating our program in reality.

3.1. *Dense matrix multiplication*

Strategy 1. Each of the input matrices is divided into $n \times n$ small square blocks equally. Then the size of each block would be $(M/n) \times (M/n)$. The output matrix would involve $n \times n$ blocks, and each of them result from the addition of n block matrix multiplications. Then, make each map task handle one block matrix multiplication. Therefore, there would be n^3 map tasks in total.

Strategy 2. The first input matrix is divided into n row strips, and the second input matrix is divided into n column strips. Each map task reads one row strip from the first matrix and one column strip from the second matrix, so can produce one (M/n)-by-(M/n) block in the output matrix. By using this strategy there would be n^2 map tasks.

Strategy 3. In this strategy, the first matrix is divided into $n \times n$ square blocks, and the second one is divided into n row strips. A map task multiplies a block from the first matrix and a row strip from the second to produce partial results for a row strip of the output matrix. n^2 map tasks would be required by this strategy.

Being used the notations M and n defined at the beginning of this section, the three strategies are compared according to the following aspects: the total input traffic on the map side, average traffic into each map task, the amount of shuffle traffic from the mappers to the reducers, the amount of computation done by each map task (in terms of number of block multiplications), memory needed for each map task, and the total number of mappers. The comparison is presented in Table 1.

Table 1. Summary 1 on the three three strategies.

	Strategy 1	**Strategy 2**	**Strategy 3**
Map input traffic (total)	$2M^2n$	$2M^2n$	M^2n
Map input traffic (average)	$2M^2/n^2$	$2M^2/n$	M^2/n
Shuffle traffic	M^2n	M^2	M^2n
Computation per map task	1	n	n
Memory per map task	$3M^2/n^2$	$2M^2/n$	$2M^2/n$
Number of map tasks	n^3	n^2	n^2

The degree of parallelism is limited by the number of physical map/reduce slots, so we can make an assumption that for each strategy the number of map tasks matches the number of physical map slots N. We substitute $N^{1/3}$ *or* $N^{1/2}$ for all n's in Table 1 and obtain Table 2.

It is worth considering that if the amount of map slots is more than the amount of map tasks, the parallelism is totally utilized. The number of multiplication done in each map task should be the same by means of different strategies, because the total amount of multiplication and the amount of map slots are fixed. However, the "Computation per map task" of Strategies 2 and 3 are more than that of Strategy 1 demonstrated in Table 2, it dues to the computation per map task calculates the amount of block multiplications done by each map task. The block size for Strategy 2 and 3 is smaller than that of Strategy 1 when the number of map slots is the same of the number of map tasks and the amount of map slots is fixed.Taking block sizes into account is easy to examine, the actual number of computation per map task is the same for all three strategies.

Table 2. Summary 2 on the three three strategies.

	Strategy 1	**Strategy 2**	**Strategy 3**
n	$N^{1/3}$	$N^{1/2}$	$N^{1/2}$
Map input traffic (total)	$2M^2N^{1/3}$	$2M^2N^{1/2}$	$M^2N^{1/2}$
Map input traffic (average)	$2M^2N^{-2/3}$	$2M^2N^{-1/2}$	$M^2N^{-1/2}$
Shuffle traffic	$M^2N^{1/3}$	M^2	$M^2N^{1/2}$
Computation per map task	1	$N^{1/2}$	$N^{1/2}$
Memory per map task	$3M^2N^{-2/3}$	$2M^2N^{-1/2}$	$2M^2N^{-1/2}$

In Section 5, the experiment results for all of the three strategies shown in this subsection by us. We will analyze the results and see if they agree with our analysis in Table 2.

3.2. *Sparse matrix multiplication*

We use a simple analogy to strategy 3 shown in Section 3.1 for sparse matrices. Figure 1 demonstrates that multiplying the corresponding row in matrix A with the whole matrix B to result in matrix C. we use one line in matrix A as the unit to partition the workload between mappers in our method. Fundamentally, we give some consecutive lines from matrix A for each map tasks so that the amount of non-zero values in those lines is approximate for different mappers. If the distribution of non-zero values in B is independent of that of A, the prospective workload for each mapper is supposed to the same roughly.

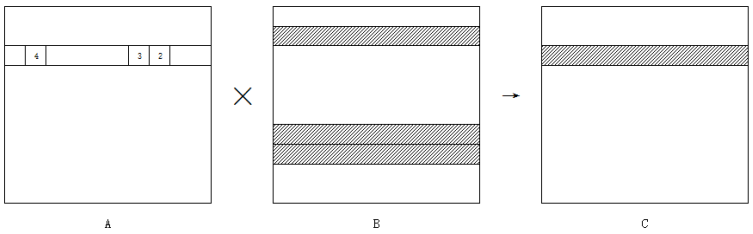


Fig. 1. Strategy for sparse matrices

4. Implementation

As for this chapter, we discuss how the input matrices are demonstrated are HDFS, how we modified the Hadoop code to distribute data properly before the map phase and how our programs can be configured.

4.1. *Matrix layout*

For dense matrices, in order to be flexible in both matrix size and block size, we apply plain row major order to store the matrices on HDFS. For a matrix of size M, entries are stored in the following order: (0,0)(0,1)...(0,M-1)(1,0)(1,1)...(1,M-1)...(M-1,0)(M-1,1)...(M-1,M-1). A mapper can get the rows and columns by means of scanning through a consecutive part of the files with this method.

4.2. *Input format*

Many different types of data input formats are provided by the Hadoop MapReduce framework. But, most of them rather focus on the data type within the input file than the input file structure. FileSplit used by all input formats includes four variables: file, start, length, hosts[].

The data corresponding to this split start from offset start to offset start+length within file file. Hosts is the physical location of this file split, which is later used to choose a data-local map task. It's obvious that this structure limits the mapper from reading data and multiple file. Therefore, we change its structure through extending the class FileSplit as follows:

Path *file1*;
Path *file2*;
long *start1[], start2[]*;
int *x, y, z*;
int *M, n*;

In our file split,we involve many information.The first is to have two file paths for the two input matrices.The second is store the input matrices in row major order, a matrix block required by a mapper is interrupted in the input file.Therefore two lines *start1[]* and *start2[]* are going to write down all the starting offsets for the blocks. *M* and *n means* the matrix size and the amount of blocks per row/column as shown in Section 3. *x*, *y*, and *z means* the block level indices of the block, making the mapper know the address of the block in the input matrix.

We carried out our own *FileInputFormat* with this new file split structure, and the *FileInputFormat* forms the file split in our desired fashion, and *RecordReader* which takes our file split as input. Then all data are offered to the mappers properly so that in the map phase, the computation can be implemented.

5. Experiments

We examined our matrix multiplication strategies on the taobao.com platform, on clusters of 4, 8, and 16 slave nodes. Each slave node can have the ability of 3 map slots and 3 reduce slots, i.e. Each 12, 24, and 48 is the maximum number of mappers that be conduct simultaneously.

5.1. *Results for dense matrices*

We take the formed square matrices as input in a causal way.We are beginning with testing the effect of map startup overhead caused multiple map waves.

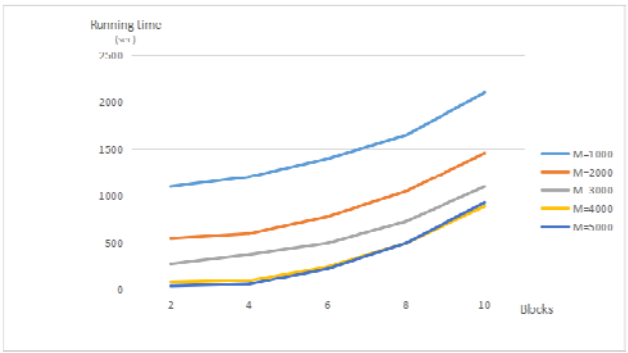


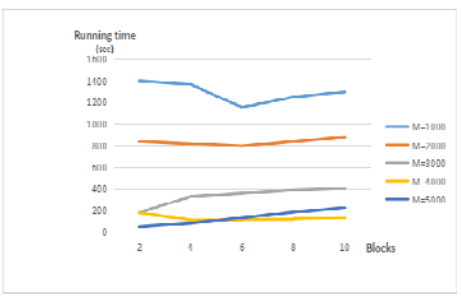
Fig. 2. Strategy 1. 4 nodes

For strategy 1, the amount of map tasks raise cubically with the add of number of blocks partitioned along a row/column (n).With a 4-node cluster, we have 12 map slots.The large amount of map tasks exceeds the amountr of map slots starting from 3

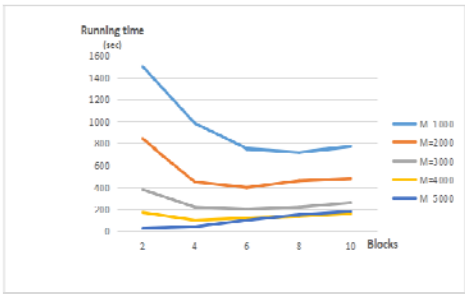
blocks along each side of a matrix. From this way, as the number of map slots is fixed,the amount of map waves adds linearly with the number map tasks, i.e. cubical with the number of blocks. We can notice a non-linear growth in running time when *n* increases linearly in Figure 2.We supposed that this non-linear growth should be owe to map startup overheads, and carried on to check on the second strategy.

For strategy 2, when n is small, the number of map slots is much bigger than that of mappers. It indicates that the parallelism given by the cluster hasn't got complete utilization. This is the reason why we see a harsh drop in running time in both Figure 3(b) and (c), when *n* is increased from 2 to 4. It is expected to achieve its minimum running time at $n \approx N^{1/2}$ on the basis of our observation and conjecture in Figure 2. The trend is not apparent in Figure 3(a) because of the small size of the cluster. In Figures 3(b) and (c), we have minimized the running time when *n* is either 4 or 6.

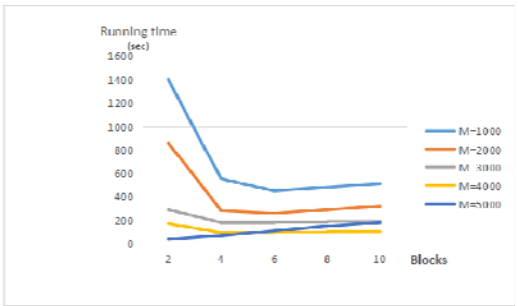
Another aspect that we should notice is the amount of input traffic to the mappers as n increases. Due to the entire storage of input matrices on HDFS , the rate of reading data from input files enters a bottleneck period. In the light of our analysis in Section 3.1, for all three strategies, the overall amount of input traffic improves linearly with *n*. Taking this into consideration, the number of blocks *n* in the minimal running time would be transferred a little to the left of $N^{1/2}$, as we can see in Figure 3(c). This effect is not as evident in Figure 3(b).



(a) 4 nodes



(b) 8 nodes



(c) 16 nodes

Fig. 3. Strategy 2. Running time vs. number of blocks *n*

We also make an examination about the influence of the size of the cluster on the running time. We use strategy 3 for this experiment. For each size of the input matrices, we design the minimal running time achieved through the adjustment of *n*. A 4-node

cluster is started and doubled the cluster size for twice. The following displays the results in Figure 4.

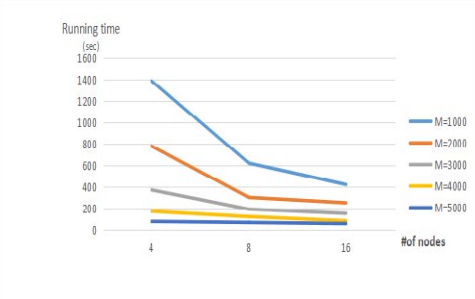


Fig. 4. Strategy 3. Running time vs.cluster size

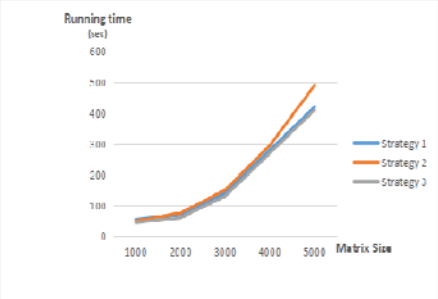


Fig. 5. Comparison between the three strategies

Perfectly to say, when doubling the cluster size, we should halve the running time. Nonetheless, there are overheads like HDFS I/O cost, shuffle cost, map startup cost, and etc. As we can see in Figure 4, when the number of nodes increases from 4 to 8, the running time is roughly halved because of the invisible overheads compared with the long running time. When we double the number of nodes again, from 8 to 16, the new running time is clearly larger than half of the original running time.

Just like what we did for the previous experiment, for each combination of problem size and strategy, we make an adjustment of n to achieve the minimal running time. The results are sketched in Figure 5. Based on our analysis on the relationship between block size and map startup overhead, the running time achieves its minimum when parallelism is entirely utilized and as few map waves are involved as possible. So the time spent on computation ought to be approximately the same for all the three strategies.

We have two discoveries from the figure. First of all, strategy 3 exceeds strategy 4 a little in all matrix sizes. Two possible reasons for this slight difference can account for this, both related to reading input from HDFS. First reason comes from the amount of input traffic for strategy 3 which is half of that of strategy 2. Second reason is that the data each mapper gets from the second input matrix is successive for strategy 3 instead of strategy 2.

The second discovery is limited in terms of memory. We can see from the figure that as the problem size increases, the running time of strategy 1 rises fastest compared with strategy 2 and 3. The reason is that when matrix size mounts up to 4000/5000, all of the minimal running time failed to achieve the theoretically optimal block size on account of the large amount of memory required. Consequently, we had to divide the matrix into smaller pieces, which results in more map waves. Since the number of mapper for strategy 1 develops more rapidly than strategy 2 and 3(cubic vs. Quadratic) more map startup overheads are turned up.

Comparing our strategy 1 with [Benson (2015)] on a 16-node cluster, on 5000-by-5000 matrices. We tried to make the parameters suitable in order to minimize the performance time for both programs. It costs our program 495 seconds to finish while theirs took

longer,and finished in 2371 seconds. We also expanded the input size to 10000-by-10000. It costed our program 3926 seconds to finish, which is equivalently 2^3=8 times the performance time to multiply two 5000-by-5000 matrices.

5.2 *Results for sparse matrices*

Our discrete matrix was generated in a stochastic way. For a $m \times m$ sparse matrix, we let each value be non-zero with probability ln m/m independently, in that way, the expected number of non-zero values per line is ln m, and m ln m for the whole matrix.

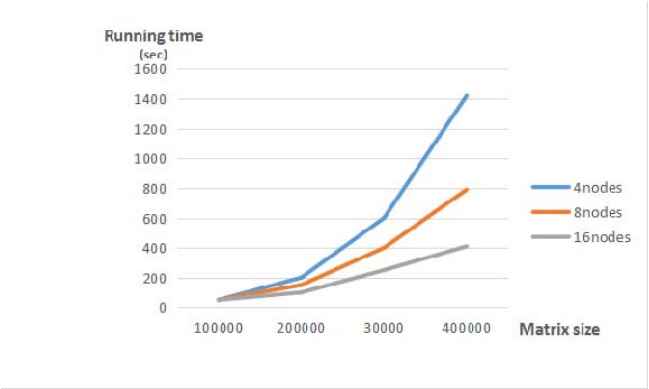


Fig. 6. Sparse Matrix Result

Under the guiding of this method,we got matrices with size from 100000 (100K) to 400000 (400K) and run the sparse matrix multiplication on the same cluster above with 4, 8 and 16 nodes, which has 12, 24 and 48 map slots. The experiment results are shown in Figure 6. If the number of mapper is set a little smaller than the number of map slot, the map phase would finish in only one wave. Therefore, there won't be too much overheads when setting up new mappers.

If we give the same number of nodes, as the matrix size develops, the input traffic for each mapper grows linearly. The amount of computing operations for matrix with m ln m non-zero values are m $\ln^2 m$. As we can see, in Figure 6, the curves a growth in performance time larger than linear.

If we give the same input to matrix size, the computation workload will turn out to be the same. As the cluster grows from 4 nodes to 8 and 16 nodes, the performance time are almost cut down by half twice. It is the result of the low overhead involved in our implementation.

6. Conclusion

In the study, the Hadoop input format has been altered in order to make sure a mapper to get non-consecutive data from multiple files. After we have done this, we can not only complete a matrix multiplication in one MapReduce job, but also avoid the preprocessing overhead at the same time. We applied this technique to three dense matrix multiplication

strategies and one sparse matrix multiplication strategy, and conducted experiments on our programs.

In summary, the performance of dense matrix multiplication in MapReduce has been impeoved by slightly modifying the infrastructure of Hadoop, and packing all work into one MapReduce job. The way how to configure the number of blocks to optimize the performance has been clarified through experiments, and possible solutions to limitations have been given in the current approach, for example, bottleneck in data transfer bandwidth and memory limit of mappers. Much work should be done in order to implement these ideas as well as to examine the impact of correlation between the input matrices on performance for sparse matrices.

References

Bosilca, G.; Ltaief, H.; Dongarra, J. (2014): Power profiling of Cholesky and QR factorizations on distributed memory systems. Computer Science - Research and Development, 29(2), pp. 139-147.

Boehm, M.; Tatikonda, S.; Reinwald, B.; Sen, P.; Tian, Y.; Burdick, D. R.(2014): Hybrid parallelization strategies for large-scale machine learning in systemml. Proceedings of the 40th International Conference on Very Large Data Bases, pp. 553-564.

Benson, A. R.; Ballard, G.(2015): A framework for practical parallel fast matrix multiplication. Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 42-53.

Das, S.; Sismanis, Y.; Beyer, K. S.; Gemulla, R.; Haas, P. J.; Mcpherson, J.(2010): Ricardo: integrating R and Hadoop. Proceedings of the Acm Sigmod International Conference on Management of Data, pp. 987-998.

Ekanayake, J.; Li, H.; Zhang, B.; Gunarathne, T.; Bae, S. H.; Qiu, J.(2010): Twister: a runtime for iterative MapReduce. Proceedings of the ACM International Symposium on High Performance Distributed Computing, pp. 810-818.

Gates, A. F.; Natkovich, O.; Chopra, S.; Kamath, P.; Narayanamurthy, S. M.; Olston, C.; Reed, B.; Srinivasan, S.; Srivastava, U.(2009): Building a high-level dataflow system on top of Map-Reduce: the Pig experience. Proceedings of the Vldb Endowment, pp. 1414-1425.

Ghoting, A.; Krishnamurthy, R.; Pednault, E.; Reinwald, B.; Sindhwani, V.; Tatikonda, S.(2011): SystemML: Declarative machine learning on MapReduce. Proceedings of the IEEE International Conference on Data Engineering, pp. 231-242.

Hasanov, K.; Quintin, J. N.; Lastovetsky, A.(2014): Hierarchical approach to optimization of parallel matrix multiplication on large-scale platforms. Journal of Supercomputing, 71(11), pp. 1-24.

Hassan, M. A. H.; Bamha, M.; Loulergue, F.(2014): Handling Data-skew Effects in Join Operations Using MapReduce. Procedia Computer Science, 29, pp. 145-158.

Konda, P.; Kumar, A.; Ré, C.; Sashikanth, V. (2013): Feature selection in enterprise analytics: a demonstration using an R-based data analytics system. Vldb Demo, 6(12), pp. 1306-1309.

Li, J.; Ma, X.; Yoginath, S. B.; Kora, G.; Samatova, N. F.(2011): Transparent runtime parallelization of the R scripting language. Journal of Parallel & Distributed Computing, 71(2), pp. 157-168.

Meng, X.; Bradley, J.; Yavuz, B.(2015): MLlib: Machine Learning in Apache Spark, arXiv preprint arXiv:1505.06807.

Oancea, B.; Dragoescu, R. M.(2014): Intergrating R and Hadoop for big data analysis. Romanian Statistical Review, 62(2), pp. 83-94.

White, T.(2010): Hadoop: The Definitive Guide. O'reilly Media Inc Gravenstein Highway North, 215(11), pp. 1-4.

Zhang, Y.; Zhang, W.; Yang, J.(2010): I/O-efficient statistical computing with RIOT. Proceedings of the 26th International Conference on Data Engineering (ICDE), pp. 1157-1160.

Zaharia, M.; Chowdhury, M.; Das, T.; Dave, A.; Ma, J.; Mccauley, M.(2012): Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, pp. 141-146.