Massively Parallel Unsupervised Feature Selection on Spark [⋆]

Bruno Ordozgoiti[1], **Sandra Gómez Canaval**[1], **Alberto Mozo**[1]

Department of Computer Systems
University College of Computer Science,
Universidad Politécnica de Madrid, Crta. de Valencia km. 7 - 28031 Madrid, Spain
`bruno.ordozgoiti,sgomez@etsisi.upm.es, a.mozo@upm.es`

Abstract. High dimensional data sets pose important challenges such as the curse of dimensionality and increased computational costs. Dimensionality reduction is therefore a crucial step for most data mining applications. Feature selection techniques allow us to achieve said reduction. However, it is nowadays common to deal with huge data sets, and most existing feature selection algorithms are designed to function in a centralized fashion, which makes them non scalable. Moreover, some of them require the selection process to be validated according to some target, which constrains their applicability to the supervised learning setting. In this paper we propose as novelty a parallel, scalable, exact implementation of an existing centralized, unsupervised feature selection algorithm on Spark, an efficient big data framework for large-scale distributed computation that outperforms MapReduce when applied to multi-pass algorithms. We validate the efficiency of the implementation using 1GB of real Internet traffic captured at a medium-sized ISP.

Keywords: feature selection, unsupervised learning, data mining

1 Introduction

Over the last few years, machine learning has proved to be a useful discipline for many applications such as automated classification, forecasting and anomaly detection. However, the data are often high dimensional, which poses certain challenges. First, this entails what is known as the curse of dimensionality. The amount of data required to train a system increases exponentially in the number of dimensions, and the sparsity of the data grows dramatically in settings in which distance measures are used. In addition, some of the present features might be redundant or noisy, which might hamper the accuracy of the model.

There exist techniques to fight these problems, some of which being based on Principal Components Analysis or the Singular Value Decomposition. These transform the data into a new, lower dimensional subspace which captures the majority of the information of the original records.

[⋆] The research leading to these results has been developed within the ONTIC project, which has received funding from the European Union's Seventh Framework Programme (FP7/2007-2011) under grant agreement n. 619633.

Another family of dimensionality reduction algorithms is the one known as feature selection. These techniques eliminate redundant or uninformative features in favor of the most relevant ones, which results in a lower dimensional and hence more manageable data set. Feature selection algorithms can be classified into filter methods, which are model-agnostic; wrapper methods, based on the performance of the model with the chosen features; and embedded methods, which are built into the training algorithm of the specific model being used.

The algorithm presented in this paper, which we will refer to as PPCSS (Parallel Partition-based Column Subset Selection) can be thought of as a filter method. It is based on the centralized algorithm described in [1], described in section 5. Our method presents several advantages over other proposals: (1) it does not rely on arbitrary partitions or concise representations to distribute the data set, which would impact the final result, (2) it is unsupervised and can be used in conjunction with any learning algorithm and (3) it provides a provable theoretical performance metric.

The rest of the paper is structured as follows: Section 2 describes the related work on parallel feature selection algorithms and Section 3 details the proposed contribution. In Section 4, a summary of big data distributed computing frameworks is shown. Sections 5 and 6 present the centralized version of the algorithm and our parallel version proposal respectively. In Section 7 we present experimental results and we conclude in Section 8.

2 Related Work

In recent years, some parallel feature selection approaches have been proposed to deal with the enormous size of today's data sets. In [3] a framework for choosing a column subset that is close to the best rank-k approximation is proposed. It relies on partitioning the data matrix $A \in \mathbb{R}^{m \times n}$ into submatrices of at least $k = \text{rank}(A)$ columns, which limits scalability to the ratio n/k. In [4] a parallel method based on mutual information is described. The data, however, are assumed to be fully discrete, and the method works separately on arbitrary subsets of the data set, making the final result a heuristic approximation. In [5] a MapReduce implementation of the minimum redundancy maximum relevance algorithm is described. This method, though, is only suitable for supervised learning models. In [7] a parallel feature selection algorithm is proposed for logistic regression. This method too is only suitable for the supervised setting, plus it requires to retrain the model at each iteration. In [6] a distributed parallel feature selection is proposed for microarray data classification. This method suffers from the same drawbacks as the previous one, and it imposes a completely random choice to be part of the final feature subset. In [8] a parallel algorithm for unsupervised feature selection is described. Their technique, however, relies on the distribution of the feature covariance matrix in full-column partitions. If the number of features is not very large, the algorithm cannot benefit from parallelization. If it is, workers might not be able to handle the partitions. Other data structures present in the algorithm present the same issue. In [9] a MapReduce implementation of a greedy method for solving the column subset selection problem is proposed. This solution relies on a random projection to represent the data in the different computing nodes. Finally, in [10] a MapReduce implementation of a method based on rough set theory is described.

3 Contribution

In this paper we describe a parallel, scalable version of the algorithm described in [1], which we have designed to be implemented on Spark. This platform is much more efficient than the widely used MapReduce paradigm when the data need to be accessed accross iterations.

Unlike other algorithms, the method we propose does not rely on concise approximations to distribute the data set or on heuristic approaches executed on arbitrary submatrices. Therefore, except for the variations inherent to its probabilistic nature, the proposed method yields the same results as the centralized version of the algorithm would if run on the same data. Moreover, the authors of the original algorithm provide a proven bound on the residual norms of the difference between the data matrix and its projection onto the span of the selected columns. To the best of our knowledge, none of the existing truly scalable versions of feature selection algorithms provide a theoretical performance metric of that nature.

The proposed algorithm functions by performing a judicious random column sampling and then computing the residual norm. In order to achieve a desirable result with a probability close to 1 it is necessary to build various samples and choose the one that provides the best residual norm. When dealing with very large data sets, the computations involved become prohibitively expensive and entail further difficulties. The norms to be computed consist of very large scale summations, which can cause numerical errors and overflow to arise. In this work we deal with some of these issues and provide guidelines for further control.

4 Parallel Computing Platforms

The increasing availability of enormous data sets has contributed to a significant surge of the popularity of large-scale data processing platforms during the last few years. One of the first frameworks to become widely popular was MapReduce [15]. The MapReduce framework is a programming model for processing large amounts of data based on two steps, *map* and *reduce*. The *map* step takes the input data and generates a set of key-value pairs according to a user-chosen criterion. The *reduce* step takes all the elements with the same key and performs an aggregation function, also implemented by the user. Each of the *map* and *reduce* operations can be executed independently on a separate machine. Therefore, together with the distributed, fault-tolerant storage capabilities of Hadoop [16], MapReduce allows for extreme scalability, and enjoys widespread use in research and industrial applications. Other systems that expose a programming API hiding distribution details are Ciel [17] and Dryad [18], the latter of which has been discontinued.

4.1 Spark

The algorithm described in this paper has been implemented on Apache Spark, a novel parallel computation platform designed to provide distribution and fault tolerance efficiently. The key idea distinguishing Spark is its in-memory computation capabilities, allowing data chunks to be reliably cached in memory across iterations. This results in significant performance improvements.

Spark revolves around an abstraction called Resilient Distributed Dataset (RDD) [12]. RDDs enable efficient data reuse by caching in main memory, which is extremely beneficial for certain types of algorithms. To efficiently enable resiliency, each RDD stores its *lineage*, i.e. the sequence of transformations necessary to build it. In fact, RDDs are loaded *lazily*, i.e. their computation is deferred until the data they contain are needed. Spark also permits different RDD *persistence* and *partitioning* strategies, which allow for substantial optimizations, and provides the *Broadcast* and *Accumulator* tools for efficient manipulation of shared variables.

5 Centralized Algorithm Description

The algorithm described in [1] provides a sub-optimal solution to the column subset selection problem (CSSP), which is formulated as follows.

Definition 1. *Given a matrix $A \in \mathbb{R}^{m\times n}$ and a positive integer k, pick k columns of A forming a matrix $C \in \mathbb{R}^{m\times k}$ such that the residual*

$$\|A - P_C A\|_\xi$$

is minimized over all possible $\binom{n}{k}$ choices for the matrix C. $P_C = CC^+$ denotes the projection onto the k-dimensional space spanned by the columns of C and $\xi = 2$ or F denotes the spectral norm or the Frobenius norm.

The challenge of the CSSP is to find a subset of columns that contains as much information of the original matrix as possible. All the possible subsets can be generated in $O(n^k)$, making it unfeasible to find an optimal solution even for relatively small choices of k when the data are represented in hundreds or thousands of dimensions.

The algorithm proposed by Boutsidis et al. in [1] works in two stages, a randomized and a deterministic one. In the randomized phase, a judicious random column sample is taken from the right singular vectors (V_k^T) of the data matrix, using probability distribution $\{p_i | i \in \{1, \ldots, n\}\}$ obtained from (1). This phase outputs a matrix $(V_k^T S_1 D_1) \in \mathbb{R}^{k\times c}$, where S_1 is a sampling matrix to keep only the chosen columns and D_1 scales these columns according to the factors previously described.

$$p_i = \frac{\|(V_k)_{(i)}\|_2^2}{2k} + \frac{\|(A)^{(i)}\|_2^2 - \|(AV_kV_k^T)^{(i)}\|_2^2}{2\big(\|A\|_F^2 - \|AV_kV_k^T\|_F^2\big)} \tag{1}$$

In the second phase, Algorithm 1 of [11] is run on matrix $V_k^T S_1 D_1$ to obtain a permutation revealing exactly the k columns to be kept.

In [1] it is proved with a probability of at least 0.7 this algorithm provides a matrix C (comprised of a column subset of A) such that

$$\|A - P_C A\|_2 \le O(k^{3/4} \log^{1/2}(k)(\rho - k)^{1/4})\|A - A_k\|_2$$

$$\|A - P_C A\|_F \leq O(k \log^{1/2} k)\|A - A_k\|_F$$

where A_k is the best rank-k approximation of A and ρ is the rank of A.

In order to attain these bounds with sufficient certainty it is necessary to retrieve numerous different column samples in the randomized phase. In [2], where this algorithm is used to apply PCA, 40 samples are drawn in order to achieve the desired result with a probability of almost 1. The computations required to project the original matrix onto the span of the chosen subsets and obtain the residual norm are computationally intense and may become extremely demanding and problematic when dealing when large datasets. The main goal of the work presented in this paper is to dramatically improve the efficiency of this phase by means of distributed computing.

6 Distributed Algorithm Description

The algorithm described in this paper has been implemented on Apache Spark, based on RDDs (refer to section 4). When a map operation is run on an RDD, the function is applied in parallel to all of its elements.

6.1 Notation

- By RDD(S) for some set S we denote an RDD whose elements are the elements of set S.
- By $A \odot B$ we denote the element-wise product of matrices A and B.
- $A_{i:}$ is the i-th row of matrix A.
- Lowercase bold letters (e.g. $\boldsymbol{d}$) represent vectors.

6.2 The PPCSS Algorithm

The PPCSS algorithm (see figure 1) is implemented in two separate modules, representing the column sampling and the residual norm computation respectively. The first step of the column sampling phase is to load the data set into an RDD -*rows*- whose elements are the indexed rows of the data matrix A, and to compute its top-k right singular vectors V_k. To compute the set of sampling probabilities $\{p_i\}$ we need to use formula 1, whose factors are obtained in parallel calling mapper *mapSamplingPartitions* then reducer *reduceSamplingPartitions* (figure 2) on RDD *rows*. This will produce $\boldsymbol{a}$, containing the squared ℓ^2-norms of the column vectors of A (note that $\boldsymbol{a}$ equals the diagonal of the gramian matrix of A, $A^T A$), and $\boldsymbol{b}$, containing the squared ℓ^2-norms of the column vectors of $AV_k V_k^T$. The corresponding squared Frobenius norms can be obtained adding up the elements of each of these vectors[1]. The sampling probabilities can now be easily computed on the master.

Afterwards, we must take 40 random samples of $c = O(k \log k)$ column indices following the obtained probability distribution, forming matrices S_i and

[1] Note that we are building a probability vector. Therefore, these elements can be rescaled to avoid overflow if the data set is truly huge. With respect to the numerical error of the sum, the pairwise nature of reduce operations constrains it to $O(\log n)$ [13]

D_i for every $i \in [1, 40]$ as described in [1], and then perform a rank-revealing QR factorization on each of the matrices $V_k^T S_i D_i \in \mathbb{R}^{k \times c}$. To obtain exactly k columns, we will keep the first k elements of the resulting permutation. For this task, we use LAPACK's DGEQP3 routine[2].

The next phase of the algorithm consists in finding the column choice that minimizes the residual norm $\|A - P_C A\|_F$. We compute the pseudo-inverse of each matrix C_i, C_i^+ using the singular value decomposition, and the product $C_i^+ A \in \mathbb{R}^{k \times n}$ using a simple MapReduce operation for large matrix multiplication that we will not describe for the sake of space. It then remains to compute $A - C C_i^+ A$, done simultaneously for all matrices C_i using a single MapReduce operation. First, we set up an RDD -*allMats* - that can be described as follows: the jth element of *allMats* is a list containing the jth row of each matrix C_i as well as the jth row of matrix A. Then, we call *mapNormAddends* (figure 3) on this RDD. Each map operation will compute one row of $C_i C_i^+ A$ for each matrix C_i (the matrices $C_i^+ A$ are transmitted to each worker via Broadcast), their differences with the corresponding row of A and the contribution to the residual norm of this row. This will result in a vector $\boldsymbol{d}$ containing the addends of the residual norm for each column choice. Reducer *reduceNormAddends* (figure 3) can then add up all of the vectors produced by the mappers to obtain the final residual norms.

The fact that the norms are computed simultaneously for all matrices C_i allows us to control overflow by subtracting the minimum vector component from all the elements of said vector in the reduce step. This way we can significantly decrease the chance of overflow without incurring additional numerical error. It should also be noted that since the reduce operations are performed in a pairwise fashion, the expected numerical error is $O(\log n)$, where n is the number of addends [13]. A slower alternative using Kahan's algorithm and Accumulators could be implemented to achieve $O(1)$ error [14].

6.3 The Complexity of PPCSS

The complexity of the algorithm on which PPCSS is based is $O(mn \min(m, n))$ for the computation of the top k right singular vectors, $O(k^3 \log k)$ for the deterministic phase and $O(mnk)$ for the final norm computation. Since we are considering the case that $m \gg n \gg k$, the running time of the algorithm is dominated by the computation of V_k and the norms, which are the operations that we parallelize. In general, the complexity of these calculations will scale linearly with the inverse of the number of nodes in the cluster.

With respect to network usage, the algorithm requires to move $O(mn + n^2 + kn)$ between the master and the workers: the data set, matrix $V_k V_k^T \in \mathbb{R}^{n \times n}$ and set $P = \{C_i^+ \in \mathbb{R}^{k \times n} | i \in [1, 40]\}$. Therefore, a large number of features could severely impact the total running time. However, we have observed that in certain domains, e.g. network traffic management, this number tends to be below a few hundred, which our algorithm seems to be able to handle adequately.

[2] We consider the case where $n \ll m$. Matrix $V_k^T S_i D_i$ is therefore of a manageable size and this operation can thus be performed locally.

```
Column sampling phase
Input: data matrix $A \in \mathbb{R}^{m\times n}$, number of columns $k$

 1. $rows$ := RDD($\{A_{i:} \mid i \in [1, m]\}$)
 2. Compute the top $k$ right singular vectors of $A$, $V_k$.
 3. ($colNorms$, $prodNorms$) :=
    $rows$.mapSamplingPartitions(broadcast($V_k V_k^T$)).reduceSamplingPartitions()
 4. Compute the sampling probabilities $\{p_i\}$ for $1 \le i \le n$
 5. Obtain 40 random samples of $c = O(k \log k)$ columns, forming matrices $S_i$ and $D_i$
    for all $i$ in $\{1, \ldots, 40\}$.
 6. for $1 \le i \le 40$
      – Run LAPACK's DGEQP3 on $V_k^T S_i D_i$ and keep the $k$ first columns, obtaining
        the column indices to form matrix $C_i$.
 7. output $\{C_i \mid i \in [1, 40]\}$

Norm minimization phase
Input: $\{C_i \mid 1 \le i \le 40\}$

 1. $P := \emptyset$
 2. for $1 \le i \le 40$
      – $P := P \cup \{C_i^+ A\}$
 3. $allMats$ := RDD($\big\{F_j = \{(C_i)_{j:} \mid i \in [1, 40]\} \cup \{A_{j:}\} \mid j \in [1, m]\big\}$)
 4. $\boldsymbol{d}$ := $allMats$.mapNormAddends(broadcast($P$)).reduceNormAddends()
 5. output $\underset{C}{\operatorname{argmin}} \|A - P_C A\|_F$, i.e. $C_i$ with $i = \underset{i}{\operatorname{argmin}}\, \boldsymbol{d_i}$
```

Fig. 1. PPCSS algorithm

```
mapSamplingPartitions
($A_{i:}$,broadcast($V_k V_k^T$))

 1. $\boldsymbol{a} := A_{i:} \odot A_{i:}$
 2. $\boldsymbol{b} := (A_{i:} V_k V_k^T) \odot (A_{i:} V_k V_k^T)$

reduceSamplingPartitions
($(\boldsymbol{a}_i, \boldsymbol{b}_i), (\boldsymbol{a}_j, \boldsymbol{b}_j)$)

 1. $\boldsymbol{a}_{out} := \boldsymbol{a}_i + \boldsymbol{a}_j$
 2. $\boldsymbol{b}_{out} := \boldsymbol{b}_i + \boldsymbol{b}_j$
 3. output $(\boldsymbol{a}_{out}, \boldsymbol{b}_{out})$
```

Fig. 2. MapReduce operations for the addends of the sampling probability distribution

```
mapNormAddends
($F_j$,broadcast($P$))

 1. $D_i := (C_i)_{j:} C_i^+ A - A_{j:} \forall (C_i)_{j:} \in$
    $F_j \backslash \{A_{j:}\}$
 2. output vector
    $\boldsymbol{d} = (D_1 \boldsymbol{u}, \ldots, D_{40} \boldsymbol{u})$, where $\boldsymbol{u}$ is a
    vector of ones

reduceNormAddends ($\boldsymbol{d_i}$,$\boldsymbol{d_j}$)

 1. output $\boldsymbol{d_i} + \boldsymbol{d_j}$
```

Fig. 3. MapReduce operations for computing the final residual norms for each column choice

7 Preliminary Experimental Results

We have tested the algorithm on a homemade cluster of 10 nodes equipped with a Quad-core processor and 4Gb of RAM each. The dataset used was a 1GB real Internet traffic capture from a Spanish medium-sized ISP consisting of 2,500,000 flow instances and 105 features.

The purpose of the first of the experiments was to test the scalability of the algorithm with respect to the number of workers. We launched the algorithm on the full 2,500,000 $\times$ 105 matrix using 1, 2, 5, and all 10 nodes. This set of executions shows that our algorithm benefits greatly from parallel computation (figure 4). As expected, the speedup decreases as the number of workers grows, and the graph suggests that a point of asymptotic lack of improvement will be reached. However, this point is expected to shift to the right as the size of the input data set grows.

In the second set of experiments we executed the algorithm on datasets of different sizes. We trimmed the original data to obtain sets of 50,000, 100,000, 200,000, 400,000, 800,000 and 1.6 million instances, and tested the algorithm on these subsets. As seen on figure 5, our method yields the best results when the data set is truly huge. For instance, the difference in execution times with 200,000 and 400,000 instances is not that large. This is to be expected, given the fact that the total time is a function of Spark's cluster management tasks, network usage and I/O operations and the complexity of the algorithm. When the size of the data set is small, cluster management time (which is more or less constant for a fixed-size cluster) and the overhead of I/O operations are significant with respect to the total time. Since the running time of the algorithm increases linearly with m (the number of flows), and both n and k (the dimensionality of the data and the number of selected features respectively) are fixed in this experiment, it is evident that data seem to be handled more efficiently by Spark when the input is large.

The third experiment was carried out to determine the impact of the parameter k (i.e. the number of selected features) on the running time. As explained in section 6.3, the algorithm depends cubically on k. Since in general $k \ll n$, this will not have a significant effect on the performance of the algorithm provided that n is not extremely large. This fact is shown by the results of the experiments reflected in figure 6, in which the value of k does not dramatically impact the total time. Nevertheless, in settings where the dimensionality is extremely high, the cubic dependence on k could be an important drawback.

8 Conclusions and Future Work

We have presented a truly scalable feature selection algorithm able to cope with enormous data sets. Our proposal is a parallel, scalable, exact implementation of an existing centralized, unsupervised feature selection algorithm on Spark. As opposed to other algorithms of this kind already present in the literature, our technique does not rely on concise representations of the original data or arbitrary partitioning that might mask certain properties and relevant characteristics. We have carried out some preliminary tests on a modest cluster and

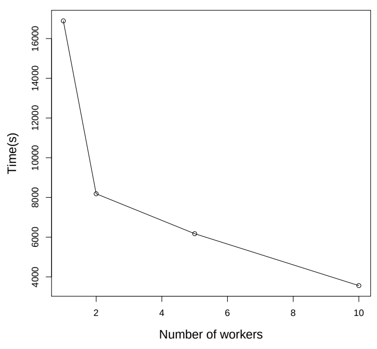

Fig. 4. Experiments with different numbers of workers

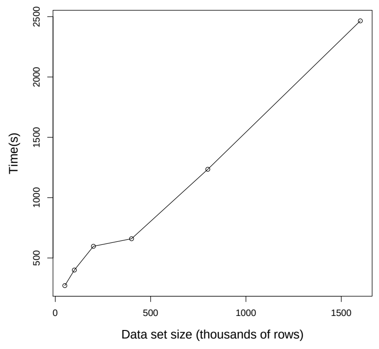

Fig. 5. Experiments with data sets of different sizes

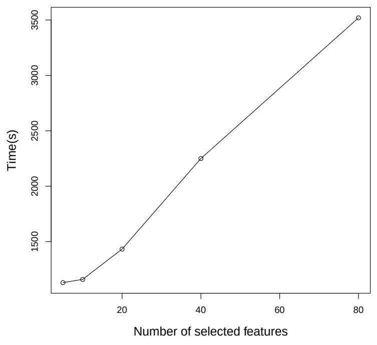

Fig. 6. Experiments with different choices of k

the results with regards to scalability and execution times are very promising. However, the performance of the algorithm is sensitive to the number of features, which impose penalties on both running time and network usage. Hence, extremely high-dimensional data sets pose significant performance concerns. It is therefore an open challenge to address scalability in that sense. The experiments have shown, nevertheless, that the algorithm can comfortably handle big data sets of over 100 features, which are frequent in domains such as network traffic management.

We plan to perform additional experiments on larger data sets, as well as to evaluate the possibilities of large-scale feature selection on certain domains that could immensely benefit from a truly scalable and efficient algorithm of this kind. We also intend to run implementations of other existing proposals in order to assess the performance of our method compared to those. In addition, we will continue our work on parallel unsupervised feature selection to address the problem of scalability with respect the number of dimensions.

References

1. Boutsidis, C., Mahoney, M. W., & Drineas, P. (2009, January). An improved approximation algorithm for the column subset selection problem. In Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms (pp. 968-977). Society for Industrial and Applied Mathematics.
2. Boutsidis, Christos, Michael W. Mahoney, and Petros Drineas. "Unsupervised feature selection for principal components analysis." Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2008.
3. Pi, Yifan, et al. "A scalable approach to column-based low-rank matrix approximation." Proceedings of the Twenty-Third international joint conference on Artificial Intelligence. AAAI Press, 2013.
4. Sun, Zhanquan, and Zhao Li. "Data intensive parallel feature selection method study." Neural Networks (IJCNN), 2014 International Joint Conference on. IEEE, 2014.
5. Reggiani, Claudio, et al. "Minimum Redundancy Maximum Relevance: MapReduce implementation using Apache Hadoop." BENELEARN 2014 (2014): 2.
6. Boln-Canedo, V., N. Snchez-Maroo, and A. Alonso-Betanzos. "Distributed feature selection: An application to microarray data classification." Applied Soft Computing 30 (2015): 136-150.
7. Singh, Sameer, et al. "Parallel Large Scale Feature Selection for Logistic Regression." SDM. 2009.
8. Zhao, Zheng, et al. "Massively parallel feature selection: an approach based on variance preservation." Machine learning 92.1 (2013): 195-220.
9. Farahat, Ahmed K., et al. "Distributed column subset selection on MapReduce." Data Mining (ICDM), 2013 IEEE 13th International Conference on. IEEE, 2013.
10. He, Qing, et al. "Parallel feature selection using positive approximation based on MapReduce." Fuzzy Systems and Knowledge Discovery (FSKD), 2014 11th International Conference on. IEEE, 2014.
11. Pan, C-T. "On the existence and computation of rank-revealing LU factorizations." Linear Algebra and its Applications 316.1 (2000): 199-222.
12. Zaharia, Matei, et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing." Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. USENIX Association, 2012.
13. Higham, N. J. (1993). The accuracy of floating point summation. SIAM Journal on Scientific Computing, 14(4), 783-799.
14. W. Kahan. 1965. Pracniques: further remarks on reducing truncation errors. Commun. ACM 8, 1 (January 1965)
15. Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." Communications of the ACM 51.1 (2008): 107-113.
16. Shvachko, Konstantin, et al. "The hadoop distributed file system." Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on. IEEE, 2010.
17. Murray, Derek Gordon, et al. "CIEL: A Universal Execution Engine for Distributed Data-Flow Computing." NSDI. Vol. 11. 2011.
18. Isard, Michael, et al. "Dryad: distributed data-parallel programs from sequential building blocks." ACM SIGOPS Operating Systems Review. Vol. 41. No. 3. ACM, 2007.