**Integrating Pig with Harp to Support Iterative Applications**

**with Fast Cache and Self-Defined Communication**

Tak-Lon Wu, Abhilash Koppula, Judy Qiu

School of Informatics and Computing   
Indiana University, Bloomington, IN, USA   
{taklwu, akoppula, xqiu}@indiana.edu

*Abstract*—Using high level language to solve big data problems saves the implementation time for most popular algorithms and repeatable experiments. Existing solutions such as Pig and Hive have provided important basic functions and extendable interfaces as UDFs, allowing for quickly writing scripts submitted to the default binding Hadoop MapReduce runtime. However, the current strategy of using high level languages to support iterative applications with Hadoop MapReduce relies on an external wrapper script, which causes a huge performance loss. Our idea is to reduce the extra job startup overheads by integrating Pig with Harp, providing fast data cache and self-defined communication patterns among iterations. The results show that we can improve to at least 2 and at most 5 times faster than the default approach.

Keywords: Pig, Iterative Algorithms, Big Data, Language, MapReduce

# Introduction

MapReduce programming model has been widely adopted by many fields of research in computer science and scientific computing. It provides gratifying features like pleasingly parallel computation, horizontal scalability, and high performance on commodity clusters and clouds. Hadoop [[1](#_ENREF_1)] is the Java-based open source project that provides the interfaces to implement the algorithms and applications. But in order to achieve the best performance, it requires advanced knowledge and solid programming skills in Java. Beyond MapReduce, some have built high level languages such as Pig [[2](#_ENREF_2)], Hive [[3](#_ENREF_3)] and Shark [[4](#_ENREF_4)] to support an expressive directed acyclic graph (DAG) computing model that contracts and runs jobs on top of MapReduce. These languages hide the complexity of MapReduce programming, instead providing functional operators and record-based data type abstraction to logically enable users to handle different types of ETL data integration and iterative applications.

So far, these high level languages systems have been used by many commercial companies including Yahoo!, Facebook, and LinkedIn, and they have proven to be efficient enough to handle daily ETL operations and ad hoc queries in many big data problems; more than half of the daily submitted MapReduce jobs are either generated as Pig or Hive scripts in these companies. However, in the case of supporting iterative applications, it is nontrivial. Most of these solutions claim to be applicable and require developers to write user defined functions (UDFs) for computing the core algorithms and wrapping the main language script inside an external control-flow script to map the iteration data from disk to memory. As a result the performance is limited due to submitting multiple rounds of MapReduce jobs with extra job startup overhead. In addition, most of these language systems are built on top of Hadoop using disk caches and disk I/O, meaning the data communication overhead is too high and soon becomes undesirable due to the overall performance loss.

In this paper, we use Pig as an example and introduce Pig integrated with Harp, a fast cache MPI-like collective communication plugin with Hadoop, to investigate possibilities to simplify the hierarchy using one main high level language and improve the performance by looking into fast data cache and better communication patterns between iterations. For scientific applications we mainly propose to use these high level languages simply as functional libraries in R [[5](#_ENREF_5)] which provides a rich set of sophisticated data analysis algorithms; therefore users only pass the parameters to those existing methods, which execute their applications automatically in distributed environments. For the developers that implement those complicated algorithms, we suggest they write a single Pig script with UDFs combined with customized data cache and communication patterns in order to remove unnecessary job restart overhead and achieve the best performance.

The rest of this paper is organized as follows. Section II introduces the general background of Harp and Pig. Section III explains our vision of system design and improvement by integrating Pig on Harp. Section IV presents our targeted use cases for scientific applications. Section V shows aspects of results based on the lines of code, performance, and coding difficulty. Section VI compares our approach with the related solutions. And last but not least, Section VII draws the conclusion.

# Background

Harp [[6](#_ENREF_6)] is a Hadoop plugin that enables the possibility of loop awareness, fast in-memory caching and self-contained communication patterns for iterative computation. It replaces the default mapper interface with a long-running mapper that can support multi-thread/multi-processes computing and in-memory caching instead of Hadoop’s default multi-processes on split key-value pairs parallel computing. In addition, Harp provides MPI-like collective communication interfaces for developers to do self-defined network shuffling rather than shuffling with HDFS I/O. These worthwhile features enable our work to gain impressive performance improvement.

Pig [[2](#_ENREF_2)] is a high level platform, extensively designed for Hadoop MapReduce large-scale data analysis applications, and Pig Latin [[7](#_ENREF_7)] is the provided language that abstracts the complicated Java MapReduce programs into dataflow programs with simple notations. Internally, submitted Pig scripts are compiled into sequences of MapReduce jobs which run locally as single thread applications or remotely on an existing Hadoop MapReduce [[1](#_ENREF_1)] runtime. In other words, a Pig program is embarrassingly parallel and easy to maintain. Pig Latin is a procedural language compared to traditional SQL for RDBMS. Figure 1 shows an example of WordCount written in Pig Latin.

Pig is a dataflow language, each line having only a single data transformation which could be nested. The WordCount program includes a total of 7 lines of code, and the syntax is straightforward and easy to understand. In general, data is loaded as records, and each field in a record is defined according to Pig’s default data types: bag, tuple, and field. The length of a record is flexible since tuple can contain a different number of fields in the same column. Other than the syntax shown in this paper, Pig Latin has more operations and syntax patterns that can be used for various data transformations. Currently, Pig misses out on optimized storage structures like indices and column groups, which may not be suitable for all applications.

|  |
| --- |
| 1 input = LOAD 'input.txt' AS  (line:chararray);  2 words = FOREACH input GENERATE  FLATTEN(TOKENIZE(line)) AS word;  3 filWords = FILTER words BY word MATCHES '\\w+';  4 wdGroups = GROUP filWords BY word;  5 wdCount = FOREACH wdGroups GENERATE group AS  word, COUNT(filWords) AS count;  6 ordWdCnt = ORDER wdCount BY count DESC;  7 STORE ordWdCnt INTO 'result'; |
| 1. WordCount written in Pig Latin [[8](#_ENREF_8)] |
| pigDataFlow |
| 1. Pig High Level Dataflow |

Whenever a user submits their Pig Latin scripts in batch mode or opens a Pig console and enters line-by-line data transformation commands in interactive mode, the overall execution flows are handled by a default compiler. This compiler translates the entered Pig Latin scripts into machine understandable operators as top-down Abstract Syntax Trees (AST) in different stages. It then visits the last compiled AST from the MapReduce Plan compiler and constructs MapReduce jobs in sequence. Figure 2 shows the dataflow and lists all major components. Similar to any programming language, Pig Latin does the syntax checking by parsing the user submitted script into a parser written in ANTLR (ANother Tool for Language Recognition) [[9](#_ENREF_9)]. The Pig main driver program converts each MapReduce operator from Map-Reduce Operator Plan *MROperPlan* objects into Hadoop JobControl objects with detail descriptions, input/output linkages, and other parameters which are then passed along to each worker node with the general system configuration in xml format. These translations generate Java jar files as MapReduce jobs respectively that contain the Pig default Map and Reduce Class, including the user defined functions if any. The package jar files are submitted to Hadoop Job Manager in sequences, and job progress is monitored ‘til finished.

|  |
| --- |
| PigYarnKmeans |
| 1. Iterative applications with Pig on Hadoop |

# Pig in Supporting Iterative Applications

Pig is good enough for general ETL applications, however, it does badly in supporting iterative applications. When writing Pig program for iterative applications, the control flow should be similar to what is shown in Figure 3. The need of an external wrapper script is necessary, because Pig syntax does not provide control flow statement. Therefore, a submitted program runs in several rounds of MapReduce jobs with extra overhead of unnecessary job startup and cleanup time which hugely decreases the overall performance. Additionally, inputs of iterative applications are normally unchanged and cacheable in every iteration, whereas Pig is DAG framework that can’t cache those inputs in memory and reuse them efficiently.

Due to the obvious fact that Pig lacks the features of loop-awareness and in-memory caching, our approach is to investigate and apply possible extensions to Pig based on the DAG computation model. There are several iterative MapReduce frameworks as candidates; they are Twister [[10](#_ENREF_10)], Spark [[11](#_ENREF_11)], HaLoop [[12](#_ENREF_12)] and Harp. We choose Harp as our initial approach as it is a simple MapReduce extension with our required features. With the Harp integration, we mainly replace the Hadoop Mapper interface with Harp’s MapCollective Mapper. Subsequently, iterative applications implemented in Pig can cache reusable data and replace the default GROUP BY operation with Harp’s collective communication interface. We compare the original reduce stages against Harp’s communication in Section V. Figure 4 shows overall dataflow that can be applied on any iterative applications.

|  |
| --- |
|  |
| 1. Iterative application with Pig on Harp |
| 1 raw = LOAD $hdfsInputDir using  PigKmeans('$centroids',  '$numOfCentroids') AS (datapoints);  2 dptsBag = FOREACH raw GENERATE  FLATTEN(datapoints) as dptInStr;  3 dpts = FOREACH dptsBag GENERATE  STRSPLIT(dptInStr, ',', 5) AS  splitedDP;  4 grouped = GROUP dpts BY splitedDP.$0;  5 newCens = FOREACH grouped GENERATE  CalculateNewCentroids($1);  6 STORE newCens INTO 'output'; |
| 1. Pig K-means on Hadoop for a single iteration |

# Use Case

The proposal applications are K-means clustering and PageRank, both popular iterative algorithms for scientific computation, but it could be extended to any algorithm as long as user defined functions are correctly implemented, e.g. naïve bayes classifier. We compare two versions of implementation for these two algorithms, one implemented on Hadoop 2.2.0 and another built on Harp 0.1.0, both scheduled on YARN resource manager.

## Pig K-means on Hadoop

Here, Pig K-means on Hadoop applies this logic and the implementation is split into three pieces; a python control flow script, a Pig data transform script for a single iteration, and two K-means user defined functions written with Pig-provided Java interface. During every iteration, our customized Loader in each Mapper loads the centroids into memory from distributed cache on disk before computing the Euclidean distances for data points. It then uses the Pig standard GROUP operation for collecting partial centroid vectors from mappers. Afterwards, it takes the average of all partitions and emits a final centroids file back to HDFS for the next iteration. Figure 5 shows a single iteration of K-means written in Pig Latin.

## Pig K-means on Harp

In case of Pig K-means running on Harp, the customized Loader in each Mapper firstly loads the initial centroids and data points one time from HDFS to memory as cache for all iterations. Then UDF computes the Euclidean distance and emits partial centroids locally. These partial centroids on each mapper are then exchanged by Harp’s communication layer. By default, our UDF uses AllReduce to synchronize among all partitions. The program reuses the same set of mapper processes ‘til break conditions have been met.

The script in Figure 6 illustrates a similar idea of using R. Users only consider the parameters provided to the existing interface, such as number of mappers, total amount of iterations, communication patterns used for global data synchronization, etc. Note that developers must have a deep understanding of using Hadoop and Harp as well as distributed system knowledge in order to achieve the best performance.

|  |
| --- |
| 1 centds = LOAD $hdfsInputDir using  HarpKmeans('$initCentroidOnHDFS',  '$numOfCentroids', '$numOfMappers',  '$iteration', '$jobID', '$Comm') as  (result);  2 STORE centroids INTO '$output'; |
| 1. Pig K-means on Harp |
| 1 raw = LOAD '$InputDir' USING  CmLoader('$noOfURLs','$itrs') as  (source,pagerank, out:bag{});  2 prePgRank = FOREACH raw GENERATE FLATTEN(out)  as source, pagerank/SIZE(out) as  pagerank;  3 newPgRank = FOREACH (COGROUP raw by source,  prePgRank by source OUTER) GENERATE  group as source, (1-$dpFactor) +  $dpFactor\*(SUM(prePgRank.pagerank)  IS NULL ?0:SUM(prePgRank.pagerank))  as pagerank, FLATTEN(raw.out)  as out;  4 STORE newPgRank INTO '$outputFile'; |
| 1. Pig PageRank on Hadoop for a single iteration |

## Pig PageRank on Hadoop

For PageRank implemented in Pig, we use less UDF functions and utilize the Pig operators and built-in functions for page rank computation to see how Pig performs. As shown in Figure 7, a Pig script for a single iteration of PageRank algorithm is created and invoked by a Java wrapper iteratively. Steps in this script involve: a) Load the given input file using custom loader into variable raw; b) Extract the outgoing URLs and emit the outgoing URL and partial page rank from source URL; c) CO-GROUP above two aliases to calculate new page rank and store it in an alias called newPgRank; d) Store this new page rank into a HDFS temp file which will be the input file for our next iteration. One drawback of this program is the default Pig runtime optimizer creates extra mappers for step d) when it calls the raw variable for the CO-GROUP operators, where it uses extra computing and memory resources.

## Pig PageRank on Harp

For PageRank implemented on Harp, we create a new data loader and write UDFs for computing the access probabilities for each web page. For initial iteration, data is loaded in a graph data structure where vertices are partitioned across all worker nodes. Each vertex has all its in-edges information by calling regroupEdges communication, and amount of out-edges is sent to all vertices by calling an AllMsgToAllVtx operation. This vertex and edge information is cached in memory for all iterations. Finally, whenever the page rank value of each vertex on a worker node is changed during an iteration, they are distributed by an AllGather communication ‘til the program meets break conditions, e.g. the end of iterations. The script shown in Figure 8 is similar to the approach of Pig K-Means on Harp.

|  |
| --- |
| 1 pagerank = LOAD '$InputDir' using  HarpPageRank('$totalUrls',  '$numMaps', '$itrs', '$jobID')  as (result);  2 STORE pagerank INTO '$output'; |
| 1. Pig PageRank on Harp |

|  |
| --- |
|  |
| 1. K-means Clustering Performance Comparison across different platforms |

# Results

We have investigated the lines of code in detailed implementations using Pig against other platforms. Also we have run a standard computation comparison for the same algorithm to see the performance difference. We construct our experiments in vertical and horizontal scale; firstly, we keep the same ratio between the amount of data points and amount of centroids, and we try to see the data loading, cache access, and computation overhead within the same environment. Meanwhile, we increase the computing resources in parallel by adding more mappers to each case in order to see the parallelism and communication overhead. Results shown in Figure 9 are obtained from our local cluster Madrid with Hadoop 2.2.0 and Pig 0.12.0. The specification and configuration are described below.

**Madrid:** An 8-nodes cluster with an extra headnode; each worker has 4 AMD Opteron 8356 at 2.30GHz with 4 cores, totaling 16 cores per node, installed with 16GB node memory and 1Gbps Ethernet network connection. It runs Red Hat Enterprise Linux Server release 6.5.

**Hadoop 2.2.0**: We run all the master services, such as resource manager, namenode, application master, etc., on the headnode. Each worker starts with node manager and datanode service, and any job can obtain up to 13GB of memory per node. By default each process spawns 1GB memory. For Harp, as its multithread computing model, we give the master process on each worker a total of 13GB memory.

**Pig 0.12.0:** We use the latest stable version released on Oct 13th, 2013 for general Pig applications. In addition we embedded Harp’s MapCollective Mapper into Pig and made the customized version run on top of Harp.

For K-means, we have set up three major batches of performance tests: a) 100 million data points against 500 centroids; b) 10 million data points against 5000 centroids; c) 1 million data points against 50k centroids. All of these are executed with different mappers and partition sizes such as 24, 48 and 96 on Madrid cluster. For PageRank, we perform a strong scaling test on a dataset with 2 million vertices, and it is executed with mappers and partition sizes of 8, 16 and 32.

## Coding Style

Table I has shown the lines of code for K-means application implemented on Pig and other platforms. In general, applications written in Pig require less code as it does not include the control flow statements. By contrast, the native Java MapReduce implementation requires more lines to define the variables and data transformation functions. But in some sense these data transformations are exactly the same code as Pig’s UDFs when implemented. In our case of K-means clustering, Pig K-means on Hadoop is implemented as a MPMD model which must include a wrapper written in the support language, e.g. Python or other supported language. For Pig K-means in Harp, the amount of code is almost the same as Harp K-means; the UDFs contain the customized data loading, computation and user-defined communication. This is similar to PageRank shown in Table II, but in the case of Pig implementation on Hadoop, we have less code as we only rewrite a customized data loader.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Hadoop  K-means | Pig  K-means on Hadoop | Harp  K-means | Pig  K-means on Harp |
| Java | 780 | 345 | 730 | 700 |
| Pig | 0 | 10 | 0 | 3 |
| Wrapper | 0 | 40 | 0 | 0 |
| Total lines | 780 | 395 | 730 | 703 |

1. K-means implemented on Pig and others

|  |  |  |  |
| --- | --- | --- | --- |
|  | Pig  PageRank on Hadoop | Harp  PageRank | Pig  PageRank on Harp |
| Java | 50 | 536 | 497 |
| Pig | 5 | 0 | 3 |
| Wrapper | 70 | 0 | 0 |
| Total lines | 125 | 536 | 500 |

1. PageRank implemented on Pig and Harp

|  |  |  |
| --- | --- | --- |
|  |  |  |
| a. 1 million data points with 50000 centroids | b. 10 million data points with 5000 centroids | c. 100 million data points with 500 centroids |
| 1. Performance details of Harp K-means and Pig K-means on Harp | | |

## Performance and Parallelism

For K-means comparison, seen in Figure 9, most of the Pig tests on Hadoop are slower than pure Hadoop cases and Harp cases. The performance difference is due to the implementation of using Pig, which generates larger intermediate data when emitting the partial centroids result as Databag instead of key-value pairs; the shuffling stage before reduce computation also takes longer. In addition, for the 1 million data points with 50K centroids, Hadoop and Pig Hadoop have a huge performance loss, as they have to reload the centroids for each iteration, and the computing centroids array grows beyond L2 & L3 cache and influences the mapper computation time. In sum, Harp performs the best as it is highly optimized. Meanwhile, Pig Harp tests closely achieve a similar performance.

We have also enlarged and compared the detail timing between Harp and Pig Harp, as shown in Figure 10. In most cases, the overhead of using Pig as an external wrapper is small, and we even have interesting findings that Pig on Harp with multi-processes computing model isn’t as bad as expected. Harp shows the advantages of its default multi-threads when we have the same L2 & L3 cache effect of in-memory cache for large centroids, e.g. 50000 centroids against 1 million data points; the pure mapper computation time is 2 times slower. Regardless, we did figure out the communication takes longer, as more processes generate more messages, and it lacks in-node global data reduction. But since the Harp communication module is highly optimized for object serialization and deserialization, the overhead in our tests is still acceptable.

|  |
| --- |
|  |
| 1. PageRank Performance Comparison |
|  |
| 1. Performance details of PageRank |

For the PageRank result shown in Figure 11, we have different aspects to see the native Pig implementation on Hadoop against Harp’s integrations. All the cases implemented on Harp run 10 times faster than the pure Pig implementation. This is because the loop-unawareness record-based computation of native Pig PageRank takes longer, data is reloaded every iteration, and compute processes are restarted every job. Additionally, as seen in Figure 12, the Pig with Harp integration performs close to the native Harp multi-threads and multi-processes implementation. Due to AllGather communication used in Harp for page rank values updated between iterations, the larger number of partitions is likely to increase the overall communication time; this is also similar to a native Pig implementation where reduce stages take longer for the case of 32 mappers.

## Coding Difficulies

Rewriting all the code from Java Hadoop MapReduce to Pig isn’t difficult by default, as Pig is designed to run data warehouse applications on top of Hadoop. The only problems we meet here are the logic of how the data is stored in Pig’s data format and how it could retrieve from these abstractions to computation. In our experience, even if this is a legacy code from other languages, as long as it is convertible to Java, the rewriting process will look for suitable Java libraries or rewrite the function in Java to replace the legacy libraries. For Pig with Harp integration, it might be a bit difficult for beginners as they need to understand the background of Hadoop, Pig, and Harp respectively.

# Related Work

DataFu [[13](#_ENREF_13)] is an Apache open source project that provides a collection of libraries for working with large-scale data in Hadoop and Pig, especially the subdivision of DataFu Pig, which provides a good set of UDFs for developers working in data mining and statistics. Our project shares these similarities, but we focus on the performance for iterative applications and research purposes using Apache open source stacks for data scientists.

Shark [[4](#_ENREF_4)] integrates Spark [[11](#_ENREF_11)] with Apache Hive [[3](#_ENREF_3)] to support the SQL community. They have implemented Hive K-means as an example shown on their project website. The use of Spark and RDDs [[14](#_ENREF_14)] provides the possibility to write iterative applications into one Scala script by first extracting the read-only data into RDDs, then computing the core iterative algorithms with Spark runtime. We believe that both Pig and Hive provide similar functionalities and can achieve the same goal although they use different languages and implementations. This makes it ideal for possible comparison in our future work. In addition, Apache Tez [[15](#_ENREF_15)] is an Apache incubator project that optimizes Pig/Hive’s script compiler to construct a complex DAG dataflow, originally compiled into multiple MapReduce jobs, into a single MapReduce job which boosts the performance and reuses the same set of mappers and reducers. But still, this approach does not support loop-aware computation and in-memory caches from the default Pig/Hive language syntax, and Pig community does not have any alpha release for version 0.12.x in this track.

HaLoop [[12](#_ENREF_12)] is another academic project that extends Hadoop to support loop-aware task scheduling and on-disk caching for iterative applications. Users of HaLoop need be less aware of the system but write and set fewer java classes for data passing between iterations, where inter-iteration data shuffling is optimized by the modified task scheduler to reuse the same physical node. Currently, HaLoop does not provide high level language support, but we believe that our integration with Harp could also be applied on HaLoop to achieve the same goals.

# Conclusion

We have successfully integrated Pig with Harp and have presented the idea of writing applications in Pig as a SPMD model instead of MPMD. Our results show that Pig with Harp can achieve nearly the same performance compared to pure Harp implementations, although the developer must be familiar with fundamental knowledge, architecture and programming skills. Moreover, we have shown the possibility of providing user-friendly libraries to users. One may harbor doubts such as, “Why don’t we use R or other libraries directly instead of integrating Pig to achieve similar goals?” Our vision is due to the fact that Hadoop or Apache open source stack are designed as the mainstream of handling big data problems. In order to achieve the best performance, we should tightly couple with these systems to avoid extra overhead such as conversion of data type and data structure, manual parallelism for applications and algorithms, etc. In addition, as Pig with Harp integration is compatible with existing Pig operators and functions, users can select the best UDFs run on different platforms and construct the ideal Pig pipeline for their daily data analysis.

We must admit that our current results haven’t considered and investigated the data access patterns and general data abstractions in the sense of using high level language to solve parallel computing problems. We may go further in this direction as future work.

##### Acknowledgment

We would like to thank our colleagues Xiaoming Gao and Bingjing Zhang of the Salsa team in Indiana University, who shared their results on the implementation for Hadoop and Harp K-means. Also, thanks to my research committee Dr. Geoffrey Fox, with whom we discussed the possibility, usability and computation mode of using high level language for parallel computing.

##### References

[1] "Apache Hadoop," <http://hadoop.apache.org/>, 2009.

[2] Alan F. Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan M. Narayanamurthy, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava, *Building a high-level dataflow system on top of Map-Reduce: the Pig experience.* Proc. VLDB Endow., 2009. **2**(2): p. 1414-1425.

[3] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy, *Hive: a warehousing solution over a map-reduce framework.* Proc. VLDB Endow., 2009. **2**(2): p. 1626-1629.

[4] Reynold S. Xin, Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica, *Shark: SQL and rich analytics at scale*, in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 2013, ACM: New York, New York, USA. p. 13-24.

[5] R Development Core Team, *R: A Language and Environment for Statistical Computing*. 2011: R Foundation for Statistical Computing.

[6] Bingjing Zhang. *Apache Harp Project*, 2014; Available from: <http://salsaproj.indiana.edu/harp/>.

[7] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins, *Pig latin: a not-so-foreign language for data processing*, in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 2008, ACM: Vancouver, Canada. p. 1099-1110.

[8] *Pig Programming Tools*, 2014; Available from: <http://en.wikipedia.org/wiki/Pig_(programming_tool)>.

[9] Terence Parr. [*http://www.antlr.org/*;](http://www.antlr.org/;) Available from: <http://www.antlr.org/>.

[10] J.Ekanayake, H.Li, B.Zhang, T.Gunarathne, S.Bae, J.Qiu, and G.Fox, *Twister: A Runtime for iterative MapReduce*, in *Proceedings of the First International Workshop on MapReduce and its Applications of ACM HPDC 2010 conference June 20-25, 2010*. 2010, ACM: Chicago, Illinois.

[11] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica, *Spark: Cluster Computing with Working Sets*, in *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '10)*. 2010: Boston.

[12] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst, *HaLoop: Efficient Iterative Data Processing on Large Clusters*, in *The 36th International Conference on Very Large Data Bases*. 2010, VLDB Endowment: Singapore.

[13] M. Hayes and S. Shah. *Hourglass: A library for incremental processing on Hadoop*. in *Big Data, 2013 IEEE International Conference on*. 2013.

[14] 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*. 2012, USENIX Association: San Jose, CA. p. 2-2.

[15] *Apache Tez*, 2014; Available from: <http://tez.incubator.apache.org/>.