Integrating Pig with Harp to Support Iterative Applications with Fast Cache and Customized Communication

Tak-Lon Wu, Abhilash Koppula, Judy Qiu
School of Informatics and Computing
Indiana University, Bloomington, IN, USA
{taklwu, akoppula, xqiu}@indiana.edu

***Abstract*—Use of high-level scripting languages to solve big data problems has become a mainstream approach for sophisticated machine learning data analysis. Often data must be used in several steps of a computation to complete a full task. Composing default data transformation operators with the standard Hadoop MapReduce runtime is very convenient. However, the current strategy of using high-level languages to support iterative applications with Hadoop MapReduce relies on an external wrapper script in other languages such as Python and Groovy, which causes significant performance loss when restarting mappers and reducers between jobs. In this paper, we reduce the extra job startup overheads by integrating Apache Pig with the high-performance Hadoop plug-in Harp developed at Indiana University. This provides fast data caching and customized communication patterns among iterations for data analysis. The results show performance improvements of factors from 2 to 5.**

Keywords—Pig, Iterative Algorithms, Big Data, Language, MapReduce.

I. INTRODUCTION

The MapReduce programming model has been widely adopted by many fields of research in computer science and scientific computing. It provides desirable features linking pleasingly parallel computation, horizontal scalability on complex parallel codes, and high performance on commodity clusters and clouds. Hadoop [1] is the Java-based, open-source project that provides the interfaces for large-scale parallel implementations of algorithms and applications. But in order to achieve the best performance, it requires advanced knowledge of the MapReduce programming model and significant programming skills in Java. Beyond MapReduce, high-level language platforms such as Pig [2], Hive [3], and Shark [4] are introduced to support an expressive, directed, acyclic graph (DAG) computing model that explicitly encodes data flow as a sequence of MapReduce jobs. These languages hide the complexity of interleaved data transformations and execution optimization of MapReduce programs. Instead, it provides high level functional operators and record-based data-type abstraction, simplifying the way for users to handle different types of data integration in data warehouses and iterative computation for scientific applications.

So far, these high-level language platforms have been used by many commercial companies, including Yahoo!, Facebook, Amazon, and LinkedIn. They have proven to be efficient in handling daily ETL (Extract, Transform, and Load) operations and ad hoc queries in many big data problems such as Terabyte-level log records analysis and massive email/text message analysis. Thousands of the MapReduce jobs submitted daily are said to be generated as either Pig or Hive scripts in these companies [2, 3]. However supporting iterative applications is nontrivial. Most of these solutions require developers to write user-defined functions (UDFs) for the computing functions and encode with an external control-flow script to map the data from disk to memory between iterations. As a result the performance is limited due to submitting multiple rounds of MapReduce jobs with extra job startup overhead. As most of these language systems are built on top of Hadoop, using disk based cache and disk I/O, the data communication overhead becomes substantial causing the overall performance loss.

In this paper, we use Pig as an example and introduce Pig integration with Harp [5], a fast caching MPI-like collective communication plugin with Hadoop. This is an attempt to simplify the programming model using a high-level language and improve the performance by providing fast data caching and better communication patterns between iterations. The user is to write UDFs and link multiple steps with the Pig script; those UDFs can themselves call libraries like R [6] or Apache Mahout [7]. Our system will provide the data caching and high performance communication between parallel processes, allowing the user to focus on semantics of applications.

The rest of this paper is organized as follows. Section II introduces the background of Harp and Pig. Section III explains our vision of system design and improvement by integrating Pig on Harp. Section IV presents use cases for scientific applications. Section V shows evaluation results based on performance as well as coding efficiency complexity., Section VI compares our approach with related work. Section VIII sums up our conclusions and future work.

II. BACKGROUND

Harp is a Hadoop plugin that enables loop awareness, fast in-memory caching, and collective communication patterns for iterative computation. It replaces the default mapper interface with a long-running mapper that supports multi-threading and in-memory caching. Compared with process-based task

scheduling in Hadoop, it can handle large intermediate data more efficiently in a shared memory. Harp provides MPI-like collective communication interfaces for customized network-based shuffling, in addition to disk-based shuffling with HDFS. These new features enable desirable processing capabilities and high performance for data intensive applications.

Pig Latin [8] is a platform designed for large-scale data analysis with Hadoop MapReduce. Pig provides a high level language that hides complicated MapReduce programs with simple notations for a dataflow program. Internally, Pig scripts are compiled into sequences of MapReduce jobs, which automates parallelization and makes the code easy to maintain. Figure 1 shows an example of WordCount written in Pig.

In a Pig dataflow, each line of code has only one data transformation, which can be nested. The WordCount program consists of seven lines of code, and the syntax is straightforward and easy to understand. At the start, data is loaded as records in a relation/outer bag, and each field in a record is defined according to Pig's default data types: bag, tuple, and field; a bag is a set of unordered columnar tuples; a tuple is a set of fields, where tuples in bag can contains flexible length of fields and fields at the same column can have different data type; and a field is the basic type of a piece of data.Other than the syntax shown in this paper, Pig provides operations and syntax patterns for various data transformations, although the current version of Pig does not support optimized storage structures such as indices and column groups.

```
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';
```

Fig. 1. WordCount written in Pig [9]

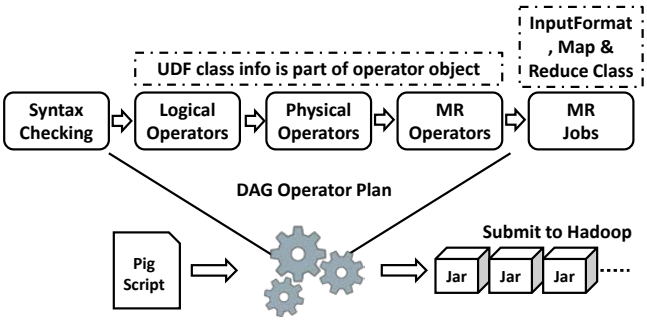


Fig. 2. Pig High Level Dataflow

Whenever a user submits their Pig scripts in a batch mode or enters line-by-line data transformation commands in an interactive mode, a default compiler handles the overall execution flows. This compiler translates the entered Pig scripts into operators and forms top-down Abstract Syntax Trees (AST) in different stages. It then visits the last compiled AST from the MapReduce operators plan compiler and constructs MapReduce jobs in order. Figure 2 shows the dataflow and lists all major steps. Similar to any programming language, Pig checks syntax by parsing the user-submitted script into a parser written in ANTLR (ANother Tool for Language Recognition) [10]. Pig's main driver program converts each MapReduce operator from Map-Reduce Operator Plan (MROperPlan) objects into Hadoop JobControl objects with detailed descriptions, input/output linkages, and other parameters, which are then passed along to each worker node with a configuration in xml format. These translations generate Java .jar files that contain the Pig default Map and Reduce classes, including the user-defined functions. The packages of .jar files are submitted to Hadoop Job Manager, and job progress is monitored until completion of the tasks.

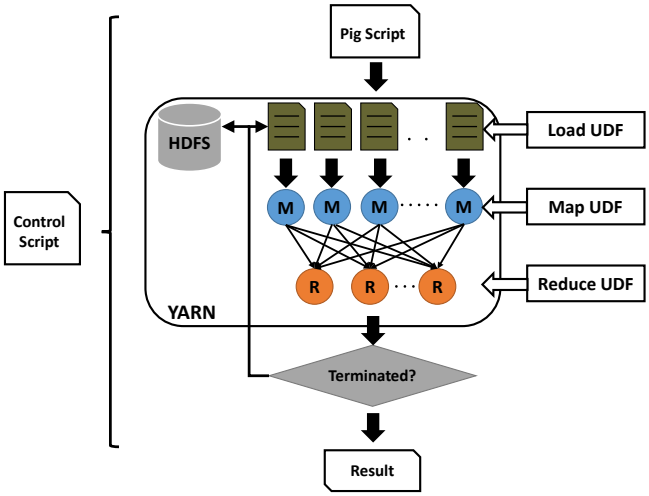


Fig. 3. Iterative applications with Pig

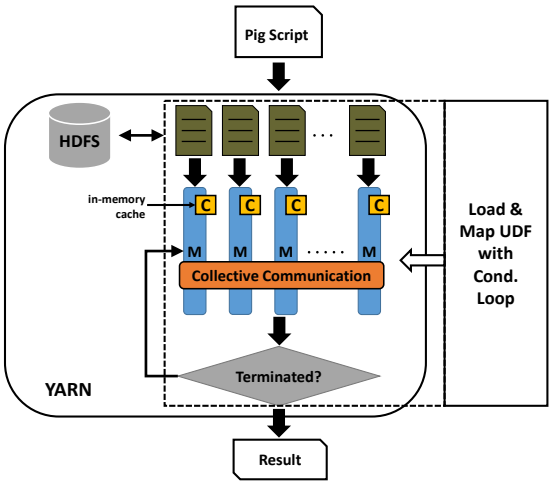


Fig. 4. Iterative application with Pig+Harp

III. PIG IN SUPPORTING ITERATIVE APPLICATIONS

Pig does a good job for ETL applications, but lacks support for iterative methods. When writing Pig programs for such applications, the control flow should be similar to what is shown in Figure 3. An external wrapper script is required, because Pig syntax does not provide a control flow statement. This causes extra overhead of job startup and cleanup time when a program runs in several rounds of MapReduce jobs.

Furthermore, inputs of iterative applications are normally unchanged and cacheable between iterations, whereas Pig has a DAG framework that does not cache those inputs in memory and reuses them inefficiently.

As Pig lacks loop-awareness and in-memory caching, we investigate a version of Pig for scientific applications based on the DAG computation model. There are several iterative MapReduce frameworks as candidates to integrate with Pig, including Twister [11], Spark [12], HaLoop [13], and Harp. We chose Harp as it is a simple MapReduce extension that supports our required iteration features. With Harp integration, we replace the Hadoop Mapper interface with Harp's MapCollective, long-running mapper to support conditional loops. Subsequently, iterative applications implemented in Pig+Harp can cache reusable data and replace the default GROUP BY operation with Harp's collective communication interface with high performance data movement. We compare the default Hadoop reduce stages withHarp's collective communications in Section 5. Figure 4 shows a dataflow that can be applied to iterative applications.

IV. USE CASES

We use K-means clustering and PageRank, as both are popular iterative algorithms for scientific computation. Our approach can be extended to other algorithms using similar user-defined functions, e.g. Naïve Bayes classifier. We compare two implementations for these two algorithms, one implemented on Hadoop 2.2.0 and the other on Harp 0.1.0, both scheduled with YARN resource manager.

A. Pig K-means

Pig K-means implementation is split into three components: a python control-flow script, a Pig data-transform script for a single iteration, and two K-means user-defined functions written with a Pig-provided Java interface. During each iteration, our customized Loader in each Mapper loads the aggregated centroids into memory as vector objects from the distributed cache on disk before computing the Euclidean distances for data points in the Loader stage. Each loader outputs assigned centroids and data points as fields in a single bag, each field in bag is defined as string data type which further splits into tuples for matching Pig's GROUP operation to collect partial centroid vectors from mappers. It takes the average of all partitions, emits to a final centroids file and saves it to HDFS. Figure 5 shows a single iteration of K-means written in Pig Latin.

```
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';
```

Fig. 5. Pig K-means script for a single iteration

B. Pig+Harp K-means

In the case of running Pig+Harp K-means, a customized Loader in each Mapper first loads the initial centroids and data points once from HDFS to memory as cache for all iterations. Then the UDF computes Euclidean distances and emits partial centroids locally. Harp's communication layer then exchanges these partial centroids on each mapper. By default, our UDF uses AllReduce to synchronize among all partitions. The program reuses the same set of mapper processes until break conditions have been reached.

The script in Figure 6 illustrates a similar idea using R. Users only provide the parameters, such as number of mappers, total amount of iterations, and communication patterns used for global data synchronization. Note that users need to have good understanding of Hadoop and Harp frameworks in order to achieve optimal performance.

```
1 centds = LOAD $hdfsInputDir using
           HarpKmeans('$initCentroidOnHDFS',
           '$numOfCentroids', '$numOfMappers',
           '$iteration', '$jobID', '$Comm') as
           (result);
2 STORE centroids INTO '$output';
```

Fig. 6. Pig+Harp K-means script

```
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';
```

Fig. 7. Pig PageRank script for a single iteration

```
1 pagerank = LOAD '$InputDir' using
             HarpPageRank('$totalUrls',
             '$numMaps', '$itrs', '$jobID')
             as (result);
2 STORE pagerank INTO '$output';
```

Fig. 8. Pig+Harp PageRank script

C. Pig PageRank

For Pig PageRank, we implement a model with fewer UDF functions in Hadoop by utilizing Pig operators. Figure 7 shows a single iteration of the PageRank algorithm, which is created and iteratively invoked by a Java wrapper. The script involves the following steps: a) Load the given input file using the custom loader into variable raw; b) Extract the outgoing URLs and emit the outgoing URL and partial page rank from the source URL; c) CO-GROUP above two aliases to calculate new page rank and store it in an alias newPgRank; d) Store new page rank in a HDFS temp file, which will be the input file for the next iteration. One drawback of this program is that the default Pig runtime optimizer creates extra mappers for the final STORE step when it calls the raw and prePgRank

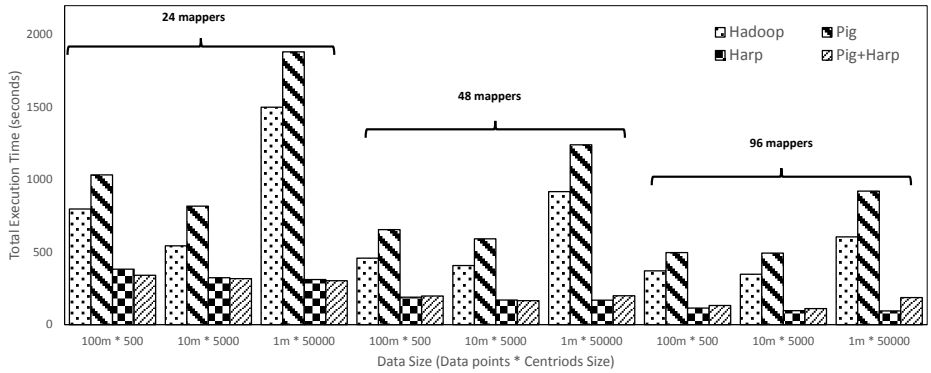


Fig. 9. K-means clustering performance comparison across different platforms

variables for CO-GROUP operators, which utilizes extra computing and memory resources.

D. Pig+Harp PageRank

In Pig+Harp PageRank, we create a new data loader and write UDFs to calculate probabilities for each web page. For the first iteration, data is loaded in a graph data structure where vertices are partitioned across all worker nodes. Each vertex receives all in-edges information by calling regroupEdges collective communication, and the number of out-edges is sent to all vertices by calling an AllMsgToAllVtx operation. The vertex and edge information is cached in memory for all iterations. Finally, the pagerank values of each vertex are updated during each iteration, and distributed by an AllGather communication until the program satisfiesbreak conditions, e.g. the end of iterations. The script shown in Figure 8 is similar to that of Pig+Harp K-Means.

V. RESULTS

We have investigated the efficiency of code in detailed implementations comparing Pig with other platforms. Also we have run a baselinecomputation for each algorithm to illustrate the performance difference. Our experiments are constructed on vertical and horizontal scales. We keep the same ratio between the amount of data points and amount of centroids, and observe the data loading, cache access, and computation overhead within the same environment. For scaling tests, we increase the computing resources in parallel by adding more mappers to each experiment in order to evaluate the communication overhead. Figure 9 shows performance results from our local cluster Madrid with Hadoop 2.2.0 and Pig 0.12.0 installations. The specification and configuration are described as below.

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

Hadoop 2.2.0: We run all master services, such as Resource Manager, NameNode, Application Master, etc., on the head node. 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. As Harp uses shared memory for a multi-threading model, we configure the master process on each worker with a total of 13GB memory.

Pig 0.12.0: it is the latest release on Oct. 13th, 2013 for Pig applications. Harp's MapCollective Mapper is embedded into Pig and made it a customized version to run on top of Harp.

We have set up three major batches of performance tests for K-means: 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 the Madrid cluster. For PageRank, we perform a strong scaling test on a dataset with 2 million vertices, and it is executed with 8, 16, and 32 mappers and partition.

TABLE I. K-MEANS IMPLEMENTED ON PIG AND HARP

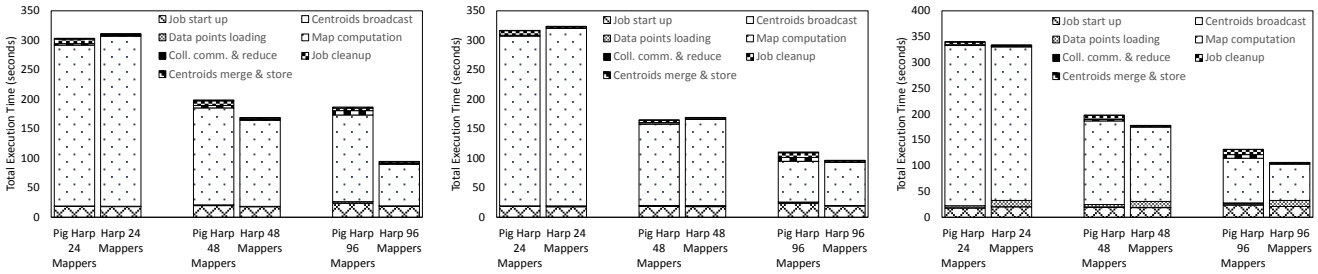
	Hadoop K-means	Pig K-means	Harp K-means	Pig+Harp K-means
K-means	36	36	39	39
Load & Format	261	250	499	662
Reduce / Comm.	142	56	34	34
Pig	0	10	0	3
Driver / Wrapper	341	40	176	0
Total lines	780	393	748	738

TABLE II. PAGERANK IMPLEMENTED ON PIG AND HARP

	Pig PageRank	Harp PageRank	Pig+Harp PageRank
PageRank	1	56	56
Load & Format	50	386	494
Reduce / Comm.	0	4	4
Pig	4	0	3
Driver / Wrapper	70	90	0
Total lines	125	536	557

A. Coding Style

Table I has shown the lines of code for a K-means application implemented on Pig and other platforms. In general, applications written in Pig require less code, as it does



a. 1 million data points with 50000 centroids b. 10 million data points with 5000 centroids c. 100 million data points with 500 centroids

Fig. 10. Performance details of Pig+Harp K-means and Harp K-means

not include the control flow statements. By contrast, the native Hadoop MapReduce implementation requires more code lines to define the variables and data transformation functions. But in some sense these complex data transformations are implemented exactly the same both in Hadoop/Harp and Pig's UDFs. In our Pig+Harp K-means examples, the amount of code is almost identical to Harp K-means; the UDFs contain customized data loading, computation and user-defined communication. This is similar to PageRank as shown in Table II, except that Pig PageRank implementation has less code, as we only rewrite a customized data loader. These tables record the "Load & Format" row that covers the lines of codes that load and store data from/to the file system. It also includes lines that transform abstracted data type to java primitive data type before any computation, and convert java primitive data type to Harp data type when using collective communication. In our projected scientific Pig prototype, this capability would be included within the framework rather than the responsibility of the user.

We stress that our tests do not demonstrate a key advantage of "Scientific Pig"; namely the ability to efficiently link multiple analysis (pipeline) steps on the same data sample.

B. *Performance and Parallelism*

Figure 9 shows K-means performance, where most of the Pig tests 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 a data bag instead of key-value pairs; the shuffling stage before the reduce computation also takes longer time. In addition, for the 1 million data points with 50K centroids tests, both Hadoop and Pig have a huge performance loss, as they reload the centroids for each iteration, and the computation of centroids array grows beyond L2 & L3 cache and impacts the mapper computation time. In summary, Harp performs the best, as it is highly optimized java implementation. Meanwhile, Pig Harp achieves a comparable performance.

We have increased the timing detail 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 it is interesting to observe that Pig+Harp running as multi-processing model also has good performance and performs as close as Harp multi-threading model. Few cases of Pig+Harp implementations, e.g. running 24 mappers on 50k centroids against 1 million data points, are running slightly faster. Those performance advances are due to more stable non-shared processor level memory address space allocation. But there is a special case for running 96 mappers on 50k centroids against 1 million data points, Harp shows the advantages of its default multi-threading computation, while Pig+Harp implementation have the same L2 & L3 cache effect of in-memory caching for large centroids; the pure mapper computation time is 2 times slower. Pig+Harp's data communication takes longer, as more processes generate more messages, and it lacks in-node global data reduction. Harp communication model is highly optimized for object serialization and deserialization, therefore the multi-processing computation and communication overhead in our Pig+Harp tests is acceptable.

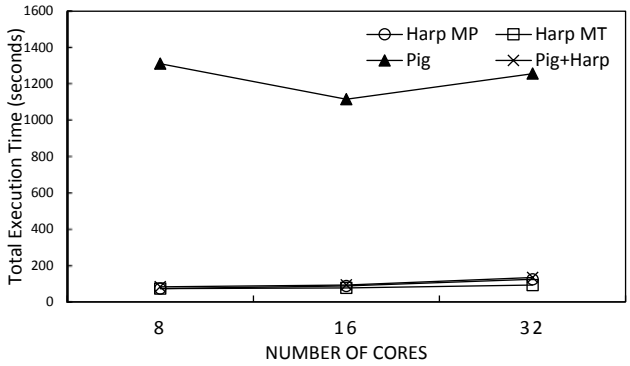


Fig. 11. PageRank performance comparison

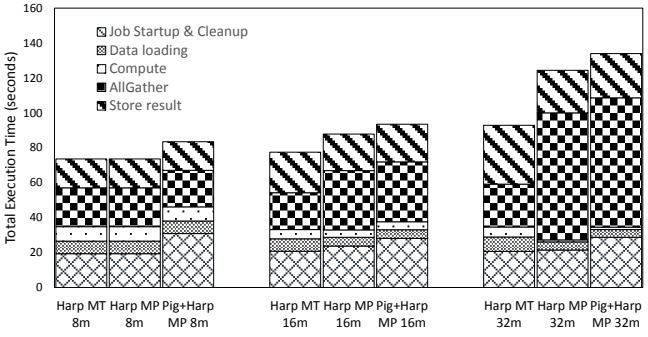


Fig. 12. Performance details of PageRank

Figure 11 displays several variations of PageRank implementations, including the native Pig versus Harp's integrations. All the cases implemented on Harp run 10 times faster than the Pig implementation. This is because the loop-unawareness record-based computation of native Pig PageRank takes longer time; data is reloaded every iteration, Pig data type

conversion time between bags and fields cost extra overhead, and compute processes are restarted with every job. Additionally, as shown in Figure 12, the Pig+Harp integration performs close to the native Harp multi-threading and multi-processing implementation. Due to AllGather communication used in Harp for PageRank values updated between iterations, alarger number of partitions is likely to increase the overall communication time; this is also similar reason of a native Pig implementation where reduce stages take longer for the case of 32 mappers.

C. Coding Complexity

Rewriting all the code from Hadoop MapReduce to Pig is not complicated by default, as Pig is designed to run data warehouse applications on top of Hadoop. The only problems we encounter here are the logic of how the data is stored in Pig's data format and how it could retrieve the correct form of data from abstracted data formats before passing it to the computation. If a legacy code is written 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.

VI. Related Work

DataFu [14] 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 standard MapReduce UDFs for developers working in data mining and statistics. Our project shares these similarities, but we focus on the performance for iterative applications with global data dependencies and research purposes using Apache open source stacks for data scientists.

Shark [4] integrates Spark [12] with Apache Hive 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 [15] provides the possibility of writing iterative applications with Scala script by first extracting the read-only data into RDDs, then computing the core iterative algorithms with the Spark runtime. We intend to compare Shark with Pig+Harp in our future work.

Cascading [16] is a Java library built on top of Hadoop to support data-parallel pipelines. It is similar to Pig but instead constructs data pipelines as DAG flow from source tap to sink tap programmatically by writing linkage for each component in a pipe that maps into MapReduce jobs. Unlike our work, Cascading naturally supports iterative applications as a dependency ordered DAG, where developers need to write the correct Riffle annotations to link the input and output as source and sink between iterations. Although Cascading considers unchanged data source/sink as reusable logic unit, it does not support in-memory data caching between iterations. Cascading as a library maintaining the full expressivity of Java is expected to be more powerful than Pig as a specialized language. We hope to look at a Harp enhanced Cascading as a technical data analysis environment in the future.

Apache Tez [17] 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. Still, this approach does not support loop-aware computation and in-memory caching from the default Pig/Hive language syntax, and the Pig community does not have any alpha release for version 0.12.x on this track.

HaLoop 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 and 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.

Apache Hadoop 2.3.0+ supports off-heap in-memory caching where the NameNode of HDFS caches given paths or files for any read-only data frequently queried by the user. Compared to our integration with Harp, we have one set of on-heap vector objects, the read-only X-dimensions data points convert from files, and our approach still saves computation cycles for such data transformation. Furthermore, lifelong running workers and MPI-like data communication in Harp saves the job restart overheads and boosts the overall performance.

VII. Conclusion

We have successfully integrated Pig with Harp and have presented the idea of writing loop-aware applications in a single Pig script. Our results show that Pig+Harp can achieve nearly the same performance compared to pure Harp implementations, although a user must have basic knowledge of and familiarity with MapReduce, Harp architecture, and programming skills. Moreover, we have shown the possibility of providing user-friendly libraries. One may harbor doubts such as, "Why don't we use RHadoop [18] or other scripting libraries directly instead of integrating Pig with Hadoop or Harp to achieve similar goals?" Our approach is motivated by the fact that Hadoop and Apache open-source stacks are designed as the mainstream tools for handling big data problems. In order to achieve the best performance, we should leverage these building blocks to maximize the usage of existing features, such as expressiveness of data type and data structure, automatic parallelization for applications, and algorithms. This motivated our switch [19, 20] from custom iterative MapReduce such as our successful Twister system [10] to development of a Hadoop plug-in to support Iterative MapReduce. To support large-scale iterative applications in Pig+Harp, we suggest developers should minimize the overhead of using Pig; one should avoid the slow record-based computation and replace aggregation operators by writing customized collective communication. In addition, as Pig+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 data analysis.

Our current results have not considered and investigated data access patterns, general data abstractions, optimization of Pig operators, or using Pig to link scientific data pipelines as an end-to-end solution in the contextof using high-level languages to solve parallel computing problems. We may go further in this direction as future work which aims at a version of Pig optimized for technical data analytics.

ACKNOWLEDGMENT

This project is in part supported by NSF Grant OCI-1032677 and NSF CAREER Grant. We would like to thank our colleagues Xiaoming Gao and Bingjing Zhang of the SALSA team at Indiana University and Dr. Geoffrey Fox for their support and comments.

REFERENCES

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

[2] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava, "Building a high-level dataflow system on top of Map-Reduce: the Pig experience," *Proc. VLDB Endow.,* vol. 2, no. 2, pp. 1414-1425, 2009.

[3] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: a warehousing solution over a map-reduce framework," *Proc. VLDB Endow.,* vol. 2, no. 2, pp. 1626-1629, 2009.

[4] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica, "Shark: SQL and rich analytics at scale," in Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, New York, New York, USA, 2013, pp. 13-24.

[5] B. Zhang. "Apache Harp Project," http://salsaproj.indiana.edu/harp/.

[6] R. D. C. Team, *R: A Language and Environment for Statistical Computing*: R Foundation for Statistical Computing, 2011.

[7] "Apache Mahout," https://mahout.apache.org/.

[8] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: a not-so-foreign language for data processing," in Proceedings of the 2008 ACM SIGMOD international conference on Management of data, Vancouver, Canada, 2008, pp. 1099-1110.

[9] "Pig Programming Tools," http://en.wikipedia.org/wiki/Pig_(programming_tool).

[10] T. Parr. "http://www.antlr.org/," http://www.antlr.org/.

[11] 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, Chicago, Illinois, 2010.

[12] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," in 2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '10), Boston, 2010.

[13] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "HaLoop: Efficient Iterative Data Processing on Large Clusters," in The 36th International Conference on Very Large Data Bases, Singapore, 2010.

[14] M. Hayes, and S. Shah, "Hourglass: A library for incremental processing on Hadoop." pp. 742-752.

[15] 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 Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, San Jose, CA, 2012, pp. 2-2.

[16] "Cascading," http://www.cascading.org/.

[17] "Apache Tez," http://tez.incubator.apache.org/.

[18] "RHadoop," https://github.com/RevolutionAnalytics/RHadoop.

[19] J. Qiu, S. Jha, A. Luckow, and G. C. Fox, *Towards HPC-ABDS: An Initial High-Performance Big Data Stack*, 2014.

[20] S. Jha, J. Qiu, A. Luckow, P. Mantha, and G. C. Fox, "A Tale of Two Data-Intensive Approaches: Applications, Architectures and Infrastructure," in 3rd International IEEE Congress on Big Data Application and Experience Track, 2014.