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, 47408
[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. The results show performance improvements of factors from 2 to 5.

Keywords

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

1. 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 implementing 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, some have built high-level languages such as Pig [2], Hive [3], and Shark [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, enabling users to handle different types of data integration in data warehouses and with less experience, iterative computation in scientific applications.

So far, these high-level languages systems have been used by many commercial companies, including Yahoo!, Facebook, Amazon, and LinkedIn, and they have proven to be efficient enough to handle 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. More than half of the MapReduce jobs submitted daily are said to be generated as either Pig or Hive scripts in these companies. However supporting iterative applications 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 of 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 [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 UDF's as now and link multiple steps with the Pig script; those UDF's can themselves call libraries like R [6] or Apache Mahout. Our system will provide the data caching and high performance communication between parallel processes.

The rest of this paper is organized as follows. Section 2 introduces the general background of Harp and Pig. Section 3 explains our vision of system design and improvement by integrating Pig on Harp. Section 4 presents targeted use cases for scientific applications. Section 5 shows aspects of results based on the lines of code, performance, and coding difficulty. Section 6 compares our approach with the related solutions. Section 7 sums up our conclusions.

2. BACKGROUND

Harp is a Hadoop plugin that enables 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-process computing and in-memory caching, instead of Hadoop's default multi-process parallel computing on split key-value pairs. 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 is a high-level platform extensively designed for large-scale Hadoop MapReduce data analysis applications on raw data. Pig Latin [7] is the provided language that abstracts the complicated Java MapReduce programs into dataflow programs with simple notation. 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 runtime. In other words, a Pig program is automatically 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 seven 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 tuples 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';
```

Figure 1. WordCount written in Pig Latin [8]

Whenever a user submits their Pig Latin scripts in batch mode or enters line-by-line data transformation commands in interactive mode, a default compiler handles the overall execution flows. This compiler translates the entered Pig Latin scripts into machine-understandable operators and forms 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 checks syntax by parsing the user-submitted script into a parser written in ANTLR (ANother Tool for Language Recognition) [9]. 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 the general system configuration in xml format. These translations generate Java jar files as MapReduce jobs that contain the Pig default Map and Reduce classes, including the user-defined functions if any. The package jar files are submitted to Hadoop Job Manager in sequences, and job progress is monitored until finished.

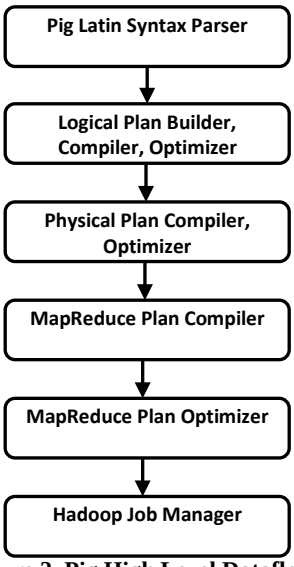


Figure 2. Pig High Level Dataflow

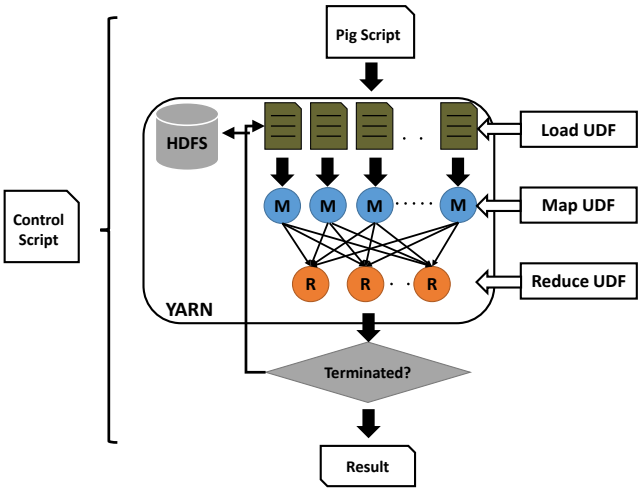


Figure 3. Iterative applications with Pig on Hadoop

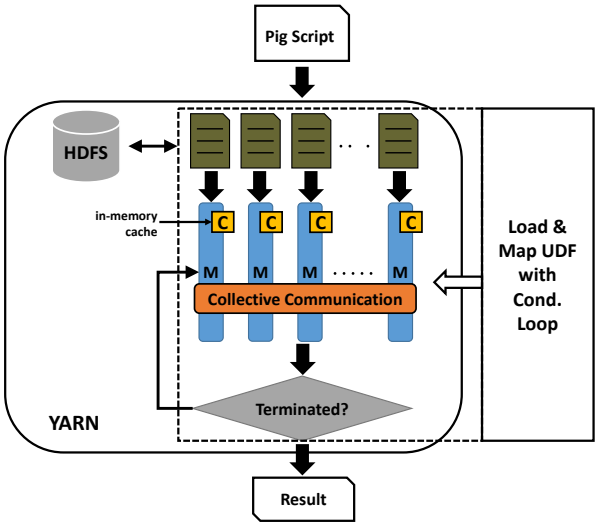


Figure 4. Iterative application with Pig on Harp

3. PIG IN SUPPORTING ITERATIVE APPLICATIONS

Pig is good enough for general ETL applications, however it does badly in supporting iterative applications. When writing Pig programs for iterative applications, the control flow should be similar to what is shown in Figure 3. The need for an external wrapper script is vital, because Pig syntax does not provide a control flow statement. Therefore, a submitted program runs in several rounds of MapReduce jobs with extra overhead from unnecessary job startup and cleanup time, which hugely decreases overall performance. Additionally, inputs of iterative applications are normally unchanged and cacheable in every iteration, whereas Pig is a DAG framework that cannot 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 framework candidates: Twister [10], Spark [11], HaLoop [12], and Harp. We choose Harp for our initial approach, as it is a simple MapReduce extension that supports our required features. With Harp integration, we mainly replace the Hadoop Mapper interface with Harp's MapCollective, long-running mapper to support conditional loops. 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 5. Figure 4 shows overall dataflow that can be applied to any iterative applications.

4. USE CASE

The proposal applications are K-means clustering and PageRank, both popular iterative algorithms for scientific computation, but our approach could be extended to other algorithms 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.

```
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';
```

Figure 5. Pig K-means on Hadoop for a single iteration

4.1 Pig K-means on Hadoop

Here, Pig K-means on Hadoop 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 a Pig-provided Java interface. During every iteration, our customized Loader in each Mapper loads the centroids into memory from distributed caches 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.

4.2 Pig K-means on Harp

In the case of Pig K-means running on Harp, the customized Loader in each Mapper first 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. 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 met.

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

Figure 6. Pig K-means on Harp

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.

4.3 Pig PageRank on Hadoop

For PageRank implemented in Pig, we use fewer 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 the PageRank algorithm is created and iteratively invoked by a Java wrapper. Steps in this script involve: 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 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 that the default Pig runtime optimizer creates extra mappers for the final step when it calls the raw variable for the CO-GROUP operators, using extra computing and memory resources.

```
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';
```

Figure 7. Pig PageRank on Hadoop for a single iteration

4.4 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 the initial iteration, data is loaded in a graph data

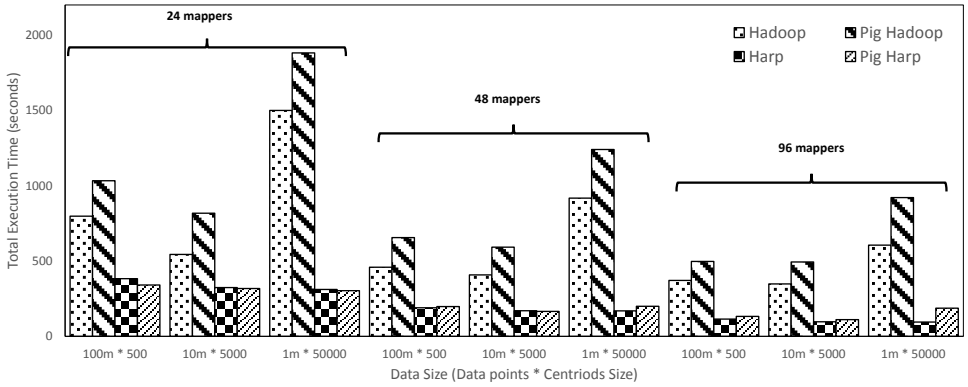


Figure 8. K-means Clustering Performance Comparison across different platforms

structure where vertices are partitioned across all worker nodes. Each vertex has all its in-edges information by calling regroupEdges collective 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 values of each vertex on a worker node are changed during an iteration, they are distributed by an AllGather communication until the program meets break conditions, e.g. the end of iterations. The script shown in Figure 9 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';
```

Figure 9. Pig PageRank on Harp

5. 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 each algorithm to see the performance difference. We construct our experiments on vertical and horizontal scales. 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 8 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-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 the 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. 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 the 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.

Table 1. K-Means implemented on Pig, Harp and Hadoop

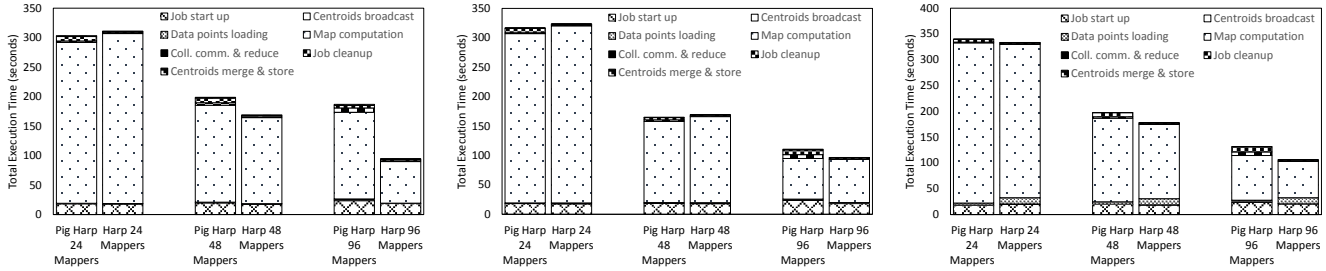
	Hadoop K-means	Pig K-means on Hadoop	Harp K-means	Pig K-means on Harp
Kmeans	36	39	39	39
Load & Format	261	250	499	662
Reduce / Comm.	142	56	34	34
Pig	0	10	0	3
Driver / Wrapper	341	40	176	0
Total lines	780	395	748	738

Table 2. PageRank implemented on Pig and Harp

	Pig PageRank on Hadoop	Harp PageRank	Pig PageRank on Harp
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

5.1 Coding Style

Table 1 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 not include the control flow statements. By contrast, the native Java MapReduce



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

Figure 10. Performance details of Harp K-means and Pig K-means on Harp

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 2, but in the case of Pig implementation on Hadoop, we write less code, as we only rewrite a customized data loader. These tables record the "Load & Format" row that covers the lines of codes that loads and stores data from/to 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 this capability would be included in system and would not be responsibility of the user.

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

5.2 Performance and Parallelism

For K-means comparison, seen in Figure 8, 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 a databag instead of key-value pairs; the shuffling stage before the 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.

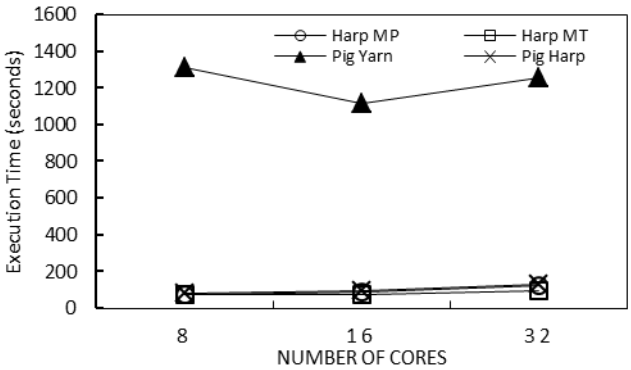


Figure 11. PageRank Performance Comparison

We have also enlarged and compared 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 we even have interesting findings that Pig on Harp with multi-processes computing model has good performance. 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. 50,000 centroids against 1 million data points; the pure mapper computation time is 2 times slower. 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.

For the PageRank result shown in Figure 11, we display several variations to compare 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, Pig data type conversion time between databag and fields cost extra overhead, and compute processes are restarted with 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.

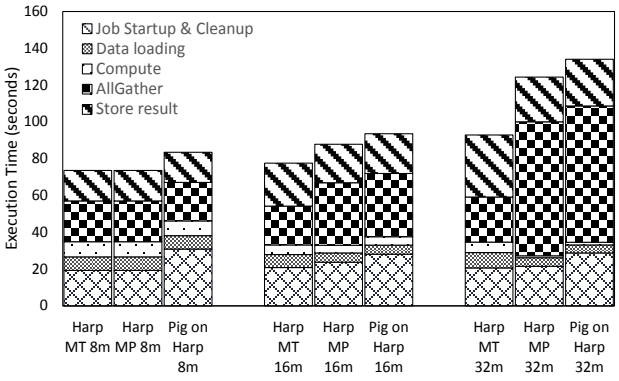


Figure 12. Performance details of PageRank

5.3 Coding Difficulties

Rewriting all the code from Java Hadoop MapReduce to Pig is not difficult in general, 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 the correct form of data from abstracted data formats before passing it 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.

6. RELATED WORK

DataFu [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] integrates Spark [11] 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 [14] provides the possibility of writing iterative applications into one 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 [15] is a Java library built on top of Hadoop to support data-parallel pipelines, it's similar to Pig but it constructs data pipeline as DAG flow from source tap to sink tap programmatically by writing linkage for each component in 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. In addition, although Cascading consider unchanged data source/sink as reusable logic unit, it does not support in-memory data caching between iterations. Some have commented that Cascading as a library maintaining the full expressivity of Java is 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 [16] 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 caches 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.

7. 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 have fundamental knowledge of and familiarity with MapReduce, Harp 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 RHadoop [17] or other scripting libraries directly instead of integrating Pig 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 [18, 19] 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 with 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 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 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 sense of 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.

8. ACKNOWLEDGMENTS

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 in Indiana University, who shared their results on the implementation for Hadoop and Harp K-means. Also, thanks to Dr. Geoffrey Fox with whom we discussed the possibility, usability, and

computation model of using high-level language for parallel computing.

9. REFERENCES

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

[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] 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, Vancouver, Canada, 2008, pp. 1099-1110.

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

[9] T. Parr. "http://www.antlr.org/," 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, Chicago, Illinois, 2010.

[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), Boston, 2010.

[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, Singapore, 2010.

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

[14] 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.

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

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

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

[18] Judy Qiu, Shantenu Jha, Andre Luckow, and Geoffrey C.Fox, "Towards HPC-ABDS: An Initial High-Performance Big Data Stack. August 8, 2014. http://grids.ucs.indiana.edu/ptliupages/publications/nist-hpc-abds.pdf.

[19] Shantenu Jha, Judy Qiu, Andre Luckow, Pradeep Mantha, and Geoffrey 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. June 27- July 2, 2014. Anchorage, Alaska. http://arxiv.org/abs/1403.1528.