2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing Workshops

Watershed reengineering: making streams programmable

Rodrigo Caetano Rocha, Renato Ferreira, Wagner Meira, Dorgival Guedes
Department of Computer Science
Universidade Federal de Minas Gerais
Belo Horizonte, Brazil
{*rcor, renato, meira, dorgival*}*@dcc.ufmg.br*

***Abstract*—Most high-performance data processing (aka big-data) systems allow users to express their computation using abstractions (like map-reduce) that simplify the extraction of parallelism from applications. Most frameworks, however, do not allow users to specify how communication must take place: that element is deeply embedded into the run-time system (RTS), making changes hard to implement. In this work we describe our reengineering of the Watershed system, a framework based on the filter-stream paradigm and focused on continuous stream processing. Like other big-data environments, watershed provided object-oriented abstractions to express computation (filters), but the implementation of streams was an RTS element. By isolating stream functionality into appropriate classes, combination of communication patterns and reuse of common message handling functions (like compression and blocking) become possible. The new architecture even allow the design of new communication patterns, for example, allowing users to choose MPI, TCP or shared memory implementations of communication channels as their problem demand. Applications designed for the new interface showed reductions in code size on the order of 50% and above in some cases, with no significant performance penalty.**

***Keywords*-big data; high-performance computing; stream processing system; programming model; parallel programming;**

I. Introduction

The explosion of data available to researchers today has led to the growth of the area of high-performance data processing (a.k.a. big-data). To make the task of developing applications that can explore the parallelism intrinsic to the data in a way accessible to experts in the domain of the data, but perhaps not in parallel programming, multiple frameworks have been proposed, with MapReduce (and its open-source implementation, Hadoop) being the most well known. Most frameworks, however, do not allow users to specify how communication between computer nodes must take place: that element is deeply embedded into the run-time system (RTS), making changes hard to implement.

Watershed [1] is a distributed stream processing system for massive data streams, inspired in the *data-flow* model. Stream processing systems comprise a collection of modules that compute in parallel, and that communicate via *data streams* [2], *i.e.*, filters obtain data from channels, transform them and send them to output channels. In its original form, Watershed also presented streams as black-boxes, elements that cannot be altered by the user. Not only that, but its orientation towards continuous stream processing made it hard to use when developing more traditional, batch/file-oriented applications.

To change that, we performed the re-engineering of the Watershed framework, to make the stream abstraction extensible by the programmer. By isolating stream functionality into appropriate classes, combination of communication patterns and reuse of common message handling functions (like compression and blocking) became possible. Our goal is that, with this redesign, users can extend the framework with other communication elements, better fitted to their algorithms.

For example, users should be able to add transformations to a data stream in an elegant fashion, without requiring new processing filters to be written, nor changes to their application code. One common case is that of message aggregation: in many applications, to reduce overhead due to the transmission of many small messages, it is often good for performance to block a large number of data items into a single message to be sent over the network. In the original version of Watershed, each developer had to add that blocking code (and its unblocking counterpart) to their application code. In the re-engineered version, it should be possible for programmers to reuse a common block/unblock element that would be added directly to the stream, not to the application code.

The new framework architecture should also allow the design of new communication patterns, for example, allowing users to choose MPI, TCP or shared memory implementations of communication channels as their problems demand. That would help also in making Watershed simpler to use in the development of batch/file-oriented applications, as streams could be extended to better serve that form of processing.

The remainder of this work describes our design decisions during this re-engineering process, and some preliminary results. Those results show that applications designed using the new API showed reductions in code size on the order of 50% and above in some cases, with no significant performance penalty.

II. Related Work

Stream processing is an area that has received some attention lately, with a few different solutions proposed, like SPC, S4, Millwheel and others. The emphasis on making the communication element (streams) more flexible can be related to the Coflow abstraction. Those are discussed next.

SPC (Stream Processing Core)[3] is a distributed stream processing middleware designed to support stream-mining applications that extract information from a large number of digital data streams. SPC offers high-level user requests for information, which are translated into one or more processing flow graphs. Similarly to the Watershed programming abstraction, a processing element receives data from a collection of streams through input ports, process and writes resulting data to output ports, thus creating new streams.

S4 (Simple Scalable Streaming System)[4] is a distributed stream processing engine inspired by the MapReduce model. The S4 design shares many attributes with SPC. Both systems are designed for big data and are capable of mining information from continuous data streams using user defined operators. The main differences are in the architectural design. While the SPC design is derived from a subscription model, the S4 design is derived from a combination of MapReduce and the Actors model.

Spark Streaming is a stream processing framework based on the programming model called *discretized streams* (D-Streams), that offers a high-level functional programming API, strong consistency, and efficient fault recovery [5]. Spark Streaming is implemented as an extension to the Spark [6] cluster computing framework, which lets users seamlessly intermix streaming, batch and interactive queries. D-Streams treats a streaming computation as a series of deterministic batch computations on small time intervals. The discretized stream concept differs from the Watershed model, where we have the notion of a continuous stream of data computation, opposed to this notion of a series of deterministic batch computations.

MillWheel [7] is a programming model, tailored specifically to streaming, low-latency systems. Users write application logic as individual nodes in a directed compute graph, for which they can define an arbitrary, dynamic topology. Collectively, a pipeline of user *computations* will form a data-flow graph, where records are delivered continuously along edges in the graph. Users can add and remove computations from the topology dynamically, without needing to restart the entire system. That concept is very similar to the working principle in Watershed, in which the filters are the computation nodes.

The stream processing systems just mentioned, however, treat streams — their communication primitives — as black boxes, embedded into their run-time systems. In that aspect, this re-engineering of Watershed tries to turn streams into a first order abstraction. In that sense, it can be related to Coflow[8], a networking abstraction to express the communication requirements of prevalent data parallel programming paradigms. Coflow makes it easier for the applications to convey their communication semantics to the network, which in turn enables the network to better optimize common communication patterns. Similarly to the concept of communication pattern offered by Coflow, in the Watershed model the users are able to specify which will be the communication pattern, but they are also allowed to develop their own communication channels.

III. Module Abstraction

In Watershed, a processing module comprises a filter, a set of input ports and a set of output ports, as illustrated in Figure 1. We can attach a stream channel to each input port and attach one or more stream channels to each output port. When a module consumes the data produced by another module, a stream channel dynamically binds them via their input and output ports, respectively.

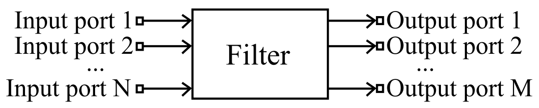

Figure 1. Abstraction of a processing module.

Filters receive data from input ports in an event-based fashion. After a filter has processed the data, it may send the transformed data through its output ports, triggering new delivery events at the next filters connected to that specific stream.

A filter can be said to be the producer for another filter when the former produces an output stream of data that is an input stream of the latter. Similarly, a filter can be said to be the consumer from another filter when the former consumes an input stream of data that is produced as an output by the latter.

IV. Stream Abstraction

The processing filters are connected via data stream channels. A stream is basically described by a sender, a deliverer, a stack of encoders and a stack of decoders, as depicted in Figure 2. Encoders and decoders can be used as pairwise data transformers during the data transmission through a stream, *e.g.*, for cryptography, data compression, data aggregation and disaggregation, etc. We can also use either an encoder or a decoder as a single, one-way transformation, *e.g.*, to encode data stream records as JSON strings, or to change a temperature stream from Fahrenheit to Celsius. The sender is responsible for sending the data through a specific transmission medium, e.g. sending the data through a network communication or through a distributed file system. The deliverer is responsible for obtaining data from the specific transmission medium and for delivering that data to a particular processing filter.

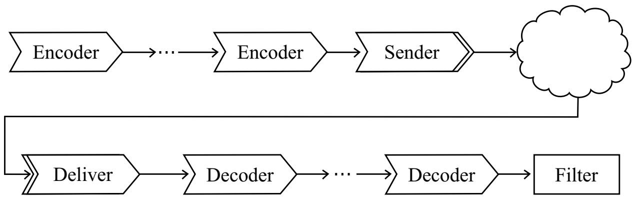

Figure 2. The decomposition of a single stream connected to a filter.

Except for the one-way transformers, the stack of encoders and the stack of decoders must agree in the sequence of execution, in such a way that the decoder of the last encoder will be the first to be executed, and vice-versa. The sender and the deliverer must also agree in the transmission medium and the communication protocol.

From the class diagram of the module components, Figure 3, we can see that the data stream is basically composed by a chain of senders and a chain of deliverers, or receivers.

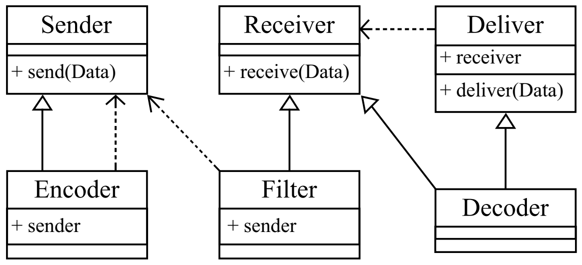

Figure 3. Simplified class diagram of the module components.

The Watershed default library of streams contains some basic implementations that are common in most of the data processing applications [8], such as: broadcast over the network; round-robin over the network; over the network labeled stream; file line reader; file line writer. The library also contains some encoders and decoders, such as data compressor, data decompressor, itemizers, and a regular expression-based filter.

V. Termination Mechanism

In its original design, Watershed considered all streams as continuous and had no explicit termination semantics. In the process of re-engineering it, we developed a termination mechanism based on Pregel's halting mechanism [9]. The termination of a module is based on every module instance *voting to halt*. A filter has an event called *onInputEnd* which is triggered when an input stream ends, and another event called *onAllInputsEnded* which is automatically triggered when all the input streams terminate. Both filters and stream deliverers have a function called *halt*. When the stream deliverer calls the *halt* function, it triggers the *onInputEnd* event of the filter. When the filter calls the *halt* function, it sends a signal to the Watershed master specifying that that particular instance of the module has voted to halt. By the time all instances of a given module have halted, the Watershed master removes the module from the execution environment.

The mechanism for propagating the termination signal of a filter through a stream channel is the responsibility of the communication protocol used to implement that particular stream, where the deliverer associated with it must trigger the *onInputEnd* event. Consumers can then decide to halt when all of its data producers finish, based on the propagation of the end-of-stream signal.

Based on the halting mechanism, we can define stream channels as either finite or continuous. Finite streams, have well defined and predetermined duration, such as a stream that reads data from a file. Continuous streams have undetermined duration, depending on an outside *actor* to finish its execution, such as a stream that reads data from a web feed, a stream that writes data into a file, or a stream that reads data from the network.

With this halting mechanism, it becomes simpler to implement batch applications on top of Watershed, as illustrated in Section VII. In a sense, the implementation of a Watershed batch application becomes equivalent to an Anthill application [10], which was one of our original goals for this work.

VI. Loading a module

When the user wants to load a new module, M_1, he sends the module descriptor to the Watershed master. The master schedules the slave nodes that will execute each instance of the module. Afterwards, the master properly sends the instance descriptors to the scheduled slave nodes. Finally, it verifies the bindings among the modules. The consumer is the one responsible for describing the stream channel components it needs, as represented by Figure 2, that will create the binding with the producer. Therefore, different consumers are able to use different stream policies to read from a same output.

For each filter M_0 that is a producer for M_1, the Watershed master sends a stack of encoders and a sender for each instance of M_0, which will be properly attached to the output port, as specified by M_1. In the same fashion, for each filter M_2 that is a consumer of a data stream produced by M_1, the master provides a stack of encoders and a sender for each instance of M_1, which will be properly attached to its output port, as specified by the input stream of M_2.

VII. Experimental Evaluation

In order to validate our redesign and the performance of the re-engineered system, we have implemented two Watershed batch applications composed by two filters. We compared their performance with other available implementations of the same algorithms, and compared the code developed in each case based on lines of code (LOC) to have a notion of code complexity in each case. The experiments have been conducted in a cluster containing five computer nodes, where one of them has performed the role of master and the remaining four behave as slave nodes. Each node runs on an Intel processor® with 4 CPU cores, a clock of 2.5GHz, and 8GB of RAM. We performed a total of five executions for each experiment.

A. *Word Frequency*

The word frequency is a batch processing application for massive datasets that is very suitable for the MapReduce abstraction [11]. Figure 4 presents an implementation in Watershed for the word frequency application that is conceptually equivalent to the MapReduce abstraction. The application consists of two filters: the WordCounter, which is equivalent to a Mapper, and the Adder, which is equivalent to a Reducer. The communication between the WordCounter and the Adder filters uses the shuffle pattern of communication [8], which delivers key-value pairs based on the result of a mapping function (usually hash) to the key. In Wathershed, that pattern is also known as a labeled stream.

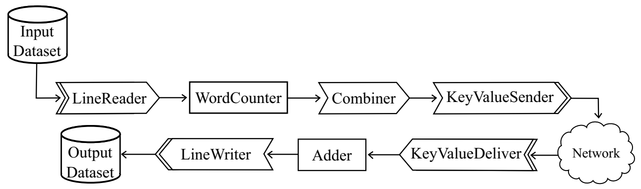

Figure 4. Word Frequency filter-stream application design.

The LineReader input stream reads the dataset line by line, with the dataset evenly distributed in the computer cluster. The Combiner is the implementation of an encoder in the stream abstraction and its implementation resembles the concept of Combiners in Hadoop MapReduce [12], in the sense that it works as a local reducer. The KeyValueSender and the KeyValueDeliverer belong to the default library of network streams and together they implement the labeled stream communication pattern.

For an input dataset of 1.1GB of English books crawled from Project Gutenberg [13], adding up to a total of about 22.5 million lines of text, the experimental results of execution time are shown in Figure 5. The dataset was partitioned into smaller data blocks of 32MB, which were distributed in the nodes of the computer cluster. For each execution, the number of instances of the WordCounter filter was the same as the number of computer nodes used in each experiment, and only one instance of the Adder filter.

Relative to the execution with only one node, by increasing the number of computer nodes, the word frequency application presents a nearly linear *speedup* with an average *efficiency* of 94.2%, as illustrated in Figure 6. On the other hand, the MapReduce implementation shows a *speedup* that is close to a logarithmic growth.

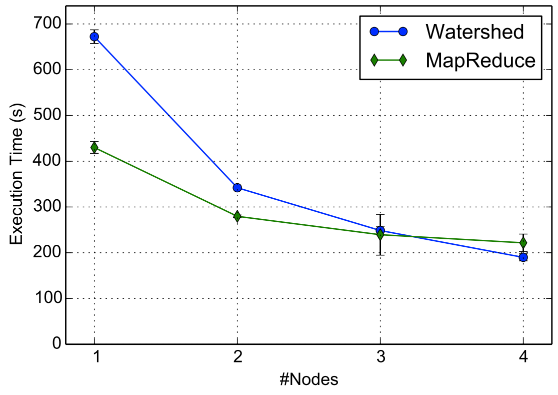

Figure 5. Word Frequency experimental analysis of execution time.

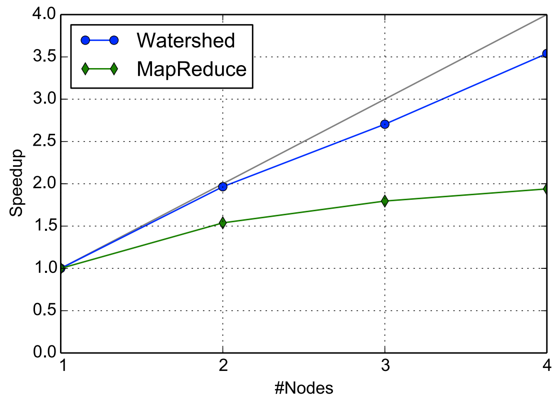

Figure 6. Word Frequency experimental speedup.

Figure 7 shows the experimental scalability of the word frequency application in respect of the size of the input data. For small input data, the latency overhead added by the Watershed framework can be much smaller than the overhead added by the Hadoop MapReduce framework, since it is heavily dependent on the file system.

The source code for the word frequency application designed for the re-engineeered Watershed interface has a total of 72 lines of code, with the advantage that the LineReader stream is a reusable component implemented by the default library of streams. However, for the previous version where the stream channels were embedded in the RTS and the user had the need to develop a filter to act as the dataset reader, the source code for the word frequency application had a total of 127 lines of code (43% LOC reduction for the re-engineered version). Just as a reference, an equivalent implementation in Hadoop MapReduce has 64 lines of code, where the same code for the Reducer can be used as the Combiner.

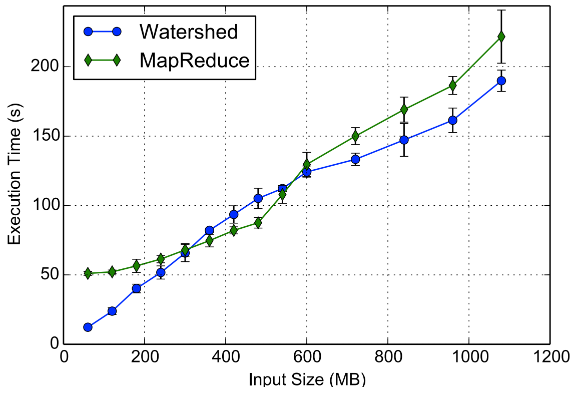

Figure 7. Word Frequency experimental scalability.

B. *k-Nearest Neighbors Classifier*

As a second experimental batch application, we have implemented a common data mining classification algorithm, the k-nearest neighbors (kNN) classifier. The kNN classifier predicts the class of a given input as the majority class among its k nearest neighbors computed from the training dataset [14]. The experiments were performed for a value of k equal to 50, a total of 5000 testing samples, and different quantities of training samples.

Figure 8 illustrates the Watershed implementation for the k-nearest neighbors algorithm. We use the same LineReader stream channel developed previously as the input stream. The input stream passes through an itemizer, which is implemented by a decoder, in order that each input data is forwarded as a key-value pair, with the input data as the value and an incremental identifier as the key. The identifier is important for the merge phase, that is when the partial results of the top k nearest neighbors are combined for each *key* composing the classification based on the final k-nearest neighbors.

For each instance of the Compare filter, the full testing dataset is read and only a portion of the training dataset. Afterwards, each instance produces a top k nearest neighbors relative to its portion of the training dataset.

In the Compare-Merge communication, the same implementation for the labeled stream has been used.

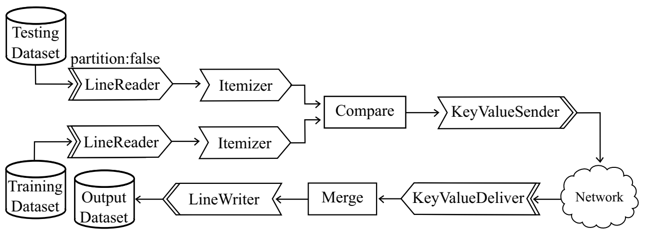

Figure 8. k-Nearest Neighbors filter-stream application design.

Figure 9 explicits an analysis of the execution time for the kNN, varying the number of computer nodes in the cluster. It follows a pattern similar to the word frequency execution time, as seen in Figure 5.

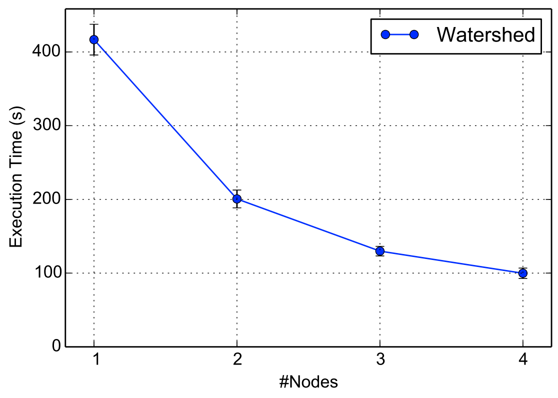

Figure 9. k-Nearest Neighbors classifier experimental analysis of execution time.

The kNN application presents a nearly linear *speedup* with an average *efficiency* of 103%, as detailed by Figure 10. The super-linear behavior is due to the increase in reduction of the partition fed to each node, as fewer items have to be read and may be all kept in memory by each node.

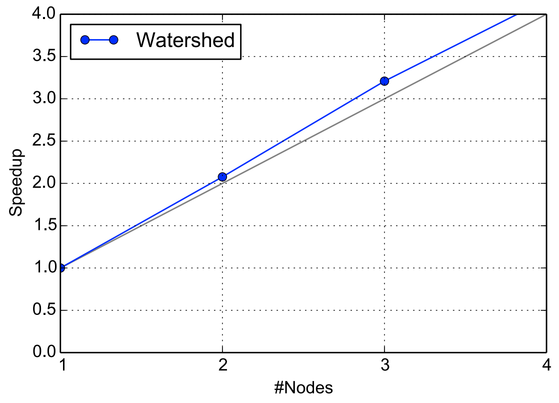

Figure 10. k-Nearest Neighbors classifier experimental speedup.

Figure 11 depicts the experimental scalability of the kNN application, based on the variation in the size of the training dataset, in the scale of thousands of samples.

The source code for the kNN application, due to the reusability of components offered by the default library in Watershed, has a total of 141 lines of code. The code size for the new interface is only 25% of the 563 lines of code of the same kNN application as implemented for the previous version of the Watershed framework.

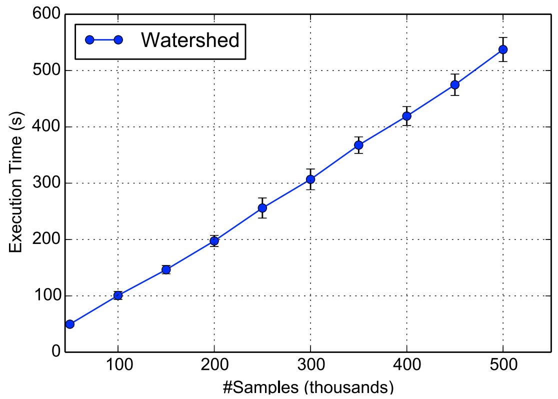

Figure 11. k-Nearest Neighbors classifier experimental scalability.

VIII. Conclusion

Multiple frameworks have been proposed to ease the task of programming big-data applications. Most of them, however, allow developers to express only their processing needs — communication is left as a black box inside the framework, to which programmers have no access. In this work we have described the process of re-engineering Watershed, our stream processing system, to make streams first-class objects in the framework. By doing that, programmers can now implement communication channels that better fit their needs, as well as to reuse code to add functionalities to an existing communication channel. Our first results show a significant reduction in code complexity, based on lines of code, while maintaining good performance, comparable or superior to the previous version.

As future work we intend to continue the development of new streams, including an HDFS stream to allow Watershed to read and write directly to the Hadoop File System. That will make data partitioning simpler for file-based applications.

Acknowledgment

This work was partially supported by Fapemig, CAPES, CNPq and InWeb, the Brazilian Institute for Science and Technology of the Web.

References

[1] T. L. A. de Souza Ramos, R. S. Oliveira, A. P. de Carvalho, R. A. C. Ferreira, and W. Meira, "Watershed: A high performance distributed stream processing system," in *Computer Architecture and High Performance Computing (SBAC-PAD), 2011 23rd International Symposium on*. IEEE, 2011, pp. 191–198.

[2] R. Stephens, "A survey of stream processing," *Acta Informatica*, vol. 34, no. 7, pp. 491–541, 1997.

[3] L. Amini, H. Andrade, R. Bhagwan, F. Eskesen, R. King, P. Selo, Y. Park, and C. Venkatramani, "Spc: A distributed, scalable platform for data mining," in *Proceedings of the 4th international workshop on Data mining standards, services and platforms*. ACM, 2006, pp. 27–37.

[4] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*. IEEE, 2010, pp. 170–177.

[5] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters," in *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*. USENIX Association, 2012, pp. 10–10.

[6] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 2010, pp. 10–10.

[7] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle, "Millwheel: fault-tolerant stream processing at internet scale," *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1033–1044, 2013.

[8] M. Chowdhury and I. Stoica, "Coflow: a networking abstraction for cluster applications," in *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*. ACM, 2012, pp. 31–36.

[9] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 135–146.

[10] R. A. Ferreira, W. Meira, D. Guedes, L. M. d. A. Drummond, B. Coutinho, G. Teodoro, T. Tavares, R. Araujo, and G. T. Ferreira, "Anthill: A scalable run-time environment for data mining applications," in *Computer Architecture and High Performance Computing, 2005. SBAC-PAD 2005. 17th International Symposium on*. IEEE, 2005, pp. 159–166.

[11] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[12] "Hadoop: Open-source implementation of mapreduce," http://hadoop.apache.org.

[13] "Project gutenberg: Free ebooks," http://www.gutenberg.org.

[14] M. J. Zaki and J. Wagner Meira, *Data Mining and Analysis: Fundamental Concepts and Algorithms*. Cambridge University Press, May 2014.