AROM: Processing Big Data With Data Flow Graphs and Functional Programming

Nam-Luc Tran and Sabri Skhiri
Euranova R&D
Belgium
Email: {namluc.tran, sabri.skhiri}@euranova.eu

Arthur Lesuisse and Esteban Zimányi
Université Libre de Bruxelles
Belgium
Email: {alesuiss, ezimanyi}@ulb.ac.be

***Abstract*—The development in computational processing has driven towards distributed processing frameworks performing tasks in parallel setups. The recent advances in Cloud Computing have widely contributed to this tendency. The MapReduce model proposed by Google is one of the most popular despite the well-known limitations inherent to the model which constrain the types of jobs that can be expressed. On the other hand models based on Data Flow Graphs (DFG) for the processing and the definition of the jobs, while more complex to express, are more general and suitable for a wider range of tasks, including iterative and pipelined tasks. In this paper we present AROM, a framework for large scale distributed processing based on DFG to express the jobs and which uses paradigms from functional programming to define the operators. The former leads to more natural handling of pipelined tasks while the latter enhances genericity and reusability of the operators, as shown by our tests on a parallel and pipelined job performing the calculation of PageRank.**

I. Introduction

With the explosion of the volume of information generated by Internet-related applications and services, companies are more and more involved in the processing of large volumes of data for operations involved in statistics, page rankings or document crawling.

One way to ease computation on such volumes is to break the process into smaller operations, each performing the computation on a subset of the data and in parallel setups. The family of computations which can be split into smaller parallel chunks is called *data parallel* [1].

The tendency which has been started first with the price drop of commodity hardware and strengthened further by the development of the Cloud Computing is to scale out computing instances rather than scale up: instead of investing in a single server instance which will handle the entire the load, enterprises design their computations to be split and processed by large clusters composed of commodity hardware. In this way many new programming models and processing paradigms have been developed in the past years in order to provide tools to ease the definition of parallel jobs running in distributed setup on large clusters of machines.

MapReduce is probably the most famous distributed processing framework. Initiated by Google[2] and made open source by Apache[1] and Yahoo!, MapReduce represents a powerful alternative to process data. However, it forces the developer to re-design the algorithm in map and reduce phases. This could represent, on one hand, an overhead in terms of computing by the unnecessary addition of tasks such as shuffle and sort, and on the other hand, an overhead in programming for matching the map and reduce functions.

In this paper we propose a better tradeoff between implicit distributed programming, job efficiency and openness in the design of the processing algorithm. We introduce AROM, a new open source distributed processing framework in which jobs are defined using directed acyclic graphs. Indeed, we believe that the DFG model is more general than the MapReduce model and makes it suitable for expressing a wider variety of jobs, especially jobs with pipelined topologies. Being furthermore implemented in a functional programming language, it also allows for generic and reusable operator constructs. AROM also provides a comprehensive API for defining the jobs as directed acyclic graphs.

The paper is organized as follows. Section 2 provides an overview of some of the current programming models for large scale distributed parallel processing. Section 3 presents the background and context to the work. Section 4 describes the related work in the field. In Section 5 we present the architecture of AROM and outline its features and execution model. Section 6 presents the results we have obtained with our current implementation. In Section 7 we then discuss our observations on the results. Finally, in Sections 8 and 9 we present our perspectives for the framework and conclude our work.

II. Models of Big Data Distributed Processing

In the current landscape of big data processing, the MapReduce model proposed by Google [2] is one of the most popular, mostly thanks to its open source framework implementation Apache Hadoop. Many Cloud service providers such as Amazon AWS[2] or Cloudera[3] now propose complete Hadoop clusters.

A. The MapReduce Model

The MapReduce model is composed of two principal phases: Map and Reduce. Figure 1 gives an overview of the

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

[2]Amazon Elastic MapReduce - http://aws.amazon.com/elasticmapreduce/

[3]Cloudera - http://www.cloudera.com/

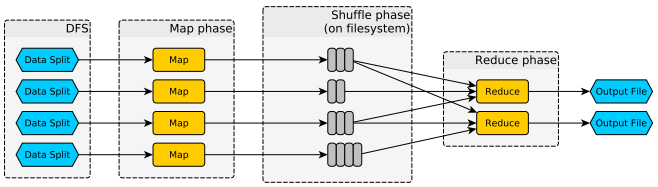


Fig. 1. The MapReduce model

MapReduce workflow. The data model in MapReduce is the key/value pair. During the Map phase, the associated Map function is executed on every key/value pair of the input data, producing in turn intermediate key/value pairs. Between the Map and the Reduce phase is the *shuffling* phase where the intermediate key/value pairs are sorted and grouped by the key. This results in a set of key/value pairs where each pair contains all the values associated to a particular key. These key/value pairs are then partitioned and grouped on the key then passed on to the Reduce phase during which the Reduce function is applied individually on each key and the associated values for that key. The Reduce phase can produce zero or one output value.

Between each phase the intermediate outputs are stored on the filesystem which is most of the time distributed for fault-tolerance but also for data localization.

The Map and Reduce functions are supplied by the user to the MapReduce framework. While the shuffling phase is closed and entirely handled by the framework, users can customize this phase to a certain extent by providing sorting and grouping directives.

Limitations: The MapReduce model is convenient to work with. Its Map and Reduce primitives, while extensive, are simple enough for most users to implement jobs with. The Map-Shuffle-Reduce workflow sets a dynamic on which the user can refer to while defining the job.

However the presence of this mandatory shuffle phase represents one of the most restrictive weakness of the MapReduce model. The sort phase is for example not necessarily required in every job. Furthermore this shuffle phase, added to the fact that the model is intended for a single source of key/value pairs input prevent a whole family of jobs such as relational operations to be implemented. Joins in particular have proven to be cumbersome to express in MapReduce [3].

B. The DFG Model

Another well known programming model for distributed large scale processing is the general Data Flow Graph. One of its most famous implementation is Microsoft's Dryad [4]. A DFG is a directed acyclic graph where each vertex represents a program and edges represent data channels. At runtime the vertices of the DFG will be scheduled to run on the available machines of the cluster. Edges represent the flow of the processed data through the vertices. Data communication between the vertices is abstracted from the user and can be physically implemented in several ways (e.g. network sockets, local filesystem, asynchronous I/O mechanisms...). Figure 4 depicts a job using the DFG notation. Such representation also outlines the paths which can be run in parallel.

The granularity of the job is thus the program represented by a vertex in the DFG. These programs can be for example pure sequential code. No data model is usually imposed, it is up to the user to handle the input and output formats for each vertex program.

Limitations and Opportunities: Manipulating DFG for defining the jobs gives users more freedom in the definition of their job workflow. This freedom in the job definition also provides the opportunity to implement jobs that are not constrained by a strict Map-Shuffle-Reduce schema. As vertex programs are able receive multiple inputs, it is possible to implement relational operations such as joins in a natural and distributed fashion. Since the jobs are not framed in a particular sequence of phases, intermediate output data do not have to be stored into the filesystem.

Furthermore, the DFG model is less constrained than the MapReduce model. We believe it is more convenient to compile a job from a higher-level language targeting a DFG framework than a MapReduce framework such as Apache Pig[5]. In fact the DFG model is more general than the MapReduce model which can be seen as a restriction on the DFG model. It is thus possible to define a semantic which translates any MapReduce job into its DFG counterpart.

C. Pipelining Considerations

Even if we could implement pipelined jobs in MapReduce, this is not as natural nor efficient when compared to a pipeline defined using DFG where each stage can be represented by a group of vertices. Figure 2 illustrates this idea.

First, each step in the pipeline would be defined as a MapReduce job or, using a trick from the original MapReduce model, the stages of the the pipelines would consist of Map-only jobs. The mandatory shuffle phase may not even be useful for the pipeline stages. The back and forth coming of the intermediate data on the distributed filesystem can also cause a significant performance penalty. Then, additional code is also required for coordinating and chaining the separate MapReduce stages and in the end the iterations are diluted, making the code less obvious to understand.

Further, another drawback of the MapReduce model for pipelining jobs is that each stage of the pipeline in MapReduce needs to wait for the completion of the previous stage in order to begin the processing. In the DFG model, provided that the vertices of the pipeline are correctly scheduled, every single output from a stage can be sent as it is produced to the downstream stage to be further processed. This may be especially efficient if the vertices processes communicate using asynchronous I/O.

The performance considerations can be particularly important in the case of applications where there are strict requirements in response times and duration of the processing [6].

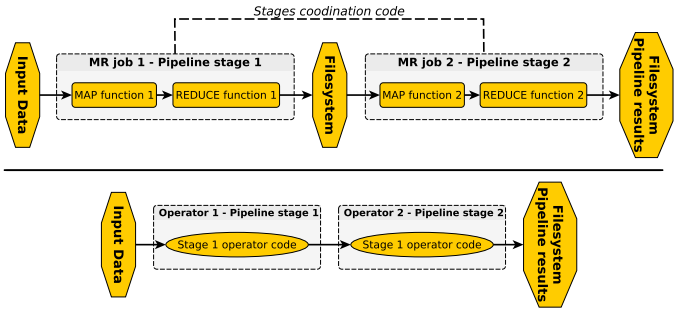


Fig. 2. Pipelining: MapReduce (top) vs DFG (bottom). In MapReduce, the first stage must complete before starting the second while in DFG the results are streamed between operators as they are produced.

III. Background and Opportunities

Reusing semantics similar to Microsoft's Dryad [4], we have developed a complete distributed processing framework using the Data Flow Graph (DFG) processing model, making it suitable for pipelined jobs.

Applications which can much benefit from pipelined topology processing comprise for example *mashups* websites which combine content from more than one source in order to generate a final result. MashReduce [6] for example makes use of MapReduce to effectively generate mashup contents targeting mobile devices. Such a mashup application can for example aggregate pictures from different locations and apply a series of operations on them prior to displaying on the device. In this case the chain of image processing and aggregation operations would be much naturally expressed using a pipeline representation.

Furthermore being implemented in a functional programming language, the AROM framework enables users to naturally express their operators using higher order functions and anonymous constructs, enforcing the genericity of the operators and their reusability.

IV. Related Work

Existing work in the field of distributed parallel processing on frameworks suited for the Cloud can be classified on two levels: developments at the engine level and contributions at the user library level.

Building on the initial MapReduce model proposed by Google [2], there are numerous projects which aim at providing higher level users libraries or integration with programming languages. Indeed MapReduce is often seen as too low-level and rigid which leads to limited reusability of the code and difficult maintenance. In this way, FlumeJava [7] provides abstractions for MapReduce programs in the form of strongly typed collections. These collections compile directly into MapReduce jobs. AROM however is implemented using Scala, a functional programming language which enables users to use higher order, anonymous functions constructs and powerful collections abstractions.

At a higher level, Google *sawzall* [8] defines a language which provides abstraction for MapReduce jobs. It provides a scripting programming language which emphasizes on filtering during the Map phase followed by aggregation during the Reduce phase and which compiles to a sequence of MapReduce jobs. One of the common use of Sawzall is to preprocess data for subsequent analysis by relational databases. Pig Latin [5] provides a higher-level language which compiles to Hadoop MapReduce jobs. Pig Latin semantics are close to the SQL, which makes it suitable for expressing queries and enforces reusability by generic statements while supporting a procedural programming style.

Other works based on the MapReduce model aims at altering the model in order to enable MapReduce to support new types of tasks. Map-Reduce-Merge [9] extends the MapReduce model with a Merge phase coming after the Reduce phase. The Merger receives the output values from the reducers and is then able to perform most of the relational primitive operations, including joins. The authors however recognizes that the fixed partitioning phase of MapReduce can limit the performance and the expressivity of the jobs. By using the general DFG model, we believe that it is possible to define relational operations more naturally thanks to the multiple inputs and to the fact that the job execution plan is expressed as directed acyclic graphs, which makes them close to the query execution plans of relational databases.

Based on the DFG model, one of the most famous implementation of a DFG based processing framework is Microsoft's Dryad [4]. However, all development has been performed behind closed doors and to date no usage of Dryad has been reported beyond Microsoft's internal usage. DryadLINQ [10] is a Microsoft project which provides Dryad with a library which enables jobs expressed in the LINQ [11] general query language to be compiled to efficient Dryad jobs. We have chosen to develop AROM as an open-source product[4] oriented for quick prototyping and research on distributed processing.

Still further developing on Dryad and the general DFG model is Naiad [12], another Microsoft project which enables the processing of iterative jobs by defining processes to perform only on the portion of updated data between two consecutive iterations. The iterations are thus performed on set of updates rather than the whole dataset. To this end, a new loop operator has been added to the original Dryad's DFG model. The main assumption however is that the iteration eventually converges, leading eventually to smaller and smaller set of differences between two iterations.

On the other hand, other implementations based on the general DFG model exist. Nephele [13] uses general DAG graphs to define the jobs. The particularity of Nephele's execution model is that the scheduler is aware of the dynamic nature of the Cloud infrastructure on which it is deployed: the VM on the cloud do not all have the same specifications (heterogeneous environment) and they can be migrated on various physical hosts through their lifecycle. Nephele's programming model is based on Parallelization Contracts (PACTs) [14] which can be seen as particular operators which perform

[4]AROM is available on github at https://github.com/nltran/arom

operations on the data prior (input PACT) to the processing by the user operator code and define the way they will be handed to the operator. Output PACTs on the other hand define filtering operations on the data coming out of an operator (for example, output only same keys, unique keys, ...). These PACTs however can be seen as particular AROM operators which can be designed generically and bundled to the user.

Especially conceived for iterative jobs is Spark [15] which is based on the concept of Resilient Distributed Datasets (RDD). The idea is that RDDs are not stored as is, but only a *handle* to an RDD which contains enough data to compute the whole RDD is stored. Iterations are then performed on the RDD which can be cached in memory if multiple iterations need to access its data. Spark is written in Scala and exposes the DryadLINQ API. It is more focused on iterative jobs and establishes a model for handling data issued from iterations. Another execution engine for data flow programs is CIEL [16] which is focused on iterative and recursive algorithms. The engine is able to spawn additional tasks at runtime and perform data-dependent control flow decisions for iterative jobs. AROM on the other hand is more centered around generic processing and the iteration support should come on top of the existing framework.

Finally, Clustera [17] is a processing framework centered around a generic scheduling model. Its goal is to remain suitable for all the types of jobs, including DAG workflows, MapReduce jobs and distributed SQL queries.

V. Design

Based on the previous observations we have decided to implement a distributed parallel processing framework based on a general DFG model. The requirements of the framework are the following:

1) Provide a coherent API to express jobs as DFG and use a programming paradigm which favors reusable and generic operators.
2) Base the architecture on asynchronous actors and event-driven constructs which make it suitable for large scale deployment on Cloud Computing environments.

The vision behind AROM is to provide a generic-purpose environment for testing and prototyping on distributed parallel processing jobs and models.

A. Job Definition

AROM jobs are described by Data Flow Graphs. A vertex in the DFG represents a program, also called *operator*, which receives data on its inputs, processes it and in turn emits the resulting data to the next operator. Edges between the vertices depict the flow of data between the operators. In order to connect the operators, we reuse the graph connectors notation presented by Microsoft's Dryad (Figure 3).

B. Operators and Data Model

The operators are modeled as purely asynchronous message-driven programs. An operator can receive and emit data on multiple inputs in the form (*i*,d) which indicates that the data

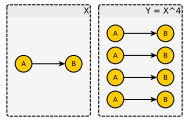
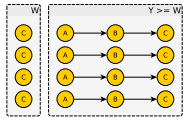
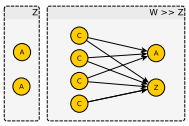
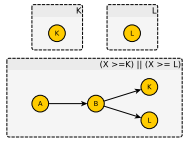
	Name	Description	Illustration
^	replication	The graph on the left hand side of the operator is replicated a number of times specified by the integer on the right hand side.	
>=	pointwise composition	Each output of the left hand side is connected to a corresponding input on the right hand side. Additional links are created in a round-robin fashion if the numbers of output and inputs do not match.	
>>	bipartite composition	Each output of the left hand side is connected to every input on the right hand side.	
||	merge	The resulting graph is the union of both graphs on either sides.	

Fig. 3. Dryad connectors

d is handed through the *i-th* incoming edge connected to the upstream operator.

Although the choice of the datatype of the data d can be left to the user, the recommended format is the *tuple*, which can be defined as an *ordered set of fields*.

C. Processing Stages

An operator has two principal phases. The first phase is the *process* in which resides the logic executed on the data units as they arrive. The second phase is the *finish* which is triggered once all the entries have been received and processed. Users can define their own logic and functions for both of these phases. Listing 1 illustrates the notation of the *process* and *finish* procedures.

The framework transparently handles the sending and receiving of signaling messages between operators such as to indicate the beginning and the end of the stream of processed data.

The scheduling of input operators can take the underlying distributed filesystem into account and schedule these operators close to the node where the split of data is physically stored. We currently support data stored on the Hadoop FileSystem (HDFS)[5].

Example: Listing 1 describes a word counting job. Three operators are defined using the operator notation. The first, `TextInput`, reads a text from the filesystem and emits each line to the downstream operator. As it is an input source and has no upstream operator, its processing takes place in the `finish` procedure where the input data is read from the filesystem and sent to the downstream operators. The `WordParser` receives a list of words as parameter and for each line of text received, parses each word contained in the line and emits those which are present in the provided list. The

[5]Hadoop Distributed File System - http://hadoop.apache.org/hdfs/

```
TextInput() {
  process(input, datum) {}
  finish() {
    [for each line l in text chunk, emit l]
  }
}

WordParser(W: {w_0, ..., w_{m-1}}) {
  process(input, line) {
    for each word u in line:
     if(u ∈ W):
       emit(i: w_i = u, 1)
  }
}

Sum() {
  sum = 0
  process(input, datum) {
    sum += datum
  }
}
```

Listing 1. The example depicts the AROM operators involved in the word counting job. The *WordParser* receives an external list of words as input. Figure 4 represents the DAG associated with the job referencing these operators. *emit(input, datum)* is the function of the framework which sends the data d to the i-th output

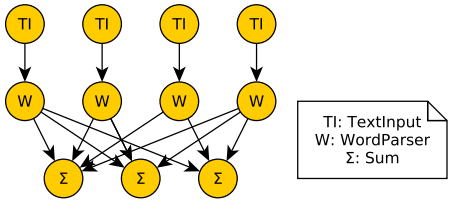


Fig. 4. The DFG representing the WordCount job in AROM. Parallelism is clear on this graph: any two vertices not linked by any edge are eligible for parallel execution. The code corresponding to each vertex is found in Listing 1.

last operator `Sum` then sums the occurrences for a given word (there should be as many sum operators as there are words in the list in this example).

Using the job definition notation, this job can be expressed as follows:

$$(TextInput >= Words(W))^k >> Sum^m$$

AROM provides a comprehensive API which allows to directly express the job using this notation once the operators are defined.

D. Enforcing Genericity and Reusability of Operators

Using paradigms from functional programming to define the operators used in AROM permits to develop generic and reusable operators. We show in Listing 2 an example of a generic filtering operator. The aim of this operator is to apply a user defined filtering function on the received data in order to determine if the data should be transmitted to the downstream operator. Using the functional programming paradigm, such an operator is simple to write: we only need to define the workflow of the operator and let the user define its own *predicate* function. The operator code also does not need to

```
Filter(predicate: D ↦ {1,0}) {
  process(input, datum) {
    if(predicate(datum)):
      emit(input, datum)
  }
}
Partition(func: D ↦ ℕ) {
  process(input, datum) {
    emit(func(datum) % nbOutput, datum)
  }
}
```

Listing 2. The example depicts generic partitioning and filtering operator in AROM. In the *Partition* operator, *func* is a user-provided function returning an integer based on the data (e.g. a hash function) where *D* is the set of all data elements. *nbOutputs* is provided by the framework. In the *Filter* operator, *predicate* is a user-provided function which performs a test on the input data and returns true or false depending on the result. Using the functional programming paradigm the operator is simply a higher-order function which calls the *func* and *predicate* functions. Anonymous function constructs can also be provided to the operator.

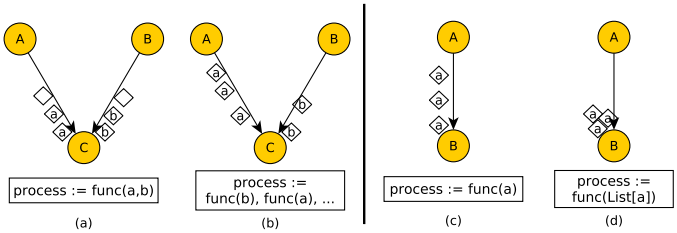


Fig. 5. Synchronous (a), asynchronous (b), scalar (c) and vector (d) operators.

handle the data types involved, it only needs the predicate function to return true or false depending on the input data. The same applies for the Partitioning operator.

In this way writing generic libraries is easy to achieve. The developers write the generic operators where only the logic concerning the workflow and the sequence of the functions are defined. The operators are then bundled into libraries and the users should only provide their own specific functions to the operator.

E. Operator Types

Different operator interfaces are available, each differing in their process cycle and in the way they handle the incoming data. Figure 5 illustrates these specificities.

a) Asynchronous/Synchronous: An *asynchronous* operator processes the entries as they come from the upstream operators, in a first-come, first-served basis. A *synchronous* operator on the other hand performs the processing only once there are entries present at each of its upstream inputs.

b) Vector/Scalar: A *vector* operator operates on batches of entries provided by the upstream operator. A *scalar* operator on the opposite operates on entries one at the time.

F. Job Scheduling and Execution

AROM is built around the master/slave architecture. The *master* node is the single entry point to the jobs while the *slaves* execute the operators. Upon receiving a job to be

executed, the Master node assigns individual operators or subgraphs to the slaves.

When an operator is scheduled on a slave, the master performs the following steps:

1) Select the slave node among those available.
2) Send the code of the operator to the elected slave.
3) Update all the slaves running a predecessor operator with the location of the new operator.

The scheduling of the operators is based on heuristics which take into account the synchronous/asynchronous nature of the operator, the source nature of the operators and the colocalization of the data with the physical host on which the slave is run. The scheduling uses a fixed capacity model which means that there is a maximum amount of operators which can be run simultaneously on the same slave.

On the data transmission between operators if the operators run on the same slave, they are located in separate threads of the JVM and the reference to the data is passed among them. If the operators are located on different physical slaves, data is streamed using sockets, and data in waiting to be sent is automatically stored on disk when it reaches a limit in memory. We use asynchronous I/O for the transmission of data between slaves for enhanced performance.

This means that in an optimal case where all the nodes of a pipelined job are scheduled, data is asynchronously passed between the operators and do not even need to be written to disk. Also, in the case of pipelined and iterative jobs, the operators of subsequent stages can be scheduled even if the previous stages are not entirely completed.

G. Specificities

AROM is implemented using Scala[6], a functional programming language which compiles to the Java JVM. The first characteristic enables the usage of anonymous and higher-order functions which makes the constructs of operator code less awkward than if they would have been written in Java. This was one of the users's complaints with FlumeJava [7]. Furthermore the fact that Scala binaries are targeted against the Java JVM enables to include Java code and reuse the existing libraries. Using the Scala language also enables the use of the powerful Scala collections API which comprises natural handling of tuples.

AROM is built around the actors model. On the lower level of the stack we use the Akka[7] actors framework which handles the serialization, the running, the supervision and the synchronization between the operators.

VI. Evaluation

We have implemented a MapReduce API which accepts jobs defined in MapReduce and translates them into AROM DFG jobs. The API fully recreates the shuffle and sort phases between the Map and the Reduce phases. In order to validate our assumptions concerning the performance of pipelined jobs we have defined a PageRank [18] computation job over the pages of the latest dump to this date of the English Wikipedia[8], representing a total of 35Gb raw XML formatted text. We have implemented two versions of the job: one which is defined in MapReduce and running against our AROM MapReduce API and one defined in AROM DFG and optimized for pipelining. Figures 6 & 7 summarize the topology of both jobs.

In both versions of the computation, a first job parses the content of the Wikipedia dump and extracts the links contained in each article. The results of this first job contain the initial PageRank values as well as all the links contained in each article. Each iteration is then implemented as an independant job which is run against the results produced at the previous iteration. In the MapReduce version of the job, the map phase emits, for each linked article, the PageRank of the original article and its number of links. The Reduce phase finally combines all these values and computes the PageRank for each article. The computation of the PageRank values happens once all the data related to an article is delivered. Results between each iteration is stored on HDFS and reloaded by the next iteration.

The pipelined AROM version of the iterations is optimized two times. First, it incrementally computes the PageRank value for a page as the data arrive to the operator. This leads to the possibility of scheduling and running the operators of the $n+1$th iteration even if the nth is not entirely complete. Then, during the first iteration the number of inbound articles for a given article is counted and passed along with the computation result to the downstream iterations. In this way, in addition to computing the PageRank as the values arrive, downstream iterations can directly send the result of the PageRank computation once all the related values have been received. Another difference is that intermediate data is not stored on the filesystem. After the last iteration a *Writer* operator is scheduled to write the output of the job back to HDFS. Values are partitioned at the output of each iteration operator.

The tests have been perfomed on a cluster of 12, 22 and 42 nodes equipped with 8-Core Intel processors, 16 Gb of memory and interconnected with Gigabit ethernet. On each setup, there is one node dedicated to the HDFS namenode and a node dedicated to the AROM master. The remaining nodes act as AROM workers and HDFS data nodes. The computation job is run over 5 iterations.

Figure 8 shows the processing times of the jobs. We can first observe that on 10 and 20 nodes the pipelined version of the job performs around 30% faster than its MapReduce conterpart. Overall, the system scales well from 10 to 40 nodes. On 40 nodes the performance gain on the pipelined job diminishes and both jobs perform similarly. We have also performed the hand-tuned MapReduce job on a Hadoop cluster. The results show that at this time Hadoop still outperforms our framework implementation.

[6]Scala - http://www.scala-lang.org
[7]Akka - http://akka.io
[8]http://en.wikipedia.org/wiki/Wikipedia:Database_download

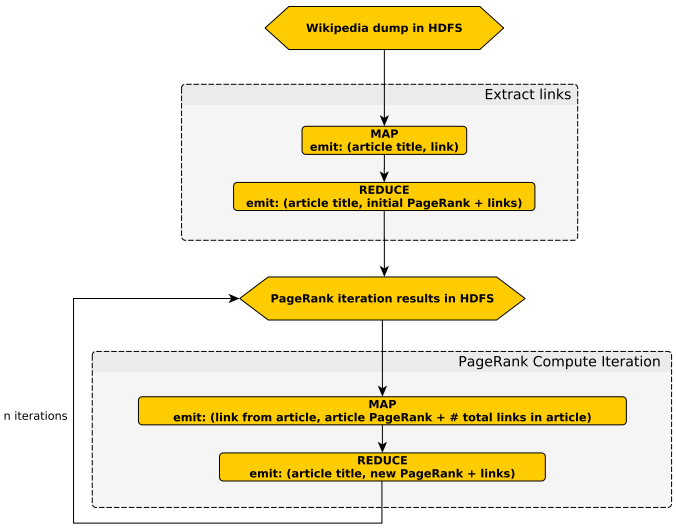


Fig. 6. The MapReduce job which computes PageRank for each Wikipedia article stored in HDFS.

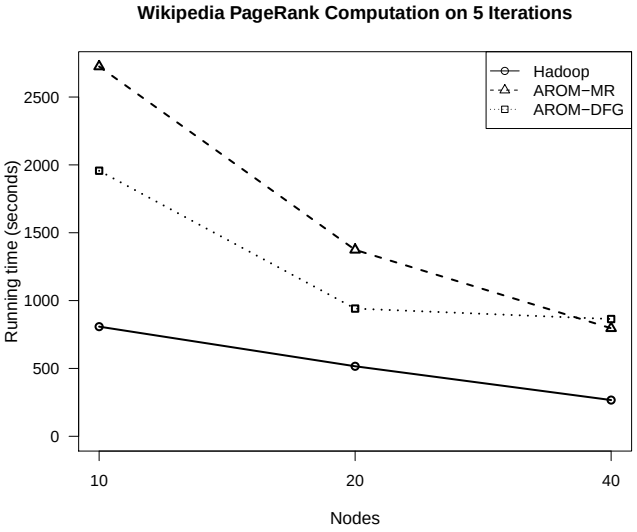


Fig. 8. Results of the PageRank calculation job on Hadoop, AROM-MR and AROM-DFG.

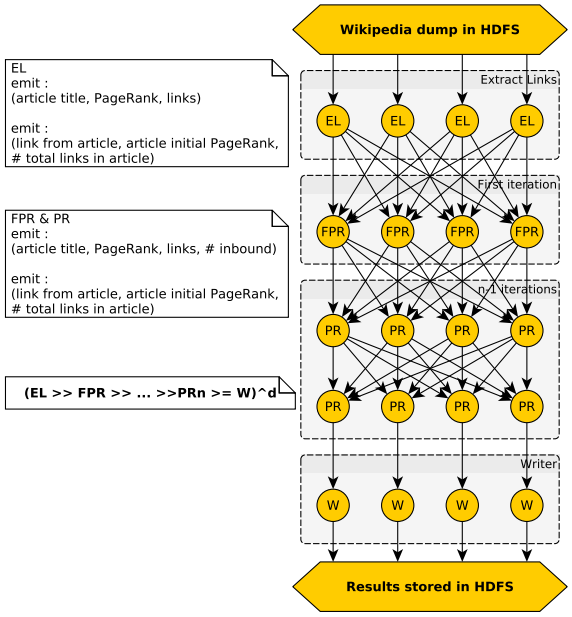


Fig. 7. The AROM version of the PageRank computation job. Every output on the EL and PR operators is partitioned on the first field which contains the title of the Wikipedia article in order to determine which downstream operator it will be sent to. *d* is the parallelization factor of the job (= 4 on this example).

VII. Discussion

In the pipelined job defined for the AROM framework, the results show better performance in part thanks to the fact that intermediate data can be directly streamed to the downstream operators and do not need to return into the DFS. This however can lead to failover issues, as the storage of intermediate data in the DFS can provide a form of checkpointing throughout the run. A solution could be to provide a checkpointing operator which receives the data from the upstream operator then transparently and asynchronously persists it on a filesystem or a NoSQL object-oriented database and finally hands it to the downstream operator. The advantage of this solution is that the user is able to define where the checkpoints are needed in the job.

We have shown that as the MapReduce model is in fact a restriction of the more general DFG model, it is possible to run jobs designed in MapReduce on a DFG framework. In that way, as the DFG model sits at a lower level than the MapReduce model, compiling jobs from a higher level language to a DFG framework offers more opportunities for optimization. Indeed, the MapReduce Map and Reduce plans are more difficult to tweak and lead to more complicated structures when used as target compilers for higher level languages, such as Pig. We have already internally developed a compiler for Pig targeting AROM as a proof of concept.

Also, we have shown that using DFG for defining the job leaves more possibility for optimization. Compared to the MapReduce version, it was possible to implement at least two optimizations. A third possible optimization would be to directly propagate the count of the inbound articles from the first iteration to all the following iterations in order to transmit the results of the computation earlier for each stage. Even if the hand-optimized Hadoop job still outperforms our results, they however show that our framework proposes a good compromise between scalability, performance and flexibility.

VIII. Future Works

One of the limitations of our current implementation is caused by our stack which prevents slaves to run more than two operators at the same time. We also suspect the communication stack to not be suitable for large scale pipelining topologies as we have identified possible contention issues at 40 nodes. We believe these facts are currently the limiting

factors in our performance against Hadoop. One of our first priorities is to migrate our implementation to a more scalable stack. In this way, we are currently evaluating the use of Akka 2.0[9] and highly scalable message passing services such as ZeroMQ[10] in our framework.

One of the future step of AROM is to further develop the scheduling. The current scheduling model is based on costs which are derived empirically and from heuristics. We also plan on adding speculative executions scheduling [2] in order to minimize the impact of an operator failure on performance. Similar to MapReduce, this speculative scheduling would identify operators that take longer time than other similar operators based on the measure of the average completion time. The operator and its share of data would then be preventively rescheduled in parallel on another host.

The scheduler should also be able to react to additional resources available in the pool of workers and extend or reduce the capacity of the scheduling in scenarios of scaling out and scaling in. This also implies that in the future the DFG execution plan should be modifiable at runtime.

IX. Conclusion

In this paper we have presented AROM, our implementation of a distributed parallel processing framework using directed acyclic graphs for the definition of jobs. The framework is based on the general DFG processing model where each node represents a program which can be run and scheduled in parallel. Paradigms from functional programming are used for the definition of operators which enables to easily define generic and reusable operators. The primary goal of the framework is to provide a playground for testing and developing on distributed parallel processing. By performing a test comparing the performance of a PageRank computation job using the MapReduce model and the DFG model, we have shown that our framework is able to scale on a large number of machines. The results also show that the pipelined job is more flexible in its definition and more open for optimization. Compared to the Hadoop implementation of MapReduce, our framework proposes a consistent tradeoff between performance, scalability and flexibility.

Acknowledgment

The cluster used for the tests has been made available by the CPER MISN TALC project with the support of the Région Lorraine.

References

[1] W. Hillis and G. Steele Jr, "Data parallel algorithms," *Communications of the ACM*, vol. 29, no. 12, pp. 1170–1183, 1986.

[2] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, pp. 107–113, Jan. 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[3] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian, "A comparison of join algorithms for log processing in mapreduce," in *Proceedings of the 2010 international conference on Management of data*, ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 975–986. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807273

[4] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," *SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 59–72, March 2007. [Online]. Available: http://doi.acm.org/10.1145/1272998.1273005

[5] 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*, ser. SIGMOD '08. New York, NY, USA: ACM, 2008, pp. 1099–1110. [Online]. Available: http://doi.acm.org/10.1145/1376616.1376726

[6] J. Salo, T. Aaltonen, and T. Mikkonen, "Mashreduce: Server-side mashups for mobile devices," in *Proceedings of the 6th international conference on Advances in grid and pervasive computing*, ser. GPC'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 168–177. [Online]. Available: http://dl.acm.org/citation.cfm?id=2008928.2008952

[7] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum, "Flumejava: easy, efficient data-parallel pipelines," *SIGPLAN Not.*, vol. 45, no. 6, pp. 363–375, Jun. 2010. [Online]. Available: http://doi.acm.org/10.1145/1809028.1806638

[8] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan, "Interpreting the data: Parallel analysis with sawzall," *Sci. Program.*, vol. 13, pp. 277–298, October 2005. [Online]. Available: http://dl.acm.org/citation.cfm?id=1239655.1239658

[9] H.-c. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker, "Map-reduce-merge: simplified relational data processing on large clusters," in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '07. New York, NY, USA: ACM, 2007, pp. 1029–1040. [Online]. Available: http://doi.acm.org/10.1145/1247480.1247602

[10] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey, "Dryadlinq: a system for general-purpose distributed data-parallel computing using a high-level language," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 1–14. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855742

[11] M. Isard and Y. Yu, "Distributed data-parallel computing using a high-level programming language," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, ser. SIGMOD '09. New York, NY, USA: ACM, 2009, pp. 987–994. [Online]. Available: http://doi.acm.org/10.1145/1559845.1559962

[12] F. McSherry, R. Isaacs, M. Isard, and D. G. Murray, "Naiad: The animating spirit of rivers and streams," Poster session of the 23rd ACM Symposium on Operating Systems Principles, 2011.

[13] D. Warneke and O. Kao, "Exploiting dynamic resource allocation for efficient parallel data processing in the cloud," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, pp. 985–997, 2011.

[14] A. Alexandrov, S. Ewen, M. Heimel, F. Hueske, O. Kao, V. Markl, E. Nijkamp, and D. Warneke, "MapReduce and PACT - comparing data parallel programming models," in *Proceedings of the 14th Conference on Database Systems for Business, Technology, and Web (BTW)*, ser. BTW 2011. Bonn, Germany: GI, 2011, pp. 25–44.

[15] 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*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863103.1863113

[16] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand, "Ciel: a universal execution engine for distributed data-flow computing," in *in Proceedings of the 8th USENIX Symposium on Networked System Design and Implementation (NSDI). USENIX*.

[17] D. J. DeWitt, E. Paulson, E. Robinson, J. Naughton, J. Royalty, S. Shankar, and A. Krioukov, "Clustera: an integrated computation and data management system," *Proc. VLDB Endow.*, vol. 1, no. 1, pp. 28–41, Aug. 2008. [Online]. Available: http://dx.doi.org/10.1145/1453856.1453865

[18] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Technical Report 1999-66, November 1999, previous number = SIDL-WP-1999-0120. [Online]. Available: http://ilpubs.stanford.edu:8090/422/

[9] Akka 2.0 released - http://akka.io/news/2012/03/06/akka-20-released.html

[10] ZeroMQ - http://www.zeromq.org/