The International Journal of Parallel, Emergent and Distributed Systems
Vol. 00, No. 00, December 2017, 1–24

RESEARCH ARTICLE

Programming Models and Systems for Big Data Analysis

Loris Belcastro, Fabrizio Marozzo*, and Domenico Talia

DIMES, University of Calabria, Italy
[lbelcastro, fmarozzo, talia]@dimes.unical.it;

(*Received October 2017*)

Big Data analysis refers to advanced and efficient data mining and machine learning techniques applied to large amount of data. Research work and results in the area of Big Data analysis are continuously rising, and more and more new and efficient architectures, programming models, systems, and data mining algorithms are proposed.

Taking into account the most popular programming models for Big Data analysis (MapReduce, Directed Acyclic Graph, Message Passing, Bulk Synchronous Parallel, Workflow and SQL-like), we analyzed the features of the main systems implementing them. Such systems are compared using four classification criteria (i.e., level of abstraction, type of parallelism, infrastructure scale and classes of applications) for helping developers and users to identify and select the best solution according to their skills, hardware availability, productivity and application needs.

Keywords: Parallel Programming models; Programming systems; Big data analysis; Cloud computing; Programming frameworks; MapReduce; Directed Acyclic Graph; Message Passing; Bulk Synchronous Parallel; Workflow; SQL-like.

1. Introduction

In the last years the ability to produce and gather data has increased exponentially. In fact, in the Internet of Things' era, huge amounts of digital data are generated by and collected from several sources, such as sensors, cams, in-vehicle infotainment, smart meters, mobile devices, GPS devices, web applications and services. The huge amount of data generated, the speed at which it is produced, and its heterogeneity in terms of format (e.g., video, text, XML, email), represent a challenge to the current storage, process and analysis capabilities. For instance, thanks to the growth of social networks and the widespread diffusion of mobile phones every day millions of people access social network services and share information about their interests and activities. Those data volumes, commonly referred as Big Data, can be exploited to extract useful information and produce valuable knowledge for science [1], economy [2], health [3] and society [4].

Although nowadays the term Big Data is often misused, it is very important in computer science for understanding business and human activities. In fact, Big Data is not only characterized by the large size of datasets, but also by the complexity, by the variety, and by the velocity of data that can be collected and pro-

*Corresponding author. Email: fmarozzo@dimes.unical.it

[0]This is an Accepted Manuscript of an article published by Taylor & Francis Group in the International Journal of Parallel, Emergent & Distributed Systems on 26/12/2017, available online: http://www.tandfonline.com/ http://dx.doi.org/10.1080/17445760.2017.1422501

ISSN: 1023-6198 print/ISSN 1563-5120 online
© 2017 Taylor & Francis
DOI: 10.1080/17445760.2017.1422501
http://www.informaworld.com

cessed [5]. So, we can collect huge amounts of digital data from sources, at a very high rate that the volume of data is overwhelming our ability to make use of it.

To extract value from such kind of data, novel architectures, programming models and systems have been developed for capturing and analyzing complex and/or high velocity data. In this scenario data mining raised in the last decades as a research and technology field that provides several different techniques and algorithms for the automatic analysis of large datasets. The usage of sequential data mining algorithms for analyzing large volumes of data requires a very long time for extracting useful models and patterns. For this reason, high performance computers, such as many and multi-core systems, Clouds, and multi-clusters, paired with parallel and distributed algorithms are commonly used by data analysts to tackle Big Data issues and get valuable information and knowledge in a reasonable time [6].

This paper addresses the main issues in the area of programming models and systems for Big Data analysis. Taking into account the most popular programming models for Big Data analysis (MapReduce, Directed Acyclic Graph, Message Passing, Bulk Synchronous Parallel, Workflow and SQL-like), we analyzed the features of the main systems implementing them. Such systems are compared using four criteria for assessing their suitability for parallel programming: *i)* *level of abstraction* that refers the programming capabilities of hiding low-level details of a system; *ii)* *type of parallelism* that describes the way in which a system allows to express parallel operations; *iii)* *infrastructure scale* that refers to the capability of a system to efficiently execute applications taking advantage from the infrastructure size; and *iv)* *classes of applications* that describes the most common application domain of a system.

The final aim of this work is to help developers identifying and selecting the best solution according to their skills, hardware availability and application needs.

The structure of the paper is as follows. Section 2 provides a description of the main requirements of Big Data programming systems and a conceptual description of the four criteria used to classify them. Section 3 describes the most widespread programming models for analyzing Big Data and compares the main systems implementing them according to the classification criteria we defined. Finally, Section 4 concludes the paper.

2. Requirements and classification criteria

This section discusses the main requirements of Big Data programming systems and describes the four classification criteria used to classify them. To cope with the need of processing large amount of data, a programming system should meet the following requirements:

- *Efficient data management and exchange.* Big Data sets are often arranged by gathering data from several heterogeneous and sometimes not well-known sources. In this context, programming systems must support efficient protocols for data transfers and for communications as well as they have to enable local computation of data sources and fusion mechanisms to compose the results produced in distributed nodes.
- *Interoperability.* It is a main issue in large-scale applications that use resources such as data and computing nodes. Programming systems for Big Data should support interoperability by allowing the use of different data formats and tools. The *National Institute of Standards and Technology* (NIST) just released the Big

Data interoperability framework[1], a collection of documents aiming at defining some standards for Big Data.

- *Efficient parallel computation.* An effective approach for analyzing large volumes of data and obtaining results in a reasonable time is based on the exploitation of inherent parallelism of the most data analysis and mining algorithms. Thus, programming systems for Big Data analysis have to allow parallel data processing and provide a way to easy monitor and tune the degree of parallelism.
- *Scalability.* With the exponential increases in the volume of data to be processed, programming systems for Big Data analysis must accommodate rapid changes in the growth of data, either in traffic or volume, by exploiting the increment of computational or storage resources efficiently.

Starting from these requirements we classified the programming systems for Big Data analysis according to the following four criteria:

1. Level of abstraction The level of abstraction of a system refers its programming capabilities to hide the low-level details of a solution (e.g., a function, a data structure, a communication protocol). In this way, developers can focus on their problem logic, without the need of implementing it by scratch. An high-level of abstraction makes it easy to build applications but hard to compile them to efficient code. Whereas a low-level of abstraction makes it hard to build applications but easy to implement them efficiently [7]. We use these requirements as a metric for classifying and assessing Big Data analysis programming systems. For comparison purposes, we distinguish three levels of abstraction:

- *Low*, when a programmer can exploit low-level APIs, mechanisms and instructions which are powerful but not trivial to use. A greater development effort is required with respect to systems providing a higher level of abstraction, but the code efficiency is very high because it can be fully tuned. It also requires a low-level understanding of the system, including working with files on distributed environments [8]. At this level, productivity of programmers is poor, whereas program performance can be effective.
- *Medium*, when a programmer defines an application as a script or a visual representation of the program code, hiding the low-level details that are not fundamental for application design. It requires a medium development effort and code tuning capabilities.
- *High*, when developers, also with low programming skills, can build applications using high-level interfaces, such as visual IDEs or abstract models with high-level constructs not related to the running architecture. At this level, program development effort is low as well as the code efficiency at run time because executable generation is harder and code mapping is not direct.

2. Type of parallelism The type of parallelism describes the way in which a programming model or system expresses parallel operations and how its runtime supports the execution of concurrent operations on multiple nodes or processors. For comparison purposes, we distinguish three types of parallelism:

- *Data parallelism*: it is achieved when the same code is executed in parallel on different data elements. Data parallelism is also known as SIMD (Single Instruction

[1]http://www.nist.gov/itl/bigdata/bigdatainfo.cfm

Multiple Data), which is a class in the Flynn's taxonomy for classifying parallel computation [9].

- *Task parallelism*: it is achieved when different tasks that compose applications run in parallel. The presence of data dependencies can limit the benefits of this kind of parallelism. Such parallelism can be defined in two manners: *i*) *explicit*, a programmer defines dependencies among tasks through explicit instructions; *ii*) *implicit*, the system analyzes the input/output of tasks to understand dependencies among them.
- *Pipeline parallelism*: it is obtained when data is processed in parallel at different stages, so as that the (partial) output of a task is passed to the next task to be processed. Pipeline parallelism is appropriate for processing data streams as their stages manage the flow of data in parallel. Because of its features, pipeline parallelism can be considered a specialization of both data and task parallelism [10].

3. Infrastructure scale This feature refers to the capability of a programming system to efficiently execute applications on a infrastructure of a given size (i.e., number of computation nodes). Some systems are designed to be used on infrastructures with a small number of nodes, while others are able to scale up to a large number of nodes. For comparison purposes, we distinguish three scales of infrastructures:

- *Small*: it refers to a small enterprise cluster or Cloud platforms with up to hundreds of computational nodes.
- *Medium*: it identifies a medium enterprise cluster consisting of up to thousands of nodes.
- *Large*: it refers to large HPC environments or high-level Cloud services with up to ten thousands of nodes.

4. Classes of applications Choosing the right programming solution for developing a Big Data analysis application is not easy, since several programming systems, libraries and languages are available today. Some solutions can be efficiently used in a specific field (e.g., stream data processing, data querying, machine learning), while others are more general so as to be used for different classes of applications.

3. Programming models

This section presents and discusses the most popular programming models for Big Data analysis and their associated languages and libraries. They are MapReduce, Directed Acyclic Graph (DAG), Message Passing, Bulk Synchronous Parallel (BSP), Workflow and SQL-like. The goal is to highlight the features, issues and benefits of each programming model and systems implementing it. A textual box at the end of each section highlights how a system is classified according to the four criteria defined in the previous section.

3.1 MapReduce

MapReduce is a programming model developed by Google [11] in 2004 for large-scale data processing to cope efficiently with the challenge of processing Big Data. The MapReduce model is inspired by the *map* and *reduce* functions commonly used in functional programming, however it was mainly designed for allowing designers to implement distributed applications based on the *map* and *reduce* operations [12]. The *map* function processes a (key, value) pair and returns a list of intermediate (key, value) pairs:

map (k1,v1) $\rightarrow$ list(k2,v2).

The *reduce* function merges all intermediate values having the same intermediate key:

reduce (k2, list(v2)) $\rightarrow$ list(v3).

In general, the whole transformation process performed in a MapReduce application can be described through the following steps (see Figure 1):

(1) A master process receives a job descriptor which specifies the MapReduce job to be executed. The job descriptor contains, among other information, the location of the input data, which may be accessed using a distributed file system.
(2) According to the job descriptor, the master starts a number of mapper and reducer processes on different machines. At the same time, it starts a process that reads the input data from its location, partitions that data into a set of splits, and distributes those splits to the various mappers.
(3) After receiving its data partition, each mapper process executes the *map* function (provided as part of the job descriptor) to generate a list of intermediate key/value pairs. Those pairs are then grouped on the basis of their keys.
(4) All pairs with the same keys are assigned to the same reducer process. Hence, each reducer process executes the *reduce* function (defined by the job descriptor) which merges all the values associated to the same key to generate a possibly smaller set of values.
(5) The results generated by each reducer process are then collected and delivered to a location specified by the job descriptor, so as to form the final output data.

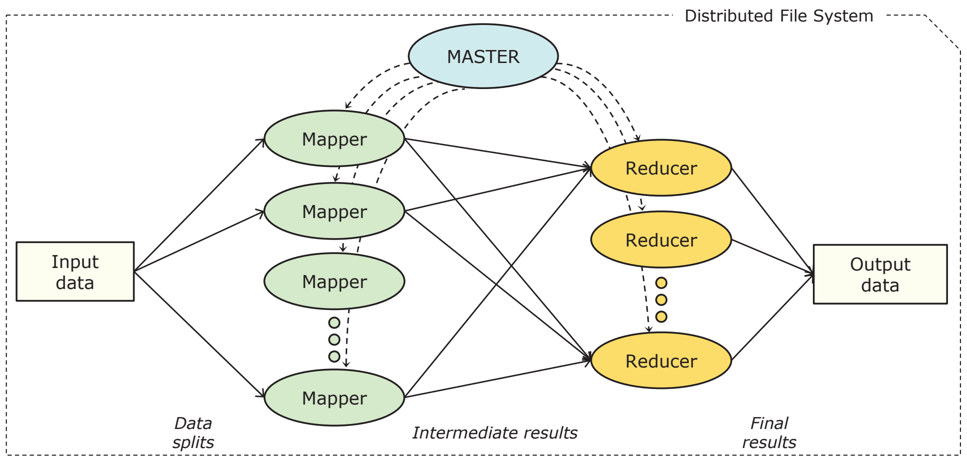

Figure 1. MapReduce execution flow.

MapReduce has been designed to be used in a wide range of domains, including data mining and machine learning, social media analysis, financial analysis, image retrieval and processing, scientific simulation, web site crawling, machine translation, and bioinformatics. Nowadays, MapReduce is considered as one of the most important parallel programming models for distributed environments. It is supported by the leading IT companies, such as Google, Amazon[1], Microsoft[2] and IBM[3], or by private Cloud infrastructures such as OpenStack[4].

3.1.1 Apache Hadoop

Apache Hadoop[1] is the most used open source MapReduce implementation. It can be adopted for developing distributed and parallel applications using many programming languages. Hadoop relieves developers from having to deal with classical distributed computing issues, such as load balancing, fault tolerance, data locality, and network bandwidth saving.

The Hadoop project is not only about the MapReduce programming model (*Hadoop MapReduce* module), as it includes other modules such as:

- *Hadoop Distributed File System (HDFS)*: a distributed file system providing fault tolerance with automatic recovery, portability across heterogeneous commodity hardware and operating systems, high-throughput access and data reliability.
- *Hadoop YARN*: a framework for cluster resource management and job scheduling.
- *Hadoop Common*: common utilities that support the other Hadoop modules.

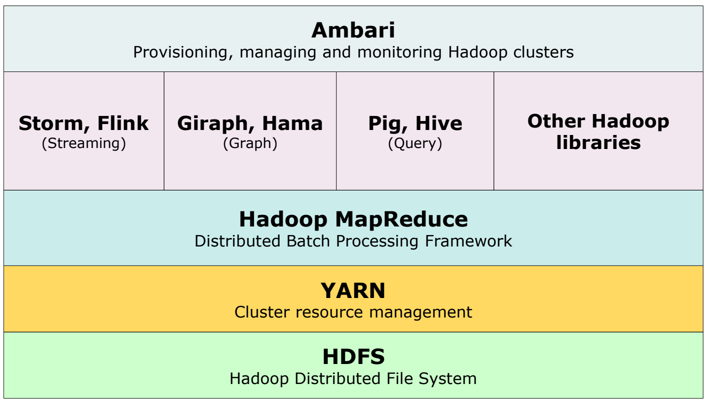
Ambari
Provisioning, managing and monitoring Hadoop clusters
Storm, Flink
(Streaming)
Giraph, Hama
(Graph)
Pig, Hive
(Query)
Other Hadoop libraries
Hadoop MapReduce
Distributed Batch Processing Framework
YARN
Cluster resource management
HDFS
Hadoop Distributed File System

Figure 2. Hadoop software stack.

In particular, thanks to the introduction of YARN in 2013, Hadoop turns from a batch processing solution into a platform for running a large variety of data applications, such as streaming, in-memory, and graphs analysis. As a result, Hadoop has become a reference for several other programming systems, such as: *Storm* and *Flink* for streaming data analysis; *Giraph* and *Hama* for graph analysis; *Pig* and *Hive* for querying large datasets; *Oozie*, for managing Hadoop jobs; *Ambari* for provisioning, managing, and monitoring Apache Hadoop clusters. An overview of the Hadoop software stack is shown in Figure 2.

Apache Hadoop provides a *low-level of abstraction*, because a programmer can define an application using APIs which are powerful but not easy to use because they are related to the computing infrastructure. It also requires a low-level understanding of the system and the execution environment for dealing with issues related to file systems, networked computers and distributed pro-

[1]https://aws.amazon.com/elasticmapreduce/
[2]https://azure.microsoft.com/services/hdinsight/
[3]https://www.ibm.com/analytics/us/en/technology/hadoop/
[4]https://wiki.openstack.org/wiki/Sahara
[1]https://hadoop.apache.org/

gramming [8]. Developing an application based on Hadoop requires more lines of code and development effort when compared to systems providing a greater level of abstraction (e.g., Hive or Pig), but the code efficiency is higher because it can be fully tuned.

Hadoop is designed for exploiting *data parallelism* during map/reduce steps. Input data is partitioned into chunks and processed by different computing nodes in parallel. It is designed to process very large amounts of data in *large-scale infrastructures* with up to tens of thousands of machines. In fact, Hadoop is used by most of the leading IT companies, such as Yahoo!, IBM, and Amazon.

Hadoop is a *general-purpose* system that enables large-scale data processing over a set of distributed nodes. It is widely used to develop iterative batch applications using many programming languages, such as Java, C, C++, Ruby, Groovy, Perl, Python.

3.2 Directed Acyclic Graph (DAG)

Directed Acyclic Graph (DAG) is an effective paradigm to model complex data analysis processes, such as data mining applications, which can be efficiently executed on distributed computing systems such as a Cloud platform. A DAG consists of a finite set of edges and vertices, with each edge directed from one vertex to another. DAGs are very close to workflows (see Section 3.5), but they do not include cycles (i.e., circular dependencies among vertices). DAGs can easily model many different kinds of applications, where the input, output, and tasks of an applications depend on other tasks (see Figure 3). The tasks of a DAG application and their dependencies can be defined using two alternative ways:

- *Explicitly*, a programmer defines dependencies among tasks through explicit instructions (e.g., T_2 depends on T_1);
- *Implicitly*, the system analyzes the input/output of tasks to understand dependencies among them (e.g., T_2 reads the input O_1, which is an output of T_1);.

DAG tasks can be composed together following a number of different patterns (e.g., sequences, parallel constructs), whose variety helps designers addressing the needs of a wide range of application scenarios.

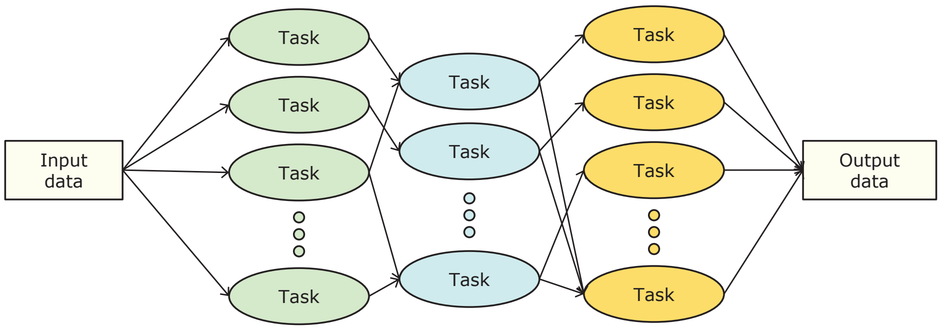

Figure 3. DAG execution flow.

3.2.1 Apache Spark

Apache Spark[1] is another top project of Apache Software Foundation for Big Data analysis. Differently from Hadoop, in which intermediate data are always stored in distributed file systems, Spark stores data in RAM memory and queries it repeatedly so as to obtain better performance for some classes of applications compared to Hadoop (e.g., iterative machine learning algorithms) [13]. A Spark application in defined as a set of independent stages running on a pool of worker nodes. A *stage* is a set of tasks executing the same code on different partitions of input data.

Spark and Hadoop are considered the leading open source Big Data systems and thus are supported by every major Cloud providers. As shown in Figure 4, different libraries have been built on top of Spark: *Spark SQL* for dealing with SQL and Data Frames, *MLlib* for machine learning, *GraphX* for graph-parallel computation, *Spark Streaming* for building streaming applications. The execution of a generic Spark application on a cluster is driven by a central coordinator (i.e., the main process of the application), which can connect with different cluster managers, such as *Apache Mesos*[1], *YARN*, or *Spark Standalone* (i.e., a cluster manager available as part of the Spark distribution). *Ambari* can be used for provisioning, managing, and monitoring Spark clusters.

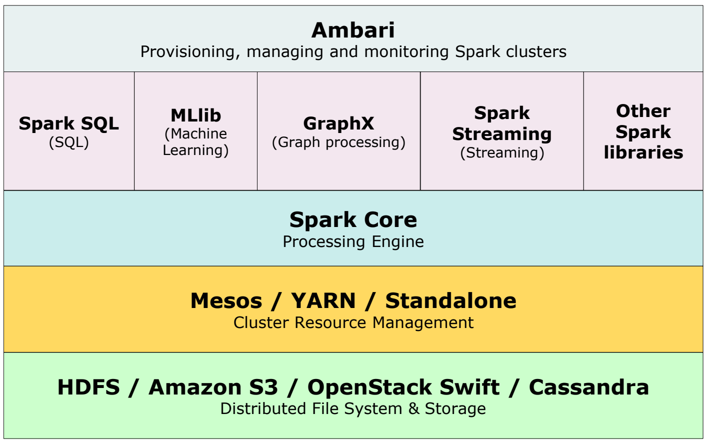
Ambari
Provisioning, managing and monitoring Spark clusters
Spark SQL
(SQL)
MLlib
(Machine Learning)
GraphX
(Graph processing)
Spark Streaming
(Streaming)
Other Spark libraries
Spark Core
Processing Engine
Mesos / YARN / Standalone
Cluster Resource Management
HDFS / Amazon S3 / OpenStack Swift / Cassandra
Distributed File System & Storage

Figure 4. Spark software stack.

Even though in some classes of applications Spark is considered a better alternative to Hadoop, in many others it has limitations that make it complementary to Hadoop. The main limitation of Spark is that datasets should fit in RAM memory. In addition, it does not provide its own distributed storage system, which is a fundamental requirement for Big Data applications. To overcome this lack, Spark has been designed to run on top of several data sources, such as distributed file systems (e.g., *HDFS*), Cloud object storages (e.g., *Amazon S3*, *OpenStack Swift*) and NoSQL databases (e.g., *Cassandra*).

Spark's real-time processing capability is increasingly being used into applications that requires to extract insights quickly from data, such as recommendation and monitoring systems. For this reason, several big companies exploit Spark for data analysis purpose: SK Telecom analyzes mobile usage patterns of customers, Ebay uses Spark for log aggregation, and Kelkoo for product recommendations.

[1]https://spark.apache.org
[1]http://mesos.apache.org/

Apache Spark provides a *low-level of abstraction*, because a programmer can define an application using APIs which are powerful but require high programming skills. With respect to Hadoop, developing an application based on Spark requires smaller number of lines of code, especially when *Scala* is used as programming language. In fact, Spark provides some built-in operators (e.g., *flat*, *flatMap*, *saveAsTextFile*, *reduceByKey*, *groupByKey*) that make easier to code a parallel application.

Spark is designed to exploit *data parallelism* into stages. Input data is divided in chunks and processed in parallel by different computing nodes. It supports also *task parallelism*, when independent stages of the same application are executed in parallel. Spark is designed to process very large amounts of data in *large-scale infrastructures* with up to tens of thousands of nodes. Several big companies and organizations uses Spark in production, like eBay, Amazon, Alibaba.

Spark is a *general-purpose* distributed computing system for large-scale data processing. It is commonly used to develop in-memory iterative batch applications using many programming languages (e.g., Java, Scala, Python, R). Many powerful libraries are built on top of Spark: *MLlib* for machine learning, *GraphX* for graph-parallel computation, *Spark Streaming* for stream processing.

3.2.2 Apache Storm

Apache Storm[1] is an open source system for real-time stream processing of large volumes of data. Storm is designed to ensure a high degree of scalability, fault-tolerance, high-speed data processing and low-latency response time.

The programming paradigm proposed by Storm is quite simple and it is based on four abstractions:

- *Stream*: it represents an unbounded sequence of tuples, which is created or processed in parallel. Storm provides some standard serializers for creating streams (e.g., integer, doubles, string), but programmers are free to create custom ones.
- *Spout*: it is the data source of a stream. Data is read from different external sources, such as social network APIs, sensor network, queueing systems (e.g., JMS, Kafka, Redis), and then is input to the application.
- *Bolt*: it represents the processing entity. Specifically, it can execute any type of tasks or algorithms (e.g., data cleaning, functions, joins, queries).
- *Topology*: it represents a job. A generic topology is created as a DAG, where spouts and bolts represents the graph vertices and streams act as the graph edges. A topology runs forever until it is stopped.

Differently from other systems, such as Hadoop, Storm is stateless and uses a “at least once” processing semantic, which ensures all the messages will be processed, but some of them should be processed more than once (e.g., in case of system failure). If developers need to implement a stateful operation or to use a “only one” processing semantic, they could use the Storm coupled with the *Trident* library.

Storm today is used by leading IT companies, such as Twitter that uses it to process many terabytes of data flows a day, for filtering and aggregating contents or for applying machine learning algorithms on stream data. However, due to its user friendly features, Storm can be adopted by small-medium companies for business purposes (e.g., real-time customer services, security analytics, and threat detec-

[1]https://storm.apache.org

tion).

Apache Storm provides a *medium-level of abstraction*, because a programmer can easily define an application by using spouts, streams and bolts. Its APIs allow to test an application in local-mode, without having to use a cluster.

Storm supports *data parallelism* (when many threads executes in parallel the same code on different chunks), *task parallelism* (when different spouts and bolts run in parallel) and *pipeline parallelism* (for data stream propagation in a topology). Storm has been designed to process very large amounts of data in *large-scale infrastructures* with up to tens of thousands of nodes. Storm is used in production by several big companies like Groupon, Twitter and Spotify.

Storm is commonly used for *real-time stream processing*. An applications based on Storm can be written using Java, Clojure or any other programming languages by exploiting the *Storm Multi-Language Protocol*.

3.2.3 Apache Flink

Apache Flink[1] is an open source stream processing system for large volumes of data. Flink allows programmers to implement distributed, high-performing and high-available data streaming applications. It provides a streaming dataflow paradigm for processing event-at-a-time, rather than as a series of batch of events, on both finite and infinite datasets. The programming paradigm is quite simple and it is based on three abstractions:

- *Data source*: it represents the incoming data that Flink processes in parallel.
- *Transformation*: it represents the processing entity, when incoming data is modified.
- *Data sink*: it represents the output of a Flink task.

The core of Flink is a distributed streaming dataflow runtime, which is alternative to that provided by Hadoop MapReduce. However, despite having its own runtime Flink can work on a cluster or Cloud infrastructure managed by YARN and access data on HDFS. Flink provides programmers with a series of APIs: *DataStream API* and *Dataset API* for transformations respectively on data streams and datasets; *Table API* for relational stream and batch processing; *Streaming SQL API* for enabling SQL queries on streaming and batch tables.

Flink today is used by important IT companies, such as Alibaba that uses it to optimize search rankings in real time, Bouygues Telecom that processes 10 billion raw events per day, or Zalando that uses it for real-time process monitoring and data management. Anyhow, due to its user friendly features, Flink can be used by small-medium companies for business purposes (e.g., real-time activity monitoring and alerting, content recommendations).

Apache Flink provides a *medium-level of abstraction*, because a programmer can easily define an application by using data sources, trasformations and data sinks. Its APIs allow programmers to manage finite and infinite sets of data, and different type of data sources like streams, datasets and relational tables.

Like Storm, Flink supports *data parallelism*, *task parallelism* and *pipeline parallelism*. Flink has been designed to process very large amounts of data in *large-scale infrastructures* with up to tens of thousands of nodes. Several big companies like Alibaba, Bouygues Telecom and Zalando use Flink in produc-

[1]https://flink.apache.org

tion.

Flink is mainly used for *real-time stream processing*. An applications based on Flink can be written using Java or Scala.

3.2.4 Azure Machine Learning

Microsoft Azure Machine Learning[1] (Azure ML) is a SaaS that provides a Web-based development environment for creating and sharing machine learning workflows as DAGs. Through its user-friendly interface, data scientists and developers can perform several common data analysis and mining tasks and automate their workflows, without needing to buy any hardware/software nor manage virtual machine manually.

Using its drag-and-drop interface, users can import their data in the environment or use special readers to retrieve data form several sources, such as Web URL (HTTP), OData Web service, Azure Table, Azure Blob Storage, Azure SQL Database. After that, users can compose their data analysis workflows where each data processing task is represented as a block that can be connected with each other through direct edges, establishing specific dependency relationships among them. Azure ML includes a rich catalog of processing tools that can be easily included in a workflow to prepare/transform data or to mine data through supervised learning (regression e classification) or unsupervised learning (clustering) algorithms.

Users can easily visualize the execution results for finding very useful information about models accuracy, precision and recall. Finally, the built models can be shared as Web services for predicting new data or performing real time predictions. Thanks to the auto-scaling feature provided by Azure, users do not have to worry about scaling models if the usage is increased.

Azure ML provides a very *high-level of abstraction*, because a programmer can easily design and execute data analytics applications by using simple drag-and-drop web interface and exploiting many built-in tools for data manipulation and machine learning algorithms.

Azure ML supports *task parallelism* because independent tasks (represented as blocks) of the same data analysis application are executed in parallel. Azure ML is thought for providing data analysis services to customers (e.g., small/medium companies) that need *small virtualized infrastructures*. The amount of computational resources assigned by Azure ML to each customer is not visible, which makes the service very simple to use but does not provide any way to configure it.

Azure ML is commonly used for data analysis applications to support *predictive analytics* and *machine learning* applications. Most applications can be defined in a totally visual way without the need to use any programming language. Optionally, programmers can include their own custom scripts (e.g., in R or Python) to extend the tools and algorithms available in the catalog.

3.3 Message Passing

The message passing model is a well-known paradigm that provides the basic mechanisms for process-to-process communication in distributed computing systems where each processing element has its own private memory. In message passing the sending process composes the message containing the data to share with the receiving process(es) including a header specifying to which processing element and

[1]https://azure.microsoft.com/en-us/services/machine-learning/

process the data is to be routed, and sends it to the network. Once the message has been inserted in the communication network, generally the sending process continues its execution (see Figure 5). This kind of send is called a *non-blocking send*. The receiving process must be aware that it is expecting data. It indicates its readiness to receive a message by executing a *receive* operation. If the expected data has not yet arrived, the receiving process suspends (blocks) until it does.

Thus, the primitives of message passing model are basically two:

- *Send(destination, message)*: a process sends a *message* to another process identified as *destination*;
- *Receive(source, message)*: a process receives a *message* from another process identified as *source*.

Message passing is used in many programming languages, operating systems and library for supporting data communication. Among those, the Message Passing Interface (MPI) is the most used library for programming message passing applications on distributed-memory parallel systems. Although MPI library has been largely used as a general parallel programming model, recently it has been exploited for implementing Big Data applications with positive results in terms of programmability and scalability [14–16].

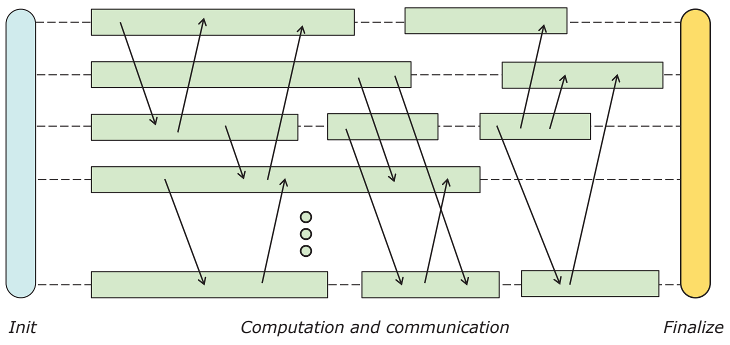

Figure 5. Message Passing execution flow.

3.3.1 MPI

As mentioned above, MPI [17] is a de-facto standard message-passing interface for parallel applications defined since 1992 by a forum composed of many industrial and academic organizations. MPI-1 was the first version of this message passing library that has been extended in 1997 by MPI-2 [18]. MPI-1 provided a rich set of messaging primitives (129), including point-to-point communication, broadcasting, barrier, reduce, and the ability to collect processes in groups and communicate only within each group. MPI has been implemented on massively parallel computers, workstation networks, clusters, Grids. Therefore MPI programs are portable on a very large set of parallel and sequential architectures.

An MPI-1 parallel program is composed of a set of similar processes running on different processors that use MPI functions for message passing. A single MPI process can be executed on each processor of a parallel computer and, according the SPMD (*Single Program Multiple Data*) model, all the MPI processes that compose a parallel program execute the same code on different data elements. Examples of MPI point-to-point communication primitives are:

- *MPI_Send(msg, leng, type, rank, tag, comm);*
- *MPI_Recv(msg, leng, type, source, tag, comm, &status);*

Group communication is implemented by the primitives:

- *MPI_Bcast (inbuf, incnt, intype, root, comm);*
- *MPI_Gather (outbuf, outcnt, outype, inbuf, incnt, intype, root, comm);*
- *MPI_Reduce (inbuf, outbuf, count, type, op, root, comm);*

For program initialize and termination the *MPI_Init* and *MPI_Finalize* functions are used. MPI provides a sort of low-level programming model, but it is widely used for its portability and its efficiency. Should be also mentioned that MPI-1 does not make any provision for process creation. However, in the MPI-2 version additional features have been provided for the implementation of *active messages*, *process startup*, and *dynamic process creation.*

MPI is becoming more and more the most used programming tool for message-passing parallel computers. However, it should be used as a an Esperanto for programming portable system-oriented software rather than for end-user parallel applications where higher level languages could simplify the programmer task in comparison with MPI. The most recent version of MPI, version 3, is under development with the objective to include also shared memory access inside a single processing element and the management of many-core systems with adaptivity and fine-grain parallelism.

MPI provides a *low-level of abstraction* for developing high performance parallel applications. Programmers can exploit only basic primitives without any high-level construct, and need to manually deal with complex issues, such as data exchanges, distribution of data across processors, synchronizations, and deadlocks.

MPI is designed for exploiting *data parallelism*, because all the MPI application's processes execute in parallel the same code on different data elements. It is used by academia and industry for running parallel applications in *medium-scale infrastructures.*

MPI is a *general-purpose* distributed memory system for parallel programming, which is commonly used for developing iterative parallel applications where nodes require data exchange and synchronization to proceed. A generic MPI program can be written by using APIs available for many programming languages (e.g., Java, Fortran, C, C++, Perl, Python).

3.4 Bulk Synchronous Parallel (BSP)

Bulk Synchronous Parallel (BSP) [19] is a parallel computation model in which computation is divided into a sequence of *supersteps* (see Figure 6). In each superstep the following operations can be performed:

(1) *Concurrent computation*: each processor performs computation using local data asynchronously;
(2) *Global communication*: the processes exchange data among them according to requests made during the local computation;
(3) *Barrier synchronization*: when a process reaches the barrier, it expects all other processes have reached the same barrier.

The communication and synchronization are completely decoupled, so as to guarantee that all the processes in a superstep are mutually independent. Moreover, this solution avoids problems due to synchronous message passing among processes (e.g. deadlocks). These features make the BSP programming model a very robust solution for developing scalable parallel applications.

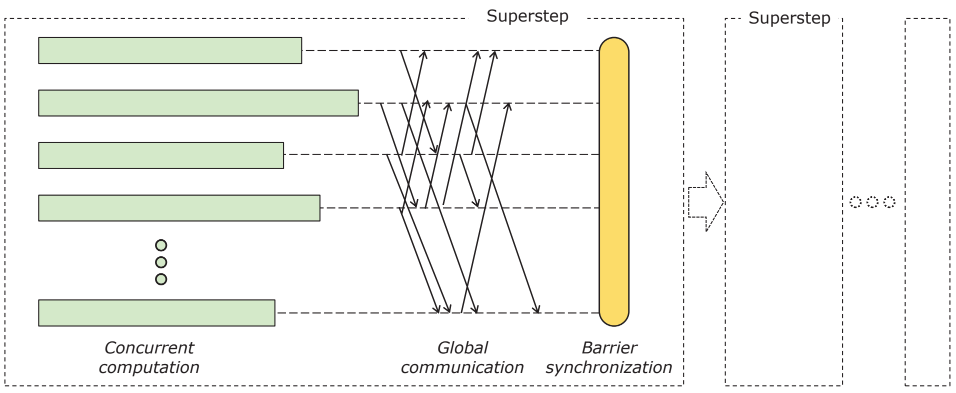

Figure 6. BSP execution flow.

3.4.1 Apache Giraph

Apache Giraph[1] is an iterative graph processing system built for developing high scalable applications. In 2012, when the first version of Giraph was released, it was considered the open source implementation of Pregel [20], a graph processing architecture developed at Google. Compared to Pregel, Giraph introduces several enhancements, such as master computation, sharded aggregators, edge-oriented input, out-of-core computation.

Giraph follows the BSP programming model and it is implemented using Hadoop as resource manager and Netty[1] for communication. Specifically, Giraph runs supersteps as a set of computations executed by map-only jobs. Each vertex executes a mapper task by performing the following operations: *i*) it receives messages sent to the vertex in the previous superstep; *ii*) it performs computation using the vertex data and received messages; *iii*) it sends messages to other connected vertices. After every vertex computation, synchronization is performed and Giraph prepares for the next superstep. The execution will halt when there are no more messages to process and all work is done.

Since Giraph runs map-only jobs, it improves performances by eliminating the reduce stage operations, which include data sorting and intermediate data storing (disk is accessed only for loading the initial graph data, writing check-points, and saving final results). Giraph is currently used at Facebook for processing very large graphs (i.e., trillion edge graphs) and for discovering relationships among users [21].

Apache Giraph provides a *low-level of abstraction*, because programmers have to implement different classes (e.g., *Reader*, *Writer*, *InputFormat*, *OutputFormat*) for each vertex of their graph applications. The code of a Giraph application is similar to MapReduce, since Giraph is a specialized programming system built on top of Hadoop.

Giraph runs map-only Hadoop jobs, so it supports *data parallelism* as Hadoop. Aiming at providing a more grained parallelism and a better CPUs usage, developers introduced the multi-threading support at worker node level. Giraph is mainly used by academia and small industry to run graph processing applications in *small infrastructures*.

Giraph is used for iterative *graph processing* applications (e.g., page rank and shared connections) according to the BSP model, which can be written using Java or Python.

[1]https://giraph.apache.org
[1]https://netty.io

3.4.2 Apache Hama

Apache Hama[2] is a Java system, based on the BSP programming model. The BSP model used by Hama allows tasks to communicate with each others through a message passing interface, with significant benefits in terms of performance. Differently from Giraph, which follows a vertex-centric programming model and depends on MapReduce paradigm, Hama uses a pure BSP engine, which makes it suitable for developing applications in many fields, such as graph computing, machine learning, complex query processing.

Differently from Hadoop, each Hama tasks do not have mandatory to sort, shuffle and reduce data: in such way, it does not have to store intermediate data on HDFS, leading to a significant lower overhead than an Hadoop. At the current state, Hama lacks of efficient algorithms for graph partition and of a more efficient system for retrieving data (e.g., message data) on the local file system [22].

Although Hama is a top project under the Apache Software Foundation since 2012, it is not still widely used, very likely because of several issues that need to be addressed yet. In fact, its community of contributors is actually small with regard to those ones of other popular systems (e.g., Hadoop or Spark) and the development is slow. However, Hama seems to be very promising: comparative studies demonstrated that Hama is able to provide higher performances compared to other related systems, such as Apache Giraph, for some types of applications [23], such as graph processing and intensive iterative applications.

Apache Hama provides a *low-level of abstraction*, because a programmer can define an application using low-level BSP primitives for computation and communication. Hama seems to be at an early stage of development, because it does not provide proper APIs (e.g., for input/output data, for data partitioning) or high-level operators that make easier to build parallel applications.

Hama supports *data parallelism* since the same computation is executed in parallel on different portions of data. It is a research project for running graph processing applications in *small infrastructures*.

Hama is used for developing iterative *graph processing* applications (e.g., graph analysis, deep learning, machine learning) based on the BSP model and written in Java.

3.5 Workflow

A workflow is a well defined, and possibly repeatable, pattern or systematic organization of activities designed to achieve a certain transformation of data [24]. Workflows provide a declarative way of specifying the high-level logic of different kinds of applications, hiding the low-level details that are not fundamental for application design. A workflow is programmed as a graph, which consists of a finite set of edges and vertices, with each edge directed from one vertex to another. For example, a data analysis workflow can be designed as a sequence of pre-processing, analysis, post-processing, and interpretation tasks (see Figure 3). Differently from DAGs, workflows permit to define applications with cycles, which are circular dependencies among vertices. Workflow tasks can be composed together following a number of different patterns (e.g., loops, parallel constructs), whose variety helps designers addressing the needs of a wide range of application scenarios. A comprehensive collection of workflow patterns, focusing on the description of control flow dependencies among tasks, has been described in [25].

[2]https://hama.apache.org

3.5.1 Swift

Swift [26] is a parallel scripting language that runs workflows across several distributed systems, like clusters, Clouds, grids, and supercomputers. Swift is based on a C-like syntax and uses an implicit data-driven task parallelism [27]. In fact, it looks like a sequential language, but all variables are *futures*, thus the execution is based on data availability. When the input data is ready, functions are executed in parallel. Its runtime, namely *Turbine*, comprises a set of services that implement the parallel execution of Swift scripts exploiting the maximal concurrency permitted by both data dependencies (within a script) and external resource availability.

The Swift language provides a functional programming paradigm where workflows are designed as a set of code invocations with their associated command-line arguments and input/output files. It also allows invocation and running of external application code and binding with execution environments without extra coding from the user. Swift exploits arrays of data that are an ordered collection of input/output data elements. Arrays can be analyzed in parallel, for example by applying a function on each element of the array in a fully concurrent and pipelined manner.

In the latest releases, a Swift program can be translated into an MPI program that uses Turbine and runtime libraries for scalable dataflow processing over MPI. The Swift compiler maps the load of Swift workflow tasks across multiple computing nodes. Users can also use Galaxy [28] to provide a visual interface for Swift.

Swift provides a *medium-level of abstraction*, because a programmer defines an application as a script (i.e., sequence of instructions). The runtime will execute script instructions in parallel according to data dependencies present into them.

Swift provides an implicit data-driven *task parallelism*. The runtime comprises a set of services that implement the parallel execution of Swift scripts by exploiting the maximal concurrency permitted by data dependencies. It also provides *pipeline parallelism*, since the (partial) output of an array is passed to the next tasks to be processed. Swift is a research project for running script applications in *medium-scale infrastructures*.

Swift is used to develop *scientific data analytics* workflows written using a C-like syntax. It has been used for modeling the molecular structure of new materials, predicting protein structures, and decision making in climate and energy policy.

3.5.2 COMPSs

COMPSs [29] is a programming system and an execution runtime, whose main objective is to ease the development of workflows for distributed environments, including Grids and Clouds. With COMPSs, users create a Java sequential application and select which methods will be executed remotely. Specifically, this selection is performed by providing an annotated interface where remote methods are declared with some metadata about them and their parameters. The runtime intercepts any call to a selected method creating a representative task and finding the data dependencies with all the previous ones that must be considered along the application run.

COMPSs applications are implemented in a sequential way, without APIs that deal with the infrastructure or with duties of parallelization and distribution (e.g., synchronizations, data transfer). This means that applications will not be based on a specific API to express the interaction with the infrastructure, therefore avoiding vendor lock-in and lack of portability of applications. Moreover, the use of sequential programming allows end users to achieve an easy way to program parallel and distributed applications. Users do not need to worry about how their program is going to be run in the computing infrastructure, because the COMPSs runtime will take care of the actual execution of the application. Instead, users only need to focus on their specific domain of knowledge to create a new program that will be able to run in the distributed environment.

Recently, PyCOMPSs [30], a new system built on top of COMPSs, has been proposed with the aim of facilitate the development of computational workflows in Python for distributed infrastructures.

COMPSs provides a *medium-level of abstraction*, because a programmer can define an application by specifying through annotations which methods will be executed as remote tasks.

COMPSs provides data-driven *task parallelism*. The runtime converts the annotated methods in remote tasks and executes them in parallel according to their dependencies. COMPSs is a research project for running data analysis applications in *small infrastructures*.

COMPSs is used to *scientific data analytics* workflows written using the Java language.

3.5.3 DMCF

The Data Mining Cloud Framework (DMCF) [31] is a software system for designing and executing data analysis workflows on Clouds. A Web-based user interface allows users to compose their applications and submit them for execution over Cloud resources, according to a Software-as-a-Service approach.

DMCF allows to program data analysis workflows using two languages:

- *VL4Cloud* (Visual Language for Cloud), a visual programming language that lets users develop workflows by programming their components graphically [31].
- *JS4Cloud* (JavaScript for Cloud), a scripting language for programming data analysis workflows based on JavaScript [32].

Both languages use three key abstractions:

- *Data* elements, representing input files (e.g., a dataset to be analyzed) or output files (e.g., a data mining model).
- *Tool* elements, representing software tools used to perform operations on data elements (e.g., partitioning, filtering, mining).
- *Tasks*, which represent the execution of Tool elements on given input Data elements to produce some output Data elements.

Data and Tool nodes can be added to the workflow singularly or in array form. A *data array* is an ordered collection of input/output data elements, while a *tool array* represents multiple instances of the same tool. Data/Tool arrays allow users to design parallel and distributed data analysis applications in a compact way. In addition, array nodes are effective to fork the concurrent execution of many parallel tasks to Cloud resources, thus improving scalability.

Regardless of the language used, the DMCF editor generates a JSON descriptor of the workflow, specifying what are the tasks to be executed and the dependency relationships among them. The JSON workflow descriptor is managed by a engine that is in charge of executing workflow tasks on a set of workers (virtual processing nodes) provided by the Cloud infrastructure. The effectiveness of these formalisms to facilitate workflow design and to enable scalability has been be demonstrated through real data analysis applications in different fields, such as bioinformatics [33] or mobility[34].

DMCF provides a *high-level of abstraction*, because a programmer can define workflows by using a high-level visual language (VL4Cloud) or a script-based language (JS4Cloud). VL4Cloud is a convenient design approach for high-level users, for example, domain-expert analysts having a limited understanding of programming. Conversely, JS4Cloud allows skilled users to program complex applications more rapidly, in a more concise way, and with greater flexibility.

DMCF provides an implicit data-driven *task parallelism*. Its runtime is able to parallelize the execution of workflow tasks by exploiting the maximal concurrency permitted by data dependencies. It also provides a *pipeline parallelism*, since the (partial) output of an array is passed to the next tasks to be processed. DMCF is a research project for running workflows in *small infrastructures*.

DMCF is used to develop *visual* and *script-based analytics* workflows. For example, it has been used to run real data analysis workflows for parallel clustering, association analysis and trajectory mining.

3.6 SQL-like

With the exponential growth of data to be stored in distributed network scenarios, relational databases highlight scalability limitations that significantly reduce the efficiency of querying and analysis [35]. Relational databases are not able to scale horizontally over many machines, which makes challenging storing and managing the huge amounts of data produced everyday by many applications. The NoSQL or not-relational database approach became popular in the last years as an alternative or as a complement to relational databases, in order to ensure horizontal scalability of simple read/write database operations distributed over many servers [36].

NoSQL databases addresses several issues about storing and managing Big Data, but in many cases they are not suitable for analyzing data. For this reason, much effort has been spent developing MapReduce solutions to query and analyze data in a more productively manner.

Although Hadoop is able to address scalability issues and reduce querying times, it is not easy to be used by low-skilled people since it requires to write a complete MapReduce applications also for simple tasks (e.g., sum or average calculation, row selections or counts), with a considerable waste of time (and money) for companies. To cope with this lack, some systems have been developed for improving the query capabilities of Hadoop and easing the development of simple data analysis applications using an SQL-like language.

3.6.1 Apache Pig

Apache Pig[1] is a high-level Apache open source project for executing data flow applications on top of Hadoop. It was originally developed by Yahoo! for easing the development of Big Data analysis applications and then moved into the Apache Software Foundation in 2007.

Pig provides a high-level scripting language, called *Pig Latin* that is a procedural data flow language allowing users to define a script containing a sequence of operations. Each operation is defined in a SQL-like syntax for describing how data must be manipulated and processed.

Pig can embed custom processing functions written in other programming languages directly into its scripts. Moreover, Pig scripts can be invoked by applications written in many other programming languages.

[1]https://pig.apache.org

The Pig engine is able to automatically optimize the execution of a script by using several built-in optimization rules, such as reducing unused statements or applying filters during data loading. In addition, Pig exploits a multi-query execution system for processing an entire script or a batch of statements at once. This aims at optimizing the execution a set of queries, which may share common data, by combining and running common tasks only once.

Pig is commonly used for developing *Extract, Transform, Load* applications, which are able to gather, transform and load data coming from several sources (e.g., streams, HDFS, files). Each Pig script is translated into a series of MapReduce jobs. The Pig interpreter tries to optimize the execution of these jobs on a Hadoop cluster.

Apache Pig provides a *medium-level of abstraction*, because a programmer can develop a data analysis application through a SQL-like script (written in Pig Latin). Compared to other systems, such as Hadoop, developers do not have to write long and complex codes to perform tasks. In fact, a Pig script can use a large collection of operators to easily perform common tasks on data, such as load, filter, join and sort.

Pig supports *data parallelism*, because data is partitioned in chunks and processed in parallel, and *task parallelism* because several queries can be executed in parallel on the same data. Pig is designed to process very large amounts of data in *large-scale infrastructures*. Companies and organizations using Pig include LinkedIn, PayPal and Mendeley.

Pig is commonly used for developing *data querying* and simple *data analysis* applications (e.g., document indexing, log processing, text mining, predictive modeling), by using data from several sources (e.g., streams, HDFS, files). A Pig script can be written by using the Pig Latin language and can embed custom processing functions written in other programming languages (actually Java, Python, and JavaScript).

3.6.2 Apache Hive

Apache Hive[1] is a popular data warehouse system built on top of Hadoop, which has been designed with the main aim of providing a scalable solution for managing and processing very large amounts of data (up to petabytes). Nowadays, it is used and supported by most important IT companies, such as Facebook, Yahoo, eBay, Netflix. It has a large community of developers that collectively ensure a rapid development of the system. Hive provides a declarative SQL-like language, called *Hive Query Language* (HiveQL), which can be used to easily develop scripts for querying data stored on HDFS. The HiveQL syntax is very similar to SQL syntax, since it reuses the main concepts of relational databases (e.g., table, row, column). Each data manipulation query is automatically translated into a MapReduce job, which allows developers to deal with Big Data without having to write long and complex programs directly in MapReduce.

Although both Hive and Pig can be used to do the same things, their goals are different. Pig is designed for programmers with a good SQL background that want to process large amounts of unstructured data. Instead, Hive is data warehouse software for data analysts to read, write, and manage large amounts of structured data residing in distributed storage. Moreover, HiveQL is a declarative SQL-like language, while Pig Latin is a procedural data flow language with a SQL-like syntax.

[1]https://hive.apache.org

Apache Hive provides a *high-level of abstraction*, because a programmer can develop a data processing application by using a SQL-like syntax, which recalls well-known concepts of relational databases (e.g., table, row, column). In addition, Hive provides many User Defined Functions (UDF) for data manipulation (e.g., *sum*, *average*, *explode*) and makes it really easy to write custom ones.

Hive supports *data parallelism* that allows to execute the same query on different portions of data. When many complex queries run in parallel, each query can be composed by several jobs, which could starve computational resources. To address this issue, Hive has been recently powered by the Cost-Based Optmizer (CBO), which performs further optimizations by taking a set of decisions according to query cost (e.g., join order, type of join to execute, number of query to be performed in parallel). Hive is used in *large-scale infrastructures* by several big IT companies such as Facebook and Netflix.

Hive is commonly used by data analysts for *data querying* and *reporting* on large datasets. A script in Hive can be written by using the HiveQL, which has a SQL-like syntax.

3.7 Summary

It is hard to summarize all the features of the systems discussed in this paper and do a comparison among them. Some of these systems have common features and, in some cases, using one rather than another is an hard choice. In fact, given a specific Big Data analysis task, it can be implemented using different programming models and systems. Some of those are widely used commercial tools, provided through Cloud services, which can be easily used by no skilled people (e.g., Azure Machine Learning). Other are open-source systems that require skilled users who prefer to program their application using a more technical approach (e.g., Apache Spark). Table 1 presents a comparison of the programming systems described above, according to the four classification criteria considered in this paper.

Concerning the *level of abstraction*, we classified the systems in three categories:

- *Low*: this category includes Hadoop, Spark, MPI, Giraph and Hama. These systems provide powerful APIs and primitives that require distributed programming skills and make the development effort high. Although developing an application using these systems requires many lines of code, the code efficiency is high because it can be fully tuned.
- *Medium*: it includes Storm, Flink, Swift, COMPSs and Pig. Such systems allow to develop parallel and distributed applications through scripts or codes built using few constructs. They require some programming skills, but the development effort is lower than systems with a low-level of abstraction.
- *High*: it includes Azure ML, DMCF and Hive. These systems provide a convenient design for high-level users with a limited programming skill. These systems allow to rapidly build data analytics applications through simple visual interfaces or scripts.

About the *type of parallelism*, we classified the systems as follows:

- *Data parallelism*: it includes Hadoop, Spark, Storm, Flink, MPI, Giraph, Hama, Pig and Hive. Such systems are designed to automatically manage large input data, which is split in chunks and processed in parallel on different computing nodes.
- *Task parallelism*: it includes Spark, Storm, Flink, Azure ML, Swift, COMPSs,

DMCF, and Pig. Such systems allow to execute in parallel independent tasks (with no data dependencies).
- *Pipeline parallelism*: it includes Storm, Flink, Swift and DMCF, which allow to send the partial output of a task to the next tasks to be processed in parallel in the different stages.

The systems have been also classified according to the *infrastructure scale* they can support. In particular, they can be classified as follow:

- *Small*: it includes Azure ML, Giraph, Hama, COMPSs and DMCF, which can be used in small enterprise cluster or Cloud platforms with up to hundreds of computational nodes
- *Medium*: it includes Swift and MPI, which have been designed to run on a medium enterprise cluster consisting of up to thousands of nodes
- *Large*: it includes Hadoop, Spark, Storm, Flink, Pig and Hive, which have been designed to be used in large HPC environments or high-level Cloud services with up to ten thousands of computational nodes.

With regard to the *classes of applications*, developers can decide to exploits some *general purpose systems* (Hadoop, Spark, MPI) or systems that have been developed to be used in specific application domains. For example, Hama and Giraph have been used for developing *graph processing* applications, Hive and Pig for data querying, Storm and Flink for *stream processing* and so on.

4. Conclusion

In the last years the ability to gather data has increased exponentially and it represents a challenge to the current storage, process and analysis capabilities. To extract value from large data repositories, novel programming models and systems have been developed for capturing and analyzing Big Data that are complex and/or are produced at high rate.

This paper analyzed the most popular programming models for Big Data analysis and the features of the main systems implementing them. In particular, such systems have been compared according to four criteria: *i*) *level of abstraction* that refers the programming capabilities of hiding low-level details of a system; *ii*) *type of parallelism* that describes the way in which a system allows to express and run parallel operations; *iii*) *infrastructure scale* that refers to the capability of a system to efficiently execute applications taking advantage from the infrastructure size; and *iv*) *classes of applications* that describes the most common application domain of a system.

The final aim of this work is to support programmers and users in identifying and selecting the best solution according to their skills, hardware availability and application needs.

References

[1] V. Marx, *Biology: The big challenges of big data*, Nature 498 (2013), pp. 255–260.

[2] L. Belcastro, F. Marozzo, D. Talia, and P. Trunfio, *Using scalable data mining for predicting flight delays*, ACM Transactions on Intelligent Systems and Technology 8 (2016).

Table 1. Comparison of most popular programming systems grouped by programming models.

	System	**Level of Abstraction**	**Type of Parallelism**	**Infrastructure Scale**	**Classes of Applications**
MapReduce	Apache Hadoop	Low	Data	Large	General purpose
DAG	Apache Spark	Low	Data/Task	Large	General purpose
	Apache Storm	Medium	Data/Task/Pipeline	Large	Real-time stream processing
	Apache Flink	Medium	Data/Task/Pipeline	Large	Real-time stream processing
	Azure ML	High	Task	Small	Predictive analytics and machine learning
MP	MPI	Low	Data	Medium	General purpose
BSP	Apache Giraph	Low	Data	Small	Graph processing
	Apache Hama	Low	Data	Small	Graph processing
Workflow	Swift	Medium	Task/Pipeline	Medium	Scientific data analytics
	COMPSs	Medium	Task	Small	Scientific data analytics
	DMCF	High	Task/Pipeline	Small	Visual and script-based analytics
SQL-like	Apache Pig	Medium	Data/Task	Large	Data querying and analysis
	Apache Hive	High	Data	Large	Data querying and reporting

[3] T.B. Murdoch and A.S. Detsky, *The inevitable application of big data to health care*, Jama 309 (2013), pp. 1351–1352.

[4] S. John Walker, *Big data: A revolution that will transform how we live, work, and think* (2014).

[5] L. Belcastro, F. Marozzo, D. Talia, and P. Trunfio, *Big data analysis on clouds*, in *Handbook of Big Data Technologies*, A. Zomaya and S. Sakr, eds., Springer, 2017, pp. 101–142.

[6] D. Talia, P. Trunfio, and F. Marozzo, *Data Analysis in the Cloud*, Elsevier, 2015, ISBN 978-0-12-802881-0.

[7] D.B. Skillicorn and D. Talia, *Models and languages for parallel computation*, ACM Comput. Surv. 30 (1998), pp. 123–169.

[8] S. Wadkar, M. Siddalingaiah, and J. Venner, *Pro Apache Hadoop*, Apress, 2014.

[9] M.J. Flynn, *Some computer organizations and their effectiveness*, IEEE transactions on computers 100 (1972), pp. 948–960.

[10] M. Bux and U. Leser, *Parallelization in scientific workflow management systems*, CoRR abs/1303.7195 (2013).

[11] J. Dean and S. Ghemawat, *Mapreduce: simplified data processing on large clusters*, Communications of the ACM 51 (2008), pp. 107–113.

[12] F. Marozzo, D. Talia, and P. Trunfio, *P2p-mapreduce: Parallel data processing in dynamic cloud environments*, Journal of Computer and System Sciences 78

(2012), pp. 1382–1402.

[13] 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*, SIGMOD '13, New York, New York, USA, ACM, New York, NY, USA, 2013, pp. 13–24.

[14] D. LaSalle and G. Karypis, *Mpi for big data: New tricks for an old dog*, Parallel Computing 40 (2014), pp. 754–767.

[15] J.L. Reyes-Ortiz, L. Oneto, and D. Anguita, *Big data analytics in the cloud: Spark on hadoop vs mpi/openmp on beowulf*, Procedia Computer Science 53 (2015), pp. 121–130.

[16] F. Liang and X. Lu, *Accelerating iterative big data computing through mpi*, Journal of Computer Science and Technology 30 (2015), p. 283.

[17] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: portable parallel programming with the message-passing interface*, Vol. 1, MIT press, 1999.

[18] A. Geist, W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, W. Saphir, T. Skjellum, and M. Snir, *MPI-2: Extending the message-passing interface*, in *Euro-Par'96 Parallel Processing*, Springer, 1996, pp. 128–135.

[19] L.G. Valiant, *A bridging model for parallel computation*, Commun. ACM 33 (1990), pp. 103–111.

[20] 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.

[21] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, *One trillion edges: Graph processing at facebook-scale*, Proceedings of the VLDB Endowment 8 (2015), pp. 1804–1815.

[22] Z. Wang, Y. Bao, Y. Gu, F. Leng, G. Yu, C. Deng, and L. Guo, *A BSP-based parallel iterative processing system with multiple partition strategies for big graphs*, in *Big Data (BigData Congress), 2013 IEEE International Congress on*, IEEE, 2013, pp. 173–180.

[23] K. Siddique, Z. Akhtar, E.J. Yoon, Y.S. Jeong, D. Dasgupta, and Y. Kim, *Apache hama: An emerging bulk synchronous parallel computing framework for big data applications*, IEEE Access 4 (2016), pp. 8879–8887.

[24] D. Talia and P. Trunfio, *Service-oriented distributed knowledge discovery*, Chapman and Hall/CRC, 2012.

[25] W.M.P. Van Der Aalst, A.H.M. Ter Hofstede, B. Kiepuszewski, and A.P. Barros, *Workflow patterns*, Distrib. Parallel Databases 14 (2003), pp. 5–51, URL `https://doi.org/10.1023/A:1022883727209`.

[26] M. Wilde, M. Hategan, J.M. Wozniak, B. Clifford, D.S. Katz, and I. Foster, *Swift: A language for distributed parallel scripting*, Parallel Computing 37 (2011), pp. 633–652.

[27] J.M. Wozniak, M. Wilde, and I.T. Foster, *Language features for scalable distributed-memory dataflow computing*, in *Data-Flow Execution Models for Extreme Scale Computing (DFM), 2014 Fourth Workshop on*, IEEE, 2014, pp. 50–53.

[28] B. Giardine, C. Riemer, R.C. Hardison, R. Burhans, L. Elnitski, P. Shah, Y. Zhang, D. Blankenberg, I. Albert, J. Taylor, *et al.*, *Galaxy: a platform for interactive large-scale genome analysis*, Genome research 15 (2005), pp. 1451–1455.

[29] F. Lordan, E. Tejedor, J. Ejarque, R. Rafanell, J. lvarez, F. Marozzo, D. Lezzi, R. Sirvent, D. Talia, and R. Badia, *Servicess: An interoperable programming framework for the cloud*, Journal of Grid Computing 12 (2014), pp. 67–91.

[30] E. Tejedor, Y. Becerra, G. Alomar, A. Queralt, R.M. Badia, J. Torres, T. Cortes, and J. Labarta, *Pycompss: Parallel computational workflows in python*, The International Journal of High Performance Computing Applications 31 (2017), pp. 66–82.

[31] F. Marozzo, D. Talia, and P. Trunfio, *A workflow management system for scalable data mining on clouds*, IEEE Transactions On Services Computing (2016).

[32] F. Marozzo, D. Talia, and P. Trunfio, *Js4cloud: Script-based workflow programming for scalable data analysis on cloud platforms*, Concurrency and Computation: Practice and Experience 27 (2015), pp. 5214–5237.

[33] G. Agapito, M. Cannataro, P.H. Guzzi, F. Marozzo, D. Talia, and P. Trunfio, *Cloud4SNP: Distributed Analysis of SNP Microarray Data on the Cloud*, in *Proc. of the ACM Conference on Bioinformatics, Computational Biology and Biomedical Informatics 2013 (ACM BCB 2013)*, September, ISBN 978-1-4503-2434-2, ACM Press, Washington, DC, USA, 2013, p. 468.

[34] A. Altomare, E. Cesario, C. Comito, F. Marozzo, and D. Talia, *Trajectory pattern mining for urban computing in the cloud*, Transactions on Parallel and Distributed Systems (IEEE TPDS) 28 (2017), pp. 586–599, ISSN:1045-9219.

[35] V. Abramova, J. Bernardino, and P. Furtado, *Which nosql database? a performance overview*, Open Journal of Databases (OJDB) 1 (2014), pp. 17–24.

[36] R. Cattell, *Scalable sql and nosql data stores*, ACM SIGMOD Record 39 (2011), pp. 12–27.