UNIVERSITÀ degli STUDI di CATANIA

DIPARTIMENTO DI INGEGNERIA ELETTRICA
ELETTRONICA E INFORMATICA

H2F: a hierarchical Hadoop framework to process Big Data in geo-distributed contexts

Author:
Marco Cavallo

Supervisor:
Prof. V. Catania

Co-Supervisor:
Prof. O. Tomarchio

Dottorato di Ricerca in
Ingegneria dei Sistemi, Energetica,
Informatica e delle Telecomunicazioni
XXX Ciclo

Abstract

The wide spread adoption of IoT technologies has resulted in generation of huge amount of data, or Big Data, which has to be collected, stored and processed through new techniques to produce value in the best possible way. Distributed computing frameworks such as Hadoop, based on the MapReduce paradigm, have been used to process such amounts of data by exploiting the computing power of many cluster nodes.

Unfortunately, in many real big data applications the data to be processed reside in various computationally heterogeneous data centers distributed in different locations. In this context the Hadoop performance collapses dramatically.

To face this issue, we developed a Hierarchical Hadoop Framework (H2F) capable of scheduling and distributing tasks among geographically distant clusters in a way that minimizes the overall jobs' execution time.

Our experimental evaluations show that using H2F improves significantly the processing time for geodistributed data sets with respect to a plain Hadoop system.

L'ampia diffusione di tecnologie ha portato alla generazione di enormi quantità di dati, o di Big Data, che devono essere raccolti, memorizzati e elaborati attraverso nuove tecniche per produrre valore nel modo migliore. I framework distribuiti di calcolo come Hadoop, basati sul paradigma MapReduce, sono stati utilizzati per elaborare tali quantità di dati sfruttando la potenza di calcolo di molti nodi di cluster.

Purtroppo, in molte applicazioni di big data, i dati da elaborare risiedono in diversi data center computazionali eterogeni e distribuiti in luoghi diversi. In questo contesto le performance di Hadoop crollano drasticamente.

Per affrontare questo problema, abbiamo sviluppato un Hierarchical Hadoop Framework(H2F) in grado di pianificare e distribuire task tra cluster geograficamente distanti in modo da ridurre al minimo il tempo di esecuzione complessivo delle applicazioni.

Le nostre valutazioni sperimentali mostrano che l'utilizzo di H2F migliora notevolmente il tempo di elaborazione per dataset geodistribuiti rispetto ad un semplice sistema Hadoop.

Contents

List of Figures

List of Tables

Chapter 1

Motivation And Background

Parallel to expansion in service offerings of IT companies, there is growth in another environment – the data environment. The volume of data is practically exploding by the day. Not only this, the data that is available now in becoming increasingly unstructured. In this chapter, we analyze the major technologies related to the high increase in the amount of produced data.

1.1 Internet of Things

Internet of Things is a neologism used in telecommunications, a term of new honeycomb (first used by Kevin Ashton, a researcher at the MIT, Massachusetts Institute of Technology), which arises from the need to name real-world objects connected to the internet.

The internet of things (or the "Internet of Things" original language) is getting more and more consensus and is increasingly an opportunity for development. Internet evolution has extended the internet to real objects and places ("things"), which can now interact with the network and transfer data and information. The object interacts with the surrounding world, as it has "intelligence", it retrieves and transfers information between the internet and the real world. From the refrigerator at home, to the clock, to the traffic lights, everyone can be considered examples of IoT. The important thing is that these objects are connected to the network, and that they have the ability to transmit and receive data. In this way, these objects become "smart," and can turn on and off "on their own" and as needed.

The areas most affected by IoT applications are Smart Home, Smart Building, Smart City and Smart Mobility, but also, and for a long time, Smart Manufacturing. In the field of energy, Smart Metering is widespread, while in the world of mobility new opportunities are coming in the Smart Car.

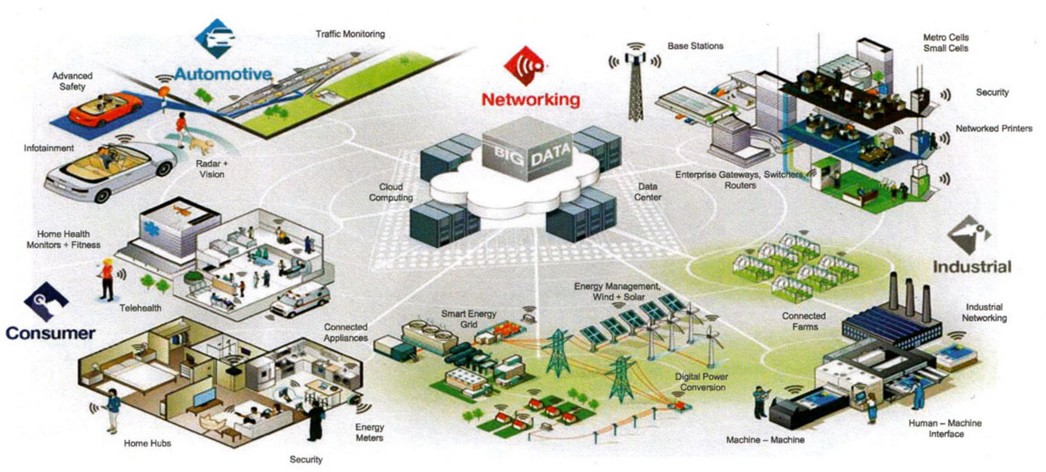


FIGURE 1.1: Internet of Things

The IoT brings "intelligence" to information processing systems. Through the Internet of things things can be remotely controlled (remote control of things), and they are capable of transmitting data from which you can extract useful information about the operation of those objects, and the interaction between these objects and who uses them (The consumer).

Over a million new IoT devices are being connected to the Internet daily, and that process is accelerating. Mobile phones and tablets equipped with multiple sensors are transmitting data on a constant basis. In the home, various automation devices like Nest thermostats and Dropcams are also contributing to the data glut.

Moreover internet of things is a growing technology in industrial applications, manufacturing companies are currently implementing this “intelligent connectivity of smart devices” in their factories and on the shop floor.

The emerging Industrial IoT gives rise to what is predicted to be a sweeping change that will fundamentally reconfigure industry. It is being called the next Industrial Revolution. This revolution follows the previous three industrial revolutions, which are usually identified as mechanization (powered by steam engines in the 1800s), mass production (powered by electricity and the assembly line in the early 1900s) and automation (powered by computers in the late 1900s). As the fourth industrial revolution, it has taken on the name Industry 4.0, in keeping with the way new versions or releases of software are usually designated. This is appropriate, considering that the latest industrial revolution is powered by the Internet and Web-enabled software applications capable of processing streams of manufacturing data.

In Industry 4.0, industrial processes and associated machines become smarter and more modular, while new protocol standards allow previously isolated control equipment to communicate with each other, enabling a hyperconnected network across multiple industrial ecosystems. All this is driving higher levels of utilization and greater flexibility for meeting customer demand.

The payoff for manufacturers who implement Industrial IoT solutions lies in better decision-making. When devices are connected, the data they generate can flow into software applications that create the information individuals can use to make choices that are timely and effective. By understanding the results of these choices more fully, decision-makers can achieve strategic objectives or benchmark performance. Decisions will be based on knowledge and wisdom, not theory or guesswork. Better decisions mean fewer mistakes and less waste.

The Internet of Things is the key technology enabling this digital transformation. Smart, always connected things with instant access to contextual information, as well as devices and applications with artificial intelligence designed to optimize processes and improve how we live, work, and interact with each other, are all changing the way we conceive, produce, deliver, and consume goods and services.

Experts predict that as many as 25 to 50 billion new IP-enabled IoT devices will be deployed and online by 2020. As a result, IoT has created an explosion of data that is designed to move freely between devices and locations, and across network environments, remote offices, mobile workers, and public cloud environments, making it difficult to consistently track and secure. Managing the huge amount of data that those billions of portable sensing devices produce every day is one of the main technological challenge.

1.2 BigData

Every time we use a computer, turn on the smartphone or open an app on the tablet, we leave our fingerprint made of data. Beyond data flows produced by IT systems and infrastructures to support production, distribution, and delivery of services, big data is a phenomenon associated with the massive evolution of individual uses and habits.

Big Data is a term that has become popular to indicate the exponential growth and availability of data, whether they are stitched or not. We talk about big data when dealing with data sets that can not be processed and analyzed, using traditional processing methods.

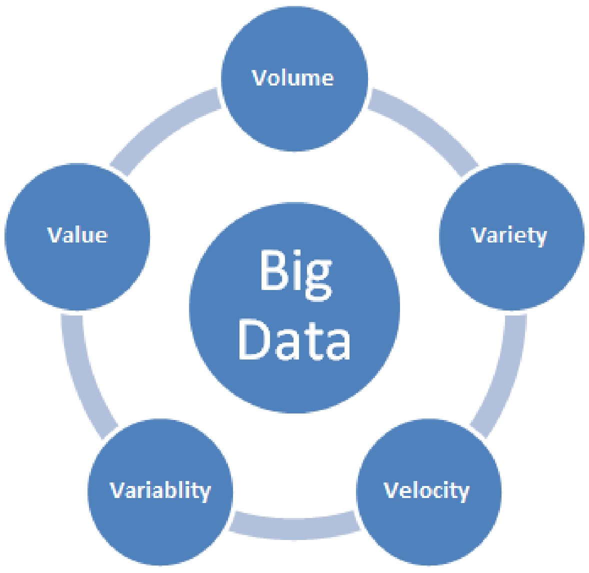


FIGURE 1.2: Big Data 5v

The analyst and researcher Doug Laney has defined the main features that can be summarized in the 3V acronym:

- Volume: the size of the datasets to handle in this context is remarkable, usually speaking of the order of hundreds of Terabytes, if not Petabyte. The main source of data is the Internet, as it involves millions of users who daily produce new data, just think of the most famous social media that collect a lot of users. Even without considering Internet data volume in many businesses is very large and current technologies to handle them have considerable costs. This has led to the need to study new storage paradigms to facilitate processing.
- Velocity: in this context we mean the velocity of generation of new data which can be very large and therefore it is indispensable to detect them, store them and analyze them as quickly as possible. The simplest solution to speed up analysis and acquisition would be to use fast and efficient memories, but the high cost per megabyte of these technologies is in contrast to the first Big Data feature, the Volume. The use of common DBMS (Database Management System) is not recommended because of the complexity of the operations implemented, which, in addition to the large size of the data involved, results in considerable response times. The solution to increase speed is to rely on parallel processing systems, such as clusters of computers.
- Variety: the data should be not necessarily structured but may have a heterogeneous nature. If you think about the Web and the multimedia content it provides, text, images, and videos, you understand that data types such as their structure are not standardized, so more general management tools are required.

To these general features of Big Data have been added two more:

- Variability: data over time can also change very quickly, even if you consider the large size you understand how difficult it is to take into account all the changes. However, an analyst needs consistent and up-to-date data to provide correct evaluations and efficient solutions.
- Value: data comes from different sources today. And it's still a business to compare, clean up and transform data through systems. However, it

is necessary to correlate and link the relationships, and data hierarchies to drive the useful information they contain.

The Big Data also comes from the ever-expanding multimedia that originates from the proliferation of fixed and mobile devices we use to live and work. According to a Cisco survey (Cisco, 2017), for example, currently 78% of the US band is occupied by video, but in 2018 it will be saturated for 84%. Videosharing and a culture of the image that brings people to share every kind of photo shoot will help those who will manage this data to better understand tastes and trends, better orienting decisions.

Big data analytics can point the way to various business benefits, including new revenue opportunities, more effective marketing, better customer service, improved operational efficiency and competitive advantages over rivals. Big data analytics applications enable data scientists, predictive modelers, statisticians and other analytics professionals to analyze growing volumes of structured transaction data, plus other forms of data that are often left untapped by conventional business intelligence (BI) and analytics programs.

On a broad scale, data analytics technologies and techniques provide a means of analyzing data sets and drawing conclusions about them to help organizations make informed business decisions.

Traditional global data center traffic is currently measured in the zettabytes, and is predicted to more than triple to 15.3 ZB annually by 2020. However, according to Forbes (Forbes, 2016), the total volume of data generated by IoT will reach 600 ZB per year by 2020, which is 275 times higher than projected traffic going from data centers to end users and devices (2.2 ZB), and 39 times higher than total projected data center traffic (15.3 ZB.)

The features of BigData, through multiple platforms and business functions, make it difficult to manage them through well-established technologies, and at the moment there is no definitive solution that can solve all the problems. Let's not forget that there is still a great deal of unused or underused data inside companies, since it is cost-intensive or because of the lack of structure that can be dealt with normal management systems in a profitable way.

1.3 Cloud

The Cloud has been evoked by many as the "right place" where produced data ought to be stored and mined. The Cloud can scale very well with respect to both the data dimension and the computing power that is required for elaboration purposes. Cloud computing has been revolutionising the IT industry by adding flexibility to the way IT is consumed, enabling organisations to pay only for the resources and services they use. In an effort to reduce IT capital and operational expenditures, organisations of all sizes are using Clouds to provide the resources required to run their applications. Clouds vary significantly in their specific technologies and implementation, but often provide infrastructure, platform, and software resources as services.

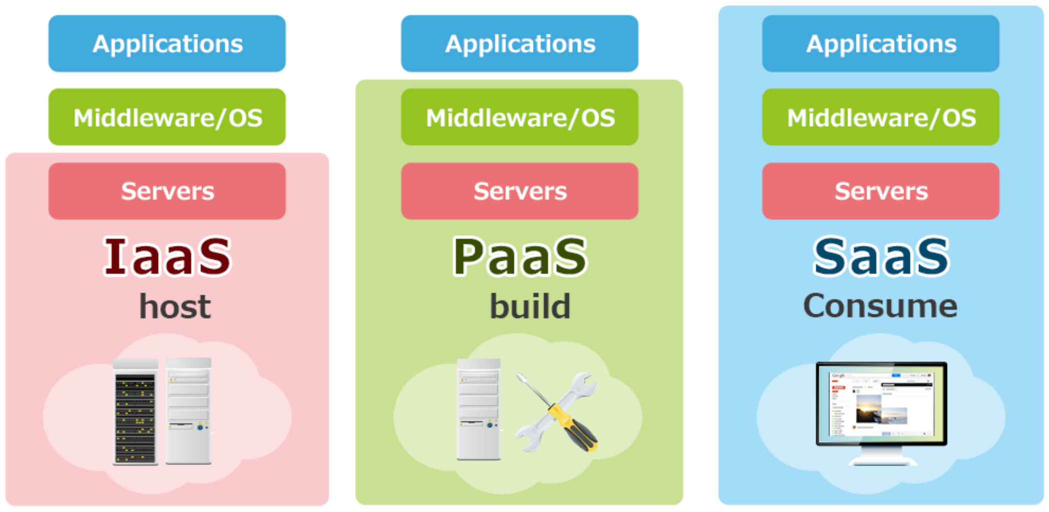


FIGURE 1.3: Cloud Infrastructure

The most often claimed benefits of Clouds include offering resources in a pay-as-you-go fashion, improved availability and elasticity, and cost reduction. Clouds can prevent organisations from spending money for maintaining peak-provisioned IT infrastructure that they are unlikely to use most of the time. Companies can easily negotiate resources with the cloud provider as required. Cloud providers usually offer three different basic services: Infrastructure as a Service (IaaS); Platform as a Service (PaaS); and Software as a Service (SaaS):

- IaaS delivers infrastructure, which means storage, processing power, and virtual machines. The cloud provider satisfies the needs of the client by virtualizing resources according to the service level agreements (SLAs);
- PaaS is built on top of IaaS and allows users to deploy cloud applications created using the programming and run-time environments supported by the provider. It is at this level that big data DBMS are implemented;
- SaaS is one of the most known cloud models and consists of applications running directly in the cloud provider

Since the cloud virtualizes resources in an on-demand fashion, it is the most suitable and compliant framework for big data processing, which through hardware virtualization creates a high processing power environment for big data.

Currently, several cloud computing platforms, e.g., Amazon Web Services, Google App Engine, IBM's Blue Cloud, and Microsoft Azure, provide an easy locally distributed, scalable, and on-demand big-data processing. However, these platforms do not take into account data locality, i.e., geo-distributed data, and hence, necessitate data movement to a single location before the computation. In contrast, in the present time, data is generated geo-distributively at a much higher speed as compared to the existing data transfer speed, for example, data from modern satellites. Many organizations operate in different countries and hold datacenters across the globe. Moreover, the data can be distributed across different systems and locations even in the same country.

We can list a few of the many real scenarios that need to face the problem of geographically and unevenly distributed big data. Social networks generate huge amounts of data worldwide. Data get stored on many data centers usually located in different countries (or continents, at least). Since the analysis has to span to many data centers, a careful design of the procedures enforcing the data analysis is needed (Facebook, 2012) in order to have reliable results within the desired time. Multinational retail corporations (Walmart, 2015) produce up to petabytes of data daily. Data generated by the transactions of thousands (or even millions) of customers purchasing goods all over the world are stored in many data centers and need to be promptly and reliably analyzed. Again, being data natively distributed over physically distant data centers (not just in one site), data computing strategies usually employed in the one-site scenarios may not be effective. There are other applications and that process and analyze a huge amount of massively geo-distributed data to provide the final output, for example: climate science, data generated by multinational companies, sensor networks, stock exchanges, web crawling, biological data processing such as DNA sequencing and human microbiome investigations, protein structure prediction, and molecular simulations, stream analysis, video feeds from distributed cameras, log files from distributed servers, geographical information systems (GIS), and scientific applications.

Several big-data processing programming models and frameworks such as MapReduce, Hadoop, have been designed to process huge amount of data through parallel computing, distributed databases, and cluster computing. In next chapter we introduce these models and analyze their behaviour in a geo-distributed environment.

Chapter 2

Big Data Computing Approaches

Technologies for big data analysis have arisen in the last few years as one of the hottest trend in the ICT scenario. Several programming paradigms and distributed computing frameworks have appeared to address the specific issues of big data systems. Application parallelization and divide-and-conquer strategies are, indeed, natural computing paradigms for approaching big data problems, addressing scalability and high performance. Furthermore, the availability of grid and cloud computing technologies, which have lowered the price of on-demand computing power, have spread the usage of parallel paradigms, such as the MapReduce (Dean and Ghemawat, 2004), for big data processing.

2.1 MapReduce

MapReduce has emerged as an effective programming model for addressing these challenges: today it is one of the most famous computing paradigm for big data widely used both in academic and in business scenarios. MapReduce is a framework for processing parallelizable problems across large datasets using a large number of nodes. MapReduce comes from the experience gained within Google in distributed computing on large amounts of data in computer clusters. The strong need for parallelization within the company has led to an abstraction that could accelerate the development of parallel applications, relegating details of parallelization, fault management, and data distribution. The MapReduce paradigm is inspired by the map and reduce

functions used within the functional programming languages that allow you to transform and aggregate a list of elements.
The map function has a parameter lists, an array of elements and a function, and returns a list of results deriving from the application of the passed function to the elements of the starting array:

$$map : ([x0, x1, .., xn-1], f(.)) \rightarrow [f(x0), f(x1), .., f(xn-1)] \tag{2.1}$$

The reduce function, also known as a fold, receives an initial value, a list of elements and a combining function, and returns a return value. The fold function proceeds by combining the elements of the data structure using the function in a systematic way:

$$reduce : ([y0, y1, .., yn-1], g(i), s) \rightarrow g(yn-1, g(yn, ..g(y0, s)...)) \tag{2.2}$$

MapReduce differs from functional computation using map and reduces for some details but the main idea remains to consider algorithms as a sequence of two-phase, transformation and group aggregation. The first significant difference concerns the data format: the MapReduce paradigm predicts that data, both in input and output, is seen as entity pairs: the first component is called the key, the second component value. Both components are not defined a priori, but are free to assume any form: the set of definitions is free and dependent on specific computation. A single computation in the MapReduce paradigm can be divided into three sequential phases: map, shuffle and reduce.

Map phase is the critical step which makes this possible. Mapper brings a structure to unstructured data. In the map phase, the mapper takes a single (key, value) pair as input and produces any number of (key, value) pairs as output . It is important to think of the map operation as stateless, that is, its logic operates on a single pair at a time (even if in practice several input pairs are delivered to the same mapper). To summarize, for the map phase, the user simply designs a map function that maps an input (key, value) pair to any number (even none) of output pairs. Most of the time, the map phase

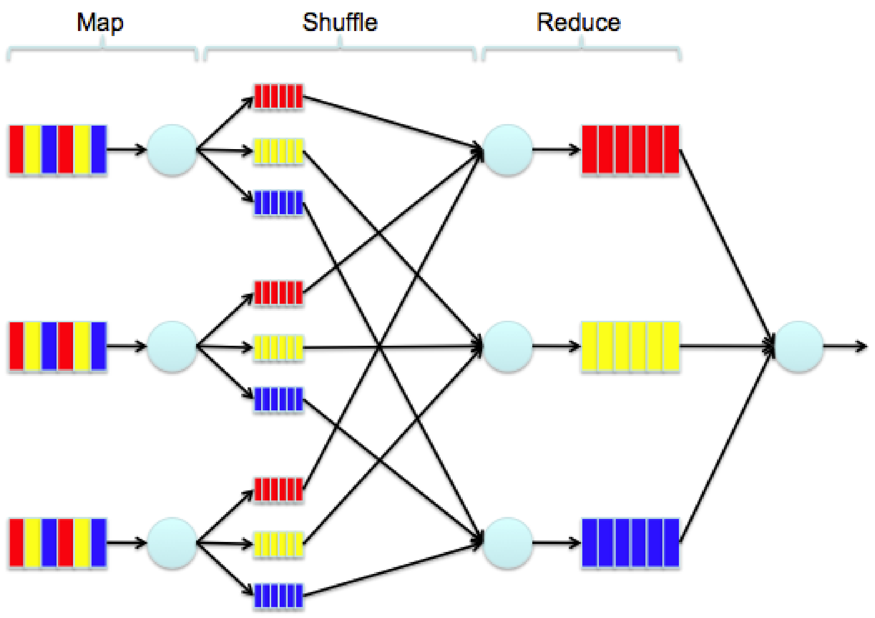


FIGURE 2.1: Overall MapReduce Process

is simply used to specify the desired location of the input value by changing its key.

The shuffle phase is automatically handled by the MapReduce framework. The underlying system implementing MapReduce routes all of the values that are associated with an individual key to the same reducer.

In the reduce phase, the reducer takes all of the values associated with a single key k and outputs any number of (key, value) pairs. This highlights one of the sequential aspects of MapReduce computation: all of the maps need to finish before the reduce stage can begin. Since the reducer has access to all the values with the same key, it can perform sequential computations on these values. In the reduce step, the parallelism is exploited by observing that reducers operating on different keys can be executed simultaneously.

MapReduce programs are not guaranteed to be fast. The main benefit of this programming model is to exploit the optimized shuffle operation of the platform, and only having to write the Map and Reduce parts of the program. In practice, the author of a MapReduce program however has to take the shuffle step into consideration; in particular the partition function and the amount of data written by the Map function can have a large impact on the performance and scalability. When designing a MapReduce algorithm, the author needs to choose a good tradeoff between the computation and the communication costs. Communication cost often dominates the computation cost, and many MapReduce implementations are designed to write all communication to distributed storage for crash recovery.

2.2 Hadoop

Apache Hadoop (The Apache Software Foundation, 2011) is an open source framework written in Java, inspired by Google's MapReduce paradigm. Its main purpose is to provide large-scale storage and processing services of large datasets in hardware commodity clusters. Hadoop was created by Doug Cutting in 2005 but actually originated in the Nutch project for the creation of an Open Source search engine. Subsequently, other support projects were created to provide additional features, we mention some of them:

- Hbase: A NoSql database distributed on Hadoop and HDFS.
- Hive: Data warehousing system on Hadoop.
- Mohaut: extension designed for Artificial Intelligence and Data Mining applications.
- Pig: a platform for creating MapReduce programs for Hadoop.

A MapReduce application on a Hadoop cluster is recognized as a Job. It is possible to locate five blocks that cost the Hadoop execution environment.

The two main ones are YARN (Yet Another Resource Negotiator), which is responsible for identifying and providing the computing resources needed for the application and Hadoop Distributed File System (HDFS), which provides a distributed and reliable file system. The YARN and HDFS infrastructures work in pairs but are completely decoupled and independent, basically HDFS communicates to YARN the position of data in the cluster.

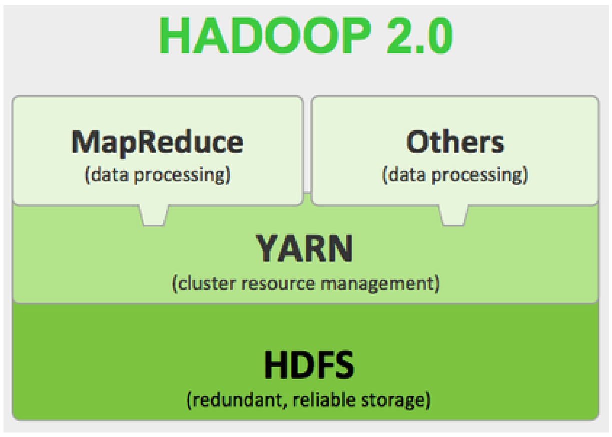


FIGURE 2.2: Overall Hadoop Architecture

The MapReduce framework is just one of the possible applications that can be implemented over YARN to leverage its services. The cluster constitutes the hardware of the system, that represents the set of hosts, called nodes which can be grouped into racks. Nodes start services that define the role within the cluster. There are two types of nodes:

- Master nodes, where coordination demons are started for HDFS and YARN.
- Slave nodes, where service demons communicate their computing and storage resources.

In order to understand the hadoop functionality we analyze in detail the two main components.

HDFS is the distributed file system of Hadoop, which will store the input and output data generated while executing the MapReduce application. HDFS is based on the POSIX standard, but sacrifices complete compatibility to provide specific services to match in pair with Hadoop MapReduce, such as:

- Large file-oriented behavior, as you work with files ranging from hundreds of MegaBytes to some Terabytes.
- Efficiency in sequential access, to support write-one paradigm, read-many-times.

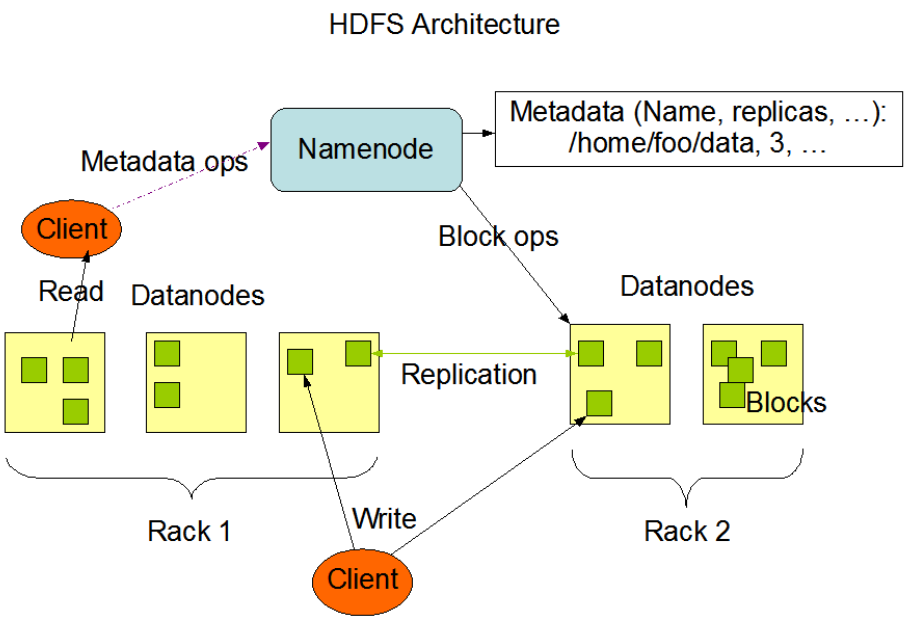


FIGURE 2.3: HDFS Architecture

File system architecture relies on the services provided by two major demons, the datanode and the namenode, and allows client entities to access transparently to reading and writing. The slave nodes execute the datanode daemon that deals with storing the data in blocks. The block size is configurable, usually 64 megabytes are used, and the blocks match files in the local file system of the slave. The namenode is unique within the HDFS cluster and represents the master in the HDFS architecture. The main role is to provide a namespace for the filesystem, to locate the data in the distributed system, and to manage the block replication factor. The master stores the metadata on all data blocks in the cluster, such as the list of datanodes that contain a copy or which files they belong to. A data exchange of heartbeat with fixed frequency is established between datanodes and namenodes. These messages, on the one hand, report to the master the state of the data nodes with information on the available storage resources, on the other hand it allows the namenode, in the responses, to provide commands to the slaves to perform the required operations.

HDFS has been designed to be a simple but fault tolerance oriented system. Replication, for example, has an active role in preventing data loss in case of sporadic failures, and heartbeat messages keep the namenode informed of data status, and in case of failure the master may choose to restore the level of default replication factor. In addition, the choice to replicate data allows access to data in parallel and allows you to choose "close" computing resources to the data to be processed. The replication factor is configurable by the client, default is set to 3, and replicas are deployed in the cluster to ensure good balancing, even though no single storage space is available.

HDFS relies on a partial knowledge of network topology through the "Rack-Awareness", which allows you to know which rack is connected to a precise cluster datanode. This information is used to deploy data block replicas to minimize the risk of data loss caused by node and rack failures. The ultimate goal is therefore to improve data reliability and availability and more efficient use of the intranet to which the cluster relies. The choice of replication distribution policy must evaluate the various trade-offs, eg placing them on different racks facilitates data retrieval in case of failure, but on the other hand increases the cost of writing to be made on multiple racks. The implementation of Rack-Awareness relies on the correct allocation of hosts to their racks, which is performed by a user-defined script during the initialization phase of the cluster. Typical operations of a filesystem are reading and writing files:

- Reading, client who wants to read a file contacts the namenode, which returns the datanodes that have a copy of the blocks that make up the file. Subsequently, the blocks are retrieved from the individual datanodes by interrogating them directly. In order to minimize the bandwidth used and read latency, HDFS tries to meet a reading request with a "close" replica to the reader. In fact, namenode provides the list of datanodes sorted by distance from the client. Namenode only provides location blocks, so clients can access data concurrently.

- Writing, client who wants to write a file first requires permission to the namenode, which first checks the access permissions and creates a metadata record for the file. The HDFS master provides the client with a list of datanodes on which to place the data blocks. The first datanode receives the data directly from the client, saves them on the local file system, and immediately sends them to the next datanode in the list on another connection. At the end, you will have a number of replicas you want and write confirmations that run the datanodes in the opposite direction that will reach the client contacting the namenode to let you know the end of the writing. The namenode waits for all data nodes to communicate, via heartbeat, that all file blocks have reached the minimum number of replicas before considering writing successfully completed.

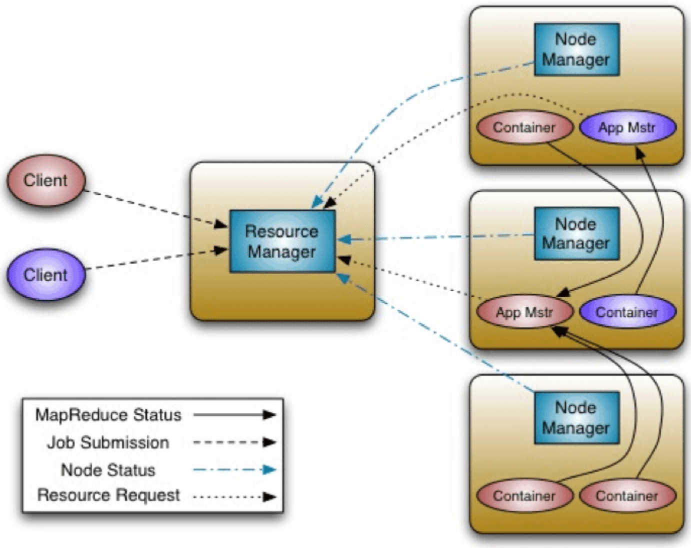


FIGURE 2.4: YARN Architecture

YARN provides an intermediate level of resource and computing management and was introduced only from version 2.0 of Hadoop. The earlier versions implemented their MapReduce paradigm directly on the services offered by HDFS, in a centralized architecture in which the master, JobTracker, had to manage the recording and supervision of resources in the cluster and manage its allocation to the various submitted applications. YARN decouplates the original tasks of the jobTracker, assigning a host the ResourceManager role for the resource management, and delegating to the other, the Application Master, the coordination of the resources obtained by the Resource Manager for the Job execution. The Resource Manger is unique within a cluster and has a cetralized and global vision of available resources, and its main task is to allocate computing units, called Containers, represented by RAM and CPU number. On slave nodes the NodeManager demon monitors local resources and manages the Container lifecycle, and returns this information to the Resource Manager during messages exchanging through Heartbeat.

A client requires an application submission through an interface provided by the Resource Manager, which models it as a JOB and send it to the Resource Scheduler. In the initialization phase, the Resource Manager allocates a Container to run the Application Manager, which manages the application lifecycle and the required resources to complete the tasks in which a Job is divided. In order to obtain Containers, The Application Master exploits the Heathbeat service to send particular resource-request messages to the Resource Manager, defining location and quality specifications. Then the Resource Manager, in accordance with the scheduling policies, will try to satisfy the demands of the Application Master.

The Resource Manager exposes two interfaces to the outside:

- one to clients submitting their applications
- one to the ApplicationMaster, to negotiate access to resources

The Resource Manager has a global view of the cluster and all its resources, which greatly simplifies the task of the scheduler if it wants to get specific requirement, such as fairness.

The Application Master encodes resource requirements in terms of a resource-request, which defines container number, priority, location, and container resources by number of cores and amount of RAM. In response to the Application Manager requests, the Resource Manager allocates containers by assigning a token to ensure resource access security and reports the status of those completed.

The main task of the Application Master is to coordinate the execution of an application in accordance with the resources in the cluster. It runs inside a container allocated by the Resource Manager during the job initialization. The Application Master periodically sends Heartbeat messages to Resource Manager to update it about its liveness status and requests. When it receives the response with the available container list, the Application Master can update the execution plan of the application by launching the Hadoop Tasks. The MapReduce Application Master defines two task types, the MAP Task, and the REDUCE Task. The Resource Manager does not interpret container status but the ApplicationMaster evaluates the success or failure of execution on a container and the progress of the application.

The "worker" nodes that perform the Hadoop tasks run the Node Manager daemon. This manages the allocated containers and monitors the status during the lifecycle. Once registered with the Resource Manager, the Node Manager sends Heartbeat messages containing their status and receives instructions to execute. In YARN, the containers, including the one who host the Application Master, are described by a Container Launch Context (CLC). This entity defines the set of environment variables, security token, payload for Node Manager services, and the needed procedures to start tasks. The Node Manager configures the environment and initialize their own monitoring system with constraints on the resources specified during allocation phase. Containers can be "killed" when Resource Manager reports that the application is terminated, when the scheduler wants to preempt, or when Node Manager notices that the container reaches the allocation limits on that node or when the Application Master no longer needs its computation. When a application is completed it is removed from all nodes. Node Manager periodically also monitors the health status of the physical node to detect any HW/SW nature problems. When a problem is detected, Node Manager changes its state into unhealthy and communicates it to the Resource Manager, allowing the scheduler to allocate no more containers to the faulty node.

2.3 Distributed computing approaches

As we introduced in the previous chapter, the number of applications that need to utilize large volumes of potentially distributed data is nowadays more and more increasing. Data get stored on many data centers usually located in different countries (or continents, at least). Since analysis has to span many data centers, a careful design of the procedures enforcing the data analysis is needed in order to have reliable results within the desired time. Being data natively distributed over physically distant data centers (not just in one site), data computing strategies usually employed in the one-site scenarios may not be effective.

Hadoop is known to perform well only when the whole data set resides in a cluster of nodes and nodes are connected to each other's by means of high-speed network links (Heintz et al., 2014). Therefore, the Hadoop framework seems to unfit the scenarios described which, instead, are subjected to two strong constraints: 1) data are scattered among many sites and 2) sites' interconnecting links are not as fast as the links interconnecting the nodes of a typical cluster.

Researchers followed two approaches to process data in geographic environments in an efficient way:

- elaborated and improved versions of the Hadoop framework which tries to get awareness of the heterogeneity of computing nodes and network links (*Geo-hadoop* approach)
- hierarchical frameworks which splits the MapReduce job into many subjobs which are sent to as many computing nodes hosting Hadoop instances which do the computation and send back the result to a coordinator node that is in charge of merging the subjobs' output (*Hierarchical* approach)

The former approach's strategy is to optimize the Hadoop's native steps and it's described in this section. The latter's will be analyzed in the next section.

Geo-hadoop reconsiders the four steps of the job's execution flow (Push, Map, Shuffle, Reduce) in a panorama where data are not available at a cluster but are instead spread over multiple (geographically distant) sites, and the available resources (compute nodes and network bandwidth) are imbalanced among. To reduce the job's average *makespan*, the four steps must be smartly coordinated. Some have launched modified version of Hadoop capable of enhancing only a single step (Kim et al., 2011; Mattess, Calheiros, and Buyya, 2013). Heintz et al.(Heintz et al., 2014) study the dynamics of the steps and claim the need for a comprehensive, end-to-end optimization of the job's execution flow. They propose an analytical model which looks at parameters such as the network links, the nodes capacity and the applications profile, and turns the makespan optimization problem into a linear programming problem solvable by means of Mixed Integer Programming techniques. In (Zhang et al., 2014) authors propose a modified version of the Hadoop algorithm that improves the job performance in a cloud based multi-data center computing context. Improvements span the whole MapReduce process, and concern the capability of the system to predict the localization of MapReduce jobs and to prefetch the data allocated as input to the Map processes. Changes in the Hadoop algorithm regarded the modification of the job and task scheduler, as well as of the HDFS' data placement policy. In (Fahmy, Elghandour, and Nagi, 2016) authors shows a Hadoop HDFS layer extension that leverages the spatial features of the data and accordingly co-locates them on the HDFS nodes that span multiple data centers.

Several other works in this area try to improve the performance of MapReduce by modifying the job scheduler. In (You, Yang, and Huang, 2011), authors propose a Load Aware scheduler that aims to improve the Resources' utilization in a heterogeneous MapReduce cluster with dynamic Workload. The Load Aware scheduler consists of two modules: 1) a "data collection module" that senses the system-level information from all nodes in the cluster; 2) a task scheduling module that estimates the task's completion time using the system-level information gathered. In (Cheng et al., 2017), authors introduce a self-tuning task scheduler for multi-wave MapReduce applications environments. The scheduler adopts a genetic algorithm approach to search the optimal task-level configurations based on the feedback reported by a task analyzer. In (Li et al., 2017), authors propose a scheduling algorithm capable of minimize inter-DC traffic of a cross-DC MapReduce job by formulating an optimization problem that jointly optimizes input data fetching and task placement. Their solution relies on an optimization model based on historical statistics for a given Job. We also considered works in literature not focusing on MapReduce, although focusing on similar scenarios. For example, authors in (Convolbo et al., 2016) propose a data-replication aware scheduling algorithm which can makes use of the replicas to optimize the makespan of a job submitted to a geo-distributed system. Their approach relies on a global centralized scheduler that maintains a local FIFO queue for all submitted jobs. The system model they propose, does not involve data transfers because of the expensive delay in geo-distributed environments. A given task can only be scheduled on a data center which already has its required data. Authors in (Yu and Pan, 27) leverage on optimal data allocation to improve the performance of data-intensive applications. Specifically, it addresses the balanced data placement problem for geographically distributed computing nodes, with joint considerations of the localized data serving and the co-location of associated data.

2.4 Hierarchical approach

The Hierarchical approach's strategy is to make a smart decomposition of the job into multiple subjobs and then exploit the native potential of the plain Hadoop. In the following we revise some relevant works from this approach.

Hierarchical approaches mainly envisions two computing levels: a *bottom level*, where plain MapReduce is run on locally available data, and a *top level*, where a centralized entity coordinates the splitting of the workload into many subjobs and gathering and packaging of the subjobs' output. A clear advantage of this approach is that there is no need to modify the Hadoop algorithm, as its original version can be used to elaborate data on a local cluster. Still a strategy needs to be conceived to establish how to redistribute data among the available clusters in order to optimize the job's overall makespan.

The hierarchical approach relies on a Global Reducer component, which is responsible for collecting and packaging all the output produced by subjobs at the bottom-level. The packaging task consists in running a new Reduce Task on the overall collected data. An application designer is requested to implement the MapReduce algorithm in such a way that the applied operations are "associative", i.e., they can be run in a recursive way both at the bottom level and at the top level of the hierarchy, and the execution order of the operations does not impact on the final result (Jayalath, Stephen, and Eugster, 2014).

In (Luo et al., 2011) authors introduce a hierarchical MapReduce architecture and a load-balancing algorithm that distribute the workload across multiple data centers. The balancing process is fed with the number of cores available at each data center, the number of Map tasks runnable at each data center and the features (CPU or I/O bound) of the job's application. The authors also suggest to compress data before migrating them among data canters. Jayalath et al.(Jayalath, Stephen, and Eugster, 2014) list the issues connected to the execution of the MapReduce on geographically distributed data. They specifically address a scenario where multiple MapReduce operations need to be performed, one after the other, on the same data. They lean towards a hierarchical approach, and propose to represent all the possible jobs' execution paths by means of a data transformation graph to be used for the determination of optimized schedules for job sequences. The well-known Dijkstra's shortest path algorithm is then used to determine the optimized schedule. In (Yang et al., 2007) authors introduce an extra MapReduce phase named "merge", that works after map and reduce phases, and extends the MapReduce model for heterogeneous data. The model turns to be useful in the specific context of relational database, as it is capable of expressing relational algebra operators as well as of implementing several join algorithms.

The framework proposed in our work follows a hierarchical approach. We envisaged two levels of computation in which at the bottom level the data processing occurs and the top level is entitled with gathering the results of computation and packaging the final result.

With respect to the literature, our proposal distinguishes for some innovative ideas. First, we address a scenario where *multiple requests* for job execution arrive and need to be handled. Also, we introduce the concept of *data granularity*. Our approach allows for the whole data set to be broken into small pieces (*data blocks*) that can be smartly moved across the Sites according to specific needs.

Chapter 3

Hierarchical Hadoop Framework

The Hierarchical Hadoop Framework (H2F) we propose, addresses the issues concerning the computation of big amounts of data unevenly sparse in a distributed computing context that is composed of heterogeneous resources. Specifically, H2F faces three dimensions of imbalance: the non-uniform size of the data scattered throughout the computing context, the inhomogeneous capability of the computing resources on which the jobs will run and the uneven speed of the network links that interconnect the computing resources. Such a threefold imbalance exacerbates the problem of carrying on computation on big data. Current parallel computing techniques employed in these tough contexts have shown to be inefficient. One for all, the MapReduce and its well known open source implementation Hadoop failed to hit good performance because they only pursue the *data locality* principle and are completely unaware of the features of the underlying (imbalanced) computing context (Heintz et al., 2014).

In the following we present the motivating scenario that guided us through the design of the H2F. Suppose that company A is running its business in multiple countries worldwide, and that in every country a point of presence (POP or Site) is maintained that collects data from the company's activities. Each POP runs a local data center equipped with some computing resources. Computing resources owned by a POP are interconnected by means of high-speed links (intra-POP links) to form a powerful cluster. POPs are connected to each other via a Virtual Private Network (VPN) that is set up over geographic network links (inter-POP links). Also, all resources (both computing and links) are *virtualised* in order to form a cloud of resources that can be flexibly configured according to the need.

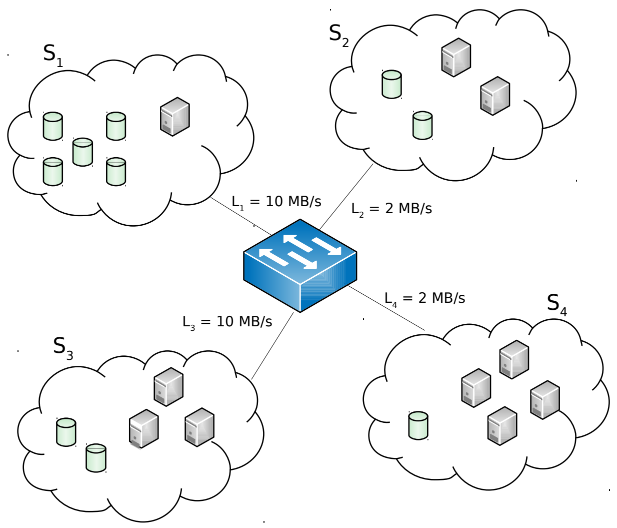


FIGURE 3.1: Motivating scenario

Suppose that each Site produces some raw data that need to be mined, and that the amount of such data varies from Site to Site. Suppose also that the computing power each Site is capable of devoting to the mining purpose is different from Site to Site. A snapshot of the just described situation is represented in the Figure 3.1. What the company would need is a way to quickly crunch all the data by means of some parallel computing techniques capable of exploiting at best the heterogeneous resources provided by its own computing infrastructure. Basically, our proposal pushes forward the idea of giving all Sites a mining task that is proportional to its computing power, i.e., the most powerful the Site the more data it will have to mine. Since the distribution of the data does not follow the distribution of the computing power (a big amount of data might happen to reside at a low-power Site) the solution we propose features the shift of data from Site to Site. Since those data movements would happen through inter-POP links, which are known to be slow, a cost is incurred. The H2F design strives to give an answer to the problem of how to exploit at best the most powerful Sites while minimizing the cost incurred for data shifts.

3.1 Design And Architecture

According to the MapReduce computing paradigm, a generic computation is called *job* (Dean and Ghemawat, 2004). Upon a job submission, a *scheduling system* is responsible for splitting the job in several *tasks* and mapping the tasks to a set of available nodes within a cluster. The performance of a job execution is measured by its completion time, or *makespan*, i.e., the time for a job to complete. Apart from the size of the data to be processed, that time heavily depends on the job's *execution flow* determined by the scheduling system (the sequence of tasks that the job is split in) and the computing power of the cluster nodes where the tasks are actually executed.

In an **imbalanced computing context** things get more complicated. When a job is submitted, the question is how to schedule at best imbalanced resources over unevenly distributed data in such a way that the job completion time get minimized. We address the context imbalance problem by adopting a **hierarchical approach** to the parallel data computation. According to this approach, a *top-level layer* is responsible for running the logic to *sense* the computing context's features and orchestrating smart computing plans, while a *bottom-level layer* is in charge of enforcing the plans while exploiting at best well known parallel computing techniques. The top-level layer, therefore, is where *smart* decisions must be taken in respect to, e.g., which of the many chunks of sparse data should be mined by a given computing resource. Basically, instead of letting the data be computed by resources that reside at the same data place, the intuition here is to move data wherever the largest possible computing power is available, provided that the benefit gained from the exploitation of the best computing resources overcomes the cost incurred for moving the data close to those resources. In order to support smart decisions at this level, we designed and implemented a **graph-based meta-model** capable of capturing the characteristics of the distributed computing contexts (in terms of computing resources, network links and data distribution) and of representing the job's potential execution paths in a structured way. The bottom-level layer, instead, is where the actual computation occurs, and is populated by *dummy* computing resources capable of running naive parallel computing algorithms.

In summary, the H2F grounds on the idea of getting the most powerful computing resources to a) *attract* as many data chunks as possible, b) *aggregate* them and c) *run* plain MapReduce routines on the overall gathered data. The proposed approach does not intend to deliver a new parallel computing paradigm for imbalanced computing contexts. It rather aims at preserving and exploiting the potential of the current parallel computing techniques, even in tough computing context, by creating two management levels that cater to the needs of *smart* computation and the needs for *fast* computation respectively.

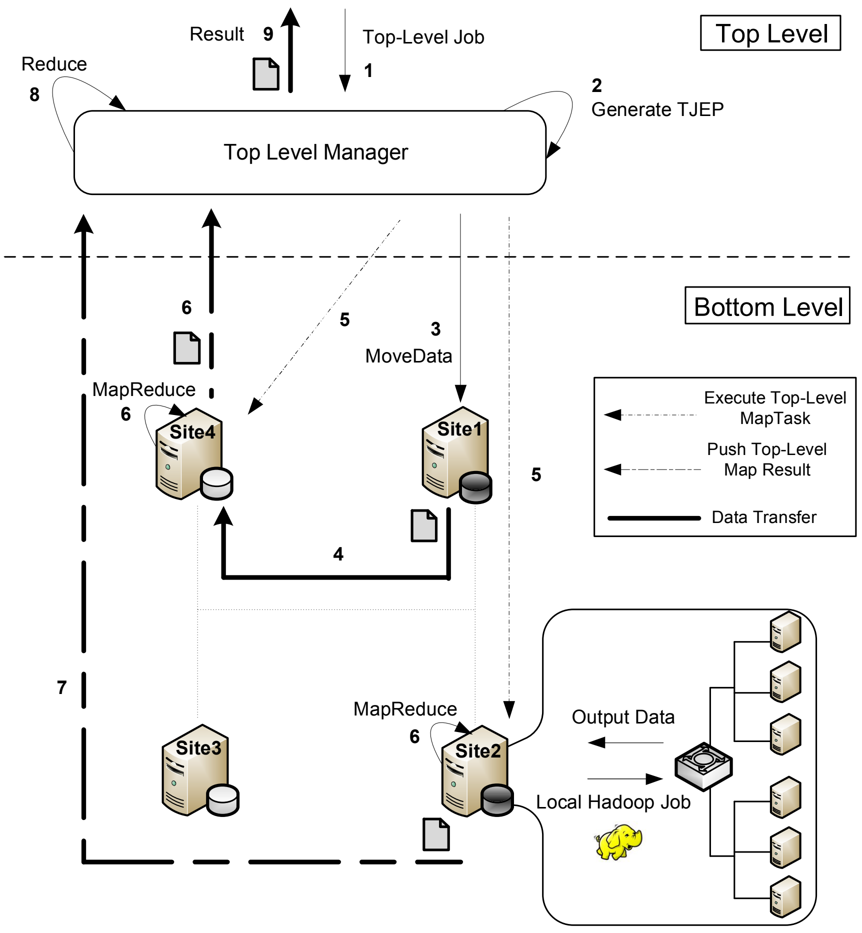


FIGURE 3.2: Job Execution Flow

The solution we propose is depicted in the scenario of Figure 3.2. Computing *Sites* populate the bottom level of the hierarchy. Each site owns a certain amount of data and is capable of running plain Hadoop jobs. Upon the reception of a subjob request from the top-level, a Site performs the whole MapReduce process on the local cluster(s) and returns the result of the elaboration to the top-level. The *Top-level Manager* owns the system's business logic and is in charge of the management of the imbalanced computing context. Upon the submission of a top-level job (i.e., a job that needs to elaborate on distributed data), the business logic schedules the set of subjobs to be spread throughout the distributed context, collects the subjob results and packages the overall calculation result. In the depicted scenario the numbered arrows describe a typical execution flow triggered by the submission of a top-level job. This specific case envisioned a shift of data from one Site to another Site, and the run of local MapReduce sub-jobs on two Sites. Here follows a step-by-step description of the actions taken by the system to serve the top-level job:

1. The Top-Level Manager receives a job submission.
2. A Top-level Job Execution Plan (TJEP) is generated using information on:
   - the status of the Bottom-level layer such as the distribution of the data set among Sites
   - the current computing availability of Sites,
   - the topology of the network and d) the current capacity of its links.
3. The Top-Level Manager executes the TJEP. Following the plan instructions, it orders Site1 to shift some data to Site4.
4. The actual data shift from Site1 to Site4 takes place.
5. According to the plan, the Top-Level Manager sends a message to trigger the run of subjobs on the Sites where data are residing. In particular, *Top-level Map tasks* are triggered to run on Site2 and Site4 respectively

(we remind that a Top-level Map task corresponds to a MapReduce sub-job).

6. Site2 and Site4 executes local Hadoop subjobs on their respective data sets.
7. Sites sends the results obtained from local executions to the Top-Level Manager.
8. A global reducing routine within the Top-Level Manager collects all the partial results coming from the Bottom-level layer and performs the reduction on these data.
9. Final result is forwarded to the Job submitter.

The whole job execution process is transparent to the submitter, who just needs to provide the job to execute and a pointer to the target data the job will have to process.

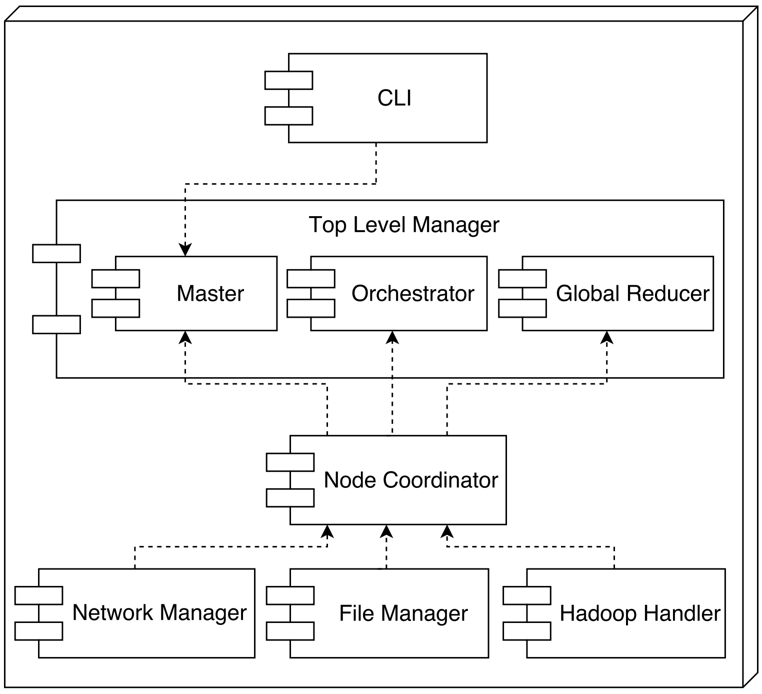


FIGURE 3.3: Overall architecture

The H2F architecture (see Figure 3.3) is made up of several modules that are in charge of taking care of the steps of the job's execution flow depicted in Figure 3.2. Each Site is an independent system that runs an instance of the architecture in one of two distinctive modes according to the role to be played. Out of the Sites, only one can activate the *orchestrating mode* that will let the Site play the Top-Level Manager role; the Top-Level Manager owns the "smart" orchestrating business logic of the system and coordinates all the system activities. The rest of Sites will have to activate the *computing mode* that will entitle them to act as Bottom-Level Executors, i.e., those who execute the dirty work of data crunching. Of course, there is nothing preventing a Top-Level Manager Site to also play the Bottom-Level Executor role.

The Top-Level Manager role is taken by enabling the following modules:

- **Master**: this module is in charge of receiving *Top-level job* execution requests, extracting the job information and forwarding them to the Orchestrator, that in turn exploits them for the generation of the Top level Job Execution Plan (TJEP). The Master enforces the generated plan. Moreover, at the end of the job processing, it receives from the Global Reducer the final result and forwards it to the job submitter.

- **Orchestrator**: is the component that generates the TJEP. The TJEP is generated by crossing information concerning the submitted job and the execution context in the form of Sites' overall available computing capacity and inter-Site bandwidth capacity.

- **Global Reducer**: collects the results obtained from the execution of sub-jobs on the local Sites and performs the reduction on those results.

The Bottom-Level Executor role is activated by turning on the following modules:

- **Node Coordinator**: owns all the information on the node (Site) status.

- **File Manager**: implements the routine for the load and the storage of data blocks and keeps track of the file namespace.

- **Hadoop Handler**: provides the APIs to interact with the features offered by plain Hadoop. It is designed to decouple the system from the underlying plain Hadoop platform thus making the framework independent of the specific Hadoop version deployed in the Site.
- **Network Manager**: handles the signaling among Sites in the network.

Finally, a command line interface component **(CLI)** is provided to users to submit a job execution request to the overall framework.

3.2 Job modelling

We have designed a graph-based representation model of a job's execution path which captures a) the characteristics of the imbalanced computing context (in terms of Sites's computing capacity, network links' bandwidth and data distribution over the Sites) and b) the job's features. By using this model, the Job Scheduler (a sub-module within the Orchestrator) automatically generates a finite number of potential job execution paths and then searches for the one that minimizes the job *makespan*. Basically, a job execution path is a sequence of actions driving the job execution. Actions may be of three types: Site-to-Site data shift, on-Site data elaboration (the Top-level *Map* phase), global reduction of elaborated data. We have spoken of "potential" execution paths as there may be many alternative paths that a job may follow in its execution. The discussion about the selection of the best alternative follows up in Chapter 4.

The job's execution path representation is based on a graph model where each *graph node* represents a **Data Computing Element** (Site), a **Data Transport Element** (network link) or a **Queue Element**. Queues have been introduced to model the delay suffered by a subjob when waiting for the computing resource to be freed by other subjobs. Arcs between nodes are used to represent the sequence of nodes in an execution path (see Figure 3.4). A node representing a computing element elaborates data, therefore it will produce an output data flow whose size is different than that of the input data; this node introduces a delay in the job process which is dependent on the computing capacity of the represented Site and on the size of the data to crunch. A node representing a data transport element just transports data, so for that node the output data size equals the input data size; obviously, a delay is introduced by this node which depends on the bandwidth capacity of the represented network link. A node representing a queue does not alter the data size but, obviously, introduces a delay in the job processing.

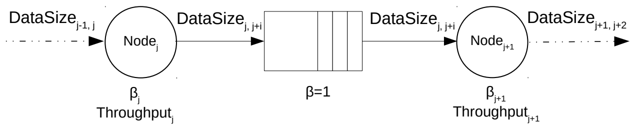


FIGURE 3.4: Nodes and arcs in the job's execution path model

Nodes are characterized by two parameters: the compression factor β_{app}, that is used to estimate the data produced by a node, and the $Throughput$, defined as the amount of data that the node is able to process per time unit. The β_{app} value for Data Computing Element is equal to the ratio of the produced output data to the input data to elaborate, while for the Data Transport Elements it is equals to 1 because there is no data computation occurring in a data transfer. The $Throughput$ of a Data Computing Element depends on the Site's computing capacity, while for the Data Transport Elements the Throughput corresponds to the link capacity. For a Queue element the β_{app} is set to 1 (no computation occurs); also, since the concept of $Throughput$ is not easily applicable to this node (it would depend on the computing process of other nodes), we will stick to the calculus of the delay introduced by the queue by estimating the residual computing time of the "competing" nodes (i.e., nodes of other jobs modeling the same contended computing resource).

In general, a node's *execution time* is defined as the ratio of the input data size to the $Throughput$ of the node. In the Figure 3.4, the label value of the arc connecting node $Node_j$ to node $Node_{j+1}$ is given by:

$$DataSize_{j,j+1} = DataSize_{j-1,j} \times \beta_j \tag{3.1}$$

A generic node $Node_j$'s execution time is defined as:

$$T_j = \frac{DataSize_{j-1,j}}{Throughput_j} \tag{3.2}$$

Both the β_{app} and the $Throughput$ are specific to the job's application that is going to be executed (i.e., the particular algorithm that elaborates on the data), and of course are not available at job submission time. To estimate these parameters, we run the job on small-sized samples of the data to elaborate (a chunk of input data, e.g. 1 GB) on test machines with known computational resources. These test results allow us to build an **Application Profile** made of both the β_{app} and the $Throughput$ parameters, which is be used as input for the elaboration of the TJEP. The estimation procedure is described in details in section 3.3, where we proposed a study of the job's Application Profile and analyzed the behavior of well known Map Reduce applications.

As already anticipated, the execution path model is influenced by the distributed computing context. The number of the job's potential execution paths depends on: a) the set of computing nodes, b) the links' number and capacity and c) the data size. In our approach the entire data set can be split into **data blocks** of same size, with the aim of moving certain data blocks from Site to Site. A study on the estimate of the *optimum data block size* is conducted in Section 3.4. Data granularity provides the advantage of having more flexible schemes of data distribution over the sites. Of course, the higher the number of the data blocks (i.e., the smaller the data block size), the higher the number of job's potential execution paths. Moreover the graph that models a job's execution path has as many branches as the number of Map Reduce sub-jobs that will be run. Every branch starts at the root node (initial node) and ends at the Global Reducer's node. We define the execution time of a branch to be the sum of the execution times of the nodes belonging to the branch; the Global reducer node's execution time is left out of this sum. Note that execution carried out through branches are independent of each other's, so branches will have different execution times. In order for the Global reducing to start, all branches will have to produce and move their results to the reducer Site. Therefore the longest among the branches' execution times determines when the global reducing is allowed to start. Finally, the execution time of the Global reducer is given by the summation of the sizes of the data sets coming from all the branches over the node's estimated throughput.

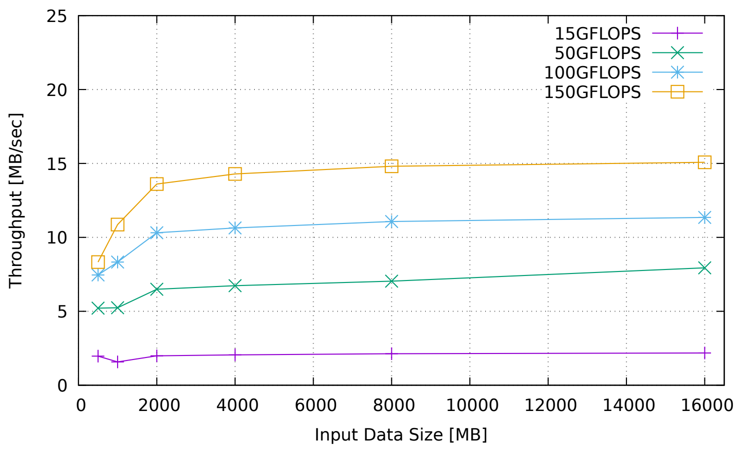
3.3 Application Profiling

The formulas (3.1) and (3.2) bind the execution time of a node (either computing or transport) to the β_{app} and the $Throughput$. Since these parameters are not available at job submission time, but are nonetheless necessary for the computation of the job's best execution path, we devised a procedure to make an accurate estimate of them. The basic idea is that once a MapReduce job has been submitted, some test machines run the job on small-sized samples of the data targeted by the job. From the study of the job's behaviour on the test machines, we are able to build the job's **Profile**, which is made up of the estimated β_{app} and $Throughput$ respectively. Eventually, that profile will be used in the computation of the job's execution time.

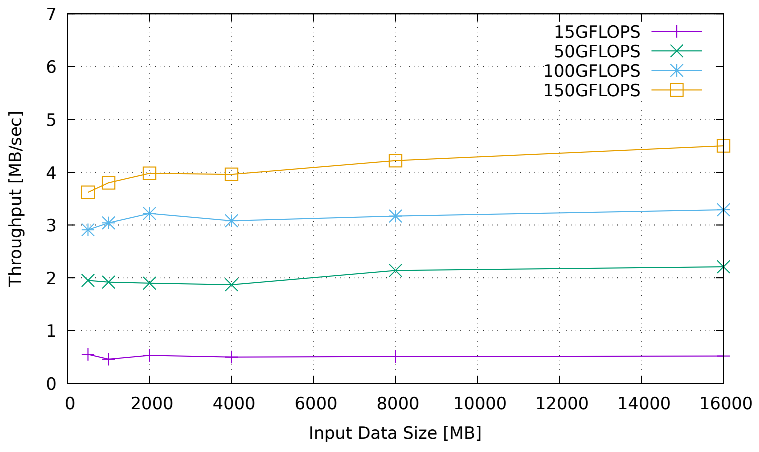
The strategy is very simple: we measure the β_{app} and the $Throughput$ on the test machines (nominal values) and try to guess what the β_{app} and the $Throughput$ would be on the real computing nodes (guessed values). Of course, the estimate procedure is prone to errors, which will unavoidably propagate up to the computation of the job's execution time. The challenge here is to keep the estimate error little enough to avoid that the Scheduler's computed execution time appreciably deviates from the one that will eventually be observed on the real computation. Chapter 5 will report on the gap between the job's execution time guessed by the Scheduler and the one observed when running the job in a real distributed computing context.

We approached the study of the job's profile by investigating on the behaviour of two well-known MapReduce applications: *WordCount* and *InvertedIndex*. We intended to study the behaviour of both the Map and the Reduce phases of each application when varying the computing power of the machine running the job and the size of the data to be crunched. Tests were run on four machines having different computing power. The Giga-flop

(Gflop) was used as unit of measure of the computing power. Specifically, the machines chosen for the tests had 15, 50, 100 and 150 Gflops respectively[1].



(a)



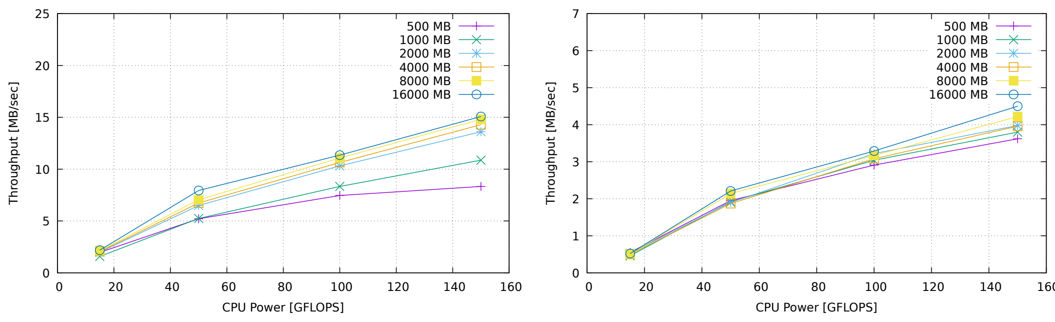
(b)

FIGURE 3.5: WordCount application: $Throughput$ VS data size:
a) Map step, b) Reduce step

Further, in the study sample data of the following sizes were used: 500MB, 1GB, 2GB, 4GB, 8GB and 16GB. In Figure 3.5 we report the values of the $Throughput$ observed with data size variation in the WordCount case. The

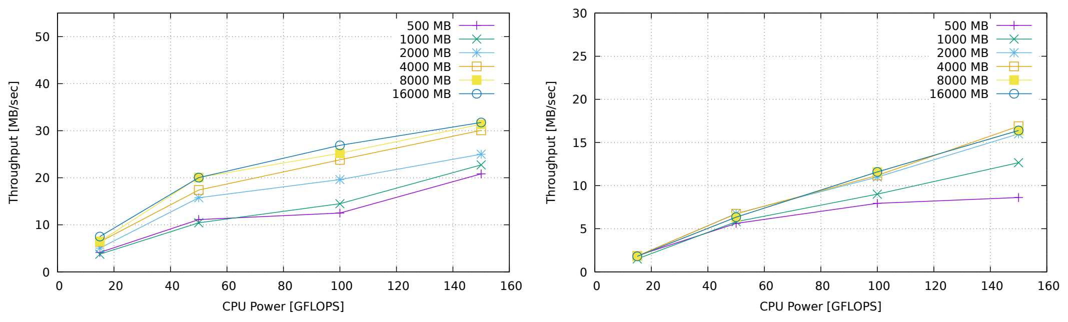
[1]The machines' computing power was assessed with the Linpack benchmark (http://www.netlib.org/benchmark/hpl/)

graphs are parameterized to the computing power. They show that, independently of the computing power value, the $Throughput$ is not affected by the variation of the data size (apart from the cases of data size smaller than 1 GB).



(a) (b)

FIGURE 3.6: WordCount: $Throughput$ VS computing power: a) Map step, b) Reduce step



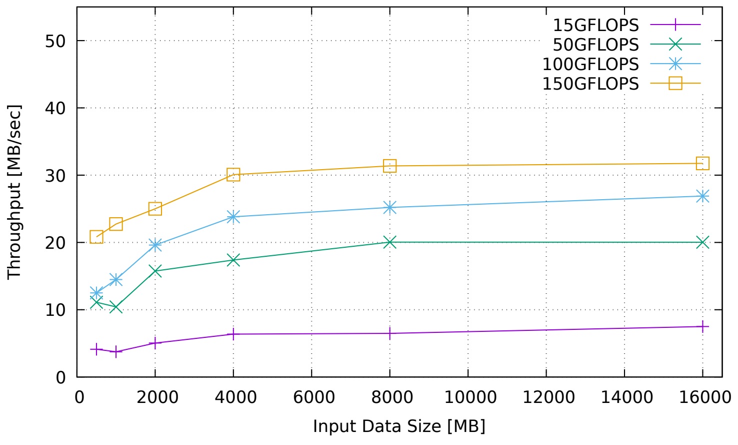
(a) (b)

FIGURE 3.7: InvertedIndex: $Throughput$ VS computing power: a) Map step, b) Reduce step

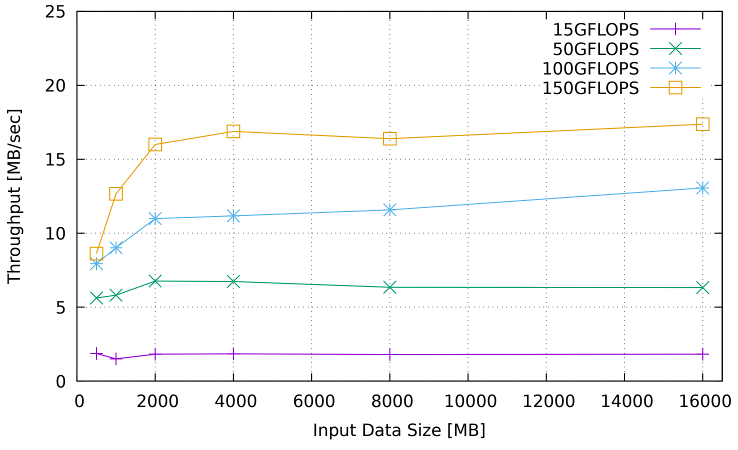
In Figure 3.6 we report the trend of the WordCount's $Throughput$ (both in the Map and in the Reduce case) with computing power (CP) variation. This time, the graphs are parameterized to the data size. Also, for almost all data sizes the $Throughput$ can be considered a linear function of the CP (only the 500MB curve in the Map graph raises an exception) and the observed slope is almost the same no matter the data size. For instance, looking at Figure 3.6(a) the Map slope value is approximatively:

$$MapSlope_{WordCount} = \frac{14.81 - 7.04}{150 - 50} = 0.078 \frac{MB}{s \cdot Gflops}$$

We repeated the experiment for the InvertedIndex application.



(a)



(b)

FIGURE 3.8: InvertedIndex application: $Throughput$ VS data size: a) Map step, b) Reduce step

In Figure 3.7 we report the $Throughput$'s trend when the CP varies. Again, we can observe a linear dependence of the $Throughput$ from the CP. For the

InvertedIndex Map the observed slop value is:

$$MapSlope_{InvIndex} = \frac{31.37 - 20.05}{150 - 50} = 0.113 \frac{MB}{s \cdot Gflops}$$

In Figure 3.8, the ($Throughput$ vs data size) graph is depicted; for the InvertedIndex application we observed that the $Throughput$ is not affected by the variation of the data size as well. From these observations, we can conclude that if we had to guess the $Throughput$ of a MapReduce application on a computing node with a certain CP, then a linear function of the CP may confidently be used to estimate it. Of course, the Map and Reduce's $Throughput$ slope have to first be computed by means of the just described procedure.

The study of the β_{app} aimed to prove that this parameter is not influenced by both the heterogeneity and the size of the input data that a computing node has to process. Again, the study was conducted on the WordCount and the InvertedIndex applications. For each application, the test envisioned two experiments. In the first experiment, the application was executed on different data samples (taken from Wikipedia) of 500MB, 1GB, 2GB and 4GB size respectively. In the second experiments, the 4GB data sample was split into smaller data blocks, and the application was run on every block.

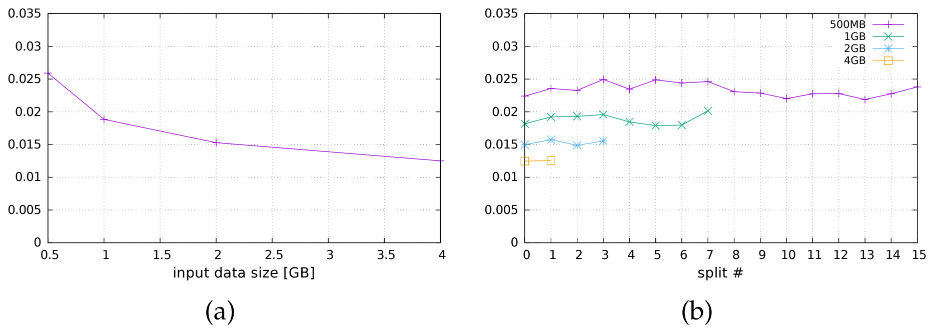


FIGURE 3.9: β_{app} trend in the WordCount application

In particular, four sub-experiments were run where the 4GB data sample was split into data blocks of 500MB, 1GB and 2GB size respectively. The results observed for the WordCount's β_{app} are shown in Figure 3.9. The graph

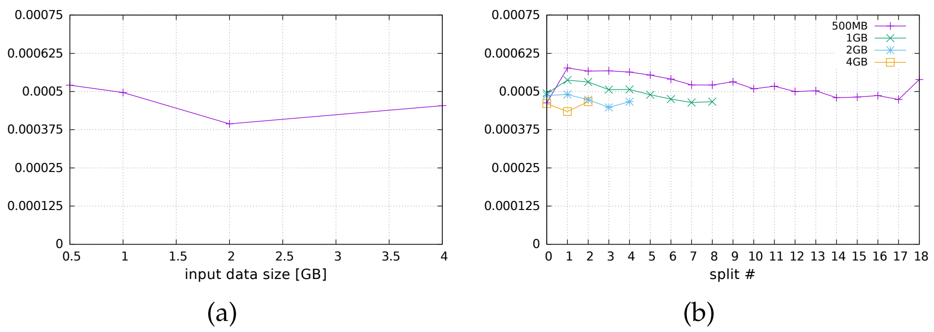


FIGURE 3.10: β_{app} trend in the InvertedIndex application

in Figure 3.9(a) shows that the variance between the maximum and the minimum of the β_{app} values is negligible, so we can deduce that the size of input data does not affect the compression factor in an appreciable way. Figure 3.9(b) reports the result of the second experiment. The reader may notice that whatever the particular data sample (split, in the figure), no appreciable variation of the β_{app} can be observed.

The same test was specularly conducted for the InvertedIndex application on a 8GB data sample (specifically, a StackOverFlow dump). Results are reported in Figure 3.10. The considerations made for the WordCount application apply for the InvertedIndex as well. To conclude, the investigation showed that the β_{app} is invariant to both the size and the type of the considered input data, and that the value of β_{app} computed on a small-sized data sample can be reasonably used as a good estimate of the β_{app} of a big data set as well. The estimate error among the β_{app} and the $Throughput$ on the test machines and the β_{app} and the $Throughput$ on the real computing nodes is negligible and does not affect Scheduler's computed execution time. Experiments described in Chapter 5 on the gap between the job's execution time guessed by the Scheduler and the one observed when running the job in a real distributed computing context support this statement and prove the feasibility of the approach.

3.4 Data Block's Size

As mentioned in Section 3.2, our approach allows the entire data set to be split into **data blocks** of same size, with the aim of moving some data blocks from Site to Site. In this section we study the impact that varying the data block size might have on the performance of the job, and propose a mechanism to estimate the data block size that maximizes the performance. Of course, the data block size is not an independent variable and its impact must be regarded along with that of other variables that characterise the distributed computing environment, namely the available computing resources and the network links' capacity.

We recall the intuition on which we grounded our proposal: in the aim of exploiting at best the distributed computing resources, a Site holding great computing capacity ought to attract as much data as possible, provided that the network paths followed by those data do not penalize too much the gathering of the data at that Site. Moving data to a Site turns out to be convenient if a) the Site offers much computing capacity and b) the network connecting the data's initial location and the Site's location provides a good capacity. To measure the fitness of a Site in regard to this purpose, we introduce the following parameters:

- CPU_{rate}: is the Site's computing power normalized to the overall context's computing power.
- $Connectivity$: represents the capability of the Site to exchange data with other sites in the computing context.

The CPU_{rate} of Site S_i is given by:

$$CPU_{rate}(S_i) = \frac{CPU(S_i)}{\sum_{j=1}^{N} CPU(S_j)}$$

The $Connectivity$ of Site S_i is defined as the arithmetic mean of all end-to-end nominal network bandwidths between the considered Site and the other Sites in the distributed computing context. It is represented as:

$$Connectivity(S_i) = \frac{\sum_{j=1}^{N} Bandwidth_{i,j}}{N-1}$$

where $Bandwidth_{i,j}$ is the nominal network capacity between node i and node j, and N is the number of Sites belonging to the distributed context. We also define the $Connectivity_{rate}$ as the connectivity of a Site normalized to the overall connectivity of the distributed context. So, for Site S_i it is given by:

$$Connectivity_{rate}(S_i) = \frac{Connectivity(S_i)}{\sum_{j=1}^{N} Connectivity(S_j)}$$

Now, for each Site we compute a score as a linear function of both the $Connectivity_{rate}$ and the CPU_{rate} described above. The *score function* is given by:

$$Score(S_i) = K * CPU_{rate}(S_i) + (1-K) * Connectivity_{rate}(S_i)$$

where K is an arbitrary constant in the range [0,1]. We ran several tests in order to best tune the K value. In our scenarios good results were observed by setting K values close to 0.5, i.e., values that balance the contribution of the network and of the computing power. Seeking for an optimum K value, though, was out of the scope of this work.

Finally, we define the $NominalBlockSize$ as:

$$NominalBlockSize = \min_{\forall i} Score(S_i) * JobInputSize \tag{3.3}$$

where $JobInputSize$ is the total amount of data to be processed by the given Job. The $NominalBlockSize$ is the data block size found by the described heuristic method. Of course, there is no guarantee that such a value is the optimum (i.e., the one that maximizes the job performance), but it is very likely that the optimum value is near the $NominalBlockSize$. A procedure is then run to seek for the optimum in the proximity of the $NominalBlockSize$.

Given the $NominalBlockSize$, we generate a set of values in the range $[0.5 \cdot NominalBlockSize, MinChunkSize]$, where $MinChunkSize$ is to the minimum size of the data chunks located at the Sites. For each data block size in the range, we compute the best job execution path by feeding the LAHC scheduling algorithm with the current context, the job information and the selected data block size. Finally, the data block size producing the minimum job makespan is selected as the best candidate.

For a better comprehension of the mechanism, we provide the following example. Let us suppose that the computing context is made up of three Sites (S_1, S_2 and S_3) interconnected through a star network topology, as depicted in the Figure 3.11. In this example, the overall data to process is 10GB large and is distributed in this way: $S_1 \leftarrow 1GB$, $S_2 \leftarrow 3GB$ and $S_2 \leftarrow 6GB$.

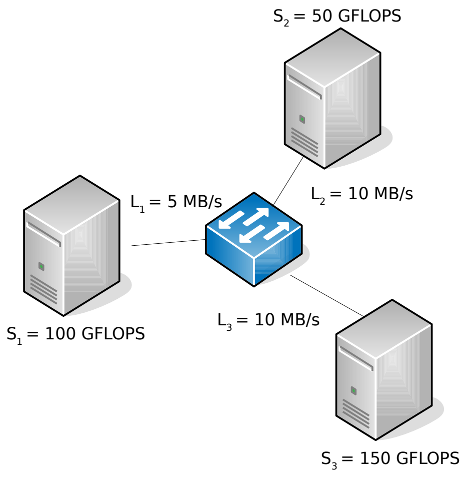


FIGURE 3.11: Example Topology

The CPU_{rate}, $Connectivity$ and $Connectivity_{rate}$ for Site S_1 will be respectively:

$$CPU_{rate}(S_1) = \frac{100}{50 + 100 + 150} = 0.33$$

$$Connectivity(S_1) = \frac{5+5}{2} = 5$$

$$Connectivity_{rate}(S_1) = \frac{5}{5 + 7.5 + 7.5} = 0.25$$

By setting K to 0.5, the score of Site S_1 will be:

$$Score(S_1) = 0.5 * CPU_{rate}(S_1) + 0.5 * Connectivity_{rate}(S_1) = 0.33$$

Similarly, the $Score(S_2)$ and $Score(S_3)$ will be respectively 0.271 and 0.43. The minimum score is then on Site S_2; according to our approach, the selected $NominalBlockSize$ will be:

$$NominalBlockSize = Score(S_2) * JobInputSize = 0.271 * 10GB = 271MB$$

In this case, the optimal size search procedure will have to seek for the optimal size in the range $[135MB, 1GB]$, as 1 GB is the size of the smallest among the data chunks (located at S_1).

In order to assess the effectiveness of the discussed explorative approach some experiments were run. Starting from the reference scenario depicted in the Figure 3.12, we designed some configurations (that are meant to represent different, concrete computing scenarios) by tuning up the following parameters: the CPU Power of each Site, the network links' capacity and the initial distribution of the data among the Sites. The experiments were done by simulating the execution of a job that needed to process 10GB of data. Specifically, we considered a sample job for which a β_{app} value of 0.5 was estimated. The parameter that is put under observation is of course the job *makespan*. The objective of the experiment is to prove the capability of the job scheduler to derive the execution path ensuring the job's optimum makespan by varying the data block size. Of course, this has to hold true independently of the distributed computing scenario that is being considered.

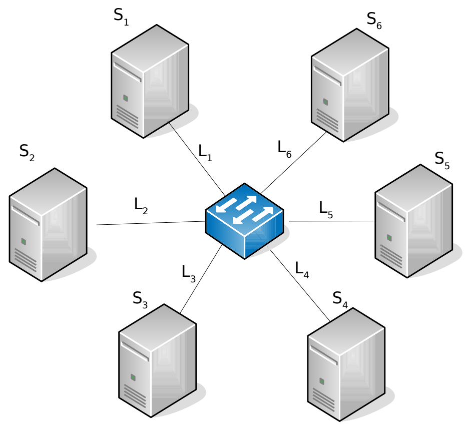


FIGURE 3.12: Reference Topology

	Sites [GFLOPS]						Links [MB/sec]					
Config	S_1	S_2	S_3	S_4	S_5	S_6	L_1	L_2	L_3	L_4	L_5	L_6
1	50	50	50	50	50	50	10	10	10	10	10	10
2	50	25	50	25	50	25	10	10	10	10	10	10
3	50	50	50	50	50	50	5	10	5	10	5	10
4	25	50	25	50	25	50	10	5	10	5	10	5
5	50	25	50	25	50	25	10	5	10	5	10	5

TABLE 3.1: Configurations used for the tests

	Config_1	Config_2	Config_3	Config_4	Config_5
NominalBlockSize	1.67GB	1.38GB	1.52GB	1.52GB	1.24GB

TABLE 3.2: Nominal Block Size computed for the five Configurations

We created five configurations, which have been reported in the Table 3.1. Each configuration represents a specific distributed computing context. As the reader may notice, going from $Config_1$ to $Config_5$ the computing scenarios get more and more unbalanced.

In the tests, for each configuration the job scheduler was fed with the context data. According to our mechanism, the optimal data block size is to be sought in the range $[0.5 * NominalBlockSize, MinChunkSize]$. For the chosen configurations, the calculus in the formula 3.3 gave values reported in the Table 3.2.

The exploration procedure extracts a finite number of points belonging to that range, and that will constitute the points to be checked. The points are selected in this way: starting from the left edge of the range, the next points to be selected are obtained by stepping to the right by a fixed quantity of 10% of the $MinChunkSize$. Of course, for a finer search a smaller step might be chosen: this would generate a higher number of points, which in turn would mean a heavier burden to the job scheduler with no guarantee of a consistent benefit.

In the first battery of tests, the job's input data were equally spread so that Sites would host 1.7GB data each. Results from tests run on the five configurations are shown in Figure 3.13, where the assessed job's makespan is plotted against the data block size. It is important to notice that, with the exception of $Config1$, for all of the configurations the graph has mainly a parabolic trend with a clearly identified minimum which is very close to the $NominalBlockSize$.

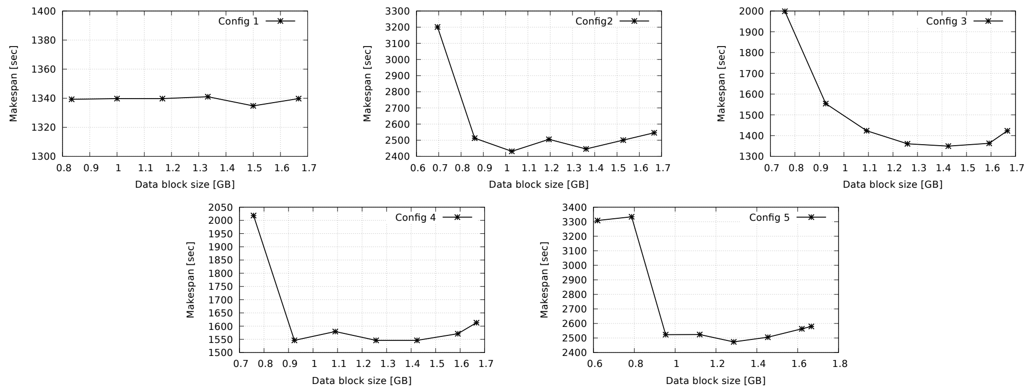


FIGURE 3.13: First battery of tests: results from the scenarios where data are evenly distributed among Sites

In the second battery of tests the configurations almost remained the same; the only modification regarded the initial data placement. In particular, a single Site was chosen to host the entire 10GB data. For the sake of completeness, we repeated the test selecting every time a different Site to host the data. It was observed that the data's initial placement is not a bias to the experiment. In the Figure 3.14 we report the results of the case where the Site chosen to host the data is S_4. This time the trend is more like a slope, but again the minimum is localised around the $NominalBlockSize$.

In the last battery of tests we split up the job's input data into two equally sized data chunks and distributed them among two Sites in the cluster. We repeated the experiment by taking into account each pair of Sites in the scenario. The choice of the Sites pair was observed to be uninfluential. In the Figure 3.15 the results obtained by equally distributing data on S_1 and S_3 are reported for each of the five configurations. The same considerations made before on the minimum apply here.

To conclude, the intuition that the optimal data block size depends on the connectivity and on the computing power distribution of the computing

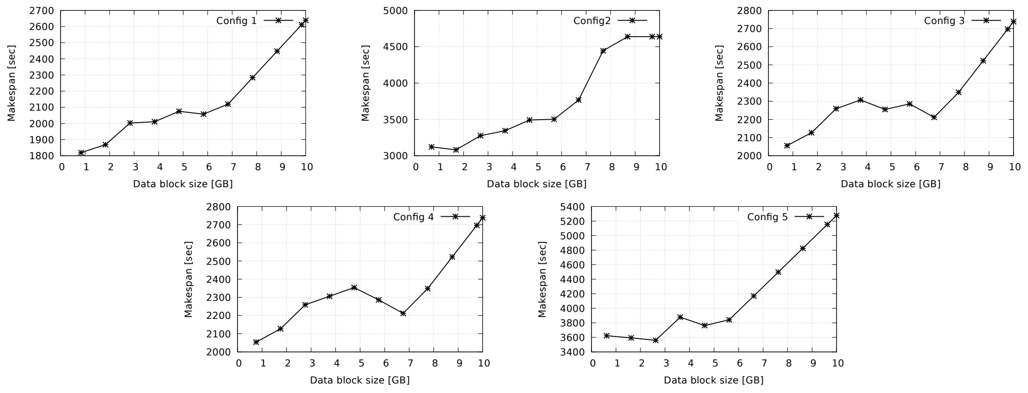


FIGURE 3.14: Second battery of tests: results from the scenarios where overall data are initially placed in S_4

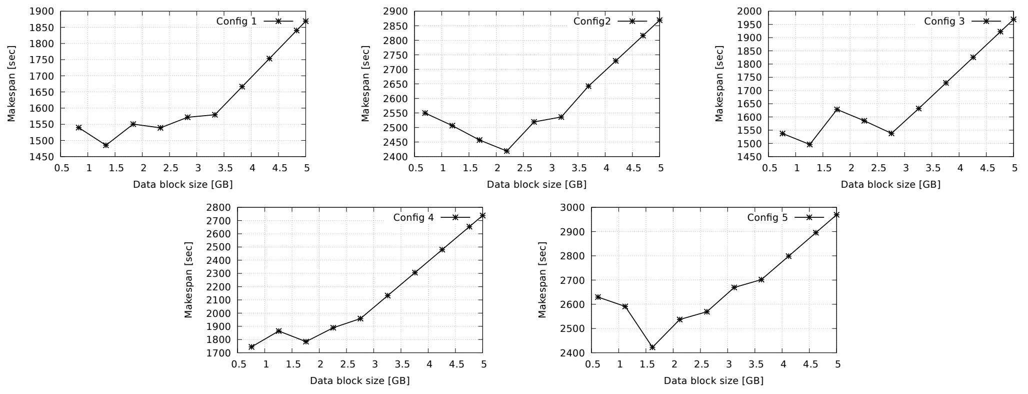


FIGURE 3.15: Third battery of tests: results from the scenarios where data are initially placed on Sites S_1 and S_3

context (formalized in the formula 3.3) was proved to be true. According to this, the procedure of the search for the optimal data block size has been embedded as a subroutine of the job scheduler in its task of searching for the best job execution path. Given that the optimum data block size was always found to be close to the $NominalBlockSize$ value, the exploration procedure will focus on a very limited boundary of that value (3 to 4 points will be checked at most).

Chapter 4

Job Scheduler

The TJEP generated by the Orchestrator contains both directives on how data have to be re-distributed among Sites and the articulation of subjobs that have to be run on the Sites. In order to compute the TJEP, the Orchestrator calls on a scheduling strategy that is capable of predicting which execution path will ensure the highest job performance in terms of completion time.

When dealing with a single job, the job scheduler strives to find the combination of available resources that guarantees the best possible performance to the job itself. This scenario describes a "happy path" that occurs when all the resources (computing elements and network links) are available to the job and do not need to be shared with other pending jobs. Unfortunately, there may be many job requests to be served even at the same time, and usually the jobs' arrival times are never known beforehand. In next section we discuss a job scheduling strategy conceived to serve multiple job requests.

4.1 Multi-Jobs Scheduling

The objective is to get the job to complete in the shortest possible time by putting at its disposal all the resources requested by the job, provided that those resources are not being used by other jobs.

The scheduler will then enforce a first-come-first-served strategy, reserving resources to jobs according to their arrival time. When a job is assigned a computing resource (a Site, in our case), the resource can not be either shared with other running jobs or pre-empted by any new job. In the case of network resources, instead, links are shared by all jobs that need to move data from Site to Site. As a result, a job may happen to be put on a wait state if the desired computing resource is busy; on the other hand, it may also experience a transfer delay if it needs to move data through a network link that is shared with other jobs.

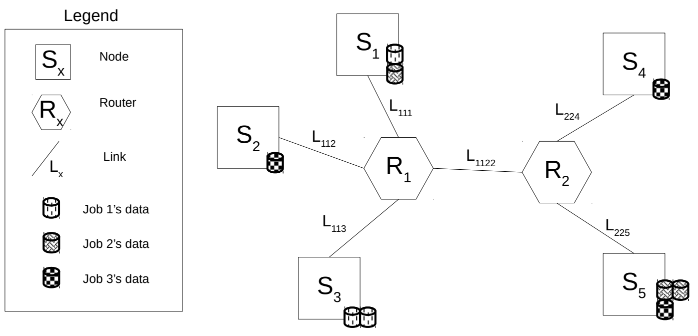


FIGURE 4.1: Reference scenario

In order to give the reader an idea of how the job scheduling works, we provide a simple example. The reference scenario is a distributed computing environment made up of five Sites which are interconnected through geographic links as represented in Figure 4.1. Upon a job submission, the job scheduler elaborates a TJEP, according to which the target data might be conveniently shifted from Site to Site and the job is split into a number of subjobs that will eventually run on the Sites where data reside. H2F considers data logically and physically divided into *data blocks* of fixed size. Thanks to this distinctive feature, H2F is capable of generating TJEPs which allow for the transfer of even small portions of data from a site to one or more sites.

When crafting a TJEP for a new submitted job, the job scheduler will have to also take into account the resource usage plan of already running jobs. In the example we provide, three jobs are submitted one after the other and are served according to their arrival time. In Figure 4.2 we show how the job scheduler handles the three job execution requests. Each top-level job is split into subjobs which are instructed to run into the Sites that hold the data. For instance, Job_1 is split into $Job_{1.1}$, $Job_{1.2}$ and $Job_{1.3}$ which will run in Sites S_1,

S_2 and S_3 respectively. Subjobs are scheduled to run on a given Site only when the data to be elaborated are available on that Site. Depending on the TJEP instructions, in fact, some data may happen to be moved to a different Site before scheduling the subjob that will have to crunch them. This is the case of $Job_{1.2}$, whose start is delayed until the target data are moved to their final computing destination (the reader may notice the time gap between Job_1's submission time and $Job_{1.2}$'s start time). Conversely, Job_3's subjobs are scheduled to run right after Job_3 is submitted, as no data shift is prescribed in its TJEP. But, while Job_1's subjobs are allowed to run at their scheduled start time (computing resources are free at that time), all Job_3's subjobs will have to wait until their requested computing resources are freed ($Job_{3.1}$ waits until $Job_{1.2}$ releases S_2, $Job_{3.2}$ waits until $Job_{2.2}$ releases S_4, $Job_{3.3}$ waits until $Job_{2.3}$ releases S_5).

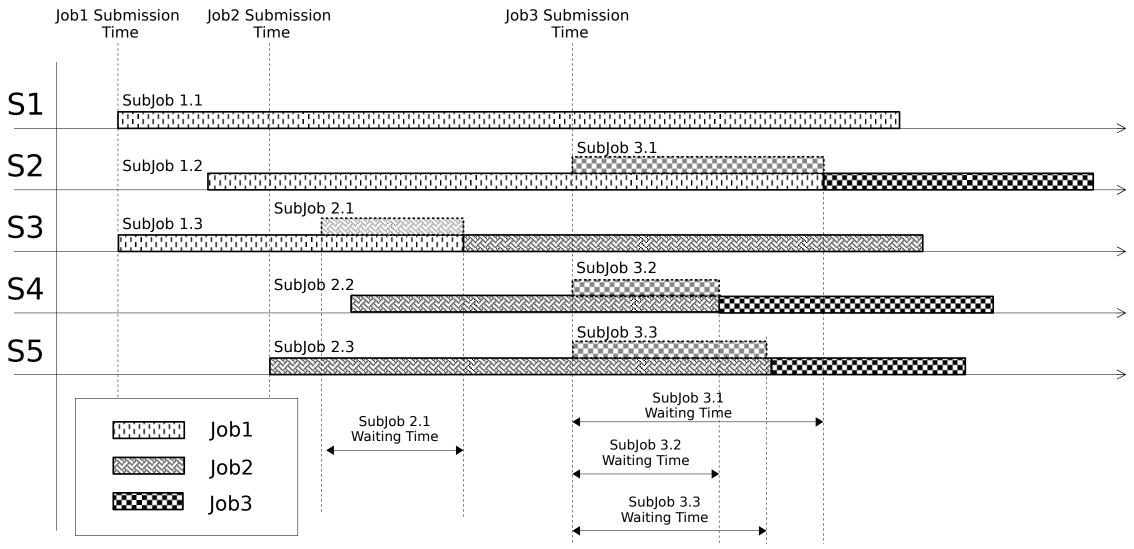


FIGURE 4.2: Multi job scheduling

In the previous chapter we described the representation model of a job's execution path, now we show an example of execution path modeling on a reference scenario. The example will refer to distributed computing context shown in Figure 4.1 and to the job submission timing shown in Figure 4.2.

We then consider three jobs requesting to execute on some data sparsely distributed among the five Sites. In this example we will assume that data

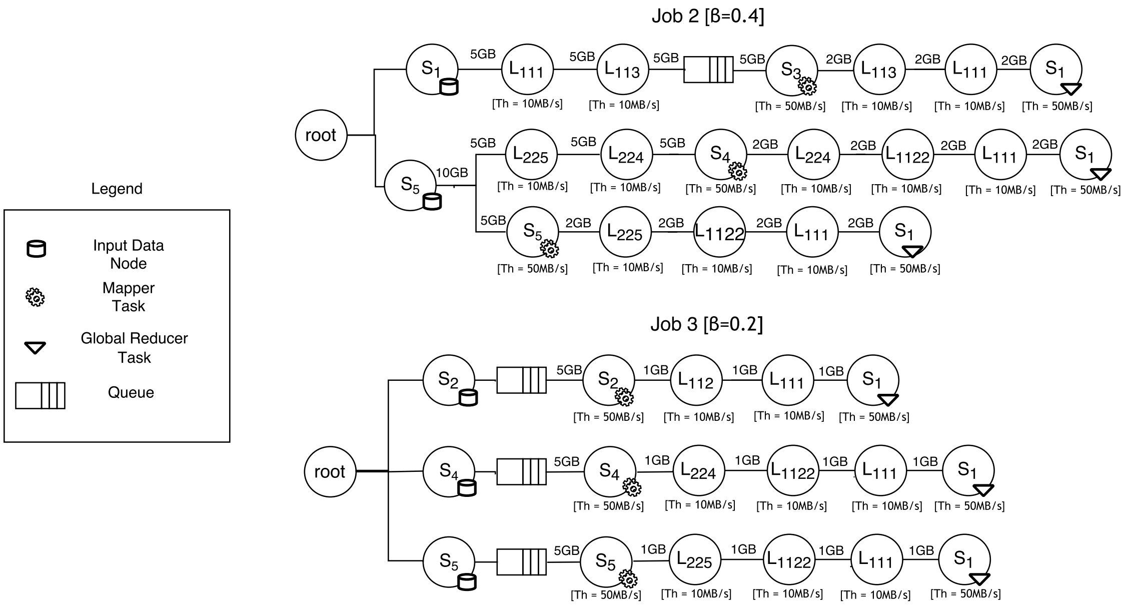


FIGURE 4.3: Graph modeling potential execution paths for Job 2 and Job 3

are partitioned in data blocks, each having a size of 5 GB, and that every job needs to access and elaborate 15 GB of data. Specifically: Job_1 is requesting to access data initially residing in S_1 (one data block of 5GB) and S_3 (two data blocks of 5GB each); Job_2 is requesting to access data initially residing in S_1 (one data block) and S_5 (two data blocks); Job_3 is requesting to access data initially residing in S_2, S_4 and S_5, each holding one data block of 5GB. Figure 4.3 shows the graph model representing two potential execution paths generated by the job scheduler for J_2 and J_3 respectively. For the sake of simplicity and clearness of the picture, we omitted to report the path for J_1. According to the model, the plan for J_2 envisions three execution branches:

1. a data block is moved from S_1 to S_3 traversing the links L_{111} and L_{113}; here, the subjob $Job_{2.1}$ will suffer a delay because the resource S_3 is locked by subjob $Job_{1.3}$ (see Figure 4.2); once S_3 has been released, $Job_{2.1}$ accesses and elaborates the data block; the output of the elaboration is shifted to S_1 through the links L_{113} and L_{111}; here, the global reduction

takes place[1].

2. a data block is moved from S_5 to S_4 traversing the links L_{225} and L_{224}; here, $Job_{2.2}$ accesses and elaborates the data block; the output of the elaboration is shifted to S_1 through the links L_{224}, L_{1122} and L_{111}; here, the global reduction takes place.

3. $Job_{2.3}$ accesses and elaborates the data block residing in S_5; the output of the elaboration is shifted to S_1 through the links L_{225}, L_{1122} and L_{111}; here, the global reduction takes place.

Likewise, the plan for J_3 envisions three execution branches. In none of the branch there is a data shift: all subjobs will work on the data blocks initially residing in the targeted Sites. Interestingly, all the subjobs suffers a delay, because the requested computing resources are found locked by other subjobs. We point out again that the paths depicted in the Figure 4.3 are only potential execution paths (not necessarily the best) for J_2 and J_3. For any job, the task of the scheduler is to explore all the potential execution paths and to identify the one that provides for the shortest execution.

4.2 Scheduling Algorithm

The job scheduling algorithm's task is to search for an execution path which minimizes a given job's execution time. Let us consider the whole job's makespan divided into two phases: a *pre-processing* phase, during which the job execution plan is defined, and a *processing* phase, that is when the real execution is enforced. Obviously, the better the plan, the faster the job's execution. Yet, the benefit of a good plan might get wasted if the time it takes to conceive the plan (the pre-processing phase) increases without control. In this section we describe the job scheduling strategies for the execution plan generation.

[1]Note that the global reduction step will start only after all the data elaborated by the subjobs have been gathered in S_1.

Our first approach's strategy was to explore the entire space of all potential execution paths for a job and to find the one providing the best execution time by leveraging on the combinatorial theory.

All potential execution paths are explored by applying combinatorics operation. First, the algorithm analyzes all computing nodes (Sites) to find the best mapper nodes' combination. A combination is a way of selecting mapper nodes from the collection of all available Sites. The algorithm computes a *k-combination* of all nodes with k ranging from 1 to the number of available Sites, where K is the number of mappers. The overall number of *k-combination* is:

$$\sum_{k=1}^{n} C_{n,k} = \sum_{k=1}^{n} \frac{n!}{k!(n-k)!} \tag{4.1}$$

For each k-combination, the algorithm computes the needed data transfers. Those transfers consists in moving *data blocks* from the Sites that hold them to the mapper nodes. The basic assumption we make is that overall data to be processed must be divided into equally sized data blocks. Therefore, Sites holding data will hold one or more data blocks. Those blocks need to be re-distributed to mapper nodes. Of course, a Site holding data may also be a mapper, therefore will happen to be assigned one or more data blocks. In order to represent all possible assignments of data blocks to mappers, we call on the *Integer Partitioning* technique (Andrews, 1976). A partition of a positive integer n, also called an integer partition, is a way of writing n as a sum of positive integers. It is possible to partition n as a sum of m addenda, in which case we will refer to it as a partition of the number n of order m. Finally, our objective is to compute the partitions of the integer n of order m, where n is the total number of data blocks and m is the number of nodes candidates to become mappers.

By the notation $P(n, m)$ we refer to the number of partitions of the integer number n in the order m. So, in the case that we have 5 data blocks to distribute over 2 Sites, two configurations are possible: 1) 1 data block on one Site, 4 data blocks on the other one; 2) 2 data blocks on one Site, 3 data blocks on the other one;. Generalizing, the overall number of partitions of a number

n in all the orders m=1,2,..,n is:

$$P(n) = \sum_{m=1}^{n} P(n,m) \tag{4.2}$$

Of course, the data blocks configuration tells us just the ways to "group" data blocks for distribution, but the distribution phase complicates the problem, as there are many possible ways to distribute group of data blocks among sites. In the example concerning the $P(5,2)$, 1 data block may go to $mapper1$ (in $Site1$) and 4 data blocks may go to $mapper2$ (in $Site2$), and by permutation, 4 data block may go to $mapper1$ and 1 to $mapper2$. So for the distribution of data blocks we have to call on the permutation theory.

Algorithm 1: Execution paths evaluation algorithm

```
Data: $P(n,m)$ : Set of all $m - order$ partitions for $n$ datablocks
Data: $C_m$ : Set of all combinations of possible $m$ mapper nodes
Result: $bestTree$ with the lowest execution time
begin
    $bestTree \longleftarrow null$
    for $c_i \in C_m$ do
        for $p_j \in P(n,m)$ do
            $tree \longleftarrow$ generateExecutionPathTree($c_i$, $p_j$)
            if $bestTree = null \bigcup$
            $tree.executionTime < bestTree.executionTime$ then
                $bestTree \longleftarrow tree$
            end
        end
    end
    return $bestTree$
end
```

In the listing 1 the reader may find the pseudo-code for the calculus of the best execution path; the code iterates on all generated trees and search for the one ensuring the best execution path. Listing 2 describes the steps to generate a single tree.

In the end, the calculus of the number of all the execution paths for a certain application must consider both the block data distribution configuration (eq. 4.2) and the combination of mappers (eq. 4.1).

Indeed, the number of generated execution paths depends on both the number of blocks to be processed and the complexity of the network. The growth in the number of blocks involves a significant increase in the number of partitions. Moreover the network topology affects the path generation since impacts on the number of combinations that grow as the number of available sites. For example, in the case of n=7 the number of generated paths will be around 18.000. For n=8 more than 150.000 configurations were obtained. Treating the problem of the generation of execution paths as an integer partitioning problem allowed us to apply well known algorithms working in constant amortized time that guarantee acceptable time also on off-the-shelf PCs (Zoghbi and Stojmenovic, 1994).

The described approach explores the entire space of all potential execution paths and find the best (minimum) execution time. Unfortunately the number of potential paths to visit may be very large, if we consider that many sites may be involved in the computation and that the data sets targeted by a job might be fragmented at any level of granularity. Of course, the time to seek for the best execution plan considerably increases with the number of fragments and the number of network's sites. That time may turn into an unacceptable overhead that would affect the performance of the overall job. If on the one hand such an approach guarantees for the optimal solution, on the other one it is not scalable.

In order to get over the scalability problem, we propose a new approach that searches for a good (not necessarily the best) job execution plan which still is capable of providing an acceptable execution time for the job. The new approach aims to keep the pre-processing phase as short as possible, though it may cause a time stretch during the processing phase. We will prove that, despite the time stretch of the job's execution, the overall job's makespan will benefit.

Well known and common optimization algorithms follow an approach based on a heuristic search paradigm known as the *one-point iterative search*. One point search algorithms are relatively simple in implementation, computationally inexpensive and quite effective for large scale problems. In general, a solution search starts by generating a random initial solution and exploring the nearby area. The neighboring candidate can be accepted or rejected according to a given acceptance condition, which is usually based on an evaluation of a cost functions. If it is accepted, then it serves as the current solution for the next iteration and the search ends when no further improvement is possible. Usually the search ends when no further improvement is possible. Different one-point search methods are distinguished by their acceptance condition, which is usually based on an evaluation of the cost functions of generated solutions. The simplest evaluation method accepts only candidates with the same or better cost function value than the current one. This method is regarded to be very fast, but not sufficiently powerful as it usually tends to get stuck quickly in a local optimum. This search procedure can be improved by employing an alternative acceptance condition which include candidates with lower scores of the function cost.

Several methodologies have been introduced in the literature for accepting candidates with worse cost function scores. In many one-point search algorithms, this mechanism is based on a so called cooling schedule (CS) (Hajek, 1988). The common property of these methods is that their acceptance condition is regulated by an arbitrary control parameter (such as temperature, threshold), which starting from an initial value at the beginning of the search it is varied in the course of the search. To terminate the search procedure this parameter must reach its final value where no worsening moves are accepted and the search converges. The variation of the control parameter is defined by a user and it can have a significant impact on the overall performance of the algorithm. A weak point of the cooling schedule is that its optimal form is problem-dependent. Moreover, it is difficult to find this optimal cooling schedule manually.

The job's execution path generation and evaluation, which represent our optimization problem, are strictly dependent on the physical context where the data to process are distributed. An optimization algorithm based on the cooling schedule mechanism would very likely not fit our purpose. Finding a control parameter that is good for any variation of the physical context and in any scenario is not an easy task; and if it is set up incorrectly, the optimization algorithm fail shortening the search time. As this parameter is problem dependent, its fine-tuning would always require preliminary experiments. Unfortunately, such preliminary study can lead to additional processing overhead. Based on these considerations, we have discarded optimization algorithms which envision a phase of cooling schedule.

The optimization algorithm we propose to use in order to seek for a job execution plan is the Late Acceptance Hill Climbing (LAHC) (Burke and Bykov, 2008). The LAHC is an one-point iterative search algorithm which starts from a randomly generated initial solution and, at each iteration, evaluates a new candidate solution. The LAHC maintains a fixed-length list of the previously computed values of the cost function. The candidate solution's cost is compared with the last element of the list: if it is not worse, it is accepted. After the acceptance procedure, the cost of the current solution is added on top of the list and the last element of the list is removed. This method allows some worsening moves which may prolong the search time but, at the same time, helps avoiding local minima. The LAHC approach is simple, easy to implement and yet is an effective search procedure. In fact, it doesn't employ any variant of a cooling schedule and, therefore, might have a wider applicability than cooling schedule based algorithms, it's free from setting "power affecting" parameters and then is almost not sensitive to initialization. This algorithm depends on just the input parameter *L*, representing the length of the list. It is possible to make the processing time of LAHC independent of the length of the list by eliminating the shifting of the whole list at each iteration.

The search procedure carried out by the LAHC is better detailed in reported in the Algorithm 3 listing. The LAHC algorithm first generates an initial solution which consists of a random assignment of data blocks to mappers. The resulting graph represents the execution path. The evaluated cost for this execution path is the current solution and it is added to the list. At each iteration, the algorithm evaluates a new candidate (assignment of data blocks and mappers nodes) and calculates the cost for the related execution path. The candidate cost is compared with the last element of the list and, if not worse, is accepted as the new current solution and added on top of the list. This procedure will continue until the reach of a stopping condition. The last found solution will be chosen as the execution path to enforce.

In the next chapter we compare the LAHC algorithm with the scheduler's algorithm based on a combinatorial approach. Objective of the comparison is to prove that the newly introduced algorithm scales better and is even capable of producing better performance in terms of reduced job makespan.

Algorithm 2: Execution path creation algorithm

```
Data: $Topology$ : network topology
Data: $DataHosts$ : Set of all nodes that owns data blocks
Result: $Tree$: execution path tree
Function: generateExecutionPathTree($Mappers, Partitions$)
    begin
        for $h_z \in DataHosts$ do
            $tree.root.addChild(h_z)$
            $nodepointer \longleftarrow h_z$
            for $m_i \in mappers$ do
                $pathtoMapper \longleftarrow Topology.findPath(h_i, m_i)$
                for $node \in pathtoMapper$ do
                    if $node.isLink$ then
                        $node.InputData \leftarrow nodepointer.OutputData$
                        $node.Throughput \longleftarrow$
                          $Topology.getLinkCapacity(node)$
                    else
                        if $node.isMapper$ then
                            $node.InputData \longleftarrow blockSizeValue * partition_i$
                              $node.Throughput \longleftarrow mapperThroughput$
                        end
                    end
                    $nodepointer.add(node)$
                    $nodepointer \longleftarrow node$
                end
                $pathtoGlobalReducer \longleftarrow$
                  $Topology.findPath(m_i, GlobalReducer)$
                for $node \in pathtoGlobalReducer$ do
                    if $node.isLink$ then
                        $node.InputData \leftarrow nodepointer.OutputData$
                        $node.Throughput \longleftarrow$
                          $Topology.getLinkCapacity(node)$
                    else
                        if $node.isGlobalReducer$ then
                            $node.InputData \longleftarrow$
                              $blockSizeValue * partition_i * \beta_{app}$
                              $node.setThroughput \longleftarrow$
                              $globalReducerThroughput$
                        end
                    end
                    $nodepointer.add(node)$
                    $nodepointer \longleftarrow node$
                end
            end
        end
        return $tree$
    end
```

Algorithm 3: Late Acceptance Hill Climbing algorithm applied to the problem of job execution time minimization

```
Produce random job execution path (initial solution) $s$
Calculate initial solution's cost function $C(s)$
Specify the list length $L$
begin
    for $k \in \{0..L-1\}$ do
        $C(k) \leftarrow C(s)$
    end
    Assign the initial number of iteration $I \leftarrow 0$
    repeat
        Produce a new job execution path (new candidate solution) $s*$
        Calculate its cost function $C(s*)$
        $v \leftarrow I \bmod L$
        if $C(s*) \leq C_v$ then
            accept candidate ($s \leftarrow s*$)
        end
        else
            reject candidate ($s \leftarrow s$)
        end
        Add cost value on top of the list $C_v \leftarrow C(s)$
        Increment the number of iteration $I \leftarrow I + 1$
    until *a chosen stopping condition;*
end
```

Chapter 5

Experiments And Results

In this chapter we describe several tests run to analyze the Job Scheduler performance and to estimate how well H2F performs against a plain Hadoop framework.

5.1 Scheduler accuracy

In order to test the effectiveness of the job scheduler, we carried out experiments in a multi-job environments. The reference scenario is a distributed computing environment made up of five Sites which are interconnected through geographic links as represented in Figure 4.1. Upon a job submission, the job scheduler elaborates a TJEP, according to which the target data might be conveniently shifted from Site to Site and the job is split into a number of subjobs that will eventually run on the Sites where data reside.

We reproduced the reference scenario with a small scale testbed environment. Specifically, we used a VirtualBox instance to create a virtual cluster composed of 5 Sites. The network infrastructure is managed by two virtual OpenFlow switches. All network link's capacity is set to 10 MB/s. Each Site is equipped with a single core vCPU with a computing power of 20 GFLOPS and 2GB of vRAM. As for the jobs, we selected three well known applications: the WordCount, the InvertedIndex and the ArithmeticMean. Those applications were chosen as a representative set (yet non-exhaustive) of the spectrum of applications that can be treated as MapReduce problems. In particular, among the three applications, the WordCount is the most I/O bound and is also very CPU intensive, while the ArithmeticMean is CPU intensive and has very few interactions with the I/O. The InvertedIndex is as CPU intensive as the other applications, but lies in the middle in terms of I/O boundness.

All the results we report in this section include an overhead time, which is the time needed to set up the computation. In the case of H2F, the overhead is the sum of up to three components: the time taken by the job scheduler to elaborate the execution path that is set to 10 seconds in the LAHC algorithm configuration (details in section 5.2); depending on the execution plan, the time to transfer raw data from a site to another one; the time taken by the site to load the received data into the local HDFS. In the case of the plain Hadoop framework, the overhead is just the time to load the raw data in the HDFS.

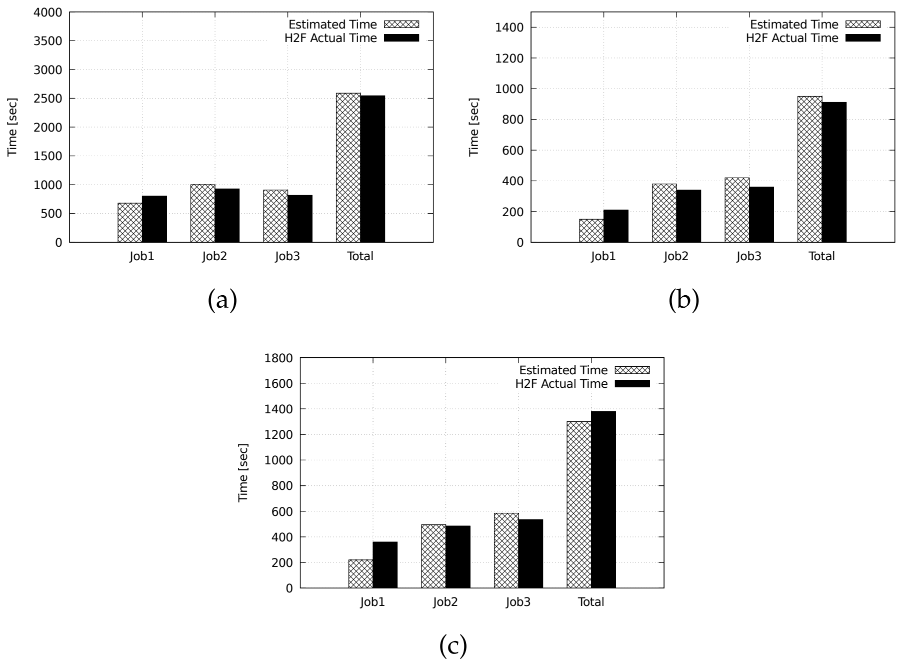


FIGURE 5.1: Estimated vs actual execution time: a)WordCount, b)InvertedIndex, c)ArithmeticMean

In this set of tests we ran fifty experiments for each application in order to prove the ability of the job scheduler to correctly estimate the execution time of the jobs. Objective of the test is to compare, for every submitted job, the execution time estimated by the job scheduler with the actual time the job takes to run and gather the final result. In each test we submitted a sequence of three jobs. Jobs' interarrival time followed a Poissonian law with a mean time of 20 seconds. Each job was instructed to process 3GB of data.

In Figure 5.1 we reported the results of three different tests where jobs were instructed to run the three applications WordCount, InvertedIndex and ArithmeticMean respectively. For all applications the error committed by the scheduler is pretty low. In fact, the gap between the actual and the expected time is around 5% on average. Also, in every test our scheduler overestimates job 1 processing time and underestimates job 2 and job 3 times. This trend can be explained by the fact that the testbed is a virtual computing environment, and of course when job 2 and job 3 are submitted there may be extra processing going on in the background (which the job scheduler is unaware of) that impacts the job performance.

5.2 Combinatorial vs LAHC

In previous section 5.1 we reported the result of tests aimed at showing the Job Scheduler's ability to estimate the job's execution time. In this section we report the result of a comparison test we ran to measure the increase of performance that the LAHC job scheduling algorithm is able to provide with respect to the combinatorial algorithm discussed in 4.2. Main objective of the test is to study the scalability of the two algorithms. To this purpose, we designed some configurations - that are meant to represent different scenarios - by tuning up the following parameters: the number of Sites populating the geographic context, the network topology interconnecting the Sites and number of data blocks distributed among the Sites. The experiments were done by simulating the execution of a job with given β_{app} and $Throughput$. Specifically, we considered a sample Job for which a β_{app} value of 0.5 was estimated. The results that we show is an average evaluated over 10 runs per configuration.

In the considered scenarios, each node is equipped with a 20 GFLOPS of computing power and each network's link has a 10 MB/s bandwidth. Further, the size of every data block is set to 500MB.

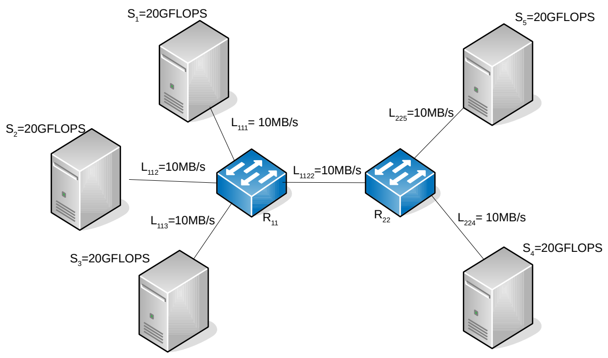


FIGURE 5.2: Network topology with 5 Sites

As for the LAHC algorithm, we also ran several preliminary tests in order to find a proper list's size value. From those tests, we observed that this parameter does not have a substantial impact on the search of the execution path by the LAHC, neither it impairs the overall LAHC performance.

For the test purpose, it was arbitrarily set it to 100. Moreover, we set the stopping condition for the LAHC to 10 seconds, meaning that the algorithm stops its search after 10 seconds of computation. This parameter's value, again, comes from preliminary tests that were run in order to figure out what an acceptable stopping condition would be. From those tests we observed that pushing that parameter to higher values did not bring any substantial benefit in the search of the job execution path. Finally, the two scheduling algorithms were both run on the same physical computing node, which is equipped with an i7 CPU architecture and a 16 GB RAM.

A first battery of tests was run on a network topology made up of five Sites interconnected to each other's in the way that is shown in Figure 5.2. From this topology, five different configurations were derived by considering the number of data blocks set to 5, 10, 20, 40 and 80 respectively. A second battery of tests was then run for another network topology that was obtained by just adding a new Site to the former topology (the new Site was attached to switch R_{22} just in between S_5 and S_4). That, of course, made things more

	Combinatorial			
# of DataBlocks	Scheduling Time [sec]	Execution Time [sec]	Makespan [sec]	Overhead [%]
5	1.49	6375	6376	0.02
10	5.63	13250	13256	0.04
20	256.02	26250	26506.02	0.96
40	3215.42	**49000**	52215.42	6.15
80	42280	**96750**	139030	30.41

TABLE 5.1: KPIs measured in the 5-node network topology with Combinatorial algorithm

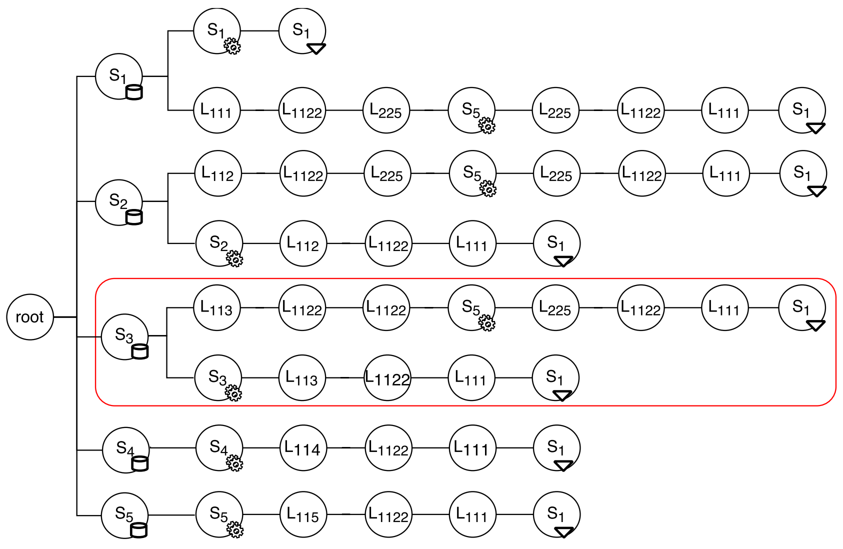
	LAHC			
# of DataBlocks	Scheduling Time [sec]	Execution Time [sec]	Makespan [sec]	Overhead [%]
5	10.32	6375	6385	0.16
10	10.01	13250	13260	0.07
20	10.01	26250	26260	0.038
40	10.01	**49750**	49760	0.020
80	10.32	**106500**	106510	0.0097

TABLE 5.2: KPIs measured in the 5-node network topology with LAHC algorithm

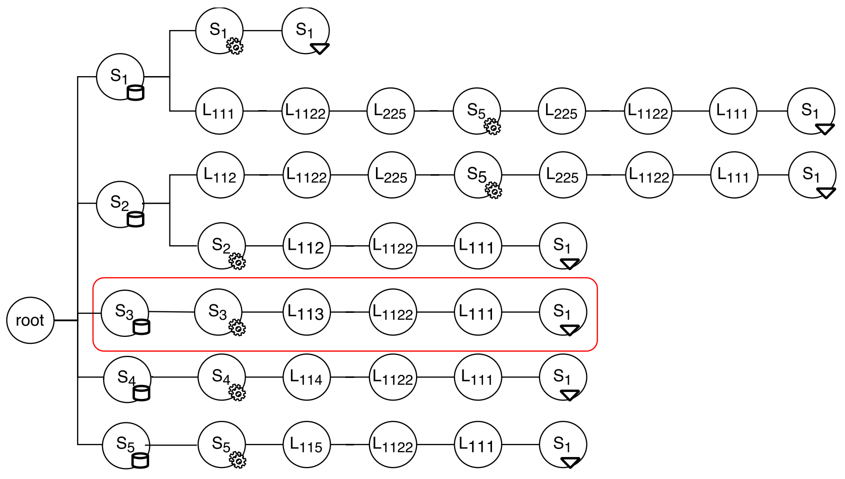
complicated since the number of combinations of all possible job execution path increases. As for this network topology, the same data block configurations were considered. Unfortunately, we had to stop the 80 data-block test as it took more than two days for the combinatorial algorithm to run without even finishing.

The KPI that is put under observation is the *job makespan*. That KPI is further decomposed in two sub-KPIs: the *job scheduling time* and the *job execution time*. Throughout the tests, the two sub-KPIs were measured. Results of the first battery of tests are shown in Table 5.1 and in Table 5.2. In the tables, for each algorithm, we report the following measures: scheduling time, execution time, makespan (which is the sum of the previous two measures) and the overhead (the percentage of the scheduling time over the makespan).

The reader may notice that in the cases of 5 and 10 data blocks respectively the combinatorial algorithm outperforms the LAHC in terms of makespan.



(a)



(b)

FIGURE 5.3: Execution paths in the 5-node network topology and the 40 data-blocks configuration: a) combinatorial; b) LAHC

In those cases, the LAHC managed to find the global optimum (we remind that the combinatorial always finds the optimum) but the LAHC overhead is higher than that of the combinatorial (which is capable of finding the solution in much less than 10 seconds). In the case of 20 data blocks, the LAHC is still able to find a global optimum, so the performance of the two algorithm terms of execution time are equal. But in this case the combinatorial took more than 200 seconds to find the solution, while the scheduling time for the LAHC is always ten seconds long. So the LAHC slightly outperforms the combinatorial in the makespan. Finally, in the cases of 40 and 80 data blocks the LAHC finds a just local optimum (the LAHC's execution time is lower than the combinatorial's). In spite of that, being the scheduling time of the combinatorial extremely long, in the makespan comparison the LAHC once more outperforms the combinatorial.

In Figure 5.3 we have reported the two execution paths found by the two algorithms in the case of 40 data blocks. While the combinatorial's is the best possible path and the LAHC's is only a local optimum, the two paths look very much alike. The only difference, which is highlighted in the red boxes, concerns the computation of the data residing in Site S_3. While the LAHC schedules the computation of the data on the Site S_3 itself, the combinatorial manages to balance to computation between Site S_3 and Site S_5, thus speeding up the overall execution time.

	Combinatorial			
# of DataBlocks	Scheduling Time [sec]	Execution Time [sec]	Makespan [sec]	Overhead [%]
5	1.87	7125	7126	0.03
10	594	12250	12844	4.62
20	10795.30	23500	34295.3	31.48
40	404980.31	**44350**	449330.31	90.13

TABLE 5.3: KPIs measured in the 6-node network topology with Combinatorial algorithm

Results of the second battery of tests are shown in Table 5.3 and in Table in Table 5.4. The reader will notice that in the new network topology, which is a little more complex than the previous one, the combinatorials has some

	LAHC			
# of DataBlocks	Scheduling Time [sec]	Execution Time [sec]	Makespan [sec]	Overhead [%]
5	10.36	7125	7135	0.14
10	10.13	12250	12260	0.083
20	10.02	23500	23510	0.043
40	10.10	**47000**	47010	0.021

TABLE 5.4: KPIs measured in the 6-node network topology with LAHC algorithm

scalability issues even with a relatively simplex data configuration (for 10 data blocks, the scheduling time takes more than 500 seconds. In the cases of 20 and 40 blocks respectively, the LAHC confirms to be the best as it is capable of finding very good execution paths (though not the best) in a very short scheduling time.

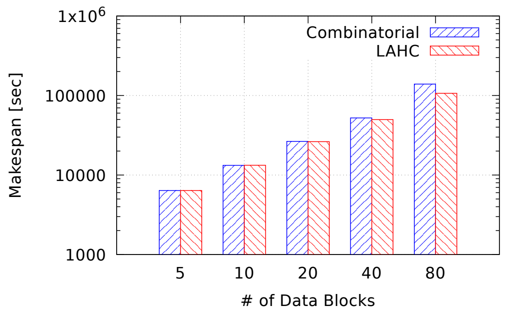


FIGURE 5.4: Makespan in the 5-node network topology

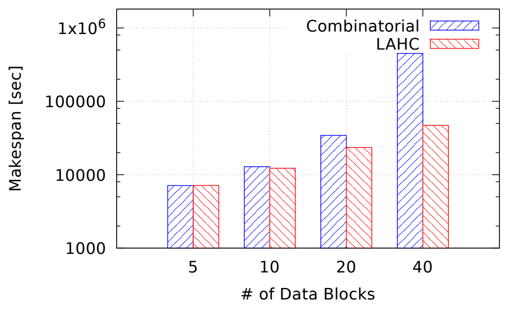


FIGURE 5.5: Makespan in the 6-node network topology

In Figures 5.4 and 5.5 we reported a graphical representation of the makespan performance of the two algorithms. In the graph, for the ease of representation, the values in y-axis are reported in a logarithmic scale. The final consideration that we draw is that the combinatorial algorithm is extremely poor in terms of scalability. In fact, the performance will degrade significantly as the number of data blocks grows. The LAHC solution was proved to scale very well. Despite the solutions it finds are local optima, even for very complex scenarios they are very close to the global optima found by the combinatorial.

5.3 Comparison standard hadoop vs H2F

In this Section we describe several tests run to estimate how well H2F performs against a plain Hadoop framework. The performance indicator upon which the comparison is made is the Job's completion time (makespan). In order to reproduce an imbalanced distributed computing context, we set up a testbed environment consisting of four Sites. Each Site represents a computing facility, may hold raw data and runs an instance of the H2F framework.

Sites are interconnected to each others through network links. The network topology of the testbed environment is depicted in Figure 5.6. For the test purpose we employed: two sites (S_3, S_4) equipped with an i7 CPU architecture and 8GB RAM, which guarantees for an estimated computing node's capacity of 150 GFLOPS [1]; two Sites (S_1, S_2) having a Core 2 Duo CPU and 4GB RAM, for an estimated computing power of 15 GFLOPS each. Therefore, speaking of computing power the testbed is natively imbalanced. Finally, each network link's nominal bandwidth is 40MB/sec.

As for the Jobs to test, we selected three well known applications: the *WordCount*, the *InvertedIndex* and the *ArithmeticMean*.

Since the H2F was designed to boost the performance of the MapReduce paradigm in distributed computing contexts where imbalance exists in terms

[1]The computing capacity was assessed with the Linpack benchmark (http://www.netlib.org/benchmark/hpl/)

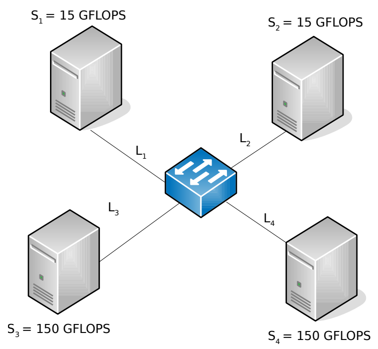


FIGURE 5.6: Network topology of testbed environment

of Sites' computing capacity, network links' bandwidth and raw data distribution respectively, we designed some context configurations (scenarios) reproducing different situations of imbalance. The scenarios were generated by tuning up the initial distribution of the raw data among the sites and the capacity of the network links interconnecting the sites.

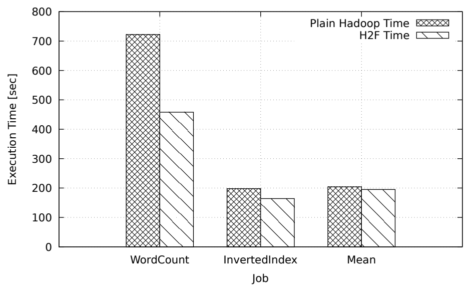
The first scenario focuses on the data distribution. The configurations we used for this test are listed in Table 5.5. Each table row represents a configuration where the displayed numbers are the the percentage of the overall input data residing in each site. So, for instance, in the $Config_1$, the 40% of data resides in S_1, the 30% in S_2, the 20% in S_3 and the 10% in S_4.

	S_1	S_2	S_3	S_4
$Config_1$	40%	30%	20%	10%
$Config_2$	40%	40%	20%	0%
$Config_3$	30%	0%	70%	0%

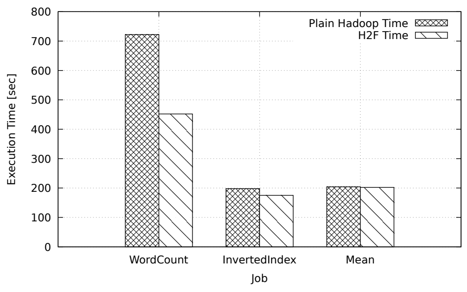
TABLE 5.5: Configurations of the data blocks' distribution

Results obtained from running the three applications on both the H2F and the Hadoop framework are reported in Figure 5.7.

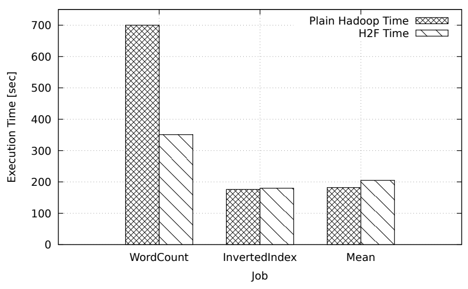
The result that we show is an average evaluated over 10 runs per configuration. A point we would like to stress is that all the depicted results include the *overhead* time, which is the time needed for preparing the data to the computation. In the case of the H2F, the overhead is the sum of four components: the time taken by the Job scheduler to elaborate the optimum execution path



(a)



(b)



(c)

FIGURE 5.7: H2F vs Hadoop: comparison of job's execution time for a) $Config_1$, b) $Config_2$ and c) $Config_3$ in the 5GB raw data scenario

and the optimum data block size; depending on the execution plan, the time to transfer raw data from site to site; the time taken by the site to load the received data onto the local HDFS; the time the Global reducer has to wait in order for the last block of map-reduced data to arrive. In the case of the plain Hadoop framework the overhead is just the time to load the raw data on the HDFS. In regard to the H2F, the time take by the Job Scheduler for the specific testbed and the considered scenarios was never over a hundred seconds. Of course, the job scheduling time may increase in the case of more complex network topologies and bigger amounts of raw data.

Taking a closer look at Figure 5.7, regardless the raw data's initial distribution, we notice that the H2F framework is able to execute the WordCount application faster than how the plain Hadoop implementation does. The execution performance of the other two applications are almost similar but H2F slightly prevails. Some considerations need to be made in respect to this result. First, the use of the H2F for this specific scenario is discouraged. But the bad performance of the H2F is due to the fact that, for the relatively small amount of data being considered (5GB), the *overhead* time has a stronger impact on the H2F than on the plain Hadoop framework. Second, the better performance of the WordCount observed in the H2F is not surprising if we consider that the application is very much CPU intensive, and the H2F is capable of outperforming the plain Hadoop despite it suffers from a longer overhead time.

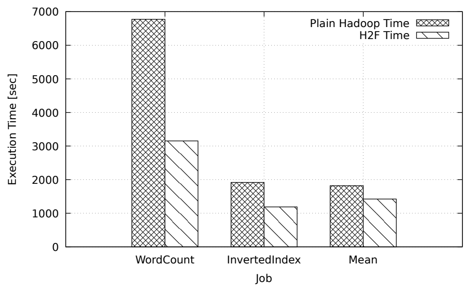
In the second scenario, the overall raw data size was set to 50 GB and while the rest of configurations remained unchanged. Collected results are shown in Figure 5.8. What we observe here is that with the increase in the raw data size, the difference of performance between the H2F and the plain Hadoop gets even more marked (15-20 % on average). In fact, with respect to the agnostic plain Hadoop, the H2F is able to identify the source of computation that are capable of boosting the computing process (because of their greater computing capacity). This time, being the computation time longer than the 5 GB case, the impact of the overhead time is very low.

Finally, we designed one last scenario where the network links' capacity is intentionally imbalanced with the purpose to show that the H2F is capable to adapt to such an imbalance and yet schedule an optimum execution path for the considered job. In Table 5.6 we report the values of the links' bandwidth for the two configurations that we designed. In the configurations, the remaining parameters are left unchanged with respect to the previous scenario (50 GB data, input data distributed among Sites as defined in $config1$).

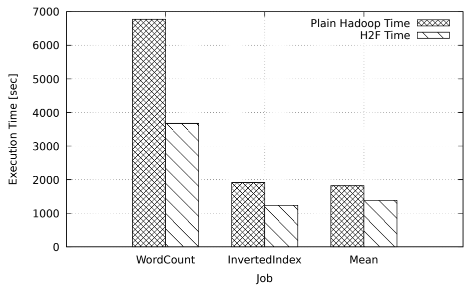
TABLE 5.6: Links' Configurations

	L_1[MB/s]	L_2[MB/s]	L_3[MB/s]	L_4[MB/s]
$Config_4$	20	20	40	40
$Config_5$	20	40	20	40

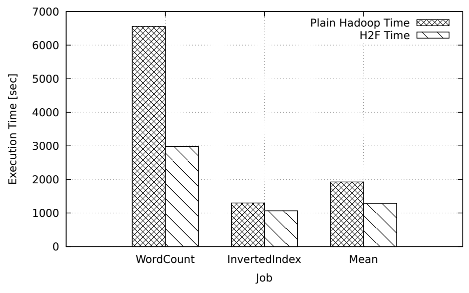
In Figure 5.9 the results of the tests run on these configurations are shown. Again, the execution times observed for the H2F are shorter than Hadoop's.



(a)

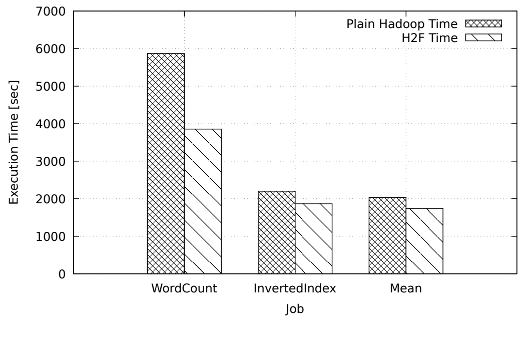


(b)



(c)

FIGURE 5.8: H2F vs Hadoop: comparison of job's execution time for a) $Config_1$, b) $Config_2$ and c) $Config_3$ in the 50GB raw data scenario



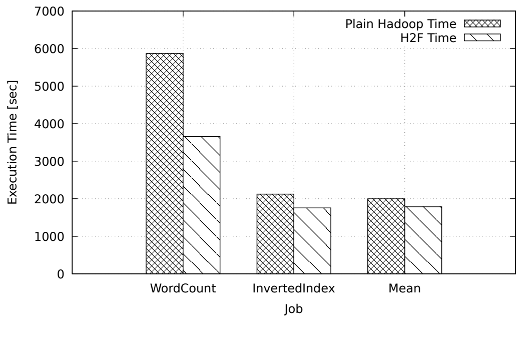


FIGURE 5.9: H2F vs Hadoop: comparison of job's execution time for a) $Config_4$ and c) $Config_5$

The aim of all these tests was to challenge the capability of the H2F to adapt to different distributed computing environments, and to also prove that it may provide better performance than the plain Hadoop framework. Despite the H2F suffers from longer *overhead* times, tests have showed that the time for the H2F to complete a job (no matter the application type) is lower that the time observed in the Hadoop case. This holds true for any of the designed scenarios.

Bibliography

Andrews, George E. (1976). *The Theory of Partitions*. Vol. 2. Encyclopedia of Mathematics and its Applications.

Burke, Edmund K. and Yuri Bykov (2008). "A late acceptance strategy in hill-climbing for examination timetabling problems". In: *Proceedings of the conference on the Practice and Theory of Automated Timetabling(PATAT)*. Montreal, Canada.

Cheng, Dazhao et al. (2017). "Improving Performance of Heterogeneous MapReduce Clusters with Adaptive Task Tuning". In: *IEEE Transactions on Parallel and Distributed Systems* 28.3, pp. 774–786. ISSN: 1045-9219. DOI: 10.1109/TPDS.2016.2594765.

Cisco (2017). *Cisco Visual Networking Index: Forecast and Methodology, 2016–2021*. https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.html.

Convolbo, M. W. et al. (2016). "DRASH: A Data Replication-Aware Scheduler in Geo-Distributed Data Centers". In: *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 302–309. DOI: 10.1109/CloudCom.2016.0056.

Dean, Jeffrey and Sanjay Ghemawat (2004). "MapReduce: simplified data processing on large clusters". In: *OSDI '04: Proceeding of the 6th Conference on Symposium on operating systems design and implementation*. USENIX Association.

Facebook (2012). *Project PRISM*. www.wired.com/2012/08/facebook-prism.

Fahmy, Mariam Malak, Iman Elghandour, and Magdy Nagi (2016). "CoS-HDFS: Co-locating Geo-distributed Spatial Data in Hadoop Distributed File System". In: *Proceedings of the 3rd IEEE/ACM International Conference* *on Big Data Computing, Applications and Technologies (BDCAT 2016)*. Shanghai, China, pp. 123–132. ISBN: 978-1-4503-4617-7. DOI: 10.1145/3006299.3006314.

Forbes (2016). *With Internet Of Things And Big Data, 92Will Be In The Cloud*. https://www.forbes.com/sites/joemckendrick/2016/11/13/with-internet-of-things-and-big-data-92-of-everything-we-do-will-be-in-the-cloud.

Hajek, Bruce (1988). "Cooling Schedules for Optimal Annealing". In: *Mathematics of Operations Research* 13.2, pp. 311–329. ISSN: 0364765X, 15265471.

Heintz, B. et al. (2014). "End-to-end Optimization for Geo-Distributed MapReduce". In: *IEEE Transactions on Cloud Computing* 4.3, pp. 293–306.

Jayalath, C., J. Stephen, and P. Eugster (2014). "From the Cloud to the Atmosphere: Running MapReduce across Data Centers". In: *IEEE Transactions on Computers* 63.1, pp. 74–87.

Kim, Shingyu et al. (2011). "Improving Hadoop Performance in Intercloud Environments". In: *SIGMETRICS Perform. Eval. Rev.* 39.3, pp. 107–109. ISSN: 0163-5999. DOI: 10.1145/2160803.2160873.

Li, P. et al. (2017). "Traffic-Aware Geo-Distributed Big Data Analytics with Predictable Job Completion Time". In: *IEEE Transactions on Parallel and Distributed Systems* 28.6, pp. 1785–1796. ISSN: 1045-9219. DOI: 10.1109/TPDS.2016.2626285.

Luo, Yuan et al. (2011). "A Hierarchical Framework for Cross-domain MapReduce Execution". In: *Proceedings of the Second International Workshop on Emerging Computational Methods for the Life Sciences*. ECMLS '11. San Jose, California, USA, pp. 15–22. ISBN: 978-1-4503-0702-4. DOI: 10.1145/1996023.1996026.

Mattess, Michael, Rodrigo N. Calheiros, and Rajkumar Buyya (2013). "Scaling MapReduce Applications Across Hybrid Clouds to Meet Soft Deadlines". In: *Proceedings of the 2013 IEEE 27th International Conference on Advanced Information Networking and Applications*. AINA '13, pp. 629–636. ISBN: 978-0-7695-4953-8. DOI: 10.1109/AINA.2013.51.

The Apache Software Foundation (2011). *The Apache Hadoop project*. http://hadoop.apache.or

Walmart (2015). *Walmart's big data*. http://www.walmartlabs.com/category/bigdata/.

Yang, Hung-chih et al. (2007). "Map-reduce-merge: Simplified Relational Data Processing on Large Clusters". In: *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*. SIGMOD '07. Beijing, China, pp. 1029–1040. ISBN: 978-1-59593-686-8.

You, Hsin-Han, Chun-Chung Yang, and Jiun-Long Huang (2011). "A Load-aware Scheduler for MapReduce Framework in Heterogeneous Cloud Environments". In: *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC 2011)*. TaiChung, Taiwan, pp. 127–132. ISBN: 978-1-4503-0113-8. DOI: 10.1145/1982185.1982218.

Yu, Boyang and Jianping Pan (27). "Location-aware associated data placement for geo-distributed data-intensive applications". In: *2015 IEEE Conference on Computer Communications, (INFOCOM 2015)*. Kowloon, Hong Kong, pp. 603–611. DOI: 10.1109/INFOCOM.2015.7218428.

Zhang, Qi et al. (2014). "Improving Hadoop Service Provisioning in a Geographically Distributed Cloud". In: *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on*, pp. 432–439. DOI: 10.1109/CLOUD.2014.65.

Zoghbi, A. and I Stojmenovic (1994). "Fast Algorithms for Generating Integer Partitions". In: *International Journal of Computer Mathematics* 80, pp. 319–332.