Modeling and Verification of Big Data Computation

Claudio Mandrioli
Matricola 849973

Relatore: Alberto Leva
Co-Relatore: Martina Maggio

POLITECNICO
MILANO 1863

Dipartimento di Elettronica, Informazione e Bioingegneria

MSc Thesis

Dipartimento di Elettronica, Informazione e Bioingegneria
Via Ponzio 34/5
20100 Milano
Italy

Contents

1

Introduction: Definitions and Problem Statement

During the 1990s and the early 2000s, the volume of data available for processing in computing systems and networks, has increased dramatically. For example, on Facebook 300 million photos are uploaded every day and 293 thousand statuses are updated every minute, on Youtube 300 hours of videos are put online every minute, modern cars are highly instrumented and produce location data and data analytics every time they are used, and many other examples could be listed.

Of course, applications involving a large amount of data are not a novelty *per se*. For example, huge sets of data come into play when solving engineering problems involving large-scale finite-element models, or computational fluid dynamics ones. The novelty in the situations mentioned above is that the data:

- Is composed of *many simple unities* unrelated to one another (think e.g. of the tweets in one day),
- Is often natively *distributed* starting from the origin (which is self-evident for example in social networks),
- Is typically *unstructured*, i.e., heterogeneous like a photo and a biographic sketch that can be uploaded by a person as a description of him/herself.

To indicate applications with the characteristics just listed, the term "Big Data" was created, and is nowadays widely adopted. Correspondingly, big data *frameworks* started emerging as the technological means to realize big data applications.

In fact, in the literature many different definitions of "big data" were given, but it is not the goal of this chapter – nor of the thesis as a whole – to enter this discussion. Therefore, for our purpose, we just say that a big data framework is a software tool that allows the user to write a big data application, exploiting modern computing architectures like those made available by the cloud. As a result, and once again at a level of detail compatible with this work, the major capability of a big data framework can be summarized as follows.

- It must be capable of manipulating data not enjoying *homogeneity* and *uniformity* properties, which is what in this context one means for "unstructured". Not being homogeneous means that data playing the same role in the application (e.g., describing a person) can have various natures, like the photo and the bio mentioned above. Not being uniform means that data can be stored in different manners, on different media, and in any case not with the organization that is typical of traditional databases.

- It must provide the user with the possibility not only of writing the application, but also of specifying the resources that will be allotted (typically, by taking them from the cloud) for its execution; in this context, it must be capable of managing different elaboration and storage units, making the application execution distributed, and to manage parallelism, typically dividing a large set of data among elaboration units.

Examples of big data frameworks are MapReduce, Apache Hadoop, Apache Spark, Apache Tez, Apache Pig, Apache Hive, Disco, Microsoft Dryad, M3R, ... A historical perspective on the mentioned big data frameworks is presented in Chapter 2.

Abstracting from their implementation intricacies, all these frameworks are very similar to one another. They focus on allowing applications to query and manipulate data that do not fit into the memory of a single computing unit, and are therefore saved in a distributed storage infrastructure. The similarity comes from sharing of the abstractions of the manipulations: hence we can investigate the idea that there is a simple model that can represent this common feature transversally to the different frameworks.

Starting from this idea, the aim of this thesis is twofold. The first objective is to capture the main characteristics of a generic big data framework – *i.e.*, the main physical phenomena that govern the processing of large datasets, where on the meaning of "physical" some words are spent later on in due course – so as to obtain a general dynamic model of the framework's operation. On the same front, the thesis proposes specializations of this general model that capture the behavior of *a specific* framework (to this end, reference is made here to Apache Spark).

The second aim of the thesis is to use the obtained model to verify properties of the execution of an application on a specific big data framework. The main idea behind this second purpose is that if the model reliably computes and exposes the quantities that are needed for the intended evaluation (e.g., completion or response times, resource allocations, and so forth) without of course actually executing the application, and therefore being immensely lighter from the computational standpoint, then subjecting the "simulated application" to model checking allows to carry out the verifications (both deterministically and stochastically) in a correspondingly reliable manner, with very significant time and effort savings. To this end, in the thesis reference is made to the UPPAAL probabilistic model checker.

1.1 Present and Expected Impact of Big Data

The kind of analysis sketched out at the beginning of this introductory chapter, were simply not possible before information technologies became as pervasive in every aspect of human life, as they are nowadays. Still, however, there are major obstacles to exploiting the so opened possibilities: for example, according to [37], we reached the point that the percentage of data a company can process with respect the data they collect is significantly decreasing. More in general, it is common perception that data are being generated at a faster rate than the machines power growth, and for this reason we cannot expect this problem to be solved by future improvements of current technologies (like larger memories and faster computation).

In detail, the obstacles just mentioned are mainly two. First, the data we are considering is intrinsically distributed as for their generation, and in any case so large that even if they were generated at a single location, collecting them in a single storing device it would not be possible. For instance the social media profiles of Facebook subscribers are stored in databases (in a number of the order of dozens) constituted each by thousands of machines, so even if the physical locality of the storing machines is the same the number of involved storing devices is impressive. Secondly, data is not produced and stored in such a way to be suitable for the required analysis; sticking to the case of a social media, if for example we want to find out which football team Facebook users support, we will have to analyze each profile, studying data of different nature at the same time: videos, texts, pictures, audio, and so on.

Obviously, to make analysis useful, a time constraint must be introduced, that is, the processing of data must be performed in a reasonable amount of time, and possibly within a predictable time window. In fact, a relevant part of data can be are volatile, this can be used for retrieving useful information only if processed in a constrained time horizon: for instance, a stock market analysis might be constrained by the date of an auction closure.

Traditional tools for data analysis are based on *homogeneity* and *structuredness* of the information, and on a concentrated environment: all the data and the computational resources are located in the same place (and therefore constraining the maximum volume of data that can be considered) and are well structured (they satisfy homogeneity and uniformity requirements, e.g. a relational database). Apparently, removing those hypothesis requires the introduction of completely new technologies for data analysis.

Summarizing, the impact of big data – both present and future – is already evident considering the kind of analysis they make possible, and correspondingly, on the progress they are (and will be) inducing both in technologies and in application/framework development methodologies.

1.2 Defining Big Data

Due to the variety of different possible applications of the big data technology, and to its potentialities, many people are interested in studying it. As a result, the term "big data" has been used in plenty of different fields simultaneously, before a rigorous definition was given. This has led to many different interpretations of its meaning, a very few of which are listed below.

- According to IBM[1], "Big data is a term applied to data sets whose size or type is beyond the ability of traditional relational databases to capture, manage, and process the data with low-latency. And it has one or more of the following characteristics – high volume, high velocity, or high variety".
- Researchers in Rome wrote that: "Big Data is the Information asset characterized by such a High Volume, Velocity and Variety to require specific Technology and Analytical Methods for its transformation into Value" [24].
- Microsoft Research[2] defined big data as "The process of applying serious computing power, the latest in machine learning and artificial intelligence, to seriously massive and often highly complex sets of information".

Although it is not the purpose of this thesis to take part into this discussion, it is worth noticing that sometimes the term often is not used referring to a technology, but rather to a generic source of information provided that this information is "big", i.e., without taking into account its nature, and the novelties that need introducing to process it fruitfully. The result is that frequently "big data" is referred to what it is used for, instead of what it actually is, and analogously, the abstract properties of their processing confused with those of the way one would like the said processing to happen.

So, we must first of all make clear that we are describing a new *technology*, that should allow us to process datasets with properties not considered before – specifically, related to the dimensions and lack of structure – and subject to constraints on the execution time. Then, given this and in the light of the works cited above, we can try to synthesize our specific definition.

It is clear that the key point of big data is that there must be something different from the previous technologies. The distinction arises in the volume and the nature of the data. Hence we say that

> Big data are datasets with volumes that do not allow to store them in a single place,
> that are constituted by elements that are small with respect the total volume of the dataset, and that might be unstructured.

[1] https://www.ibm.com/analytics/us/en/technology/hadoop/big-data-analytics/

[2] https://news.microsoft.com/2013/02/11/the-big-bang-how-the-big-data-explosion-is-changing-the-world/

The combination of these properties must be such that a specific technology for collecting and processing them is required. In this sense, big data become a technological challenge and remain independent from the possible application field allowing us to involve all of them at the same time. According to the given definition, a few examples of big data are given below.

- Profiles of the people registered to a certain social media (needless to say that the volume of information is massive and specifically it is strongly unstructured as some profiles might be characterized mainly by videos while others by pictures and so on),
- The tweets of Twitter subscribers from USA during the election day (this dataset can be considered structured as all tweets have the same structure but still the volume is such that we can consider it big data),
- The purchases made on Amazon by people in Milan in a certain period of time,
- All the records of the airplane tickets bought by people flying from Malpensa in 2016.

Instead, although "big", we do not consider the following cases to fall into the "big data" category.

- An enormous matrix (elaborating a matrix generally requires considering at the same time if not all most of its elements, hence we cannot talk about small elements that constitute the object big data),
- Data produced by a Finite Element Method simulation (same considerations made for the matrix),
- The database of a bank (such dataset might be massive but is strongly structured hence not requiring specific technologies).

The definition is based on the fact that traditional solutions do not accomplish the task of processing datasets with the properties listed before, hence new technologies must be presented from both the hardware and the software point of view. For the former the solution is offered by the so called "cloud", that can offer the computational power of theoretically all the computers around the world[3]. This has the drawback of introducing all the complexities related to managing and coordinating those resources, and this must be tackled using the software.

For the sake of completeness, we end this section by reporting and briefly commenting another common definition used for big data, i.e., five adjectives known

[3] Resorting to the cloud is a natural choice as the data are already stored in the same distributed environment.

as "the five V": Volume, Variety, Velocity, Veracity and Value. The first two terms surely point out the main dimensions that characterize big data, velocity instead to the author looks like a requirement, not a property: every datum intrinsically has a finite life after which it loses its value. Veracity and value definitely belong to another level of analysis not related to the technology. This is proven by the fact that it is not possible to give a non ambiguous and rigorous definition of them and is a clear example of how this term is often overlapped with what it is used for as we said before. We opted instead for a technology-based definition, as it best fits the scope of this work.

1.3 Big Data Frameworks and Applications

To process those high volume, distributed and non-structured datasets, it is necessary to define programs that are able to manipulate those objects. We will call those programs from now on *big data applications* (BDAs). A BDA defines the operations that the user (the programmer) wants to execute on the dataset.

In order to manipulate big data, programs will use some instruments or tools provided by a framework (that analogously we will call *big data framework*, BDF). Those instruments consist in functions that manipulate the big data objects, each time the programmer uses them the framework autonomously allocates the resources available in the cloud in order to execute the program. The framework must therefore provide the programmer the functionalities to manipulate the big data object and coordinate the computational resources and the data movements.

Although not strictly necessary from the conceptual point of view, for apparent reasons of opportunity and practical viability, BDAs take the resource they require from the cloud. Exploiting the cloud also means exploiting parallelism: multiple computational units allow for faster execution of a program under the bounds of the degree of parallelism of the program itself. Big data applications are characterized by the execution of relatively few and simple operations on a massive amount of relatively small inputs. By "relatively few and simple operations" we refer to the fact that the processing of the single datum is small both in execution time and required computational resources with respect the overall execution of the application. This naturally allows for an high degree of parallelism, but we still need to have a rigorous definition of it in order to properly exploit it. In general defining the degree of parallelism of a computer program is not trivial and often close to impossible due to complexity and variety of operating systems (and how operations are actually executed). However in the specific case of big data applications, parallelism appears in two specific forms.

Two Forms of Parallelism

A BDA is defined by a set of operations to be performed on a dataset with the properties listed above. Given this we can find parallelism in it in two ways: (i) it

can be intrinsically part of the definition of the program, (ii) a required operation might be intrinsically parallel with respect the input data. In the first case, it is the programmer who defines two or more operations that do not strictly require to be sequential, this may happen in the following cases:

- To perform different operations on the same dataset, in this case data can be read twice and processed independently[4],
- To perform different operations on different data, this case is (conceptually) the most parallel possible where the required operations don't even share the input set.

Differently the second form of parallelism is a property of a single operation instead of the computational flow of the program. The programmer may require an operation on a given dataset that has some degree of parallelism, i.e. the operation can be separately performed on the different input elements. This happens in relation to how much the processing of a given datum is dependent from the processing of other data.

This form of parallelism may appear in different ways and levels. Think of an operation to be performed in different steps with decreasing level of parallelism like computing the maximum of a vector of numbers: it can be done separately on the elements of a partition of the whole vector and then iterated on the maxima of each sub-vector. Different thing is if the programmer wants to increase by 1 each of the elements of the vector: the increment is a single operation that involves each of the input elements individually. It is important that for each possible operation the degree of parallelism is well defined in order to be able to properly exploit it.

Apparently the higher the degree of parallelism of the overall BDA is, the more we will be able to exploit the cloud resources. Both the kinds of parallelism can contribute to the parallelism of the overall computation but it is clear that the two must be managed in very different way. Moreover to the programmer this distinction shouldn't appear and must be completely managed by the framework. This is not trivial and further increases the computational overhead required to exploit the cloud.

1.4 Scenario

Now that we have stated what is big data, what we want to do with it and which tools we expect to use, we can well define the *scenario* that we are considering in the context of the big data technology. The technological challenge is to use distributed computational resources to process a high number of small (relatively to the whole dataset) unstructured data stored in a distributed environment. The environments

[4] Usually called Multiple Instructions, Single Data.

that store and process the data might be the same or not or may partially overlap, apparently the first case is the most favorable one and a good framework is supposed to exploit this opportunity. In this problem statement there are different sources of complexity:

- Distributed computational resources,
- High volume and consequent distributed storage,
- Data unstructureness.

In the list above we can make an important distinction. The high volume and consequent distributed environment that has to store and process the data are known (within some bounds) at the beginning of the execution of the BDA. Instead the data nature and therefore the computational weight of each datum is unknown until they are actually processed. So if we want to perform the processing in a controlled time and not just deploy the program and hope for the best, dynamic control[5] of resource allocation is required. This necessity is further strengthen if we consider that the cluster (the cloud) on which the program is run has its own dynamics and if we admit that its structure may vary during the execution. This naturally calls for a control-theoretical approach. We underline once more that the resource management must be completely and autonomously managed by the by the framework and hidden to the programmer so that he can focus on defining the analysis he wants to perform.

1.5 Purpose of This Thesis

In this work we propose a systemic approach to the problem of big data. The abstraction is that we want to process high volume datasets constituted by relatively small unstructured data in a cluster constituted by an arbitrary number of computational units and we want to do it in a controlled manner. This is a purely technological objective, by itself not related to the specific application. As stated in the previous sections, however, there are several elements of complexity, what we want to do is to point out the intrinsic complexity and dynamics of the problem, irrespectively to the specific implementation of the solution. This is usually called a "first-principle" approach where we look for the physics of the abstract problem that is common to every specific realization of it. This approach brings a strong advantage: we can arbitrarily choose which parts and at which level we want to describe of the reality and therefore arbitrarily choose the complexity of our description (model).

Modeling reality allows us to verify properties of it up to the point that the description is consistent with reality. Specifically our problem is characterized by both discrete and continuous components that respectively model the algorithmic

[5] People with a computer science background are likely to call it adaptation.

(the program) and dynamic (the processing) aspects of a big data application. Automated tools for querying those models about properties expressed in temporal logic already exist and they are called model checkers. The methodology of using those instruments is called *model checking* [23, 9]. When also statistical modeling is introduced we talk instead about *statistical model checking* [17, 18, 28].

In this work, we will use this description of big data applications exactly to do that. This should allow us to provide a theoretical upper bound for what the performances of a BDA can be. We will provide a high level description of those programs that allows without actually processing the data to have an estimate of the performance[6] of the framework. Model checking the high level description naturally requires much less time and computational resources with respect actual execution of the application: as we are talking about applications that may run for hours, days or even weeks this can be considered a useful tool.

Finally, there is another advantage that can be taken from this approach. BDF up to now have been created without actually being aware of the "inner physics" of the problem and this is natural as the problem itself was not well defined. In the writer opinion nowadays technology is mature enough to allow a complete comprehension of the problem's physics and therefore a more efficient re-build of the frameworks.

Structure of the thesis

The thesis is structured in six chapters. In the first chapter (this one) we gave a definition of big data, stated our problem in the so identified *scenario*, and outlined the objectives of this work. In the second chapter we critically go through the "short" history of big data: this will provide us the concrete concepts that characterize big data frameworks (in contrast with the purely abstract ones presented in this chapter). In the third chapter, a first model for a BDA is presented, and successively – in the fourth one – its implementation is described and validated. The fifth chapter presents another model that extends the one already presented with some analysis of its meaning and power[7]. In the last chapter some final considerations are made about the work done, and guidelines for future work are provided.

[6] Performance of a BDF regards the required computational time with respect the allocated resources.

[7] This is a clear example of what stated a couple of paragraphs before about the possibility to arbitrarily choose the level of abstraction of our representation of reality.

2

Historical Perspective on Big Data

This chapter provides an overview of the big data frameworks' landscape, in an attempt to better understand the concept behind *big data* and the characteristics that frameworks are required to have for proper functioning.

The term *big data* was first used in 1944, to refer to the imminent explosion of stored information, and then remained unused until the nineties. In the beginning of the third millenium, Google publies a paper describing the Google File System [13]. While the paper does not explicitly mention the term big data, the content of the work precisely fits the paradigm of modern big data frameworks. The Google File System introduces a new technology to handle the complexity of processing amount of data that previous technologies could not process. In the third millenium, this technology has gained traction and has been far more important, as the amount of collected data in many different domains steadily increases.

Apparent economical interests in the big data scenario are accompanied by a general lack of information about big data frameworks. To date, companies like Amazon[1] or SAS[2] do not release any information about the frameworks and infrastructures that they use to process data. The exceptions in this sense seem to be Google, which published some of the details behind their data processing framework [33], and Microsoft Research, which presented details of the Dryad framework [15]. Our analysis is necessarily focused on open-source frameworks. We review and analyze the technological solutions used in modern big data frameworks. Our analysis concentrates on what we have described as *the inner physics of the problem*, or the necessary characteristics that any big data framework must possess. This approach allows us to exploit the same modeling framework to capture the characteristics of different frameworks, and to reason about their similarities and differences using a unified approach.

[1] https://aws.amazon.com/

[2] https://www.sas.com/it_it/home.html

The surveyed technologies have reached a level of maturity and complexity that allows us to draw general conclusions that are valid irrespectively of the specific framework characteristics. Also, we believe that this investigation is crucial towards building a new framework from scratch, taking advantage of the knowledge and the experience gathered with the use of successful frameworks. We believe that structuring clearly the current technological knowledge is indeed the first step towards formalizing properly the needs and desired characteristics of a stable, usable, and well-performing big data framework.

The second part of the chapter presents the state of the art of big data technologies, focusing on two main aspects: (1) the programming model, and (2) the representation of the applications in the different frameworks. We describe these two aspects and relate them to the two different levels of parallelism discussed in Section 1.2. We are then ready to establish what are the *first principles* behind a generic software application that operates on big data.

2.1 Historical Perspective

In this section we survey relevant artifact that are linked to milestones in the big data technology. Notice that providing a complete history of big data frameworks and of their characteristics is out of the scope of this thesis. However, we will use the surveyed knowledge to understand and describe the first principles behind big data processing. The mentioned milestones are the following:

- Google File System [13] (2003): In this work large datasets are taken into account. The amount of data processed at each time is however really small.
- MapReduce [11] (2004): This work presents the first programming model suited to process a big amount of data simultaneously.
- Dryad [15] (2007): Applications that have to operate on big data are described for the first time using the formalism of Directed Acyclic Graphs, precisely highlighting the flow of information.
- Resilient Distributed Dataset [34] and Spark [35] (2010): These two contributions determine how in-memory operation change the big data scenario. Big data applications can be sped up, dividing the data set in structures that are kept in nodes' memory for fast access.
- Apache Tez (2015): This contribution introduces libraries for data-flow based engines and the concept that all the frameworks are built upon the same set of basic components and concepts.

The Google File System

We start our historical overview from the Google File System Paper [13]. The authors tackled the problem of designing *"a scalable distributed file system for large distributed data-intensive applications"*. The purpose of a file system is to determine how data are placed in the available storage medium. This work's peculiarity is that the storage medium is distributed across an arbitrary number of machines.

Among the novelties introduced by the paper, the authors mention explicitly the problem of managing large datasets of the volume of terabytes, comprising billions of objects. Albeit the authors do not talk explicitly about big data, their infrastructure and the characteristics of their problem precisely fit the definition that we have given of *big data*. Their file system is structured to partition datasets in relatively small chunks (the standard size of a chunk of data is 64 MB). Said chunks are then replicated and distributed to different locations to achieve fault tolerance, with a standard number of 3 replicas per chunk. A single master node is in charge of managing the location of all the chunks, in the form of metadata. This structure has then been used by the Hadoop File System[3] (HDFS) – the open-source implementation of the Google File System.

HDFS is nowadays used by most (if not all) the open source big data frameworks.

MapReduce

The introduction of the *Map Reduce* programming model [11] is arguably the main turning point in big data analytics. The functional style of the Map Reduce programming model – and of its homonym framework – is adequate to process large data sets on large clusters, matching precisely the definition of big data.

The basic idea behind the Map Reduce framework is to split the computation on the data in two defined steps: a *map* phase, and a *reduce* phase. The map phase processes elements one by one, applying one or more functions to the original data to generate intermediate `<key,value>` pairs. During the reduce phase, the intermediate values are merged based on their key parameter. Between the two operations, the dataset must be re-organised and the intermediate data having the same key must be localized. This operation is usually referred to as *shuffle*. During a shuffle operation, the master node is informed of the location of all the data with the same key, to aid the reduce operation. A shuffle does not necessarily produce any data movement, but rather determines how to partition the intermediate dataset and where it is located on the large cluster of available nodes. The intermediate data are then stored on the distributed file system, with a write operation, to re-organize them and partition them in such a way that the reduce operation can be executed locally in parallel on many different nodes.

Map Reduce introduced some fundamental concepts of big data analytics: (1) the necessity of a specific programming model, (2) the complexity related to the

[3] https://hadoop.apache.org/docs/r1.2.1/hdfs_user_guide.html

distributed computation, that should be handled by the big data framework.

The necessity of a specific programming model for big data is directly linked with a rigorous definition fo the application parallelism. The map operation is – by definition – fully parallel. In fact, the process of each *datum* is independent, and the function specified in the map invocation can be applied independently to each of them. Partitions of data can be generated arbitrarily and there is no need to group data in any way to perform the operations. On the contrary, the reduce operation is not fully parallel, but its degree of parallelism is determined by the amount of different keys in the data set. The operation on each key can be executed in parallel, as the processing of all the values associated with the same key is independent of the values given to other keys. In some particular cases (e.g., if the reduce operation satisfies the associative and commutative properties) the degree of parallelism can be increase further.

Map Reduce – as any following big data framework – uses this knowledge to automatically parallelize the operations that can be run in parallel and hide this process and its additional complexity from the programmer. The programmer can then focus on the analysis to be performed, without being concerned about the low-level framework implementation details.

Dryad

The Dryad framework [15] from Microsoft Research represents an alternative to write *"efficient parallel and distributed applications"* and achieve scalability with respect to the size of the input data.

The main novelty introduced by Dryad is the high-level description of big data applications by means of a dataflow graph, where vertices define computational steps and directed edges model the data path along the steps. To model an application properly, cycles are not allowed, as this would imply executing operations on data that will only be available as a result of the operation itself. The results is a description of big data applications as Directed Acyclic Graphs (DAGs). The idea of describing application using DAGs was successful and has been implemented in several frameworks, like Spark [35], Nephele [32], Hyracks [7]. However, Microsoft abandoned the Dryad project, discontinuing the framework and starting to contribute to alternatives.

The contribution of this paper – the DAG description – is crucial as it allows to rigorously define the parallelism that is intrinsic in the different operations: using edges and vertices the programmer is forced to explicitly state dependencies between the different operations. This enables efficient automatic parallelization of the application execution.

Resilient Distributed Datasets (RDD) and Spark

While the program writing phase is important, as testified by the DAG introduction, another area that called for improvement was the performance of the computation.

Slow computation on big data would have determined the failure of the entire big data idea, while fast computation would have enabled many different uses of big data.

The introduction of Resilient Distributed Datasets (RDDs) [34] introduced in-memory computation on large clusters. The call for a new memory management system came as a result of frequent requests for intermediate results (e.g., machine learning, data mining, and iterative algorithms refine the computation on intermediate data). Intermediate results are critical as the reading and writing procedures on them are time consuming, due to the volume of data. There is therefore a strong need to minimize the amount of stored intermediate results. A viable solution to this problem is in-memory execution [26, 36].

The general idea behind in-memory execution is to store the intermediate results in the computer Read Access Memory (RAM), instead of writing on the hard drive. Writing to disk is from 10 to 100 times longer than reading and writing from memory, allowing for significant advantages. While in a single node the savings are apparent, however, the presence of a network and of distributed computing nodes generates complications in unleashing the potential performance benefits.

In-memory execution does not directly process data from memory, but keeps track of how the input *datum* of the following operation should be retrieved, or of the so-called *data lineage*. Starting from a *datum* saved somewhere in the file system of a node, and knowing the set of operations to be performed, in-memory computation can be used to get the desired partial result. This allows to access the intermediate results without actually having to store them to disk, and using only the knowledge on how to retrieve the data from the node that either stores it or last processed it. A key enabler of the performance improvement is the *lazy evaluation*, or the fact that the proper evaluation of the operations on the *datum* happens only at the latest time instant, when the *datum* is needed.

This performance-boosting solution is implemented in the Spark framework. Spark distinguishes between two types of operations on the given data: *transformations* and *actions*. The former are the operations performed lazily, generating an output distributed dataset object from an input distributed dataset object. The latters are the operations that actually require the execution of the piped transformation and of the final action, the operation that generates the requested value. Transformation manipulate an input RDD object into an output RDD object, while actions generate a traditional object (an integer, a list, a string, etc) from an input RDD. Tables 2.1 and 2.2 show the extension of the Map Reduce programming model provided by Spark, respectively displaying the transformations and the actions that can be used in the framework.

Apache Tez

Apache Tez [27] is the most recent work presented here. We presented to enforce the reader's confidence in the similarities between the different big data frameworks.

More precisely, Apache Tez helps us showing that all the big data frameworks are built on the same concepts and components. Tez, in fact, is a *"library to build data-flow based engines"* and can be used to produce code to be executed using different frameworks: Map Reduce, Hive, Pig, Spark, Flik, Cascading, and Scalding. Tez confirms the *"intuition that most of the popular vertical engines can leverage a core set of building blocks"*.

This idea is the basis of our systemic approach, and reinforces the importance of looking for the common characteristics of all the frameworks, the first-principles that rule the processing of big data. Specifically, according to the creators of Tez, the common elements to the different frameworks are:

- A description of big data applications in the form of DAGs,
- The environment in which they run, Hadoop[4] and the Hadoop File System HDFS 2.1, and a policy for resource allocation – in most cases Apache YARN [29],
- Mechanisms for recovering from hardware failures,
- Mechanisms for publishing metrics and statistics.

We will not delve into further details of Apache Tez, but will limit our historical perspective to highlighting the consciousness that different frameworks are based on the same basic blocks. The following section is devoted to the clarification of these first principles.

2.2 First-Principles of Big Data Processing

The two most important breakthroughs in big data technologies so far have been: (1) the Map Reduce programming model, and (2) the DAG abstraction for program description. The validity of this statement is supported by the need to exploit parallelism in big data applications. The two mentioned contributions are related to one of the forms of parallelism found in big data applications, that we defined in section 1.3. On one hand we have *application-level parallelism*, or parallelism that arises from the possibility of executing multiple parts of the application in parallel (e.g., retrieving the marks given to students in two different exams and computing their averages can be done in parallel). On the other hand we have *data-level parallelism*, or parallelism that is intrinsic in the operation that is being executed on the data (e.g., adding one to every element of a vector can be done in parallel).

This section focuses on how to distribute the computation over the cluster and should help us undestanding what are the first principles of processing big data.

[4] http://hadoop.apache.org/

While capturing first-principle phenomena for a physical problem is easy and first-principle equations come directly from the law of physics, in computing problems finding first-principle phenomena is a difficult task. It is of course always possible to describe microscopic behavior, at the level of the transistors, and model computation using the laws of electromagnetism. However, this results in unprecedented complexity, and the given models are of little or no use to control the macroscopic phenomena like data moving from one location to another, or resources being allocated to a computational unit. These macroscopic laws, in computing systems, are themselves part of the design choices and should be therefore devised at the design level [3, 21].

In fact, first-principles laws are useful in computing systems too. There is a nice parallelism between physical problems and computing problems, in that these laws depend on the chosen level of abstraction, which can be chosen according to the aspects that need to be captured for the given problem solution. For instance, to model the behavior of a gas we could either decide to model the single molecules dynamics or to use the ideal gas theory. In the same way in computing we can go from modeling the physics of transistors to modeling the values of the variables during the computation. Our choice should then depend on which aspects of the problem are relevant for our study.

In big data applications, the relevant phenomena are related to (1) the processing of data, (2) data locality, and (3) the transmission of data through the network. In this section we will justify this statement starting from the design choices related to the programming model and the DAG description that characterize most (if not all) of the big data frameworks.

Exploiting Parallelism

Big data applications run on clusters (networks of computing machines) and usually do not get exclusive access to the hardware resources, being these computational or networking resources. There are several managers that handle the requests for resources received from concurrently running applications. Google, for instance, uses Borg [30]; the Hadoop environment adopted YARN [29] or Mesos[5]. These managers or resource schedulers receive resource requests and allocate units of computational resources in the cluster, e.g. `<2GB RAM, 1 CPU>`. The choice of the amount of resources to be allocated is almost invariantly formulated at the application level and met whenever possible at the cluster level.

A big data application that runs on a cluster distributes the different computation steps, i.e., different processing operations to be executed on the big data. These computation steps are defined as the couple *metadata* (defined in Section 2.1) and *operations*. The computational units receive the information (the couple of metadata and operations) and perform the given operation on the data that is identified by the metadata. According to the properties defined in Chapter 1, operations and metadata

[5] http://mesos.apache.org

are small with respect to the actual data. When the computation steps are sent across the network their contribution to the network load is negligible with respect the contribution of the transfer of the actual data.

A good distribution of the computational load will try to assign to a physical node (a computer) the processing of the data stored in the same physical node. This is based on the principle that *moving the computation is cheaper than moving the data* (see the HDFS architecture guide[6]). This is one of the reasons why the DAG is a useful abstraction of the big data application, as it allows to clearly identify which operations have to be performed on which data.

The DAG abstraction describes the flow of data through the operations and to the file system calls. The representation of big data applications in forms of DAGs at the present stage is not unique. Equivalent DAGs represent the same sequence of computation step, the difference between them representing different execution choices. Those choices are related to what the DAG describes of the given application. A natural consequence of this property is that the DAG can be manipulated to improve the computational efficiency of the application through better parallelism exploitation. Section 3.1 gives a more accurate description of the DAG model.

We illustrate the above concept with an example. An application is composed of three different operations. The first is operated on an input set. The second and third are independent operations that are to be executed on the data that is the result of the first operation. The big data framework can then choose different execution alternatives.

As a first approach, the input is read and the first operation is performed in parallel on parts of the input by different computational units (that differ from the machines[7]). The output of this first step is written on the file system. As a following step, the computational units are divided in two groups and to each group is assigned one of the two following operations, that can be executed in parallel. Following this approach, the intermediate dataset is read twice by the machines of the different groups and processed in parallel. The two final results are separately written on the file system.

As a second approach, the computational units can be immediately divided in two groups. Each group reads the input set and perform sequentially the first (equal for the two sets) operation and a second assigned operation among the two. Following this second approach, the first operation is performed twice on each *datum*, but there is no writing of the intermediate result on the file system – a procedure that is computationally heavy and imposes a lot of additional overhead.

There are other different possibilities depending also on eventual mathematical properties of the given operations. Each of these possibilities results in a different

[6] https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

[7] In the author's opinion, it would be proper to define the computational units as the purely sequential computing machines that constitute the cluster (i.e. the cores of the machines). In current big data frameworks, these computational units are running processes and are implemented as java threads hence not univocally associated to a CPU. Chapter 5 contains a few remarks on the matter.

DAG. Deciding when a DAG is optimal given a specific infrastructure is a problem outside the scope of this thesis. In here, we are concerned only with capturing all the given possibilities, to ensure a complete description of the problem.

The example shows very well the difference between the two forms of parallelism. In the first approach "the first instruction is performed in parallel on parts of the input". This is the operation-level or data-level parallelism described in Chapter 1, Section 1.3. For example, in the case of a map operation, the data can be partitioned arbitrarily and execution of operations could happen in parallel. This is usually called *multiple data, single operation* parallelism. In the example, the second step, i.e., the two parallel operations, is an example of application-level parallelism, also called *single data, multiple operations* parallelism. At this step of the execution both the forms of parallelism are present and the framework must manage and exploit them at the same time. The computational units must be divided in two groups and the complete input set must be divided among the elements of both the first and second group.

2.3 Spark Programming Model

Different programming models for big data ara available [38, 16], but at the present stage, these programming models are all inspired by the Map Reduce paradigm, and define some extensions on top of it, sharing most of the functionalities. For instance, the frameworks based on SQL-like[8] queries (like Hive and Pig) translate those queries in a set of operations described by a DAG [38].

In our model validation experiments, we use Spark as a big data framework, due to its customization and extensibility. We here then report the set of operations that are available in the Spark framework. As reported in Section 2.1, the Spark programming model is based on the RDD memory abstraction.

The main available operations are listed in tables 2.1 and 2.2[9]. Operations are divided between *transormations* and *actions* with the distinction reported in Section 2.1. From the computational point of view, the most relevant feature of each operation is its degree of parallelism. We are also interested in identifying the situations that require to shuffle the data. The former is related to the possibility to parallelize the processing as an intrinsic property of each functionality (or to data-parallelism), the latter to the necessity of performing write and read operations for re-organizing the data in the cluster.

To exploit data-level parallelism, Spark partitions the input set of an operation (or a *stage*, as we will discuss next) and distribute the operation among the computational units – the so called *tasks*. A task consists in a set of data-chunks of the original input and in the instructions to perform on them.

[8] Structured Query Language (SQL) is the language used for interrogating traditional databases.
[9] http://spark.apache.org/docs/latest/rdd-programming-guide.html

In Spark, the necessity of shuffling the data at a given point of the execution of the application is enforced if the next operation requires a different partition of the dataset (with respect to the one used for the pervious operation). In this case data are centrally re-organized through serialization of the RDD in the file system, and successive re-distribution of data chunks to the computational units. Spark evaluates this necessity of shuffle operations when building the application's DAG. Consider for instance the sequence of a map and a reduce operations: between the two a shuffle must be performed in order to aggregate the data with the same key. The DAG must capture this aspect[10]. What requires a shuffle operation is that a different partition of the dataset is required.

For our purposes, we are interested in how each operation is characterized in terms of level of parallelism that allows to framework to execute it separately in the different chunks of input data. We are also interested in knowing which operations require input from the file system or write their output on the file system.

Spark's application DAG

As written in Section 2.1, the use of dataflow diagrams in big data applications was introduced in the Dryad framework. Dryad enforced a dataflow programming model, creating a one-to-one mapping between the application and its DAG. The programmer is supposed to use a precise programming syntax to build the DAG – and the application itself. Each vertices defines a set of sequential operations to be performed on data (which are encoded as the edges entering vertices). The edges exiting from nodes point to the nodes that use the output of the given node as input. In Dryad, the DAG is only a model of the program, and does not represent anything about its execution. In successive frameworks, the DAG was automatically extracted and built by the big data framework based on the application code.

Since in our experiments we use Spark, this section explains how the DAG is generated from the application code in the Spark framework.

The DAG is used to represent the parallelism that characterizes the application we are considering and the read and write operations. Spark uses the actions to partition the application in *jobs*: this means that each job, given the definition of transformations and actions, starts from one or more RDD (a big data object), and performs different transformations on them, concluding with the action that generates a value.

The sequence of transformations in each job is described by a DAG. A transformation is applied on the input RDD – separately on its subsets (the performing of each of those is what we named in section 2.1 as task). If multiple operations can be sequentially performed on the same subset of data, they are grouped and performed

[10] A particular case is the one when the reduce operation satisfies the associativity property: nodes could start performing the reduce operation of the data stored in themselves and only after this perform the shuffle, possibly strongly reducing the volume of data to redistribute through the network.

Table 2.1 Spark: list of transformations.

Transformation	Meaning
map(func)	Return a new distributed dataset formed by passing each element of the source through a function func.
filter(func)	Return a new dataset formed by selecting those elements of the source on which func returns true.
flatMap(func)	Similar to map, but each input item can be mapped to 0 or more output items (so func should return a Seq rather than a single item).
mapPartitions(func)	Similar to map, but runs separately on each partition (block) of the RDD, so func must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.
mapPartitionsWithIndex (func)	Similar to mapPartitions, but also provides func with an integer value representing the index of the partition, so func must be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T.
sample(withReplacement, fraction, seed)	Sample a fraction fraction of the data, with or without replacement, using a given random number generator seed.
union(otherDataset)	Return a new dataset that contains the union of the elements in the source dataset and the argument.
intersection(otherDataset)	Return a new RDD that contains the intersection of elements in the source dataset and the argument.
distinct([numTasks])	Return a new dataset that contains the distinct elements of the source dataset.
groupByKey([numTasks])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs.
reduceByKey(func, [numTasks])	When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function func, which must be of type (V,V) => V. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.
aggregateByKey(zeroValue) (seqOp, combOp, [numTasks])	When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.
sortByKey([ascending], [numTasks])	When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument.
join(otherDataset, [numTasks])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through leftOuterJoin, rightOuterJoin, and fullOuterJoin.
cogroup(otherDataset, [numTasks])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (Iterable<V>, Iterable<W>)) tuples. This operation is also called groupWith.
cartesian(otherDataset)	When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements).
pipe(command, [envVars])	Pipe each partition of the RDD through a shell command, e.g. a Perl or bash script. RDD elements are written to the process's stdin and lines output to its stdout are returned as an RDD of strings.
coalesce(numPartitions)	Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset.

Action	Meaning
repartition(numPartitions)	Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network.
repartitionAndSortWithPartitions(partitioner)	Repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys. This is more efficient than calling repartition and then sorting within each partition because it can push the sorting down into the shuffle machinery.

Table 2.2 Spark: list of actions.

Action	Meaning
reduce(func)	Aggregate the elements of the dataset using a function func (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
collect()	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
count()	Return the number of elements in the dataset.
first()	Return the first element of the dataset (similar to take(1)).
take(n)	Return an array with the first n elements of the dataset.
takeSample(withReplace, num, [seed])	Return an array with a random sample of num elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.
takeOrdered(n, [ordering])	Return the first n elements of the RDD using either their natural order or a custom comparator.
saveAsTextFile(path)	Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file.
saveAsSequenceFile(path) (Java and Scala)	Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).
saveAsObjectFile(path) (Java and Scala)	Write the elements of the dataset in a simple format using Java serialization, which can then be loaded using SparkContext.objectFile().
countByKey()	Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
foreach(func)	Run a function func on each element of the dataset. This is usually done for side effects such as updating an Accumulator or interacting with external storage systems.

together in a single task. When a transformation requires input from multiple partitions, data must be re-organized, i.e. stored and re-partitioned, via a shuffle.

Shuffles delimit what in the Spark framework is called a *stage*. Each stage corresponds to a node in the DAG. More formally, a stage is a sequence of transformations at the end of which a shuffle (or an action in the case of the final stage of the job) must be performed. The flow of data into the stages is described by the directed edges: an edge from stage 1 to stage 2 defines that the output of stage 1 is the input of stage 2.

As frequently happens in interactive algorithms and machine learning applications different jobs may require the same operations on the same data (hence producing the same intermediate result). Spark avoids performing twice these operations, re-using the already produced RDD — the key concept of in memory execution of Spark defined in section 2.1. This means that the DAGs of different jobs can be connected[11].

The framework can then operate manipulations on the DAG to optimize its execution, but we will not delve in further detail as it is not strictly related to our work. The key points in Spark are the automatical retrieval of the DAG from the application code and the fact that this description is not unique. Optimizations performed on the DAG are usually aimed at minimizing read and write operations. Also, they do not strictly depend on the cluster and on the available resources, but rather on the data and the sequence of operations.

2.4 Performance and Simulation of Big Data Frameworks

The previous sections have provided an overview of how a big data framework works. It is a complex, distributed system that runs in a complex environment, a cluster of ten, hundred of even thousands of machines. Predicting the behavior of both the software and the hardware is a non-trivial task. However, as we stated in Chapter 1, Section 1.5, the ability to predict and even control the execution of an application would give us a clear advantage. This need is further witnessed by the existence of several works, related to either performance modeling or simulation of big data frameworks. Here we report just a small part of them in chronological order.

- Mumak (2009)[12]: is the Apache open source simulator for Hadoop.
- MRperf (2009)[13]: is a Map Reduce simulator by Virginia Tech researchers,

[11] Spark does not actually connect the DAG of different jobs: in case the described scenario happens it introduces the stages twice, but executes them only once re-using the available data. This is equivalent to connecting the DAGs.

[12] https://issues.apache.org/jira/browse/MAPREDUCE-728

[13] http://research.cs.vt.edu/dssl/mrperf/

the purpose of this simulator is to help programmers evaluating different set-up choices on the Hadoop cluster.

- SimMR (2011) [31]: another Map Reduce simulator oriented at allowing the analysis of different schedulers.
- YARNsim (2015) [22]: a simulator oriented at emulating the behavior of MapReduce in the YARN environment introduced in 2013.
- I/O modeling for big data applications over cloud infrastructures (2015) [25]: This works discusses a performance analysis and points out the difficulties in capturing the implementation intricacies of big data applications, proposing a purely input output approach for performance modeling.
- Analysis, Modeling, and Simulation of Hadoop YARN Map Reduce (2016) [8]: a statistical modeling of Map Reduce in the YARN Hadoop environment.
- Stage Aware Performance Modeling of DAG Based in Memory Analytic Platforms (2016) [14]: a gray-box approach for analyzing the performance of Spark applications based on the DAGs produced by the framework. The execution time of each stage of the DAG is statistically modeled running the operations on subsets of the input.

These are only a few works that demonstrate the importance of the topic. What is crucial is that none of these approaches exhibits an explicit system-theoretical approach, resulting in the fact that none of the presented models is dynamical. The proposals are purely statistical approaches or simulation-based analysis tools. They do not investigate abstractions or first-principle relations between the involved quantities. Despite that, similar considerations (e.g., the relevance of the degree of concurrency or of the data locality) are recurrent in the different studies. Models of phenomena are often derived from data observations, not from a systemic analysis of the involved quantities. For instance YARNsim [22] evidences a lognormal distribution of the execution time for map tasks with the average value linearly related to: (1) blocksize of input chunks, (2) size of intermediate data and (3) level of concurrency. In our opinion these work lack investigation on why these relations emerge, probabilistic distributions of certain quantities, and system dynamics.

3

Modelling Big Data Frameworks

To process the large amount of data typically involved in big data computations, data storage needs to be distributed onto different physical nodes in a cluster and duplicated for fault tolerance. At the same time, to obtain competitive performance, data processing must exploit parallelism. The combination of these two factors supports the use of a distributed software architecture with multiple storage and computation nodes.

These nodes concurrently contribute to the processing of the data. With the term data processing we indicate the execution of an *application* that contains one or more calls that operate on the large data set, typically executing instructions that read, write, or modify the data.

We will use first-principle models, describing only the core physics of the big data framework – which does not depend on the specific implementation framework. Traditionally first-principle models are defined as the models based on balance equations. In our model this kind of equation are in fact present in three ways: when modeling (1) the progression of the computation, (2) data transmission among the different computational steps and (3) resource allocation.

In this Chapter, we derive a generic abstract model for the execution of an application in a big data framework, and we propose some specialization of the abstract model to describe specific framework implementations, like Apache Spark. We will use a toy example to describe the main steps for retrieving the model from the code of a big data application.

3.1 Big Data Application

The source code of a big data application contains multiple calls to functions that manipulate the data, like `map`, `filter`, `reduce`, and their derivatives. These calls are resolved in the scope of the framework, that has access to the code that it should

```
1  text = read_input(in1_location)                          // Call #1
2  counts = text.flatmap(fun line -> line.split(" "))       // Call #2a
3    .map(fun word -> (word, 1))                            // Call #2b
4    .reducebykey(fun a, b: a+b)                            // Call #2c
5  counts.write_output(out1_location)                       // Call #3
6  articles = {a, an, the}
7  no_articles =
8    text.flatmap(fun line -> line.split(" "))              // Call #4a
9        .filter(fun word not in articles)                  // Call #4b
10 no_articles.write_output(out2_location)                  // Call #5
```

Listing 3.1 Pseudo-code for a big data application example.

(potentially optimize and) execute. Sequences of instructions containing framework calls are interleaved with other operations that the application should execute.

For example, the code for the application shown in Listing 3.1 contains five sequences of big data framework calls. The first one reads an input text from a given location (`Call #1`). The second one splits each line of the text into words with a `flatmap` (`Call #2a`), transforms each word into a tuple with the word as a key and a number 1 as a value with a `map` operation (`Call #2b`) and counts the number of occurrencies of each word with a `reducebykey` call (`Call #2c`). The third call writes the the counts to an output location (`Call #3`). The fourth call parses the text dividing it into words with the `flatmap` function (`Call #4a`) and then removes the articles using a `filter` function (`Call #4b`). Finally, the last call saves the text without articles to a given location (`Call #5`).

Looking at the specific calls, it is possible to identify the input and the output data of each operation. These create a logic link between the different calls, that depend upon each other for execution. We can represent each of the framework calls as the node of a graph, and the dependencies as directed edges between nodes.

Arrows represent dependancies related to big data sets. Those data sets are generally transmitted through the different functions being written on the distributed file system, hence a write and a read operations are required. If the intermediate *datum* is small enough that it doesn't require distributed storage the it doesn't require to be written on the file system and can instead be stored locally[1]. For this reason in such cases no arrow connection between the nodes related to the function calls is required. Under the light of this statement the DAG becomes a representation of both the program and its execution.

DAG Definition

The result of the analysis in the previous paragraph is a Directed Acyclic Graph, which both represents the order of execution of the function calls in the framework

[1] In general further processing of those data wont even require more calls of the framework functions.

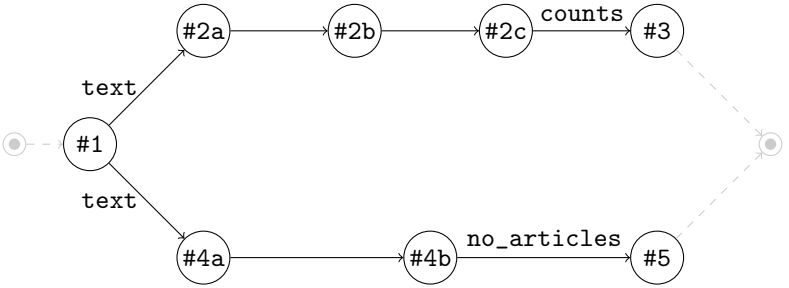

Figure 3.1 A Directed Acyclic Graph models the framework calls of the application shown in Listing 3.1.

and the read write operations to the file system. This abstraction is more general than the one presented in Chapter 2 for frameworks like Apache Spark, Apache Tez, and Dryad as it is generated from the generic code and not defined by the framework. For this reason it can theoretically be applied also to the execution of application in frameworks that do not explicitly implement the DAG.

Nodes of the DAG represent calls that should be executed, while arrows represent synchronization points between the function calls executions, that model the need for input data that should be written on the file system by the execution of previous function calls. Figure 3.1 represent the DAG of the code block in Listing 3.1. Dashed arrows and gray marks are added only for convenience to represent the entry and exit point of the code block execution and to highlight that this is only a code block in a possibly larger application. On top of the directed edges, we have identified the name of the variables associated with the data exchanged by the framework function calls. Assuming we know the control flow of the big data application, we can model the big data framework calls that are going to be executed by any application using a DAG[2].

Formally, se use the following notation.

- **Definition (Directed Graph):** A Directed Graph is a tuple $< V, E >$ where $V = \{v_1, v_2, \ldots v_n\}$ is a set of n vertices or nodes, and E is a set of directed edges, each of which can be written as $v_x \rightarrow v_y$, indicating a connection between the nodes v_x and v_y both in V, where v_x is the start point and v_y is the end point of the edge.
- **Definition (Path):** A path in a directed graph is a sequence of vertices that are connected with directed edges. For example, if $V = \{v_1, v_2, v_3\}$ and $E = \{v_1 \rightarrow v_3, v_2 \rightarrow v_3, v_3 \rightarrow v_1\}$, a possible path is the sequence $[v_2, v_3, v_1, v_3]$.

[2] Our aim is to verify properties related to the execution time of the big data application. The assumption of knowing the control flow is therefore not a limiting one, as we can run the verification process for all the possible control flows of the application and observe the worst-case scenario. We still need it to know previous the the application execution which framework calls will be executed.

- **Definition (Cycle):** A cycle is a path that starts and ends with the same node.
- **Definition (Directed Acyclic Graph):** A Directed Graph that has no cycles is also referred to as a Directed Acyclic Graph (DAG).
- **Definition (Input Edges of a Node):** For each node v_x we denote with $\mathscr{E}(v_x)$ the set of directed edges that have the node as the end point. $\mathscr{I}(v_x) = \{v_i \rightarrow v_j \in E \mid v_j = v_x\}$.
- **Definition (Output Edges of a Node):** For each node v_x we denote with $\mathscr{O}(v_x)$ the set of directed edges that have the node as the start point. $\mathscr{O}(v_x) = \{v_i \rightarrow v_j \in E \mid v_i = v_x\}$.
- **Definition (Node Predecessors):** A node v_x is said to belong to the set of *predecessors* of a node v_y – denoted with $\mathscr{P}(v_y)$ – when the arc $v_x \rightarrow v_y$ is included in the set of arcs of the DAG, *i.e.*, if $v_x \rightarrow v_y \in E$. In other terms, $\mathscr{P}(v_y) = \{v_x \mid v_x \rightarrow v_y \in \mathscr{I}(v_x)\}$.
- **Definition (Node Successors):** A node v_x is said to belong to the set of *successors* of a node v_y – denoted with $\mathscr{S}(v_y)$ – when the arc $v_y \rightarrow v_x$ is included in the set of arcs of the DAG, *i.e.*, if $v_y \rightarrow v_x \in E$. In other terms, $\mathscr{S}(v_y) = \{v_x \mid v_x \rightarrow v_y \in \mathscr{O}(v_x)\}$.
- **Definition (Initial Nodes):** The set of initial nodes $\mathscr{I}$ is defined as the set of nodes whose predecessors sets are empty. Formally, a node $v_x \in \mathscr{I}$ iff $\mathscr{P}(v_x) = \varnothing$.
- **Definition (Final Nodes):** The set of final nodes $\mathscr{F}$ is defined as the set of nodes whose successors sets are empty. Formally, a node $v_x \in \mathscr{F}$ iff $\mathscr{S}(v_x) = \varnothing$.

We refer to the application DAG as defined above (that is with a one-to-one correspondence between nodes and function calls) using the term "fine-grained DAG". Irregardless of the execution framework, for every block of code, we can always determine the fine-grained DAG of the application. In some frameworks, the (fine-grained) DAG is manipulated (either at compile or run time) to enable further optimizations, with the aim of reducing computation time, and increasing efficiency. This improvement is possible as the DAG is not only a model of the program but captures also the input output operations on the file system.

The information included in the DAG for each node is (1) the function call that the nodes represent, (2) its input edges, and (3) output edges. The function call represents the operation that the big data framework is applying to the data and can be selected among the ones available for the specific big data framework. The input and output edges – $\mathscr{I}(v_x)$ and $\mathscr{O}(v_x)$ – model the data needed for the computation and produced by the computation.

DAG Manipulations

If we let a single node represent a sequence of function calls instead of a single one, we can define optimizations on the DAG. In this case, we interpret the edges as the definition of the input ad output sets of the given sets of operations and we can define all the possible mutations that are applicable to a given graph. The transition from a single function calls to a set of calls models the execution of multiple operations on the same partition of the data without the need for intermediate operations involving writing results. This enables optimizations like recomputing a set of data instead of transferring the data over the network. We now investigate the possible maniplations of the DAG.

- **Join of two nodes:** a node can be joined with its successor if the set of successors contains only that specific node and the node is the only predecessor of its successor. In place of the two nodes, one single node remains in the DAG, representing the sequence of operations of the two joint nodes. The predecessors of this joint node are the predecessors of the first node and the successors of the joint node are the successors of the original successor node. The new DAG describes exactly the same sequence of operations but hides the intermediate result that would be generated between the two operations, avoiding the read and write operations of it. For this reason the join two nodes manipulation is applicable only if the one to one relation is satisfied. The necessity but not sufficiency is related to the possibility that two successive operations might require read and write operations in between for data shuffling. Formally, given a DAG $D =< V, E >$ and two nodes $v_x, v_y \in V$ such that $\mathscr{S}(v_x) = \{v_y\}$ and $\mathscr{P}(v_y) = \{v_x\}$ we can define a new DAG $D' =< V', E' >$ with $V' = V \bigcup v_z \setminus \{v_x, v_y\}$ and E' such that $\mathscr{P}(v_z) = \mathscr{P}(v_x)$ and $\mathscr{S}(v_z) = \mathscr{S}(v_y)$[3].

- **Duplicate a node with respect to the input:** every node with strictly more than one successor can be split in two nodes characterized by the same operations and same predecessors. The sets of the successors of the new nodes must be a partition of the successors set of the single node. The two resulting nodes are associated with the same operation that defined the starting one. This manipulation models the simple choice of executing twice the same operations on the same dataset (possibly in parallel on different computation units). For this reason it can always be applied. Formally, given a DAG $D =< V, E >$ and a node $v_x \in V$ with $\mathscr{S}(v_x) = \{v_{y,i} | i = 1, s\}$ we can define a new DAG $D' =< V', E' >$ where $V' = V \bigcup \{v_{x.i} | i = 1, \ldots, s\} \setminus v_x$ and E' is such that $\mathscr{P}(v_x) = \mathscr{P}(v_{x,1}) = \ldots = \mathscr{P}(v_{x,s})$[4] and $\mathscr{S}(v_{x,i}) = \{v_{y,i}\}$, $i = 1, \ldots, s$.

[3] $\mathscr{P}(v_x)$ and $\mathscr{S}(v_y)$ are defined in D while $\mathscr{P}(v_z)$ and $\mathscr{S}(v_z)$ are defined in D'

[4] $\mathscr{P}(v_x)$ is defined in D while $\mathscr{P}(v_{x,i})$ are defined in D'

- **Duplicate a node with respect to the output:** this manipulation is the dual of the former. It is operated on a node with strictly more than one predecessor and consist in applying the operation separately to the different outputs of the predecessors. The successors set of each of the new nodes must coincide with the one of the single node. The two resulting nodes are associated with the same operation of the starting one. Differently from the previous manipulations, this one cannot always be applied depending on the nature of the function calls that are associated to the node to be duplicated[5]. Formally, given a DAG $D =< V, E >$ and a node $v_y \in V$ with $\mathscr{P}(v_y) = \{v_{x,i} | i = 1, p\}$ we dan define a new DAG $D' =< V', E' >$ where $V' = V \bigcup \{v_{y.i} | i = 1, \ldots, p\} \setminus v_y$ and E' is such that $\mathscr{S}(v_y) = \mathscr{S}(v_{y,1}) = \ldots = \mathscr{S}(v_{x,p})$ and $\bigcup_{i=1,\ldots,p} \{\mathscr{P}(v_{y,i})\} = \{v_{x,i} | i = 1, \ldots, p\}$.

- **Switch two nodes:** a node can be switched with its successor if this is the only successor of the considered node and the node is the only predecessor of its successor. Moreover, it is required that the two operations satisfy the commutativity property. Formally, given a DAG $D =< V, E >$ and two nodes $v_x, v_y \in V$ such that $\mathscr{S}(v_x) = \{v_y\}$, $\mathscr{P}(v_y) = \{v_x\}$, $\mathscr{P}(v_x) = \{v_{pr,i} | i = 1, \ldots, p\}$ and $\mathscr{S}(v_y) = \{v_{sc,i} | i = 1, \ldots, p\}$ we can define a new DAG $D' =< V, E' >$ where E' is such that $\mathscr{S}(v_y) = \{v_x\}$, $\mathscr{P}(v_x) = \{v_y\}$, $\mathscr{P}(v_y) = \{v_{pr,i} | i = 1, \ldots, p\}$ and $\mathscr{S}(v_x) = \{v_{sc,i} | i = 1, \ldots, p\}$

The described manipulations are strictly related to how the graph models the application in terms of the sequence of operations that are given a specific set of input and output data. Using these manipulations, we are able to describe all (but not only) the possible graphs that are related to the given code. The fundamental property that must be guaranteed while manipulating the DAG is that the resulting programs always provide the same result. This concept is used to define the *equivalence* between two DAGs.

- **Definition: Equivalent DAGs** Two DAGs are said to be equivalent when the sequences of operations they define applied to the any input produces the same output[6].

By means of those manipulations we can model how the sequences of framework calls discussed can be grouped into different "blocks of calls", depending on the big data framework compiler logic and on memory access.

In the example of Figure 3.1 we can apply the join two nodes manipulation to both the branches of the DAG: three times to the upper branch grouping operations

[5] The operation must allow independent processing of each of the resulting subsets of the input set, that is the degree of parallelism must allow it.

[6] Note that equivalence is related to what the DAG describes of the definition of the program, not of its execution.

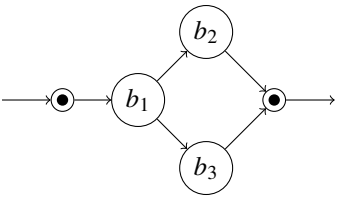

Figure 3.2 Result of DAG manipulation by joining of nodes.

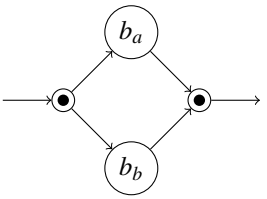

Figure 3.3 Resulting DAG after further manipulation.

`#2a`, `#2b`, `#2c` and `#3` and two times to the lower branch grouping `#4a`, `#4b` and `#5`. The resulting DAG is shown in figure 3.2 where therefore node b_1 is related only to operation `#1`, node b_2 is related to `#2a`, `#2b`, `#2c` and `#3` and node b_3 is related to `#4a`, `#4b` and `#5`. Figure 3.2 shows the result of this manipulation.

Further manipulation could be the choice to duplicate with respect the input the first node and join both the resulting nodes to the subsequent ones obtaining the DAG shown in figure 3.3 . Where node b_a is related to operations `#1`, `#2a`, `#2b`, `#2c` and `#3` and node b_b is related to `#4a`, `#4b` and `#5`. Figure 3.3 shows the resulting DAG after applying the additional manipulation.

Note that `read_input` and `write_ouput` operations defined in the code of the example do not strictly correspond to the input output operations of the big data framework. In fact not only when executing those commands data are written and read data from the file system, but the programmer is simply defining the inputs and the outputs of the overall application. For instance between the `map` and `reducebykey` operations a shuffle must be performed as defined in Chapter 2, and the dataset must be re-partitioned. The re-partitioned dataset will then be read by the executors of the reduce operation. Another important consideration is the DAG shown in Figure 3.2 is not compatible with an actual execution of the application as the shuffle operation between the two function call just mentioned is not represented. A coherent manipulation of the DAG would instead be: joining the couples of nodes `#2a`, `#2b` and `#2c`, `#3` on the upper branch, and joining all the three nodes `#4a`, `#4b` and `#5` on the lower branch resulting in the DAG represented in figure 3.4.

Big data frameworks are able to use the DAG abstraction to decompose the processing of the application in different threads and exploit the parallelism and locality

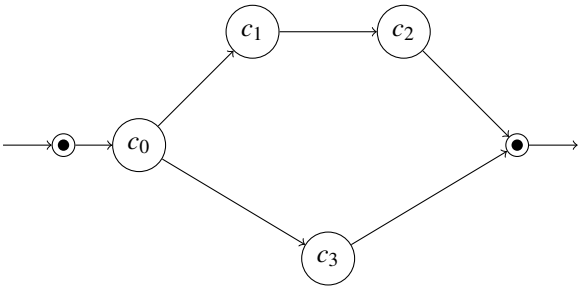

Figure 3.4 DAG manipulated taking into account the shuffle.

of data to improve computing performances. The application is first decomposed in the different computational steps[7] corresponding to the nodes of the DAG. Each of the computational steps is decomposed in different parallel processes[8]. This is how the DAG abstraction allows the big data framework to exploit the two forms of parallelism. In the next section we will propose a dynamical model for the distributed processing of each of the computational steps.

3.2 Model of the Computational Step

The DAG abstraction allows us to model the application as a set of computing steps (the nodes) that synchronize and exchange information (the edges). The processing of each of those steps is distributed according to the degree of parallelism allowed by the operations that define it. This means that there are different executors for the same computational step and that each of them performs the given operations on a subset of the input of the overall node[9].

The allocation and synchronization of these executors is the level at which we wish to exploit data locality, i.e., a good big data framework will try to allocate the executors in the cluster close to where the data they are supposed to process are located. On the one side, data locality has a strong impact on the execution of the application. On the other side, it strongly depends on the specific execution instance and on the cluster on which the application is run. For those reasons we chose for our model a higher level of abstraction that is oblivious to this phenomenon[10].

We generically define the given resources as a single variable and relate the progress of data processing to this single variable, being it executors, cores, CPU time and so forth. The resources are then assigned to the overall computational step (or stage in Spark jargon). If we want to regulate the amount of resources with a

[7] In Spark, computational steps are called stages.
[8] In Spark, parallel processes are called tasks.
[9] The union of the subsets must be the original input set.
[10] The retrieved model can be extended and overcome this limitation, as explained in Chapter 5.

controller, it must also be clear that the executors are precisely the allocated resource. Using the control terminology, the executors are the control variable, not the processes to which we allocate resources. Under the light of those considerations we model the processing of each node aggregately with respect the distributed executors.

Each computational step will go through different phases during the processing phase. First of all, the step must synchronize with its predecessors and retrieve the metadata containing information on data locality. Once the predecessors have terminated, the node executors can be allocated and then start collecting and processing the data. During the processing phase, the big data framework allocates resources to the computational step, possibly verifying the progress of the execution by communicating with its successors. Control comes in here, determining the choice of the amount of resources to be given to the computational step at each point in time. Depending on the specific big data framework, resource allocation is carried out using different scheduling policies. In our perspective, this is a *control* problem, where there is a target processing rate and the resource manager should allocate resources closing a control loop on the progress rate error[11]. Once the processing of the considered computational step is finished, the executors communicate where they stored the produced data to the framework and the framework coordinates the end of the computational step with the start of the successors.

The sequence of states of the computational step is represented by the finite state automaton shown in Figure 3.5. The resource manager (the component of the framework that manages the allocation of computational resources) is represented in Figure 3.6. We can now go through each of the states and provide a more rigorous definition.

Waiting for Predecessors (WFP)

This state captures the behavior a node before its input set is ready for processing. It is assumed that it would be meaningless to start the computation of a node before all the predecessors have finished. This is reasonable as before the end of the computation of the predecessors the input-set is not even completely defined.

The node is therefore waiting for its predecessors to asynchronously provide the metadata corresponding to the input data. The exchange of information is modeled through the transition `get metadata from predecessor` and it is synchronous with a transition in the predecessor (see state `communicating with successors`). The amount of exchanged information is relatively small, and we can neglect the time consumed for the communication (which only involves metadata). When all the predecessors terminate and the input set is ready, the node can move to the next state taking the transition `all predecessors have terminated`.

[11] This feedback approach is actually implemented in xSpark, an extension of the Spark framework developed at Politecnico di Milano.

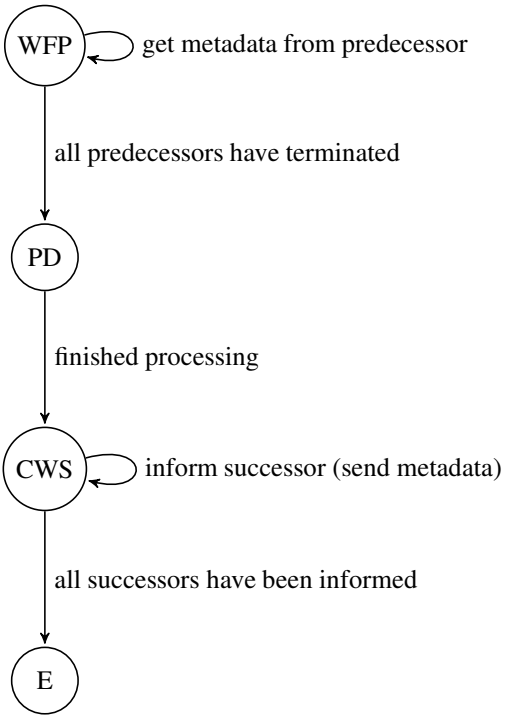

Figure 3.5 State machine representing the states of the single computational step.

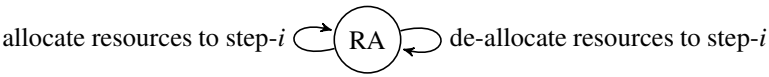

Figure 3.6 State machine of resource allocator.

Processing Data (PD)

In this state we capture the processing of the data. The processing speed depends on the allocated resources. Resources $u(t)$ – like CPU time and cores – are assigned to executors, that process the data. We model the progress rate as an integrator where the integration rate is a generic function of the allocated resources $g(\cdot)$.

An example is

$$\dot{e}(t) = g\left(e(t), u(t), d(t), \theta\right), \tag{3.1}$$

where $e(t)$ represents the state variable (the computation progress), $u(t)$ are the allocated resources, $d(t)$ is a positive quantity that represents a time varying efficiency of the computation and θ is a vector of parameter describing the cluster.

The time varying efficiency is a disturbance on the system representing all the possible external inputs affecting the computation progress, from the varying data load to the external disturbances acting on the cluster's machines. The function $g()$ must be non negative[12] and saturated with respect to the input resources according to the maximum available computational parallelism in the overall cluster — i.e. the number of available sequential computational units, the cores — and the maximum parallelism allowed by the operations represented by the node.

The procedure of writing data on the file system is considered to happen immediately after the data are generated. It therefore overlaps with the processing time and does not require specific modeling[13]. This holds under the assumptions that (1) the writing action doesn't affect the processing rate, and (2) that the writing rate is higher than the processing rate – i.e. the bottleneck is the processing.

When the variable $e(t)$ reaches the threshold representing the volume of input data the node can take the transition `finished processing` and move to the next state.

Communicating with Successors (CWS)

This state is the dual of the `Waiting for predecessors` one, and models how the node synchronizes with successors. The node takes the transition `inform successor` once for each of the successors synchronizing with them and sharing the metadata of its output-set. In the real case the metadata should describe data locality and volume, to our level of abstraction is sufficient the latter. The volume of generated data is defined by the *fraction factor* (a parameter of the node – and of the set of operations) multiplied by the volume of the input-set. Once all successors are informed the node moves to the final state `End` (E) where it is eventually available in case backup information are needed.

Resources and Resource Allocation Modeling

The model of the overall application is constituted by a set of parallel switching systems, equal to the one just presented. The DAG describes the required synchronization among these different switching systems, determining the successor and predecessor relationships. Each of the switching systems models independently the computation of a single step. The different switching systems share the resources to the same cluster, hence the sum of the allocated resources $u(t)$ to each step must be saturated according to the cluster dimensions.

The Resource Manager can allocate resources with different policies, usually the programmer has several choices for each framework. The resource allocation policy determines both when the the transitions `allocate resources` and `de-allocate`

[12] A negative value of $g()$ implies the loss of processed data during the application execution. We assume this cannot happen, as the File System takes care of replicating data and guarantees fault tolerance.

[13] This is even more valid when the computational unit is on a machine that works also as storage node.

`resources` are taken and how much resources to which stage are allocated. In our control-theoretical perspective this is the control law.

A final note about the modeling of resources is that, as we stated in Chapter 1 Section 2.2, those applications are not the only running on the cluster. We can easily represent this making the saturation level of total resources vary in time.

3.3 Frameworks: the Resource Allocation

In this section we investigate how to apply the model just defined to the execution of applications defined in different frameworks. The basic ingredients for defining the model are (1) the DAG definition and (2) the behavior of the Resource Manager.

For defining the resource manager behavior, we need to analyse the scheduler of the cluster environment in which applications run. All of the framework that we will model run in an Hadoop environment in which the programmer is allowed to plug different scheduling policies. The standard policy is the FIFO scheduler that does not use any feedback signal and simply allocates resources to all the tasks it receives in the arrival order. This can be implemented in our model activating the transition `allocate resources`, as long as the processing is not terminated and there is resource available. Alternatives to this behavior are the "Fair scheduler", developed by Facebook, and the "Capacity scheduler", developed by Yahoo!. These can be modeled as an arbitrary activation of `allocate resources` (or `de-allocate resources`), providing the amount of resources to allocate at each activation.

We also model the allocation policy implemented in xSpark [4]. xSparx implements a feedback-based resource allocator, described in the following.

xSpark Resource Allocator

The xSpark resource allocator implements a cascade control loop. The first loop is at the level of the overall application and acts on the deadline of the single sages – adjusting the deadlines for proper allocation. The second loop is local to each computational step and allocates the resources.

Application Level Control Loop The deadline control loop generates the deadline for each computational step (stage) using a *heuristic approach*, based on the deadline of the overall application $\mathscr{D}_{\mathscr{A}}$, supplied by the user. Denoting with $\mathscr{T}_j$ the time elapsed from the beginning of the processing at the beginning of stage j, the deadline $\mathscr{D}_j$ for the stage j is generated as show in Equation 3.2.

$$\mathscr{D}_j = \frac{\alpha \cdot \mathscr{D}_{\mathscr{A}} - \mathscr{T}_j}{w_j} \tag{3.2}$$

Here, α is a constant parameter between 0 and 1, used to implement a given margin with respect of the overall deadline, and w_j is a weight associated with each stage, determined according to Equation 3.4. Denoting with $\mathscr{R}_j$ the number of

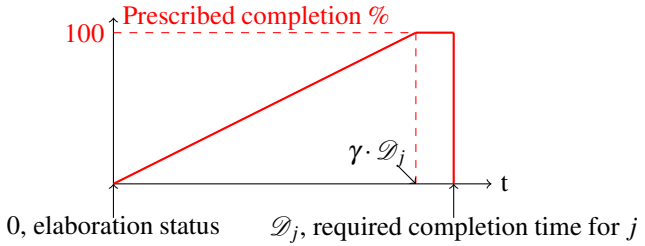

Figure 3.7 Processing progress reference for stage j.

remaining stages to complete at the moment of the start of the execution of stage j, with r_j the ratio between the expected duration of the stages that still need to be executed and the duration of stage j's computation, and with d_j the expected duration of stage j, the ratio is determined as

$$r_j = \frac{1}{d_j} \cdot \sum_{i=j}^{n} d_i, \tag{3.3}$$

and the stage weight is then defined as

$$w_j = \beta \cdot \mathscr{R}_j + (1-\beta) \cdot r_j. \tag{3.4}$$

In Equation (3.4), β is a constant determined conducting a sensitivity analysis, in xSpark its value is set to $\frac{1}{3}$.

Stage Level Control Loop The inner control loop uses the deadline generated by the external control loop to compute the reference of processing progress, with some safety margin defined by the user like the one plotted in Figure 3.7. The margin is implemented forcing the processing to attempt ending at time $\gamma \cdot \mathscr{D}_j$ instead of $\mathscr{D}_j$ with $\gamma \in (0,1]$.

Using this reference and the actual progress, the processing error is computed. Based on this error a discrete Proportional and Integral (PI) controller is implemented, to decide how much resource to allocate. The necessity of a PI is driven by the reference signal being a ramp. For asymptotic ramp setpoint tracking, two integrators are required in the loop: one is provided by the process when the progress rate is integrated to obtain the actual amount of processed data, the second is therefore included in the controller.

The resource allocation dynamics is modeled as shown in Equation (3.5). The input $c(k)$ is the amount of requested resources at time k and the output $\rho(k)$ is the processing progress rate at time k. The dynamics includes a pole p and an arbitrary function $f()$: the pole is estimated using step-response analysis and $f()$ is experimentally estimated.

$$\rho(k) = p \cdot \rho(k-1) + (1-p) \cdot f(c(k-1)) \tag{3.5}$$

Given the progress rate the actual amount of processed data $e(k)$ at time k is obtained by simply integrating it as shown in Equation 3.6, where q is the control timestep.

$$e(k) = e(k-1) + q \cdot \rho(k-1) \tag{3.6}$$

Or equivalently using percentages (D_o is the total amount of data to process):

$$a_{\%}(k) = 100 \cdot \frac{e(k)}{D_o} = a_{\%}(k-1) + \frac{100 \cdot q}{D_o}\rho(k-1). \tag{3.7}$$

For the tuning of the controller we re-interpret the problem in the continuous time domain, as it is more natural. Moreover specifications are given as times and not time-steps hence a continuous time representation is apparently more appropriate. First we write the discrete time transfer function of the system applying the Z-transform to Equations 3.5 and 3.7 and substituting $u(k) = f(c(k))$.

$$\frac{a_{\%}(z)}{\rho(z)} = \frac{100q/D_o}{z-1}, \quad \frac{\rho(z)}{u(z)} = \frac{1-p}{z-p} \tag{3.8}$$

The two are in series, therefore the overall transfer function is:

$$\frac{a_{\%}(z)}{u(z)} = \frac{100q(1-p)/Do}{(z-1)(z-p)}. \tag{3.9}$$

The discrete transfer function of the PI is:

$$C(z) = K \cdot \frac{z-\xi}{z-1}, \tag{3.10}$$

where K and ξ are the parameters to tune.

Second we back-apply the forward difference method and obtain the continuous time transfer functions: eq. 3.11 for the process and 3.12 for the controller.

$$P_c(s) = \frac{\mu_P}{s(1+s\tau_P)}, \qquad \mu_P = \frac{100K_f}{Do},\ \tau_P = \frac{q}{1-p}, \tag{3.11}$$

$$C_c(s) = \mu_C \frac{(1+s\tau_C)}{s}, \qquad \mu_C = \frac{K(1-\xi)}{q},\ \tau_C = \frac{q}{1-\xi}. \tag{3.12}$$

To guarantee stability the Bode magnitude diagram of the loop transfer function $L(j\omega)$ is forced to behave like in Figure 3.8. $f()$ is then substituted with a positive, bounded, unknown gain K_f for which we assume a nominal value $K_{f,nom}$. Then the controller parameters are set: the time constant τ_C is chosen proportional by $\eta > 1$ to τ_P and the controller gain so that the cutoff frequency (for the nominal value of K_f) is the logarithmic mean of τ_C and τ_P. This provides under nominal conditions the values for the cutoff frequency and phase margin reported in 3.13.

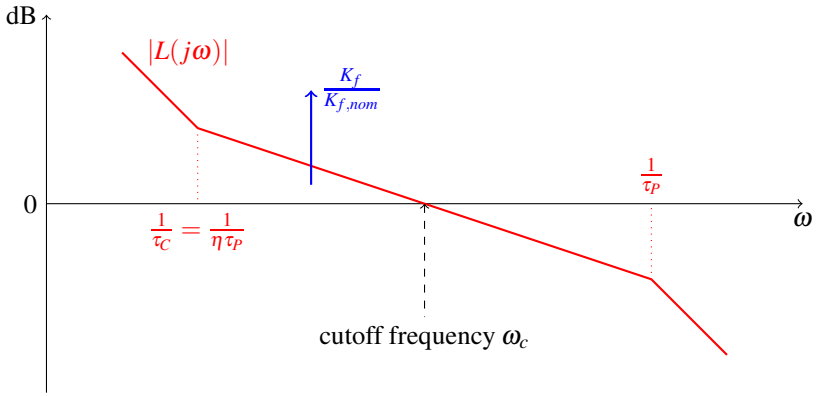

Figure 3.8 Bode magnitude diagram of the required open-loop frequency response.

$$\omega_c = \frac{1}{\eta\sqrt{\tau_P}}, \qquad \varphi_m = \arctan\left(\sqrt{\eta}\right) - \arctan\left(\frac{1}{\sqrt{\eta}}\right). \tag{3.13}$$

Once retrieved the parameters are converted back to the discrete-time domain obtaining the following formulas:

$$K = \frac{Do(1-p)}{100qK_f\sqrt{\eta}}, \quad \xi = \frac{\eta + p - 1}{\eta}. \tag{3.14}$$

With a value of η around 10 – corresponding to ϕ_m around 55° – as a reasonable default the controller behaves satisfactorily provided that q is small enough with respect to the required completion time. The discrete-time controller in state space form reads:

$$\begin{cases} x_C(k) &= x_C(k-1) + (1-\xi)\left(a^{\circ}_{\%}(k-1) - a_{\%}(k-1)\right) \\ c(k) &= Kx_C(k) + K\left(a^{\circ}_{\%}(k) - a_{\%}(k)\right) \end{cases} \tag{3.15}$$

The number of required cores may transiently assume negative values or exceed the number of available resources, this plainly means that the control variable is saturated (as we stated also in Section 3.2). Hence the anti-windup must be implemented according to eq. 3.16.

$$x_C(k) = \frac{c(k)}{K} - a^{\circ}_{\%}(k) + a_{\%}(k). \tag{3.16}$$

Our model captures the xSpark resource allocator activating transition `allocate resources` (or `de-allocate resources` if the control action is negative) at every control time-step q and computing and allocating the resources computed according to the PI control law.

Figure 3.9 Map Reduce applications DAG.

3.4 Frameworks: the DAG

This section briefly describes how to retrieve the execution DAG of each application implemented in different frameworks.

Map Reduce

We presented the general features of the Map Reduce framework in Chapter 2, Section 2.1. The framework allows developers to implement applications divided in subsequent map and reduce computational steps with a shuffle in between the two steps. According to our definition of the fine-grained DAG (see Section 3.1), we obtain a simple graph like the one in Figure 3.9. The first node of the graph models the map procedure while the second models the reduce operation. The edge connecting the two captures instead the shuffle procedure.

Spark (and xSpark)

We presented the Spark framework in Chapter 2, Section 2.3 and mentioned the xSpark extension. From the DAG perspective, the two frameworks are equivalent. They make explicit use of the DAG for parallelism exploitation at processing time, using a similar definition to the one we provided. Specifically the two frameworks also define the computational steps (the nodes of the graph) – called stages – using write, re-organize and read operations. The application DAG is therefore also modeling its execution. The actual implementation of xSpark, executes the stages sequentially, therefore doesn't exploit the application-level parallelism.

Tez

Apache Tez, introduced in Chapter 2, Section 2.1, provides the programmer with a DAG API to build and represent the big data application. Analogously to our definition, DAG edges in Tez represent both the logical connection and the physical exchange of data between the computational steps (even though here this is not strictly related to the shuffle procedure, that as we said is the only situation in which data motion is strictly required). Moreover, the Tez engines allow for either static or dynamic (i.e., at runtime) definition of the graph. The model that we propose can be used to represent a Tez DAG. Also, as Tez is not only a framework but a library for building engines for DAG-based frameworks, all the frameworks supported by Tez are compatible with the model defined in this thesis.

SQL-like Frameworks

As evidenced in [38], SQL-like programming models rely on frameworks that translate the query in jobs for other DAG based frameworks, hence validating once more that the DAG is a power abstraction to capture a general big data application. For instance, Hive translates queries in Map Reduce jobs connected as a DAG (in this case each of the nodes corresponds to a map and a reduce operation, duplicated to capture the shuffle that is executed between the two operations). The retrieved DAG is compatible with the proposal of this thesis.

4

Implementation and Validation

Now that we have described the proposed model for big data applications, we can proceed to its implementation and validation. As the final aim of this work is to perform model checking and verify properties of the different applications, we implemented the model using the statistical model checker *UPPAAL*[1]. This software tool serves well for our purpose, because it allows the study of stochastic switching systems, and the properties we want to verify are exactly of that nature, like for example probability distributions and expected values of convenient indicas characterizing the evolution of the system. This Chapter first introduces the UPPAAL software tool, and then provides some simulations to assess the validity of the UPPAAL implementation our model.

The aim of model validation in the long term research project to which this thesis belongs, is mainly to show the validity of our way of reasoning and approach to the big data problem. We want to show that by means of simple "first-principle" abstract models we can recreate the behavior of the applications and the frameworks. As stated in the introduction, we expect that understanding those concepts will eventually pave the way the implementation of a brand new generation of big data frameworks.

4.1 Model Checking and UPPAAL

In the last few years, model checking techniques have been attracting a strong interest in both the academic and the industrial world. Those tools are of the "push a button" kind, meaning that once the model and the properties one wants to verify are defined and implemented in the formalism of the tool, all one needs to do is to push a button and the software will tell whether or not the system satisfies the

[1] http://www.uppaal.org/

property. According to this statement the key ingredients of a model checking tool are:

- An operational formalism to describe the the system to be analyzed,
- A descriptive formalism to describe the properties to be verified,
- An algorithm that can check if a system described using the given operational formalism satisfies a property defined in the given descriptive formalism.

This is generally called the *dual-language* approach, as it is based on the idea of describing the system using an operational language (like automata or timed automata), and the studied properties using a descriptive language (most frequently, a logic or a temporal logic).

The idea of model checking can be extended allowing a stochastic behavior in the system description, but this is done at a very high cost, as the model checking problem for such systems is undecidable. The approach proposed in the literature to overcome this limitation is called *statistical model checking* (SMC) [20]. The idea of SMC is to run a finite number of simulations of the system, and then use results from statistics to verify properties of the system within some interval of confidence. This classifies SMC between exhaustive model checking and testing. In fact, while on one side SMC it is not exhaustive as pure model checking, on the other side it is more expressive and requires by far less memory and time for computing.

UPPAAL is one of those tools. There are several works that present it both from a practical[2] and methodological point of view [19] [5]. The tool was extended to implement also statistical model checking, and such features are presented and explained in [10]. Finally, for what concerns the purely theoretical aspects and foundations of the model checking methodology, we refer to [12].

Systems in UPPAAL: the Operational Language

The modeling formalism implemented by UPPAAL SMC is a stochastic extension of the timed automata [1]. Timed automata extend the basic finite-state automata by introducing *clocks*, i.e., variables that generally evolve according to the differential equation $\dot{x} = 1$ where x is the value of the clock. Clocks may influence the behavior of automata (being part of the transition's conditions or state's invariants) and vice versa (transitions may reset or alter the value of clocks). This way, timed automata allow to extend the finite state automata formalism in order to capture their evolution over time.

A further extension of this formalism is the introduction of stochastic behavior for the systems involved. The stochastic extension implemented in UPPAAL replaces the two non-deterministic aspects of timed automata formalism with a probabilistic behavior. We now spend a couple of words on this important aspect.

[2] https://www.it.uu.se/research/group/darts/uppaal/small_tutorial.pdf

First, non-deterministic time delays that determine the permanence in a state are substituted by probabilistic distributions. Those distributions can be either uniform (for bounded intervals) or exponential (for unbounded intervals)[3]. For instance, in the automaton in Figure 4.1, once the condition A for transition activation is satisfied, there is no information on when the transition will actually be taken: it could be immediately or never. Instead, in the stochastic automaton in Figure 4.2, when condition A is satisfied the system waits for a time determined by an exponential probabilistic distribution with initial rate equal to 1. This way the time-delays instead of being non-deterministic are refined by probability distributions.

Secondly, when the automaton can chose between two possible transitions, it does that accordingly to an (apparently discrete) probabilistic distribution, instead of non-deterministically. The automaton in Figure 4.3, when condition A is satisfied, will non-deterministically take one of the two transitions and move to states *b* or *c*. If say 100 simulations are run, we cannot assume anything about how many times the system will move to state *b* or *c*. Instead, the automaton represented in Figure 4.4 will chose between states *b* and *c* according to a discrete probabilistic distribution where probability of moving to state *b* is 0.7 and probability of moving to *c* is 0.3 . Hence, if 100 simulations are run, the system will approximatively end 70 times in *b* and 30 in *c*.

We can use the first kind of statistical modeling to describe time delays that can be related to communication between nodes (i.e., network efficiency) and in general to all the phenomena that we cannot (or do not want) to model by means of first-principle equations. Instead the statistical choice between transitions could be used to model the choice between different possible DAGs for the same application, hence overcoming the necessity of knowing in advance the control flow of the application—see Chapter 3, Section 3.1.

Inside a model, the user can define more than one automaton, and each of them will evolve concurrently. UPPAAL allows to implement synchronization channels that impose simultaneous execution of transitions in different automata. For a transition with a channel to be executed also another process must be ready to execute a transition associated to the same channel. One transition must be marked as receiver and the other as sender. The updates (or in general the code) associated to those is executed first on the sender side, and then on the receiver side.

According to the so defined formalism, the system definition results in the end in a *network of stochastic timed automata* (NSTA).

Properties in UPPAAL: the Descriptive Language

UPPAAL allows to query models in different ways. The queries can regard either evaluation of expressions or satisfaction of properties. Expressions are user-defined functions that depend on clocks or variables in the system.

[3] This constrain is dictated by the possibility of using statistical results for analyzing the results. This limitation frequently appears when dealing with our kind of problems.

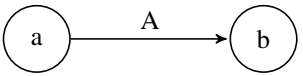

Figure 4.1 Non-deterministic Behavior: transition A can be taken or not at any time instant (e.g., it could be taken at time *i* or it could also never be activated).

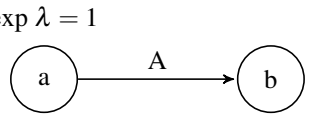

Figure 4.2 Stochastic Behavior: transition A is taken after a delay determined according to an exponential distribution with initial rate 1.

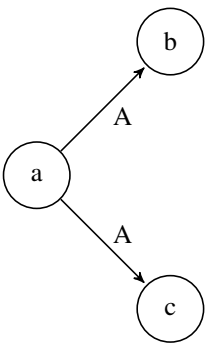

Figure 4.3 Non-deterministic Behavior: when A is satisfied, the choice of the transition to be taken is arbitrary and the system could reach either state b or c.

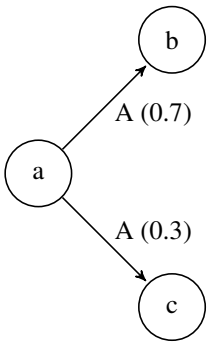

Figure 4.4 Stochastic Automaton: when A is satisfied and the automaton leaves state a, it goes with probability 0.7 to state b and with probability 0.3 to state c.

Properties are statements about the system expressed with an extension of the temporal logic, namely the Mixed Interval Temporal Logic (MITL) [2]. This logic allows to express properties over bounded domains for the clocks; this is not binding for our analysis, as we want to study applications that execute over a finite time. The available operators are the ones usually implemented in any propositional logic, like "*not*" $\neg$, "*and*" $\wedge$, "*or*" $\vee$ plus the temporal operators "*eventually in the future*" $\Diamond$ and "*always in the future*" $\Box$ with the usual quantifiers "*exists*" $\exists$ and "*for all*" $\forall$ [4].

The available queries for statistical model checking are the following.

- **Simulate the defined system** : this functionality makes the model evolve for a finite time, defined by the user, and visualizes the value of defined expressions. This is not a standard model checking query, but allows the programmer to get a feedback about whether the system behaves as he expected when he implemented it.

[4] Note that apparently this is neither an exhaustive nor rigorous definition of the logic implemented by UPPAAL, for such descriptions we refer to the cited works. Here we report the main concepts that give an intuition which kind of properties we can (and want) describe.

- **Probability estimation** : this query asks UPPAAL to produce an approximation interval $[p-\varepsilon, p+\varepsilon]$ for the probability p that a property ψ is verified. $p = P(\psi)$ with confidence $(1-\alpha)$, where ε and α are defined by the user. Hence, given a property, ψ UPPAAL evaluates the probability that ψ is verified.

- **Hypothesis testing** : this query asks UPPAAL to test the null-hypothesis $H : p = P_M(\psi) \geq \theta$ against the alternative hypothesis $K : p = P_M(\psi) < \theta$. In this case, the user is asking to test whether or not the probability that a property is verified is bounded by a certain value. UPPAAL asks the user to define also an execution bound M for the runs that are performed when testing the hypothesis[5].

- **Probability comparison** : this query simply asks UPPAAL to verify among two different properties which has the higher probability of occurring. Bounds must be provided on the simulated time of the queries.

- **Expected value** : this query asks UPPAAL to estimate the expected value for the minimum or the maximum of a given expression. This query requires the user to provide a number of runs to execute for the estimation and, like the previous queries, a bound on the simulated time.

Extension to Switching Systems

UPPAAL SMC has also been extended to allow the control of the progress rate of the clocks. In each state of the modeled automata, the user can impose the progress rate of the clocks using the variables defined inside the model, including the clock value itself, thus implementing ordinary differential equations (ODE). This is done inside the environment "*state invariants*", where the user can define progress rate of clocks and properties that the system must satisfy when in this state. In case those properties are not satisfied and the process cannot evolve without violating them, a run-time error will be generated. We use this functionality to implement the progress of the processing queue, and in general the differential equations inside the model.

4.2 Model Implementation

In this section we delve into the details of how we implemented the model presented in Chapter 3 in UPPAAL, for capturing the behavior of the xSpark framework presented in Section 3.3.

A model in UPPAAL is constituted by a set of concurrent processes defined as automata. For the definition of a model, three environments are available:

[5] This bound can be expressed in different ways; suffice here to state that it is generally a bound over the simulated time.

- An environment for global declarations accessible both in reading and writing by all the processes (data-types, constants, variables, clocks, channels and functions),
- A set template definitions of the automata, each with its own environment for local definitions accessible only by the automaton itself (constants, variables, clocks and functions),
- An environment for the instantiation of the automata (or equivalently of the processes).

Our model defines two automata templates: one for the single computational step, and another for implementing an observer that checks whether the application meets or misses the deadline. We now go through all the environments and templates.

Global Declarations

The global declarations include all the informations that are shared among the computational steps. Those include:

- The DAG of the application,
- The definition of the cluster on which the application runs,
- Control parameters of the framework,
- The communication channels that allow synchronization of the computational steps,
- A couple of commodity functions.

The DAG The DAG is defined using a matrix of booleans called `flow_diagram`. Each row and each column directly corresponds to a stage of the application, and vice versa: for instance, stage k corresponds to row k and column k. The (i, j) element of the matrix is true when stage j is a predecessor of stage i, false otherwise. Symmetrically, we can say that (i, j) is true when i is a successor of j. This implies that on the generic column j, true values evidence the successors of stage j, while on generic row i, true values evidence the predecessors of stage i.

Cluster and Allocated Resources The cluster is defined as a set of machines of given and fixed cardinality. Each machine has a fixed number of cores, and each core provides a certain (maximum) computational power; the processing rate that one may desire to govern is ten expressed as jobs (in the broadest sense of the term) per second per core. The resources, when allocated, are quantized according to the `machine_quantum` variable defined in this environment, as this is a potential control parameter of the overall application.

Allocated resourced are described as a vector of integers with as many elements as the number of stages—a choice that eases the UPPAAL operation, and is consistent with the inherently quantized nature of any resource. Specializing to a cluster, each element quantifies the percentage of machines allocated to a given stage aggregated with respect to the machines. This means that we cannot distinguish which machine is allocated to which stage. It is worth noticing that this would not be a meaningful distinction anyway, under the hypothesis of uniform resources and stages[6].

Application Level Control Parameters In the global declarations we find the control parameters related to the application-level control loop(s). Those include the overall application deadline and the α parameter, that implements the desired margin with respect to the completion of the overall application – as was defined in Chapter 3, Section 3.3. A boolean vector for keeping track of the completed nodes is implemented. It is used for the computation of the weight of each stage in the heuristic for stage deadline generation, eq. 3.4.

Channels A matrix of channels with dimensions equal to the number of stages on both row and columns is implemented. As channels impose simultaneous execution of transitions in different processes (automata), they come into play when synchronization between stages is required. Channel (i, j) is used when stage j (marked as sender) wants to communicate with stage i (marked as receiver), we will get back on this when describing the behavior of the single computational step in Sections 4.2 and 4.2.

Functions The commodity functions implemented here are two. One returns the number of missing stages, information required in the deadline loop that implements the heuristic shown in Chapter 3, Section 3.3. The second one receives as input a double and returns the smallest integer that is bigger than the input. This latter function is used when values of the kind of *double* are assigned to *int* variables.

Computational Step Template

The automaton template of the single stage implemented in UPPAAL is shown in Figure 4.5. As we can see, there is here one main difference with respect to the theoretical one presented in Figure 3.5: the transition leaving the `processing` state and defining a self-loop. This transition implements the resource allocator. In fact, resource allocation is activated only in this state of the stage automaton, and in xSpark the stage-level control of each computational step is independent of the controls of the other stages. For this reason there is no need for a parallel process representing it, and doing so would increase the complexity without implementing specific functionalities. Still, all the stages allocate resources from the same pool (the global

[6] This is a necessary hypothesis as we want the framework model to be agnostic with respect to this information. The resource allocation logic (i.e. the control) should guarantee adaptation with respect to those variabilities or, using control-theoretical terms, should reject those disturbances.

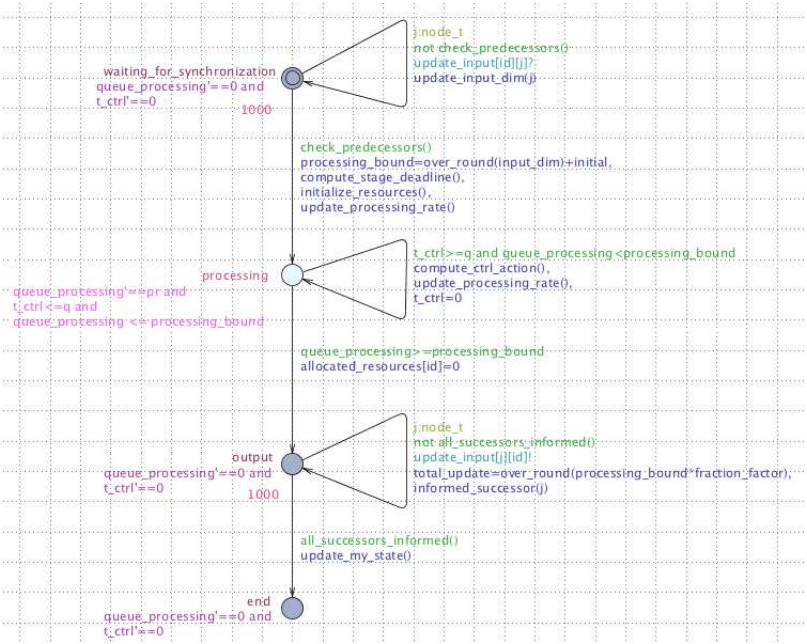

Figure 4.5 Stage automaton as implemented in UPPAAL.

variable), hence the representation of the "competition" among parallel stages for the resources is preserved.

Stage Declarations As we said, we can declare variables that live within the scope of the single process. This means that those variables are accessible only by the process they are associated to. Moreover when allocating the single stage we can define some input parameters for the process. For the stage template those are:

- The integer `id` that is a univocal identifier for the stage,
- The integer `initial` defining the volume of input data that do not come from predecessors[7],
- The `fraction_factor` defining the ratio between volume of input data and output data[8],

[7] This is used for the starting stages, that do not receive the data from other stages but process the raw input of the application.

[8] This depends on the actual data and operations that are being executed so we cannot provide a general model for it. Here we define it as an input in order to have the same data volumes of the application executions of which we have data. In the general case where we do not have this information; it can

- The weight parameter `w` corresponding to the $w(s_k)$ parameter defined in Chapter 3, eq. 3.3.

In our model the variables of the stage template can be divided in two groups: one comprising the variables that define the state of the stage computation, related to the progress of processing itself, the second with the variables related to the control, hence defined by the framework.

In the first group we find:

- The variables `input_dim` and `processing_bound` that define the volume of the data set to process,
- Two boolean vectors `received_predecessors` and `inofrmed_successors` that respectively keep track of which predecessors have synchronized at the beginning of the computation and which successors have synchronized at the end,
- The clock `queue_processing` that is the "core" variable of the stage: it describes the amount of processed data (i.e. the e variable in eq. 3.1 defined in Section 3.2),
- The constant `pr_pole` defining the pole of the resource allocation dynamics described in Chapter 3, eq. 3.5.

Instead the variables related to the control are:

- The constants `gamma` and `beta` implementing respectively the γ parameter defined in Chapter 3, Section 3.3 and the β parameter used in eq. 3.4,
- The stage `deadline` and the `weight` parameter (corresponding to the $weight(s_k)$ used in eq. 3.2) both computed at the beginning of the processing,
- The variable `reference_rate` computed according to the deadline and γ parameters,
- The variable `start_time` used for storing the time at which the processing of the stage started,
- The variable `ci_old` implementing the state variable of the controller,
- The clock `t_ctrl` and constant `q` used for implementing the control time step (the constant corresponds to the q defined in Chapter 3, Section 3.3),

be statistically modeled by substituting a parameter computed at the end of the processing according to some probabilistic distribution.

- The variables `estimated_cores`, `stage_saturation` and the constant `overscale` used by xSpark for bounding the total amount of resources available for the single stage.

We now go through each of the states and transitions of the template.

Waiting for Synchronization This state implements the state described in Section 3.2. Here no data processing and no control is happening, so the rate of the clocks `t_ctrl` and `queue_processing` are fixed at zero (defined in the invariants of the state).

In this state the automaton may activate two transitions with mutually exclusive conditions. One of those is described by the self-loop for synchronizing with the predecessors, it is in fact associated to the channel `update_input[id][j]` and can be taken when condition `not check_predecessors()` is valid. The function `check_predecessors()` returns a boolean value that is *true* when all the predecessors have terminated and synchronized with the present node, *false* otherwise[9]. When the transition is taken, the function `update_input_dim()` is executed: this function updates the variable `received_predecessors` and increments the variable `input_dim` reading the value of the global meta-variable[10] `total_update`. The variable was in fact first updated by the predecessor (see Section 4.2), exploiting the fact that the transition code is first executed by the sender and then by the receiver.

When all the predecessors have terminated and synchronized with the given stage, the output of function `check_predecessors()` returns true, hence the automaton may take the transition to the Processing state. When taking this transitions different functions are executed. The `processing_bound` variable is updated summing the input dimension received from the predecessors and the input of the template `initial`. Function `compute_stage_deadline()` computes the stage deadline, the reference rate, the estimated cores and the stage saturation: in general we can say that this function initializes the control of the stage. Function `initialize_resources` initializes the allocated resources to the node according to the estimated required cores (also the state of the controller is initialized accordingly). Finally, the function `update_processing_rate()` computes the processing rate reading the allocated resources and implementing eq. 3.5.

Processing As defined in Section 3.2, in this state the actual processing of data is modeled. In fact the rate of the clock `queue_processing` is set equal to variable `pr` that was previously initialized by the function `update_processing_rate()`.

[9] The function simply compares the vector variable `received_predecessors` defined before with the predecessors defined by the DAG.

[10] A meta-variable is a variable that is not part of the state hence the programmer cannot rely on it having the same value across the execution of different transitions. When possible it is used to reduce the state-space of the implemented system and improve efficiency of model checking.

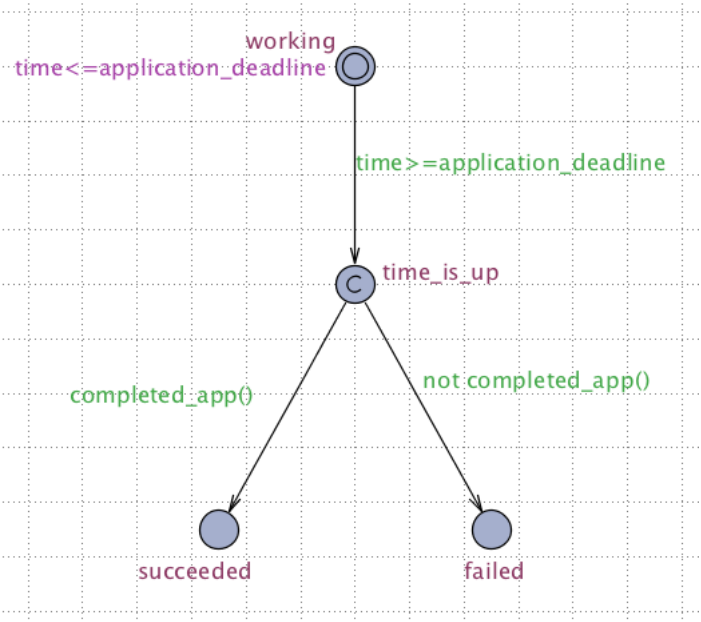

Figure 4.6 Test automata for evaluation of deadline satisfaction.

Two other invariants are implemented in this state: one defining an upper-bound for the `queue_processing` clock so that as soon as it reaches the threshold the automaton is forced to leave the processing state and the other constraining the `t_ctrl` clock, thus forcing the automaton to activate the resource allocation transition (the self-loop) at every control-step.

The two transitions leaving the processing state have mutually excluding conditions, according to whether the processing is terminated or not. In fact one models the control action performed at every control step during the processing and the other is taken only when the processing has been completed.

The resource allocation transition when taken, apart from resetting the `t_ctrl` variable, executes two functions: `compute_ctrl_action()` that computes and performs the control action (i.e. allocates the resources) and the function `update_processing_rate()` that we already defined in previous sections. The computation of the control action is performed through the implementation of eq. 3.15 and 3.16 and the discretization of the control action according to the defined core quantum (performed using function `discretize_core_allocation()`).

When instead the processing of the data is finished – corresponding to the condition `queue_processing>=processing_bound` – the automaton takes the transition to the next state. While doing so, the allocated resources are released.

Output This state is the dual of the initial one. Here the process is ready to synchronize and communicate to the successors its output data. This is done executing the transition that defines a self-loop synchronized with the one in the initial state executed by the successor.

The value assigned to the `total_update` variable – that will be read by the successor – is the `processing_bound` times the `fraction_factor` this defines the volume of output data. This is done until the function `all_successors_informed()` that compares the vectors of successors (taken from the matrix defining the DAG) and the vector of informed successors returns *true*. At this point the other transition to the `end` state is taken and the stage updates its execution state in the global variable `completed_nodes`.

Deadline Observer

A common way to define properties in model checking, is the usage of *test-automata* [6]. This idea is mainly motivated by the desire of avoiding expressing properties of the systems with the logic, and instead use simpler automata that evidently enjoy the said properties. Doing so is not by itself more powerful than using the logic (in the sense that one cannot express more properties), but can be easier to understand for the user. More precisely, the approach is to define an automaton inside the model and make it behave in such a way that it detects (in the sense that it changes its state) the occurrence of specific events or sequences of events. Then the query can be easily expressed on the final state of the defined automaton. Properties expressed this way frequently are of the kind of bounded liveness, but this is not exclusive.

In this work, we use this idea for checking whether an application meets or misses the deadline. This is a trivial task, as the deadline is defined as a global variable and it is therefore accessible from all processes, the deadline observer included. This simple automaton is shown in Figure 4.6. The automaton does nothing until the time reaches the deadline, at this moment the invariant `time < application_deadline` of the initial state is not valid anymore, and the process is forced to leave the state. This cannot happen earlier, as the transition that leaves this state has the condition `time >= application_deadline`. The next state, the one called `time_is_up` is marked as *committed*, this means that when in this state the process is forced to leave it without any delay. The two transitions leaving this intermediate state have mutually excluding conditions about whether the application has finished running or not. In the first case the application ends in state `succeeded`, otherwise in state `failed`.

Using this automaton, we can simply ask the model checker about the probability of runs such that the deadline observer ends in the state `failed` and directly retrieve the probability the application meeting the deadline or not.

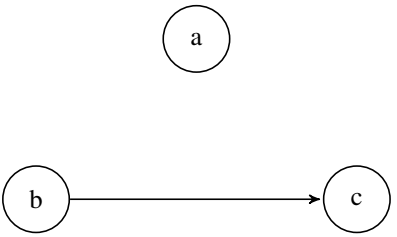

Figure 4.7 Sort by Key: execution DAG

4.3 Model Validation

We present a model validation example using the data from an execution of the application *sort by key* in the framework xSpark. The application simply performs an ordering of the input dataset, hence the volume of data remains constant through the execution – fraction factors of all stages are unitary. Specifically the processing consists in 3 stages composed in a DAG like the one in Figure 4.7. Recall that even though the execution of stage a could be in parallel with the sequential execution of stages b and c, the actual implementation of xSpark performs the stages sequentially, specifically in the order a, b, c.

We recall that we are not interested in what the application actually does or what the data represent. Our aim is to use the DAG abstraction of the application, the volume of data and some parameters describing the cluster on which it will be run to verify properties about the execution time and the resource allocation.

The parameters needed for the implementation of the application in our model are:

- The volume of input data (number of records),
- The DAG,
- The fraction factor of each node,
- All the xSpark control parameters presented in section 3.3,
- The control step,
- The resource quantization,
- The dimensions of the used cluster (number of machines, number of cores per machine),
- A statistical estimation of the core power (i.e. the number of records per second processed on average by a single core)
- The pole of equation 3.5 representing the dynamics of allocated resources.

The data available from the real execution include all the necessary information apart from the last two in the list. So we arbitrarily set the pole at $p = 0.5$, meaning that after one time step the cluster has allocated half of the required resources and approximately estimated the processing power per core from the available plots as described in the next section.

Estimation of Core Power We first underline the fact that missing precise information about this parameter does not affect the validity of our analysis, and there are two reasons for this. One is that we want to be transversal to it, we aim at verifying properties irrespectively to which data are going to be processed and to which operations will be performed on them – those information are in fact captured by the processing power. The second (intertwined) reason is that this is achievable thanks to the control-theoretical approach that was used during the design of the xSpark framework.

The parameter we need to give a value to, is the number of records processed per second by one core. We will do this using the available information shown in the plot in Figure 4.8. Each stage processes 500'000'000 records – as we said this number remains constant along the execution – the first one does it in 37 seconds allocating on average 20% of the available resources. The total available resources are 4 machines with 16 cores each, hence the average allocated cores are around 12.8 . Dividing the number of records by the number of cores and the execution time, we obtain about 1050'000 records per second per core. Applying an analogous reasoning to the third stage that apparently is the heavier from the computational point of view we have instead 500'000'000 records with an average of 35% of the resources allocated processed in 85 seconds resulting in an average of 250'000 records processed per second by a core.

In our model we implement the processing power as a random variable uniformly distributed between 250'000 and 1'050'000. We also re-compute it at every control-step to include the fact that this parameter may vary in time.

This analysis allows us by the way to appreciate the variety of processing performances of the single stage, that the control of the application has to face even in the case of such a simple application.

Validation

We first show how our model captures the dynamics of the real system without implementing the stochastic characterization of the processing power, performing ten simulations. As we can see in Figure 4.9 and 4.10, where the allocated resources and the processing progress are respectively plotted, the model captures the (in this case linear) behavior of the system. Moreover, the different simulations overlap due to the deterministic nature of the implemented model.

We now run 50 simulations with randomized value for the processing power; results are plotted in Figure 4.11 and 4.12. The plot now looks like a "cloud" of possible executions of the application, that contains the actual one. Specifically, we

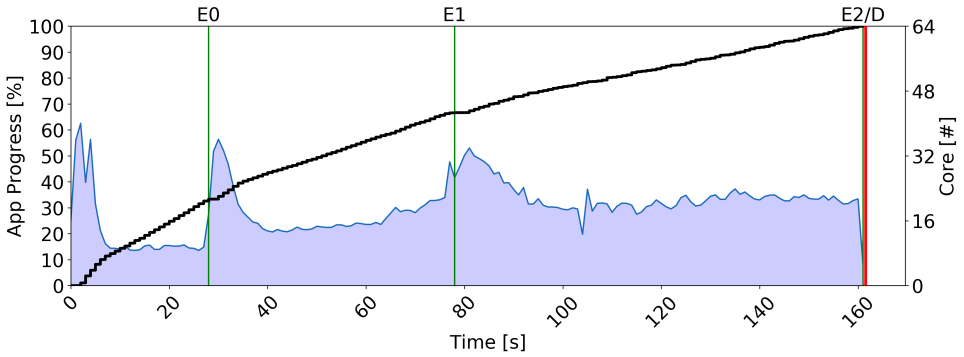

Figure 4.8 Execution progress and resource allocation profiles in the execution of sort by key application.

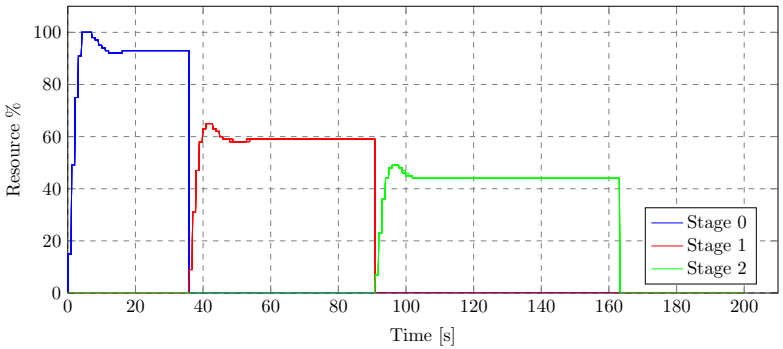

Figure 4.9 Non stochastic simulation of the implemented model: plot of allocated resources.

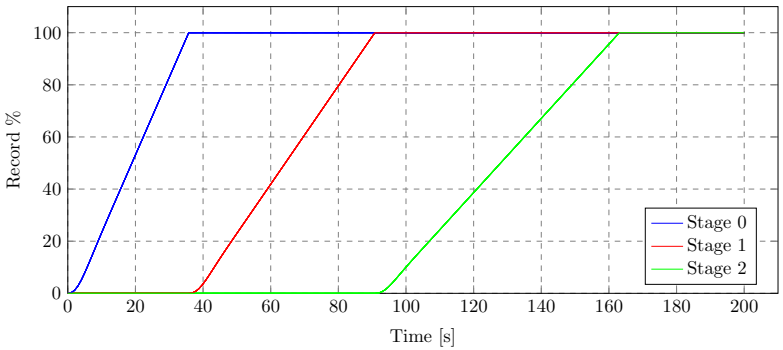

Figure 4.10 Non stochastic simulation of the implemented model: plot processing progress per stage.

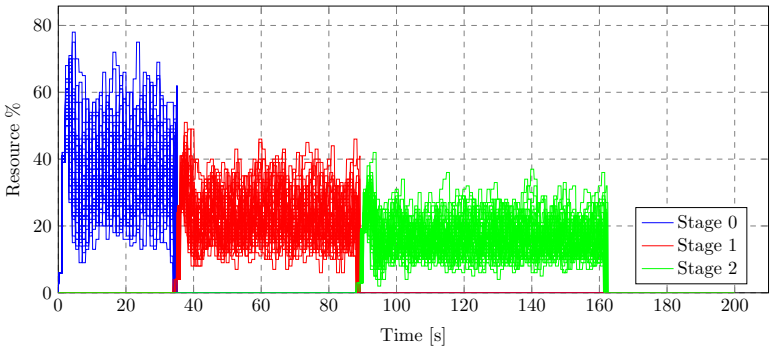

Figure 4.11 Simulation of the implemented model, including the stochastic modeling of the processing power: plot of allocated resources.

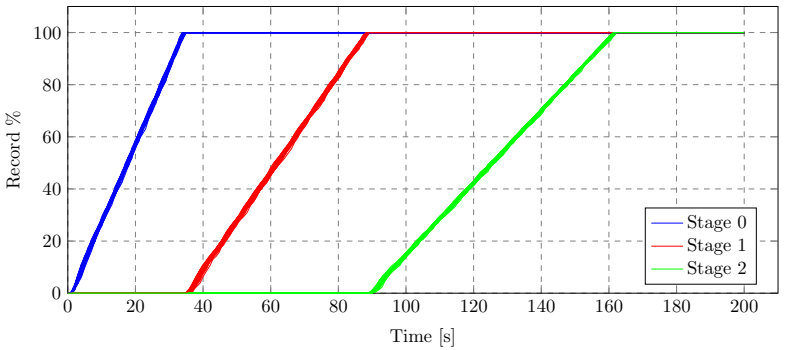

Figure 4.12 Simulation of the implemented model, including the stochastic modeling of the processing power: plot of processing progress per stage.

can appreciate how the control of the application rejects the disturbance introduced by the stochastic variation of core power, making the application terminate always in a neighborhood of the prescribed deadline.

Examples of Queries

In this section we investigate a couple of examples of queries that can be performed on the model. This is immediate thanks to the "push a button" nature of model checking; now that the model is implemented, we only need to formulate the queries in the prescribed descriptive language, and ask the software tool whether the system satisfies them or not.

The most immediate property to ask is the probability of failing the deadline. This is expressed in temporal logic (the logic was defined in Section 4.1) as:

$$Pr[<= 200](<> dead\_obs.failed). \tag{4.1}$$

Figure 4.13 Result of the query about the probability of the application missing the deadline.

Figure 4.14 Result of the query about the maximum expected value for the allocated resources.

This query asks for the probability that the deadline observer at a certain time during the execution reaches the state `failed`. The executions are bounded to last exactly 200 simulated seconds, as the deadline is set at 170 sec this does not affect the outcome of the analysis. The result is shown in Figure 4.13. As the cluster is well dimensioned and the control well tuned, the tool returns an almost null probability for this event.

A more interesting query could be the maximum expected value for the total amount of allocated resources. This is expressed with the query:

$$E[<=200;60](max : sum(i : node\_t)allocated\_resources[i]), \tag{4.2}$$

meaning that we ask to perform 60 simulation of the system lasting 200 seconds and provide an upper bound estimation for the amount of cores allocated to all the stages. The result is shown in Figure 4.14. The maximum expected core allocation is a little bit more than 40 cores over the 64 available allowing us to state for example that on average there will always be around 20 cores available for running other applications.

5

Model Extensions

The model introduced in Chapter 3 was shown to be able of reproducing application data precisely enough, and therefore to be suitable for verifying the properties of interest via model checking applied to the simulator of an application instead of the application itself.

However, although the model fullfils the goal set for this thesis, it also contains some quite relevant simplifications, that it may be interesting to relax in a view to applying the devised modeling principles to a more fine-grained analysis. The purpose of this chapter is to propose an extended model to overcome the said simplifications.

5.1 Limits to Overcome

The limitations of the model proposed in Chapter 3 – hereinafter the *basic* model, as opposed to the *extended* one presented in the following – are basically two.

- The operation of a stage is represented monolithically, without describing its partition into the elaboration and the reading/writing of data.
- The processing of data proceeds at a speed proportional (among other factors some of which unpredictable) to the allocated resources, but this fact is represented as if there were only one elaboration unit with more or less resources, not evidencing the presence of several units possibly operating in parallel.

To avoid possible confusions, it is worth specifying that in fact the basic model does represent parallelism, but only at the level of the DAG, i.e., in how the program realizing the application is written. The level of parallelism related to the single operations, instead, is not modeled, since – adopting for a moment the Spark jargon – there is no evidence of the *executors* spawned in parallel, on different cores, to carry out a stage.

Indeed, for several analysis activities, the basic model is more than enough. For example, when aiming at the first-cut sizing of an elaboration architecture, one may

not yet have precise enough an idea of how many units will be required, and thus desire to obtain that number exactly from the simulation of a model in the basic form. Or, ideas may not be yet clear on the communication and storage part of the architecture, and therefore one may decide to just give a rough quantification of the consequent time overhead, and include it in that of the processing time.

The basic model, on the other hand, clearly reveals its shortcomings when the purpose of the simulation and checking activity is not sizing an architecture, which has to be done in a cautious manner anyway, but rather providing reliable and precise enough estimates – possibly based also on the profiling of convenient benchmark applications – of the data processing times really observed at runtime.

5.2 The Extended Model

In the basic model a stage is viewed as a single entity that has to process a given amount of data. The processing speed is governed by allotting this entity more or less elaboration units, which is viewed as equivalently providing a single more or less powerful one. There is no evidence of the physical means by which this is realized.

In the extended model, when a stage starts, the data to be processed is split into a number of subsets. The processing of one of these subsets, that can take place in parallel with others, is called a *task*. The processing speed of the stage is governed by spawning more or less *executors*, that can be thought here as sequential elaboration units taking care of one task, thereby allowing more or less tasks to run in parallel.

In both cases the number of available elaboration units is clearly limited, as the cluster has a finite number of cores, whence the allocation problem. However, the extended model can for example describe the situation in which some executors come to transiently reducing their operating speed, which is apparently impossible for the basic one.

Coming to the breakdown of the data elaboration, in the basic model there is no "data transfer" if not from a stage to the downstream ones—in other words, only when locality is violated. In the extended model we conversely need to account for the time spent in attributing their subsets to the stage tasks. We therefore separate the reading of data from its elaboration, describing the former as a delay. The same is not required for the writing operation, as this takes place contextually with the elaboration itself. In fact, as soon as part of the output is produced, this is directly written to the application file system, which affects the overall processing time only to a nearly negligible extent.

For the sake of completeness, one may argue that even though the writing procedure itself does not require a relevant time with respect to the overall computation time, it may load the network, and thus indirectly affect the reading time. However, in this respect, a couple of factors must be considered: first, differently from the

reading procedure, the writing one can be forced to take place on a cluster node close to the one that processes the data or even on the same node, resulting in a marginal load for the network: second, the volume of output data is usually smaller than the volume of input data—at least, on an average basis.

Despite the remarks just made. one could still not desire to neglect the writing time. The extended model can be very easily adapted to allow this, but the matter is left to future works.

The extended model is composed of a state machine, representing the operating states traversed by the elaboration units. This state machine governs the evolution of a continuous-time dynamic system describing the units themselves. Differently from the basic model, as the number of active units can vary, the order of the continuous-time part of the model can vary as well, making the extended model a hybrid – not switching – dynamic system.

5.3 Implementation of the Extended Model

The extended model, with respect the basic one, changes the way the processing of data is described. For this reason the differences between the two are confined to how the processing of data is captured.

The model extension includes the deploying of tasks that process the data in parallel. To implement this, we use an UPPAAL functionality that allows to dynamic allocate (or using UPPAAL jargon, *spawn*) of processes within the model, hence to capture an hybrid system (as it does not require to know in advance how many processes will be launched).

From the implementation point of view, there are two main differences: the behavior of the stage in the processing state is different, and the dynamic template of the executors is introduced. We now present the introduced features.

Stage Implementation Differences Figures 5.1 and 5.2 show the stage automaton in the extended model: as we can see, the behavior differs from the basic model only in the processing state. When in this state, if the processing is not terminated, it takes a transition at every control step – analogously to the basic model – and moves to another state named `spawn tasks`. When taking this transition the control action – i.e., the number of resources (executors) to allocate – is computed: for reasons explained in the next section, in this case we allocate only a constant number of those. The update of the `pr` variable instead is implemented here to make this model comparable to the basic one, the way it affects the processing will be shown when describing the executor template. The state spawn executors is marked as *committed* that means that when in this state no delay is allowed and the system must immediately take a transition exiting the state. In this state the automaton cycles on the self-loop transition and once at a time allocates an executor. This is done until either the overall resources are saturated or the number of executors matches the number requested by the controller. Then, the process takes the transition back to

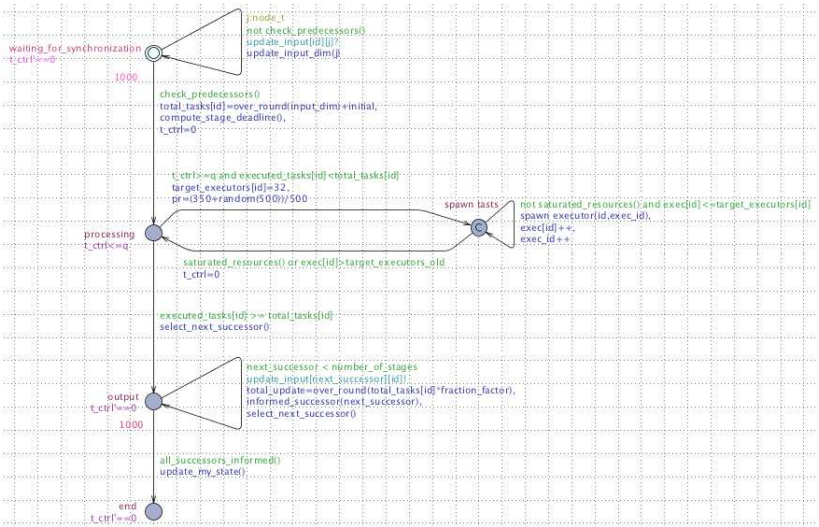

Figure 5.1 Stage automaton of the extended model implemented in UPPAAL

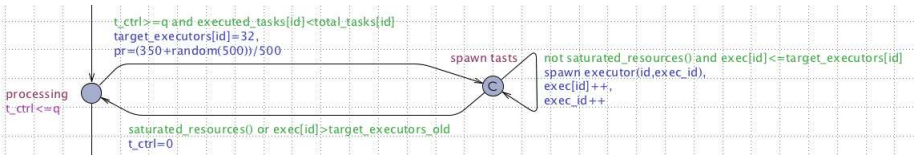

Figure 5.2 Zoom on the processing phase in the stage automaton of the extended model

the processing state. When eventually a controller is implemented, the anti wind-up procedure must be performed in this transition, as only at this moment the actual amount of allocated resources is known.

Executor Template The implementation of the template of an executor is shown in figure 5.3. This automaton is supposed to capture the sequence of procedures that are involved in the processing of a single task. When spawned, the automaton is in a initial "committed" state, which it is forced to leave immediately. The automaton hence moves to the reading state, using a transition that computes the reading delay—i.e., the time it will have to wait before being able to move to the processing state. As at the moment we do not have any model for the reading delay, hence here we made the hypothesis of a delay that is constituted by a constant term plus a linear function of the number of deployed executors. This idea should be verified using real data from adequately instrumented applications, or by trying to (finely) model the "inner-physics" of the network. When the process leaves the reading state

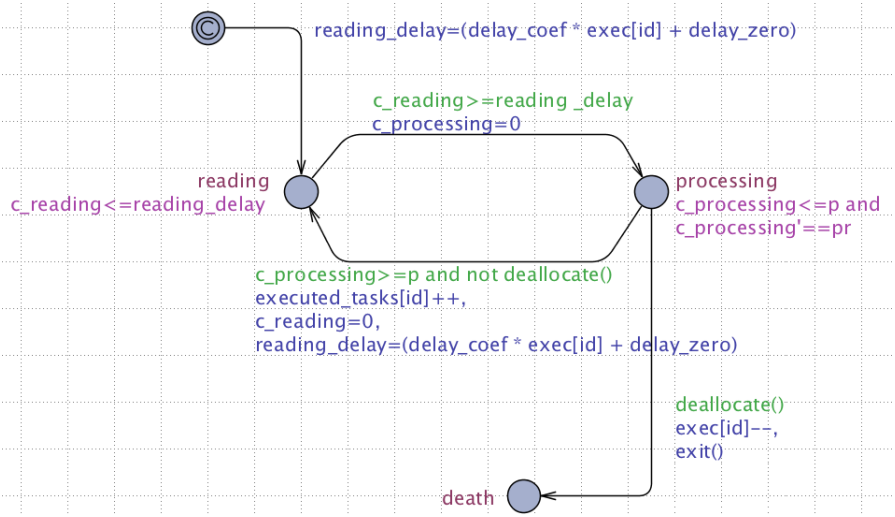

Figure 5.3 Executor Template implemented in UPPAAL

it moves to the one modeling the processing. The processing delay is implemented in a slightly different manner: the threshold is constant and the clock rate varies. The rate is determined at every control step as it was in the basic model in order to achieve comparability between the two. Also the variability is properly chosen to emulate the previous one, how this is done is described in section 5.4. Once the processing is completed, the executor increases by one the number of overall completed tasks in the stage, and start to process another task, if instead before this happens the amount of allocated resources is reduced the executor deallocates itself using function `exit()`.

We chose to implement only the reading and writing procedures, but we can see that the model could be extended introducing for instance another state capturing other procedures.

5.4 Testing the Extended Model

In this section we briefly present two tests. The first one is aimed at showing that the extended model, although different from the basic model as for its implementation, does in fact comprehend the simpler model. This is achieved by simulating an application with the basic model and then with the extended one, but allowing for only one elaboration unit in parallel, and setting the data reading delay to zero. Consistency of the two models is verified as for both the execution traces, and the probabilistic assessment of a property.

The second test is conversely aimed at evidencing the additional representation

capabilities of the extended model. This is obtained by starting from the same situation just described, and showing how the property assessment is affected by the introduction of a data reading delay.

Since the aim of both test is to assess the descriptive capabilities of the models, no control is involved—hence we "disable" the controller and impose a constant amount of allocated resources.

Test one – Consistency

The aim of this test is to verify that the extended model discussed in Section 5.2 is able to capture the physical behavior described by the basic model presented in Chapter 3. We achieve this objective by presenting one simulation with equivalent parameters for both models—the core power for the basic model (as defined in section 4.3), and the equivalent processing time per task in the extended model. Our results show that, despite the more complex dynamics introduced in the extended model and the uncertainty encoded in the mentioned parameters, the results in terms of execution times are comparable. In the extended model, reading delays are set to zero, in order to match the basic model behavior.

Our simulations involve the same application presented in 4.3, composed of three stages, each with a fraction factor of 1, i.e., preserving the total amount of records to process. The application is given 500M input records that have to be processed by the three stages sequentially. Both in the extended model and in the basic one, the application is given 32 cores. This is simply a number for the basic model, while in the extended model case, having access to 32 cores implies that the application can run 32 tasks in parallel. Each of the tasks of the extended model is given 500K records to process, therefore a total number of 1000 tasks is to be run for completion.

For each second of the simulation, in both models, the stochastic parameters are updated from a uniform random distribution. In the case of the basic model, the core power (the amount of record processed in a second per core) is uniformly distributed between 350 and 850. For the extended model, the stochastic behavior is encoded as the number of tasks completed per second per core. To obtain the same random distribution, we therefore utilize the same random variable parameters and divide them by the number of records per task (500K).

$$\left[\frac{record}{s \cdot core}\right] \cdot \left[\frac{task}{record}\right] = \left[\frac{task}{s \cdot core}\right] \tag{5.1}$$

Figures 5.4 and 5.5 show respectively the results of 50 different simulations of the basic and extended models. As can be seen, the completion times of first stage are almost identical, the range being between 25 and 30 seconds. For the second and third stage, differences become slightly more evident – justifying to some extent the necessity of a model that includes more details providing results of finer granularity. Nevertheless, it is worth noticing that the basic model, despite

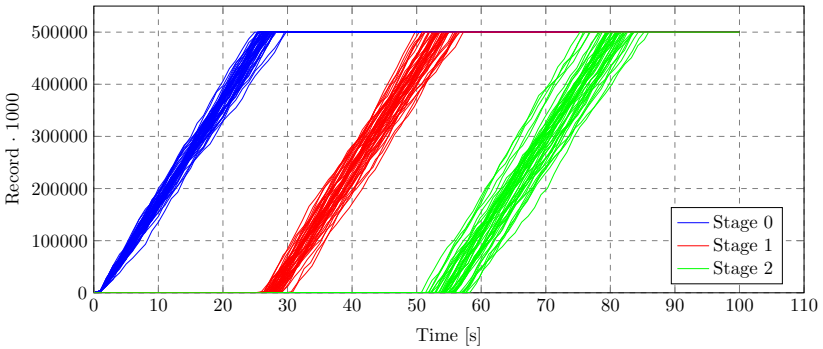

Figure 5.4 Basic model, no delay.

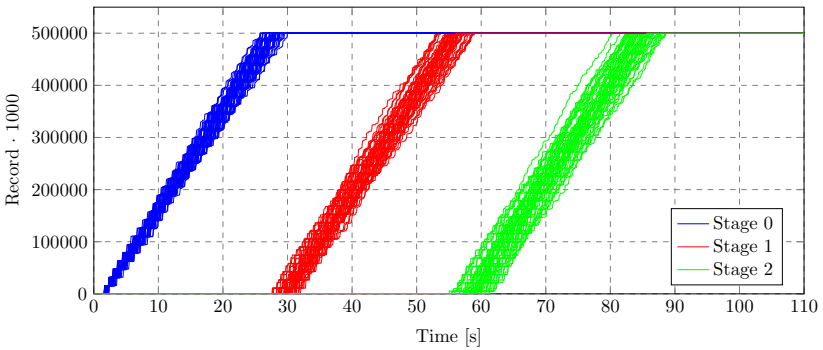

Figure 5.5 Extended model, no delay.

being notably simpler, catches the results of the extended one by far less than an order of magnitude, therefore proving to be well suitable for a first cut evaluation.

For example, observing Figure 5.5 and in particular the curves for the beginning of the stage 1 execution and the progress rate of stage 2, one can notice the quantization of the progress of the computation. This models very well the availability of measurement and information within the big data framework – it is in fact impossible to properly be informed about the progress of the execution if not for the completion time of tasks. This is also evident in the execution traces from the run shown in Figure 4.8. The extended model better capture the nature of the feedback signal, therefore being more suited for fine grain evaluation of the performance of the big data framework together with the computation infrastructure.

Test two – Extension

The aim of this second test is to show the characteristics of the extended model (and of the execution on the real hardware) that are not captured by the basic model. We achieve this by setting a nonzero reading delay and verifying the performance degra-

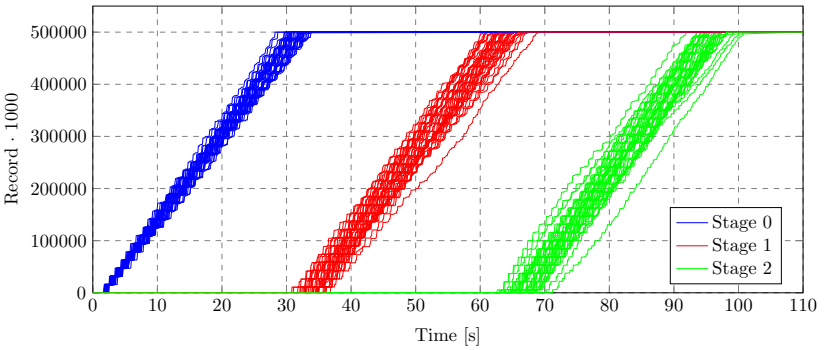

Figure 5.6 Extended model, delay

dation (longer execution times). Specifically, the model factors in a constant delay for the setup of the reading channel and a delay that is proportional to the number of the running tasks. This second term allows the extended model to capture an increase in network load – which results in lower reading speed – due to concurrency. All the other parameters (computing speed, allocated resources, etc.) are kept constant. Figure 5.6 shows the simulations obtained with the extended models and a delay of $0.1s$ for the channel setup and an additional delay of $0.0001s$ per running task. The figure displays, in fact, longer execution times for each of the stages and therefore a longer completion time for the overall application.

6

Conclusion and Future Work

The aim of this work was to provide a first-principle modeling approach to big data applications and frameworks. The main motivation for this is to make model checking feasible in the addressed context. Specifically, once the model is proven able to represent the characteristics of the evolution of a big data application informatively enough, it is possible to carry out model checking operation on the simulator of the application instead of the application itself. Since the former is far less complex and computationally intensive, the advantages are apparent. Indeed, important properties can be assessed at a very small fraction of the cost, and model checking can be considered even for applications that in the absence of a suitable simulation model would simply be intractable.

Once the model has been proposed, including some relevant extensions to the basic principles, and tests have been presented to demonstrate its adequacy to the intended purpose, it is now the time to take a wider perspective.

In fact, this thesis is part of a long term research project, the major aim of which is not only to model and simulate the existing technology, but ultimately to devise and realize a brand new generation of big data frameworks, where the system- and control-theoretical attitude here just introduced is brought to guide the entire design. As a result, several research directions are open, that can be broadly classified in three sets:

- Application modeling,
- Architecture/hardware modeling,
- Framework design and realization.

The first activity set is basically the continuation of the analysis carried out in this work, in a view to exploiting the devised abstractions to improve the software architecture of the applications.

The second set is conversely a step ahead with respect to this work. Once the software-related abstractions are assessed, the idea is to extend the modeling strategy to the hardware that executes the application, determining and adopting the convenient level of descriptive detail. This is surely an ambitious objective, as the involved architectures are typically cloud-based.

The third set, finally, amounts to turn methodological results on modeling, simulation and control, into engineering practices for the control-based design of big data frameworks.

Despite the activities just mentioned are nowadays largely to be defined, the experience gathered in this work allows to spend some final words at least on the first two, thereby sketching out, with the detail feasible to date, future activities in this context.

6.1 Application Modeling

From this viewpoint, the first problem to address is how to generate the DAG from the code of an application automatically, and for different frameworks. On one hand this has an operational importance, as it avoids to input the same information twice and reduces the possibility of modeling errors. On the other hand, and more important, the solution of this problem will provide further support to the generality and the descriptive capabilities of the proposed modeling paradigm.

A second relevant issue is whether or not, and in the affirmative case how, it is possible to manipulate a DAG so as to generate another one, functionally equivalent but with some optimality properties. Besides its obvious usefulness, a solution to this problem would inherently allow to define a "theoretically optimal" performance for the application, possibly – and hopefully – independent of the dataset and the cluster.

As a result of addressing the points above, one should be able to define a programming model (or extend existing ones) so that the DAG can be generated by the compiler instead of by the programmer (in an implicit way), and automatically completed with the information required to set up and run its simulation (at both the basic and the extended level) and model checking.

6.2 Architecture/Hardware Modeling

This activity will aim at completing the description of the application with that of the hosting computing machinery (especially in terms of network access and data management), in the broadest sense of the term. A significant effort will be required to relate this work to the huge existing *corpus* of computing systems modeling, fulfilling however the specific needs – evidenced in the previous chapters – of the big data context.

The expected results is that quantities relative to the file system, the network and the computing nodes, instead of being provided – normally in a conservative manner – by the user, can be obtained automatically. Solving this problem would make it possible to rapidly and systematically evaluate an application and the hosting architecture at the same time, and therefore to pave the way toward the realization of new big data frameworks that take advantage *right form the start of their design* of the control-theoretical approach adopted here.

Bibliography

[1] R. Alur and D. L. Dill. “A theory of timed automata”. *Theor. Comput. Sci.* **126**:2 (1994), pp. 183–235. ISSN: 0304-3975. DOI: 10.1016/0304-3975(94)90010-8. URL: http://dx.doi.org/10.1016/0304-3975(94)90010-8.

[2] R. Alur, T. Feder, and T. A. Henzinger. “The benefits of relaxing punctuality”. *J. ACM* **43**:1 (1996), pp. 116–146. ISSN: 0004-5411. DOI: 10.1145/227595.227602. URL: http://doi.acm.org/10.1145/227595.227602.

[3] K.-E. Årzén, A. Robertsson, D. Henriksson, M. Johansson, H. Hjalmarsson, and K. H. Johansson. “Conclusions of the artist2 roadmap on control of computing systems”. *SIGBED Rev.* **3**:3 (2006), pp. 11–20. ISSN: 1551-3688. DOI: 10.1145/1164050.1164053. URL: http://doi.acm.org/10.1145/1164050.1164053.

[4] L. Baresi, S. Guinea, A. Leva, and G. Quattrocchi. “A discrete-time feedback controller for containerized cloud applications”. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2016. ACM, Seattle, WA, USA, 2016, pp. 217–228. ISBN: 978-1-4503-4218-6. DOI: 10.1145/2950290.2950328. URL: http://doi.acm.org/10.1145/2950290.2950328.

[5] G. Behrmann, A. David, and K. G. Larsen. “A tutorial on uppaal”. In: M. Bernardo et al. (Eds.). *Formal Methods for the Design of Real-Time Systems: International School on Formal Methods for the Design of Computer, Communication, and Software Systems, Bertinora, Italy, September 13-18, 2004, Revised Lectures*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, pp. 200–236.

[6] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. “Uppaal—a tool suite for automatic verification of real-time systems”. In: *Proceedings of the DIMACS/SYCON Workshop on Hybrid Systems III : Verification and Control: Verification and Control*. Springer-Verlag New York,

Inc., New Brunswick, NeW Jersey, USA, 1996, pp. 232–243. ISBN: 3-540-61155-X. URL: http://dl.acm.org/citation.cfm?id=239587.239611.

[7] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica. "Hyracks: a flexible and extensible foundation for data-intensive computing". In: *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*. ICDE '11. IEEE Computer Society, Washington, DC, USA, 2011, pp. 1151–1162. ISBN: 978-1-4244-8959-6. DOI: 10.1109/ICDE.2011.5767921. URL: http://dx.doi.org/10.1109/ICDE.2011.5767921.

[8] T. C. Bressoud and Q. Tang. "Analysis, modeling, and simulation of hadoop YARN mapreduce". In: *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)*. 2016, pp. 980–988. DOI: 10.1109/ICPADS.2016.0131.

[9] Clarke, Grumberg, and Peled. *Model Checking*. MIT Press, Cambridge, MA, 2000.

[10] A. David, K. G. Larsen, A. Legay, M. Mikuăionis, and D. B. Poulsen. "Uppaal smc tutorial". *Int. J. Softw. Tools Technol. Transf.* **17**:4 (2015), pp. 397–415. ISSN: 1433-2779.

[11] J. Dean and S. Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*. OSDI'04. USENIX Association, San Francisco, CA, 2004, pp. 10–10. URL: http://dl.acm.org/citation.cfm?id=1251254.1251264.

[12] C. A. Furia, D. Mandrioli, A. Morzenti, and M. Rossi. *Modeling Time in Computing*. Springer Publishing Company, Incorporated, 2012. ISBN: 3642323316, 9783642323317.

[13] S. Ghemawat, H. Gobioff, and S.-T. Leung. "The google file system". In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. SOSP '03. ACM, Bolton Landing, NY, USA, 2003, pp. 29–43. ISBN: 1-58113-757-5. DOI: 10.1145/945445.945450. URL: http://doi.acm.org/10.1145/945445.945450.

[14] G. P. Gibilisco, M. Li, L. Zhang, and D. Ardagna. "Stage aware performance modeling of dag based in memory analytic platforms". In: *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*. 2016, pp. 188–195. DOI: 10.1109/CLOUD.2016.0034.

[15] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. "Dryad: distributed data-parallel programs from sequential building blocks". In: *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. EuroSys '07. ACM, Lisbon, Portugal, 2007, pp. 59–72. ISBN: 978-1-59593-636-3. DOI: 10.1145/1272996.1273005. URL: http://doi.acm.org/10.1145/1272996.1273005.

[16] J. C. Jacksona, V. Vijayakumarb, M. A. Quadirc, and C. Bharathid. "Survey on programming models and environments for cluster, cloud, and grid computing that defends big data". In: 2015.

[17] J.-P. Katoen. "The probabilistic model checking landscape". In: *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS '16. ACM, New York, NY, USA, 2016, pp. 31–45. ISBN: 978-1-4503-4391-6. DOI: 10.1145/2933575.2934574. URL: http://doi.acm.org/10.1145/2933575.2934574.

[18] H. L. S. Younes. *Planning and Verification for Stochastic Processes with Asynchronous Events.* PhD thesis. 2004, pp. 1001–1002.

[19] K. G. Larsen, P. Pettersson, and W. Yi. "Uppaal in a nutshell". *International Journal on Software Tools for Technology Transfer (STTT)* **1**:1-2 (1997), pp. 134–152. ISSN: 1433-2779. DOI: 10.1007/s100090050010. URL: http://dx.doi.org/10.1007/s100090050010.

[20] A. Legay, B. Delahaye, and S. Bensalem. "Statistical model checking: an overview". In: H. Barringer et al. (Eds.). *Runtime Verification: First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 122–135. DOI: 10.1007/978-3-642-16612-9_11. URL: https://doi.org/10.1007/978-3-642-16612-9_11.

[21] A. Leva, M. Maggio, A. V. Papadopoulos, and F. Terraneo. *Control-Based Operating System Design*. Institution of Engineering and Technology, 2013. ISBN: 1849196095, 9781849196093.

[22] N. Liu, X. Yang, X.-H. Sun, J. Jenkins, and R. Ross. "YARNsim: simulating hadoop YARN". In: *ACM Digital Symposium on Cluster, Cloud, and Grid Computing*. 2015.

[23] J. Maluszy¿ski. *Model Checking*. MIT Press, 1997, pp. 39–39. ISBN: 9780262291323. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6279110.

[24] A. D. Mauro, M. Greco, and M. Grimaldi. "A formal definition of big data based on its essential features". *Library Review* **65**:3 (2016), pp. 122–135. DOI: 10.1108/LR-06-2015-0061.

[25] I. Mytilinis, D. Tsoumakos, V. Kantere, A. Nanos, and N. Koziris. "I/o performance modeling for big data applications over cloud infrastructures". In: *2015 IEEE International Conference on Cloud Engineering*. 2015, pp. 201–206. DOI: 10.1109/IC2E.2015.29.

[26] H. Plattner and A. Zeier. *In-Memory Data Management: An Inflection Point for Enterprise Applications*. 1st. Springer Publishing Company, Incorporated, 2011. ISBN: 3642193625, 9783642193620.

[27] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino. "Apache tez: a unifying framework for modeling and building data processing applications". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. ACM, Melbourne, Victoria, Australia, 2015, pp. 1357–1369. ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2742790. URL: http://doi.acm.org/10.1145/2723372.2742790.

[28] K. Sen, M. Viswanathan, and G. Agha. "On statistical model checking of stochastic systems". In: K. Etessami et al. (Eds.). *Computer Aided Verification: 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005. Proceedings*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 266–280. ISBN: 978-3-540-31686-2. DOI: 10.1007/11513988_26. URL: https://doi.org/10.1007/11513988_26.

[29] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. "Apache hadoop yarn: yet another resource negotiator". In: *Proceedings of the 4th Annual Symposium on Cloud Computing*. SOCC '13. ACM, Santa Clara, California, 2013, 5:1–5:16. ISBN: 978-1-4503-2428-1. DOI: 10.1145/2523616.2523633. URL: http://doi.acm.org/10.1145/2523616.2523633.

[30] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. "Large-scale cluster management at google with borg". In: *inproceedings of the Tenth European Conference on Computer Systems*. EuroSys '15. ACM, Bordeaux, France, 2015, 18:1–18:17. ISBN: 978-1-4503-3238-5. DOI: 10.1145/2741948.2741964. URL: http://doi.acm.org/10.1145/2741948.2741964.

[31] A. Verma, L. Cherkasova, and R. H. Campbell. "Play it again, simmr!" In: *Proceedings of the 2011 IEEE International Conference on Cluster Computing*. CLUSTER '11. IEEE Computer Society, Washington, DC, USA, 2011, pp. 253–261. ISBN: 978-0-7695-4516-5. DOI: 10.1109/CLUSTER.2011.36. URL: http://dx.doi.org/10.1109/CLUSTER.2011.36.

[32] D. Warneke and O. Kao. "Nephele: efficient parallel data processing in the cloud". In: *Proceedings of the 2Nd Workshop on Many-Task Computing on Grids and Supercomputers*. MTAGS '09. ACM, Portland, Oregon, 2009, 8:1–8:10. ISBN: 978-1-60558-714-1. DOI: 10.1145/1646468.1646476. URL: http://doi.acm.org/10.1145/1646468.1646476.

[33] J. Yang. "From google file system to omega: a decade of advancement in big data management at google". In: *2015 IEEE First International Conference on Big Data Computing Service and Applications*. 2015, pp. 249–255. DOI: 10.1109/BigDataService.2015.47.

[34] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. "Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing". In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. NSDI'12. USENIX Association, San Jose, CA, 2012, pp. 2–2. URL: http://dl.acm.org/citation.cfm?id=2228298.2228301.

[35] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. "Spark: cluster computing with working sets". In: *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*. HotCloud'10. USENIX Association, Boston, MA, 2010, pp. 10–10. URL: http://dl.acm.org/citation.cfm?id=1863103.1863113.

[36] H. Zhang, G. Chen, B. C. Ooi, K.-L. Tan, and M. Zhang. "In-memory big data management and processing: a survey". *IEEE Transactions on Knowledge and Data Engineering* (2015).

[37] P. Zikopoulos and C. Eaton. *Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data*. 1st. McGraw-Hill Osborne Media, 2011. ISBN: 0071790535, 9780071790536.

[38] A. Y. Zomaya and S. Sakr, (Eds.). *Handbook of Big Data Technologies*. Springer, 2017. ISBN: 978-3-319-49339-8. DOI: 10.1007/978-3-319-49340-4. URL: https://doi.org/10.1007/978-3-319-49340-4.