Università degli Studi di Salerno
Dipartimento di Informatica

Dottorato di Ricerca in Informatica
(Ciclo XIV - Nuova Serie)

Tesi di Dottorato in Informatica

The Role of Distributed Computing in Big Data Science: Case Studies in Forensics and Bioinformatics

Candidato:
Gianluca Roscigno
Matr.: 8880900108

Tutor:
Prof. Giuseppe Cattaneo

Coordinatore: Prof. Gennaro Costagliola

A.A. 2014/2015

To my wonderful family

"*In theory, theory and practice are the same.*
In practice, they are not."
Albert Einstein

Acknowledgements

First of all I would like to thank my tutor Prof. Giuseppe Cattaneo for taking care of me for all these years of study, and for the active collaboration during my research activities. It has been an honor to be his Ph.D. student. I would also like to thank him for giving me the opportunity to work in his research group.

I would like to thank Dr. Umberto Ferraro Petrillo of the Department of Statistical Sciences of University of Rome - *La Sapienza* for the active collaboration during my research activities.

I would also like to thank Prof. Raffele Giancarlo of the Department of Mathematics and Computer Science of University of Palermo for having assisted my research in the Bioinformatics field.

The activities related to *Source Camera Identification* (SCI) project are a joint work with the "*Centro Nazionale per il Contrasto alla Pedopornografia Online*" (CNCPO), part of the "Dipartimento della Pubblica Sicurezza" within the Italian Ministry of Interior.

I would like to thank the Department of Statistical Sciences of University of Rome - *La Sapienza* for computing time on the *TeraStat* cluster and for other supercomputing resources.

Special thanks also goes out to the students and research scientists of the laboratory "*Benchmarking & Algorithm Engineering*" of the Prof. G. Cattaneo.

I would also like to thank my colleagues of studies, Antonio and Arcangelo, and the Ph.D. School Director, Prof. Gennaro Costagliola.

Lastly, I would like to thank my family for all their love and encouragement. Thank you.

Abstract

The era of Big Data is leading the generation of large amounts of data, which require storage and analysis capabilities that can be only addressed by distributed computing systems. To facilitate large-scale distributed computing, many programming paradigms and frameworks have been proposed, such as MapReduce and Apache Hadoop, which transparently address some issues of distributed systems and hide most of their technical details.

Hadoop is currently the most popular and mature framework supporting the MapReduce paradigm, and it is widely used to store and process Big Data using a cluster of computers. The solutions such as Hadoop are attractive, since they simplify the "transformation" of an application from non-parallel to the distributed one by means of general utilities and without many skills. However, without any algorithm engineering activity, some target applications are not altogether fast and efficient, and they can suffer from several problems and drawbacks when are executed on a distributed system. In fact, a distributed implementation is a necessary but not sufficient condition to obtain remarkable performance with respect to a non-parallel counterpart. Therefore, it is required to assess how distributed solutions are run on a Hadoop cluster, and/or how their performance can be improved to reduce resources consumption and completion times.

In this dissertation, we will show how Hadoop-based implementations can be enhanced by using carefully algorithm engineering activity, tuning, profiling and code improvements. It is also analyzed how to achieve these goals by working on some *critical points*, such as: data local computation, input split size, number and granularity of tasks, cluster configuration, input/output representation, etc.

In particular, to address these issues, we choose some case studies coming from two research areas where the amount of data is rapidly increasing, namely, *Digital Image Forensics* and *Bioinformatics*. We mainly describe full-fledged implementations to show how to design, engineer, improve and evaluate Hadoop-based solutions for *Source Camera Identification* problem, *i.e.*, recognizing the camera used for taking a given digital image, adopting the algorithm by *Fridrich et al.*, and for two of the main problems in Bioinformatics, *i.e.*, *alignment-free sequence comparison* and extraction of *k-mer cumulative or local statistics*.

The results achieved by our improved implementations show that they are substantially faster than the non-parallel counterparts, and remarkably faster than the corresponding Hadoop-based naive implementations. In some cases, for example, our solution for k-mer statistics is approximately $30\times$ faster than our Hadoop-based naive implementation, and about $40\times$ faster than an analogous tool build on Hadoop. In addition, our applications are also *scalable*, *i.e.*, execution times are (approximately) halved by doubling the computing units. Indeed, algorithm engineering activities based on the implementation of smart improvements and supported by careful profiling and tuning may lead to a much better experimental performance avoiding potential problems.

We also highlight how the proposed solutions, tips, tricks and insights can be used in other research areas and problems.

Although Hadoop simplifies some tasks of the distributed environments, we must thoroughly know it to achieve remarkable performance. It is not enough to be an expert of the application domain to build Hadop-based implementations, indeed, in order to achieve good performance, an expert of distributed systems, algorithm engineering, tuning, profiling, etc. is also required. Therefore, the best performance depend heavily on the cooperation degree between the domain expert and the distributed algorithm engineer.

Sommario

L'era dei "Big Data" sta dando vita alla generazione di grandi quantità di dati, che richiedono capacità di memorizzazione e di analisi le quali possono essere indirizzate solo dai sistemi di computazione distribuita. Per facilitare la computazione distribuita su larga scala, sono stati proposti molti paradigmi di programmazione e soluzioni, come MapReduce e Apache Hadoop, che in modo trasparente risolvono alcuni problemi relativi ai sistemi distribuiti e nascondono molti dei loro dettagli tecnici.

Hadoop è attualmente il più popolare e maturo framework che supporta il paradigma MapReduce, ed è ampiamente usato per memorizzare e processare grosse quantità di dati adoperando un insieme di computer. Le soluzioni come Hadoop sono attraenti poiché semplificano la "trasformazione" di un'applicazione non parallela a quella distribuita, adoperando strumenti generali e senza richiedere molte competenze. Tuttavia, senza qualsiasi attività di ingegnerizzazione degli algoritmi, alcune applicazioni realizzate non sono del tutto veloci ed efficienti, e possono soffrire di diversi problemi ed inconvenienti quando sono eseguite su un sistema distribuito. Infatti, un'implementazione distribuita è una condizione necessaria ma non sufficiente per ottenere prestazioni notevoli rispetto ad una controparte non parallela. Quindi, è necessario valutare come le soluzioni distribuite vengono eseguite su un cluster Hadoop, e/o come le loro prestazioni possono essere migliorate per ridurre il consumo delle risorse e i tempi di completamento.

In questa tesi mostreremo come le implementazioni basate su Hadoop possono essere migliorate utilizzando attentamente le attività di ingegnerizzazione degli algoritmi, di messa a punto, di profilazione e i

miglioramenti al codice. È anche analizzato come è possibile raggiungere questi obiettivi lavorando su alcuni *punti cruciali*, tali come: la computazione locale ai dati, la dimensione della partizione di input, il numero e la granularità dei sotto-problemi, la configurazione del cluster, la rappresentazione dell'input/output, etc.

In particolare, per indirizzare queste questioni, noi scegliamo alcuni casi di studio provenienti da due aree di ricerca dove il problema del grande ammontare dei dati sta crescendo, ossia *Digital Image Forensics* e *Bioinformatica*. Noi descriviamo principalmente vere e proprie implementazioni per mostrare come progettare, ingegnerizzare, migliorare e valutare soluzioni basate su Hadoop per il problema della *Source Camera Identification*, cioè il riconoscimento della fotocamera usata per scattare una data immagine digitale, utilizzando l'algoritmo di *Fridrich et al.*, e per due dei principali problemi in Bioinformatica, ovverosia il *confronto senza allineamento delle sequenze* e l'estrazione delle *statistiche globali o locali dei k-meri*.

I risultati ottenuti dalle nostre implementazioni migliorate mostrano che esse sono sostanzialmente più veloci delle corrispondenti applicazioni non parallele, e notevolmente più veloci rispetto alle corrispondenti semplici implementazioni basate su Hadoop. In alcuni casi, per esempio, la nostra soluzione per le statistiche dei k-meri è all'incirca 30 volte più veloce rispetto alla nostra semplice implementazione basata su Hadoop, e circa 40 volte più veloce rispetto ad un analogo strumento costruito su Hadoop. Inoltre, le nostre applicazioni sono anche *scalabili*, ossia i tempi d'esecuzione sono (approssimativamente) dimezzati quando si raddoppiano le unità di computazione. Infatti, le attività di ingegnerizzazione degli algoritmi basate sull'implementazione di miglioramenti astuti, e coadiuvate da accurate attività di profilazione e messa a punto, possono portare a migliori risultati sperimentali, evitando possibili problemi.

Noi anche evidenziamo come i miglioramenti, i consigli, gli stratagemmi e gli approfondimenti proposti possono essere adoperatati in altre

aree di ricerca e problemi.

Sebbene Hadoop semplifica alcune attività degli ambienti distribuiti, dobbiamo accuratamente conoscerlo per raggiungere prestazioni degne di nota. Non è sufficiente essere un esperto del dominio applicativo per costruire implementazioni basate su Hadoop, infatti, al fine di ottenere buone prestazioni, un esperto di sistemi distribuiti, d'ingegneria degli algoritmi, di messa a punto, di profilazione, etc. è anche richiesto. Quindi, le migliori prestazioni dipendono pesantemente dal grado di cooperazione tra l'esperto di dominio e l'ingegnere degli algoritmi distribuiti.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Nowadays, technologies provide to decision-makers the ability to collect a huge amount of data, making possible to deal with problems that, only a few years ago, were out of their reach. This trend is shown by the spread of terms like petabytes (PB)[1], exabytes (EB)[2] and zettabytes (ZB)[3] (see [35]), which are quickly replacing terms like megabytes (MB)[4], gigabytes (GB)[5] and terabytes (TB)[6] to denote a large amount of data. Such a wealth of data, called *Big Data*, requires the development of tools and methodologies with a high scalability degree and able to process virtually unbounded amounts of data.

In this dissertation, with the term *scalability* we indicate that the execution times of an application are (approximately) halved by doubling the computing units (*e.g.*, processors or computers).

In addition, the drop in hardware cost has allowed to put together clusters of commodity computers with huge computational power and storage, which are used to save and process Big Data in distributed manner (see Chapter 2 for details).

[1] One petabyte (PB) is approximately 10^{15} bytes of information.
[2] One exabyte (EB) is approximately 10^{18} bytes of information.
[3] One zettabyte (ZB) is approximately 10^{21} bytes of information.
[4] One megabyte (MB) is approximately 10^{6} bytes of information.
[5] One gigabyte (GB) is approximately 10^{9} bytes of information.
[6] One terabyte (TB) is approximately 10^{12} bytes of information.

1.1 Big Data

Short et al. [248] estimated that enterprise servers processed 9.57 ZB of data globally in 2008, that is 12 GB of information daily for the average worker, or about 3 TB of information per worker per year. The companies in the world on average processed 63 TB of information in 2008. In 2012 the 90% data in the world were created approximately in 2011-2012 years ([245, 210]). Since 2012, the use of the word Big Data in the USA has increased of $1,211\%$ on the Internet [109]. According to Cisco Systems company [68], the global mobile data traffic reached 2.5 EB per month at the end of 2014, up from 1.5 EB per month at the end of 2013. In addition, one EB of traffic traversed the global Internet in 2000, and in 2014 mobile networks carried nearly 30 EB of traffic. In fact, to manage these increasing data, the US NSA built a large datacenter at Bluffdale (Utah), capable of storing yottabytes (YB)[1] of data [31].

The challenges of the era of Big Data are represented by three Vs, *i.e.*, *Volume* (large size of data), *Velocity* (fast data creation) and *Variety* (heterogeneous structured and unstructured sources of data). Thanks to a dramatic increase in the volume, velocity and variety of data, the term Big Data has emerged as new research area. In addition to the three Vs to the Big Data definition, some authors (*e.g.*, [271]) also introduce the *Veracity*, that is an indication of data integrity and the ability for an organization to trust the data, and be able to confidently use them to make crucial decisions.

Although the term "big" in Big Data implies such, it is not simply defined by volume, but it also is about complexity [271]. Many small datasets, that are considered Big Data, not consume much physical space, but they are particularly complex in nature to analyze. At the same time, there could be large datasets that require significant physical space, but they may not be complex enough to be processed.

[1] One yottabyte (YB) is approximately 10^{24} bytes of information.

1.1.1 Real Life Scenarios

Nowadays science and industry are undergoing a profound transformation. In fact, large-scale and different datasets present a huge opportunity for data-driven decision making. Besides the sheer volume of data, they come in a variety of data formats, origin, quality, and so forth. In fact, Big Data comes from heterogeneous data sources, ranging from structured data (such as the traditional databases) to unstructured data, such as images, audio, video, textual information obtained through web scraper, email, telephone conversations, surveys, readings from sensors, transactions, complex simulations, data taken from *Online Social Networks* (OSNs) or blogs, scientific experimental data, user statistics, and so on.

Some examples of areas, where the Big Data problem is spreading, are: scientific measurements and experiments (astronomy, physics, genetics, bioinformatics, etc.), peer-to-peer communication (text messaging, chat lines, digital phone calls, etc.), broadcasting (news, blogs, etc.), Online Social Networks (Facebook, Twitter, etc.), authorship (digital books, magazines, Web pages, images, videos, etc.), administrative (enterprise or government documents, legal and financial information, etc.), business data (e-commerce, stock markets, business intelligence, marketing, advertising, etc.).

Therefore, not only computer scientists, but also bioinformaticians, physicists, sociologists, economists, political scientists, mathematicians, etc. require to storage, access and process large quantities of data produced during everyday activities.

In particular, an area where there is a large amount of data is the scientific field. According to *Guarino* in [129], *Data Science* is an emerging field basically growing at the intersection between statistical techniques and machine learning, completing this toolbox with domain specific knowledge, having as fuel big datasets. Hal Varian, Google's Chief Economist, in [276] has said: "*Data Science is the ability to take data - to be able to understand it, to process it, to extract value from it, to visualize it, to communicate it*".

1.2 Overview on Big Data Computing

As mentioned in the previous section, the term Big Data refers to collections of large datasets, which are so large to require highly specialized tools implementing different approaches with respect to traditional ones. The four Vs put pressure on developers to become comfortable with new programming paradigms. In fact, ad hoc solutions are required to capture, store, manage, share, analyze and visualize huge amount of data.

A single computer cannot store a very large amount of data. In fact, a computer would take a long time just to read these data assuming that it has fast access to the files. The solution is to divide storage and work on multiple machines.

In the Big Data era, storing huge amounts data is not the biggest challenge. In fact, efficient parallel and distributed computations are necessary to meet the scalability and performance requirements required by analyses on Big Data. The goal of an efficient implementation is to reduce resources consumption and completion time as much as possible. Indeed, a good distributed algorithm tries to efficiently exploit, at each instant of time, the set of hardware resources available.

Large-scale data analysis tasks need to run on a cluster (or group) of computers, splitting the input dataset across the different nodes. Many problems working on Big Data can individually act on each data item, that is, they are *embarrassingly parallel*, *i.e.*, little or no effort is needed to split the problem into a number of independent subproblems that can be simultaneously solved. For example, in some cases there could be little or no dependency between the subproblems.

Some applications operating on Big Data only spend a little bit of CPU time on each data item, but they work with an enormous number of data items. On the other side, other applications may spend large amount of CPU time on each data item. In any case, processing each data item in parallel can reduce the final response time.

Therefore, Big Data requires a massive computing power and dedicated storage system. In the last decade we have seen a huge deployment of cheap computers clusters to run data analytics workloads. Some applications that require months, with about $1,000$ commodity computers networked could require a few hours.

A programmer could develop an application working on Big Data using the traditional parallel programming constructs for multi-processor systems. In fact, these systems has always been considered a good solution to reduce computational time, but using a parallel implementation does not imply shorter execution times. Whereas shared-memory architectures can led to efficient implementations on a rather limited number of processors, distributed architectures require more specific solutions (mapping between algorithm and architecture) because significant overheads may be introduced that could compromise the efficiency of the entire solution. Unfortunately the shared-memory approach is not very *scalable*, *e.g.*, doubling the number of working processors, the execution times of an application not are always halved. In fact, these architectures can present many bottlenecks (due to bus congestion), therefore this approach is suited only for a limited amount of processors. For instance in this architecture all processors can share a single storage saturating the Input/Output (I/O) bus capacity and requesting the use of locking strategies to protect shared-memory areas. In many Big Data applications, I/O is concurrently performed by all processes, which leads to I/O bursts [89]. This causes resource contention and substantial variability of I/O performance, which significantly impacts the overall application performance. On the other hand, distributed architectures are inherently more scalable because each computer can access local resources without racing conditions with others. This can radically improve the architecture scalability with a large number of nodes. In order to get efficient solutions, the implementation strategy should carefully consider the underlying architecture exploiting data locality and reducing inter-node communications. There are some issues to address in distributed programming, such as: a lot of programming work, communication, coordination, managing and working with very large files, recovery from failure, status reporting, debugging, improvements, locality, scalability. Scaling out[1] using a cluster of commodity machines could be better for some Big Data analyses than scaling up[2] by adding more resources to a single server.

A computational strategy that is becoming popular for processing Big Data

[1] *Scale out* (or scale horizontally) means to add more nodes (*i.e.*, computers) to a distributed system.

[2] *Scale up* (or scale vertically) means to add resources to a single node in a system (*e.g.*, adding CPUs or memory to a single computer).

in distributed environments requires to break down the size of a problem into a (possibly large) number of smaller subproblems, to be solved using *MapReduce* (MR) paradigm ([81, 82]). This approach is fostered by the development of MapReduce distributed computing frameworks allowing to develop in a relatively simple way and without dealing with some of the most intricate aspects of distributed programming, such as inter-process data communication. Many problems on Big Data fit a MapReduce paradigm, and they can be solved using modern distributed middleware solutions which address many issues. In fact, adopting a MapReduce framework (*e.g.*, Apache Hadoop [14]), a developer can reduce the effort required to produce these distributed implementations operating on Big Data. In addition, an important element in all Big Data applications is the requirement for a scalable and distributed file system where input and output data can be efficiently stored and retrieved.

Nowadays, there is currently considerable enthusiasm around the MapReduce paradigm for large-scale data analysis. In fact, the sheer volume of data to process has led to interest in distributed processing on commodity hardware resources. For example, Google uses its MapReduce framework to process over 20 PB of data per day [82]. Clearly, large clusters of commodity computers are the most cost-effective way to process exabytes, petabytes, or terabytes of data.

Data Volume in Real Jobs *Appuswamy et al.* in [21] have measured that the majority of real world analytic jobs process less than 100 GB of input. For example, at least two analytics production clusters at Microsoft and Yahoo have median job input sizes under 14 GB ([95, 235]). In particular, 174,000 jobs submitted to a production analytics Microsoft cluster in a single month in 2011 were analyzed by the authors. The median job input dataset size was less than 14 GB, and 80% of the jobs had an input size under 1 TB. Although there are jobs operating on terabytes and petabytes, these still are the minority.

In addition, *Ananthanarayanan et al.* in [9] show that at least 90% of the Facebook jobs have input sizes under 100 GB. *Chen et al.* in [62] have studied some Hadoop workloads for Facebook and Cloudera customers. They show that a very small minority of jobs achieves terabyte scale or larger, and most jobs have input, intermediate, and output file sizes in the megabytes to gigabytes range.

Although some applications should be fed with large amounts of data, they still work on few data, because, generally, they are still too inefficient to manage them.

1.3 Motivation and Main Objectives of the Thesis

Nowadays, whereas Big Data are rapidly generated, they must also be analyzed in short amount of time exploiting distributed computing. The Big Data era provides new challenges developing algorithms for distributed systems. The spread of commodity computers clusters has allowed to design parallel implementations adopting new programming paradigms (*e.g.*, MapReduce) and distributed computing frameworks (*e.g.*, Apache Hadoop). In fact, many organizations are using Hadoop for developing applications working on Big Data exploiting clusters of computers. Methodological insights into the design, engineering and experimentation of scalable algorithms for Hadoop-based distributed systems are required to improve the execution times of the analyses on Big Data.

Hadoop could be at hand, however, to get acceptable results, much engineering work should be done to improve an algorithm in each step of the execution. One may even get the feeling that, with the use of sophisticated software like Hadoop, it is very easy to "transform" ("*porting*") a non-parallel program into a distributed one, with immediate performance gains. Although this may be the case, fundamental question relating to how well those transformations use the computational resources available is, at best, must be addressed. Tuning and profiling steps, in addition to the engineering and experimentations phases, are required in a Hadoop-based application. Therefore, a deep knowledge of the framework also should be addressed before to implement a distributed code.

In 2002 *Moret* [201] has said that in the last 30 years have seen enormous progress in the design of algorithms. However, comparatively little of it has been put into practice, even within academic laboratories. Indeed, the gap between theory and practice has continuously widened over these years. The algorithms and data structures community require to return to implementation as one of its principal standards of value. *Experimental Algorithmics* studies algorithms and data structures by joining experimental studies with the traditional theo-

retical analyses. The experimentation is also the key to the transfer of research results from paper to production code, providing as it does a base of well-tested implementations.

In 1999 *Cattaneo and Italiano* [52] have outlined that despite the wealth of theoretical results, the transfer of algorithmic technologies has not experienced a comparable growth. Algorithm designers are starting to pay more attention to the details of the machine model that they use, and to investigate new and more effective computational measures. In fact, more attention has been devoted to the engineering and experimental evaluation of algorithms, exploiting the *Algorithm Engineering* approach. It consists of the design, analysis, experimental testing and characterization of robust algorithms, and it is mainly concerned with issues of realistic algorithm performance. It also studies algorithms and data structures by carefully combining traditional theoretical methods together with thorough experimental investigations. Experimentation can provide guidelines to realistic algorithm performance whenever standard theoretical analyses fail. *Cattaneo and Italiano* also said that the experimentation is a very important step in the design and analysis of algorithms, as it tests many underlying assumptions and tends to bring algorithmic questions closer to the problems that originally motivated the work.

The intent of this thesis is to provide studies and insights, supported by implementations and experimentations, into the development of fast and efficient distributed algorithms on Apache Hadoop. Our goal is to show that when porting algorithms on Hadoop, algorithm engineering, careful profiling and tuning activities are often required to fully exploit the real potential of the distributed computing system.

1.3.1 Benchmark Problems, Methods and Results

In particular, we provide in this dissertation some methodological insights into the design, engineering and experimentation of scalable and efficient algorithms for Hadoop-based distributed systems in two research areas: *Digital Image Forensics* and *Bioinformatics*. These fields were selected because they are treating more and more data than ever before. In fact, the number of images uploaded on Web

are increasing (*e.g.*, [253]), therefore it is required to rapidly analyze them for investigative purposes; and the *Next-generation DNA sequencing* (NGS) machines are generating an enormous amount of sequence data to analyze (*e.g.*, [145]).

In addition, we provide the experience of Hadoop using two types of computers clusters: a commodity cluster of 33 PCs and one of 5 multi-processor workstations, as examples of those used in real environments.

Digital Image Forensics Nowadays the digital images are more pervasive in everyday life. In fact, there is an enormous amount of photos exchanged through Online Social Networks, and the crimes related to digital images are spreading. The digital image forensics is focused on the acquisition and the analysis of images (or videos) found on digital devices or Web for investigation purposes. It may be useful, for example, for establishing if a digital image has been altered after it has been captured (*e.g.*, [51, 53, 54, 60, 98, 301]), if it contains hidden data (*e.g.*, [59, 110]) or what camera has been used to capture that image (*e.g.*, [28, 46, 48, 122, 181]). The digital image forensics field is very active, as shown by the many contributions proposed, and valuable solutions are available in literature. However, with the spread of images in the Web, it is required to assess how these solutions scale when dealing with Big Data and/or how their performance can be improved to respond faster to investigators. In addition, only few scientific contributions concern the processing of large amounts of images in distributed environments for forensics purposes.

In Chapter 4 is presented our work done for efficiently engineering on Hadoop a reference algorithm for the *Source Camera Identification* (SCI) problem, *i.e.*, recognizing the camera used for acquiring a given digital image, also distinguished between camera of the same brand and model. We have chosen the algorithm by *Fridrich et al.* [181] as our benchmark. A first distributed implementation has been obtained, with little effort, using the default facilities available with Hadoop. However, its performance, analyzed using a commodity cluster produced discouraging effects. A careful profiling allowed us to pinpoint some serious performance issues related to a bad usage of the cluster resources. Several theoretical and practical improvements were then tried, and their effects were measured by accurate experimentations. This allowed for the development of alternative implementa-

tions that were able to improve the usage of the underlying cluster resources as well as of the Hadoop framework, thus resulting in a much better performance than the original naive implementation. In addition, our proposal is scalable, and faster than our non-parallel version.

Bioinformatics NGS has led to the generation of billions of sequence data, making it increasingly infeasible for sequences analyses to be performed on a single commodity computer. The evident mismatch between sequencing capability versus storage and CPU power poses new and staggering computational challenges to the future of genomic data [153]. Fundamental for their solution are both the full use of the power of the hardware available, and the deployment of fast and efficient algorithms. Due to remarkable advances in the development of software systems supporting them, the use of distributed architectures in bioinformatics has started to be investigated (*e.g.*, [90, 105, 139, 171, 190, 192, 207, 226, 241, 244, 288]) because of the intrinsic limitations that are found while expanding the hardware capabilities, in terms of CPU processing power, memory and storage, of a single computer. Instead, a distributed approach would allow both to deal with large genomic datasets as well as to reduce the computational time required to solve a problem at a desired scale, by proportionally increasing the number of processing units. Unfortunately, the mentioned studies are exploratory and the real impact that distributed architectures can have on bioinformatics, as well as all the technical challenges one has to overcome to get competitive results, has not even been delineated.

Sequence comparison, *i.e.*, the assessment of how similar two biological sequences are to each other, is a fundamental and routine task in computational biology and bioinformatics. Classically, alignment methods are the *de facto* standard for such an assessment. Due to the growing amount of sequence data being produced, a new class of methods has emerged: *Alignment-free* methods. Research in this ares has become very intense in the past few years, stimulated by the advent of NGS technologies, since those new methods are very appealing in terms of computational resources needed. Despite such an effort and in contrast with sequence alignment methods, no systematic investigation of how to take advantage of distributed architectures to speed up some alignment-free methods,

has taken place. Another issue to be analyzed is related to the possibility of using a distributed architecture to solve problem instances that are hard to solve on a single commodity machine because of memory constraints.

The aim of this research project is to advance the state of the art in this field by identifying and/or developing alignment-free algorithms based on the MapReduce paradigm and able to perform efficiently when run on very long genomic sequences. In Chapter 5 we initially provide a contribution of that kind, by evaluating the possibility of using the Hadoop distributed framework to speed up the running times of these methods, compared to their original non-parallel formulation. In particular, a distributed framework for the development of alignment-free sequence comparison methods based on word counts is proposed. Our experimental results show that the execution times of our solution scale well with the number of used concurrent processing units.

Another important case study presented in Chapter 5 is the collection of k-mer statistics (or counting) for genomic sequences exploiting a Hadoop cluster. A k-mer is one of the all the possible substrings of length k that are contained in a string. For example, let us assume a string ACTAGACGAT and $k = 3$. The possible k-mers for the given string are: ACT, CTA, TAG, AGA, GAC, ACG, CGA and GAT. Since the chosen problem is at the start of many bioinformatics pipelines, we set the foundation for the development of efficient distributed pipelines that use k-mer statistics. In particular, let $\mathcal{S}$ be a set of sequences, we are interested in collecting *Local Statistics* (LS), *i.e.*, how many times each of the k-mers appears separately in each of the sequences in $\mathcal{S}$, or *Cumulative Statistics* (CS), *i.e.*, how many times each of the k-mers appears cumulatively (globally) in sequences in $\mathcal{S}$. We propose a set of highly engineered distributed algorithms for both LS and CS managing different values of k, such as 3 or 31. Although both versions of the problem are algorithmically very simple, the sheer amount of data that has to be processed in a typical application has motivated the development of many algorithms and software systems that try to take advantage either of parallelism or of sophisticated algorithmic techniques or both.

The comparison between our proposal and different Hadoop-based solutions are also shown. In particular, the results highlight that our algorithm is faster and efficient than these solutions. In addition, our analyses show that the execution

times of our solution scale well with the number of used concurrent processing units. Moreover, a careful profiling of some of the most successful methods that have been developed for CS in parallel environments, from which it is evident that they do not scale well with computational resources, is also presented. The bottlenecks responsible for these problems are also identified.

General Results Taking advantage of the experience gained in these fields, we also give suggestions for researchers on how to improve scalability and efficiency of a distributed implementation on Hadoop. In fact, it is also possible to use the strategies used in this dissertation to develop efficient Hadoop-based variants of algorithms belonging to different domains. An interesting future direction for our work would be the formalization of this methodology and its experimentation with other case studies.

1.4 Organization of the Thesis

In Chapter 2 is presented the parallel and distributed computing with emphasis on emerging software solutions to process Big Data, such as Apache Hadoop [14], a distributed MapReduce computing framework. A brief history of parallel and distributed systems is also presented. In addition, are outlined the performance measurement metrics in parallel and distributed environments, such as *Speed up* [8] and *Size up* [132], which are useful to measure the scalability and the efficiency of a distributed implementation.

In Chapter 3 is presented an overview on MapReduce paradigm, and then it is discussed Apache Hadoop, that is the most popular and used MapReduce framework. In addition, are discussed the main features of Hadoop and its distributed file system. The chapter is concluded with a section describing how to profile, tune up and try to improve Hadoop-based distributed applications to obtain a high-level of scalability and efficiency.

In Chapter 4 is shown a very popular algorithm in digital image forensics field for Source Camera Identification problem, that is the algorithm by *Fridrich et al.* [181]. Here is described our work done to efficiently speed up the running times of *Fridrich et al.* approach using a Hadoop application running on a commodity

cluster. The first implementation has been developed in a straightforward way with the help of the standard facilities available with Hadoop. However, its performance produced discouraging effects. In fact, a closer investigation revealed the existence of several performance issues, therefore, we describe how to put in practice an engineering methodology aiming, first, at pinpointing the causes behind the performance issues we observed, and, second, at solving them through the introduction of several theoretical and practical improvements.

In Chapter 5 are presented our Hadoop-based efficient implementations to solve two problems in bioinformatics field: *Alignment-free Sequence Comparison* and *K-mer Statistics*. Initially, is presented an overview on biology and Big Data, and then are described the aspects related to the sequence comparison. Then our activities for developing and experimenting a Hadoop-based implementation for word-based alignment-free sequence comparison methods are presented. Subsequently, we deeply focus on the problem of the extraction of k-mer local and cumulative statistics. Here, we describe and validate a very fast and efficient Hadoop implementation for k-mer counting. Experimentations of the some successful parallel solutions for k-mer counting that have been developed for cumulative statistics are also addressed. These methods do not scale well with computational resources, in fact, the bottlenecks responsible for this are also identified.

In Chapter 6 is reviewed the work done in this thesis, by focusing on the experience gained and general lessons learned useful to other researchers when they would like to implement an algorithm on Hadoop.

In Appendix A is described some information related to Chapter 5, while in Appendix B are listed the publications written during Ph.D. studies.

Chapter 2

Parallel and Distributed Computing

In this chapter are addressed some aspects related to parallel and distributed computing. In particular, in Section 2.1 is presented the state of the art, while in Section 2.2 is discussed the history of parallel and distributed systems. Storing, processing and analyzing Big Data in parallel and distributed environments is addressed in Section 2.3. The emerging distributed architectures and distributed computing middleware solutions are treated in Section 2.4. Lastly, Section 2.5 presents the performance measurements in parallel and distributed environments.

2.1 State of the Art

In general, most real-life large problems can be split up into a large number of small subproblems that can be solved individually, therefore, also at the same time.

A *sequential* program (also called *stand-alone* or *serial* in this dissertation) is a single-thread program, *i.e.*, non-parallel, able to be only run on a single CPU core at a specific time instant. A CPU may have one or more cores to perform tasks at a given time, while a CPU core is the hardware on a computer that executes a stream of machine instructions. In addition, a modern computer can have a single-core or multiple cores. Given a sequential program that performs a complex computation, the execution time can be reduced designing its concurrent counterpart able to exploit multiple computational resources at the same time,

such as more CPU cores[1] in a same machine and/or more machines. For improving the performance of an application, it can be necessary to design a parallel program, do not increase CPU clock speeds.

In *parallel computing*, a computational job is generally split in several, often many, very similar subtasks that can be processed independently on a same machine, and whose results are combined afterwards, upon completion. Here all cores of a same machine may access to a shared-memory to exchange information between them ([219]). According to *Kaminsky* in [154], parallel computing is concerned with designing computer programs having two characteristics: they run on multiple processors/cores, and all the cores cooperate with each other to solve a single problem. Therefore, a *parallel program* runs on multiple CPU cores at the same time, with all the cores cooperating with each other to solve a single job.

Multi-processor machines have always been considered a good solution to speed up the response time, but engineering a parallel algorithm on a hardware architecture with more computational resources does not imply shorter execution times. In fact, a parallel application (*i.e.*, program) is *scalable* if the execution time linearly decreases adding more processors/cores.

A *Distributed System* (DS) is composed of many independent (autonomous) computers (also called nodes) that communicate over a network. They interact with each other in order to achieve a common goal (see *e.g.*, [33, 74, 262]). A *node* is an independent and autonomous computer with its own CPU cores, its own main and external memory, and its own network interface. Therefore, in distributed computing, each computing node has its own private memory, the components are located on networked computers, and communicate and coordinate their actions by message-passing. Some important characteristics of a distributed system are: concurrency of components, lack of a global clock, and independent failure of components [74].

Kaminsky ([154]) states that a "*parallel computer*" can consist of a single node, or of multiple nodes. A *cluster* is a multi-node parallel computer (*i.e.*, a distributed system), where the main memory is distributed and cannot be

[1]Generally speaking, the terms *single-core processor*, *core* or *computing unit* can be used interchangeably.

shared by all the CPU cores, requiring to the application to do inter-process communication to move data from node to another, and, therefore, increasing the program overhead. Whether a program needs more main memory, it is possible to add more nodes to the cluster. In fact, memory hungry program can scale up to much larger problem sizes on a cluster. An easy way to utilize the parallelism of a cluster is to run multiple independent instances of a sequential program on each subproblem, and, subsequently, to aggregate the results.

Therefore, distributed computing uses a network of many computers, each accomplishing a portion of an overall task, to achieve a computational result much more quickly than with a single computer. Generally, modern distributed systems use a distributed computing middleware (also called framework), which enables computers to coordinate their activities and to share the resources of the system, so that end users perceive the system as a single and integrated computing facility.

Generally speaking, there are different approaches to parallel or distributed programming, such as: parallel algorithms in shared-memory model, parallel algorithms in message-passing model and distributed algorithms in message-passing model. A same system may be characterized both as "parallel" and "distributed". In fact, in a generic distributed system, the cores of a same computer run concurrently in parallel to solve the same problem ([182]).

2.1.1 Flynn's Taxonomy

Flynn's taxonomy [104] is a classification of computer architectures, proposed by Michael Flynn in 1966. In this section we review the four classes defined by Flynn.

Single Instruction stream Single Data stream (SISD) This class indicates a standard computer which exploits no parallelism in either the instruction or data streams, *e.g.*, traditional single-processor machines such as old PCs or old mainframes.

Single Instruction stream Multiple Data streams (SIMD) This class indicates a computer which exploits multiple data streams against a single instruction stream to perform operations which may be parallelized. SIMD is adopted in *Streaming SIMD Extension* (SSE) or Altivec macros, some database operations, some operations in data structure libraries, array/vector processor and *Graphics Processing Unit* (GPU). In particular, the *vector processors* take one single vector instruction that can simultaneously operate on a series of data arranged in array format. Therefore, SIMD is a paradigm in which the parallelism is confined to operations on corresponding elements of vectors.

Multiple Instruction streams Single Data stream (MISD) This class indicates multiple instructions which operate on a single data stream. Heterogeneous components operate on the same data stream and must agree on the result, such as the pipeline architectures and fault-tolerant computers. For example, a fault-tolerant machine executes the same instructions redundantly (task replication) in order to detect and solve errors, *e.g.*, the Space Shuttle flight control computer uses this paradigm.

Multiple Instruction streams Multiple Data streams (MIMD) This class indicates multiple independent processors/cores or machines simultaneously executing different instructions on more data. This paradigm assumes multiple cooperating processes executing a program. Generally, distributed systems and multi-core processors are examples of MIMD architectures.

The *MIMD architectures with private memory* are a computational model where different computing units with private memory communicate through the network by message-passing, *e.g.*, clusters of workstations communicating through a *Local Area Network* (LAN). The first private memory distributed computer was the *Cosmic Cube*[1] with 64 computing nodes, where each node having a direct, point-to-point connection to 6 others like it. Subsequently were developed other architectures, such as hypercubes, meshes and dataflow machines.

In fact, in a MIMD architecture with private memory, each processor has its

[1]The Caltech Cosmic Cube was a parallel computer, developed by Charles Seitz and Geoffrey C. Fox from 1981 onward.

own local memory, therefore, data are transfered from one processor to another through message-passing. To avoid to connect each processor directly to each other, each is just connected to a few processors. Thus, some systems, such as hypercubes and meshs, were designed to reduce these links. For example, in a hypercube system with 4 processors, a processor and a memory are placed at each vertex of a square, while the processors are placed in a two-dimensional grid in a mesh network.

In addition, the MIMD class can be split into two subcategories:

- **Single Program Multiple Data streams (SPMD)**
  Multiple autonomous computing units simultaneously run the same program on different data ([23, 78]). Tasks are split up and run simultaneously on multiple cores with different input in order to obtain results faster.

- **Multiple Programs Multiple Data streams (MPMD)**
  It allows to run different programs on each of the multiple autonomous computing units. Typically, they simultaneously operating at least two independent programs: one called *slave* (*host*) program and one called *master* (*manager*) program. For example, the master can run a program that sends data to all the other computing units which all execute a slave program. The slaves then return their results directly to the master process.

2.1.2 Parallel Systems with Shared-Memory versus Distributed Systems

Roughly speaking concurrent architectures can be split in two main classes: Shared-Memory Systems versus Distributed Systems. In this section we propose an handsome evaluation of the differences existing between the shared-memory and the distributed computer architectures.

Whenever some data requires to be very frequently accessed, a common alternative is shared-memory architecture within a single-node, where the computing elements share the same memory bus. Shared-memory architecture is able to exploit parallelism on a machine equipped with multiple CPU cores by running several execution units at the same time. The main memory is shared among

all units, this allows two units to communicate by just sharing a same variable without any explicit data transmission. A consistent access to shared variables is typically guaranteed by locking mechanisms that allows two or more units to use a same shared variable in a safe and predictable way. Some examples of this architectures are the *Symmetric Multi Processor* (SMP) computers.

A distributed architecture differs from a shared-memory system in that each computing unit of the architecture is autonomous and share no memory or storage resources with the others. The execution of a distributed algorithm is typically achieved thanks to a network connection that allows two or more units (nodes) to communicate by exchanging messages. Each node could also have many CPU cores.

The shared-memory approach is usually more convenient than the one based on a distributed architecture, when the number of execution units is relatively low. This holds because the communication between different units in the distributed case suffers of much higher latencies and lower bandwidth than the corresponding cost paid in the shared-memory case. On the other side, the performance gain achievable by increasing the number of computing units (*i.e.*, scalability) in the shared-memory case is inherently limited by the underlying hardware architecture because of several serial bottlenecks, such as the bus used to access the shared-memory or the disk used for I/O operations. In fact, shared-memory systems do not well scale because these architectures present many bottlenecks (due to bus congestion), therefore this approach has been proved to be suited only for a limited number of processors. For instance, in this architecture all processors share a single storage and the I/O bus capacity can easily be saturated. Moreover sharing resources requires the introduction of a locking mechanism to protect them against unwanted concurrent accesses. Many of these bottlenecks do not exist in a distributed setting, thus allowing a distributed algorithm to achieve a virtually unlimited scalability on a large range of applications.

Whereas shared-memory architectures can led to efficient implementations on a rather limited number of processors, distributed architectures require more specific engineering activities (to map the algorithm against the target architecture). This is necessary because significant issues can arise compromising the efficiency of the entire solution. The distributed architectures are inherently more scal-

able because each processing element (node) can access local resources without incurring in racing conditions with other nodes. This can radically improve the architecture scalability supporting large number of nodes. The distributed systems can be more fault tolerant because if a node fails the system yet works. However in order to get efficient solutions, the implementation strategy should consider the underlying architecture exploiting data locality, and reducing inter-node communications and the related overheads.

2.2 History of Parallel and Distributed Systems

Law in [149] has said that the idea of harnessing the unused CPU cycles of a computer is as old as the first networks that later became the Internet. The use of concurrent processes that communicate by message-passing has its roots in architectures studied in the 1960s [10]. Initially, supercomputers were used to solve huge computational problems, but when the price of personal computer declining rapidly, and supercomputers still very expensive, an alternative was necessary.

In the following, the evolution of parallel and distributed systems is briefly presented (for details see *e.g.*, [146, 149, 233]).

Since 1945 until mid 1980s the computers were large and expensive. For example, a mainframe cost millions and a minicomputer cost tens of thousand.

In the 1960s and the 1970s the supercomputers were shared-memory multiprocessor systems, with multiple processors working side-by-side on shared data. In 1964 were produced the IBM/360 mainframe systems. This system performed large computation and massive processing, but communication was rare. In fact, it communicated by manually mounting data into a tape and transfer it from one system to another.

The 1969 saw the birth of the *Advanced Research Projects Agency NETwork* (ARPANET), that was the predecessor of the Internet. In the 1970s were spread: ARPANET Email[1], Ethernet for Local Area Network, mainframes and centralized hosts, minicomputers and user terminals. In the same years were born the

[1] ARPANET Email was invented in the early 1970s and probably the earliest example of a large-scale distributed application.

decentralized stand-alone systems, deployed mainly in organizations. They were not really a from of distributed system, but they were the predecessor of *Enterprise Resource Planning* (ERP) systems.

The first programs working in a net were a pair of applications called *Creeper* and *Reaper* invented in the 1970s. Creeper [281] was possibly one of the first programs that resembled an Internet worm. It ran on the old Tenex *Operating System* (OS) and spread through the ARPANET. Another similar program, called Reaper, was created to fight the Creeper infections. Reaper was the first *nematode*, that is a computer virus that attempts to remove another virus. In particular, Creeper was a worm program and it used the idle CPU cycles of processors in the ARPANET to copy itself onto the next system and then delete itself from the previous one. It was modified to remain on all previous computers, and so Reaper was created which traveled through the same network and deleted all copies of the Creeper.

In the 1980s the workstation servers became demanding and increase exponentially. In addition, the drop in hardware price did spread the *Personal Computer* (PC). The massively parallel architectures started rising, and message-passing interfaces and other libraries were developed. In fact, in the early 1980s, the idea of using parallelism to solve several tasks at the same time was spread.

In the 1980s the INMOS *Transputer* (Transistor Computer) was a microprocessor architecture for parallel computing systems that used integrated memory and serial communication links. These chips could be wired together to form a complete parallel computer.

In the mid 1980s was born the *Connection Machines* (CMs) supercomputers series of Thinking Machines Corporation (TMC) for massively parallel computing. For example, the first version, called *CM-1*, has used up to about $65,000$ single bit processors interconnected to exchange data in a hypercube architecture.

Since mid 1980s the diffusion of microprocessors and computer networks LAN and *Wide Area Network* (WAN) have determined the dissemination of the distributed systems. In fact, the microprocessors offered a better price and performance than mainframes, and some distributed systems had more total computing power than a mainframe. Indeed, the computing resources in a distributed system can be added in small increments. However, little software existed in these

years for distributed systems, and, in some case, the network saturated or caused other problems.

In the mid 1980s the *Caltech Concurrent Computation* project built a supercomputer for scientific applications from 64 Intel 8086/8087 processors. This system showed that extreme performance could be achieved with *Commercial Off-The-Shelf* (COTS) microprocessors (that is, they are ready-made and available for sale to the general public).

In 1983 was delivered the *Goodyear Massively Parallel Processor*, a *Massively Parallel Processing* (MPP) supercomputer built for the NASA. It was designed to deliver enormous computational power at lower cost than other existing supercomputer architectures. It used thousands of simple processing elements, rather than one or a few highly complex CPUs. In this system each processor performs the same operations simultaneously, on different data elements.

The first Internet-based distributed computing project was started in 1988 by the DEC System Research Center. This application sent tasks to volunteers through email, who would run these programs during idle time, then they sent the results back and, finally, they got a new task. For example, in the 1990s some tasks were factoring and prime number searching, and encryption cracking.

In the 1980s the interest in distributed computing is also evidenced by two conferences. In fact, in 1982 the *Symposium on Principles of Distributed Computing* (PODC) was the first conference in the distributed computing. Then, in 1985 was held the first European counterpart, that is the *International Symposium on DIStributed Computing* (DISC).

Between late 1980s and early 1990s was born *Parallel Virtual Machine* (PVM), a framework designed to allow a network of heterogeneous computers to be used as a "single distributed parallel processor". PVM adopted a runtime environment and a library for message-passing, tasks and resources management, and fault notification. It was used by various types of computers, such as: shared-memory or local-memory multiprocessors, vector supercomputers, specialized graphics engines, or scalar workstations and PCs, which were interconnected by many types of networks.

During 1980s and early 1990s were developed inexpensive PCs and networking hardware. Until the beginning 1990s, the parallel computing was still expensive,

in fact, each vendor had its own proprietary hardware architectures, parallel programming languages, and parallel software libraries. Indeed, various message-passing environments were developed in the early 1980s. However, some were developed for special purpose computer architectures and/or networks. Therefore, the parallel computing was mostly used for scientific, engineering and academia applications.

In the early 1990s the client-server architectures and *World Wide Web* (WWW) were born, and the first web sites were also developed. In 1991 was born *Common Object Request Broker Architecture* (CORBA), a distributed computing protocol, that enable the applications to tun on any hardware in anywhere and enable the programs can be written in any language that has mappings with *Interface Description Language* (IDL). Subsequently, the introduction of Java RMI demised CORBA.

In 1992 some researchers agreed to develop and then implement a common standard for message-passing. In this way, the *Message Passing Interface* (MPI) was born as a standardized and portable message-passing system used to work on a wide variety of parallel computers. In the mid 1990s, for MPPs and clusters systems, a number of application programming interfaces converged to MPI. In fact, since the mid 1990s, PVM and other libraries were supplanted by MPI standard.

For shared-memory multi-processor computing systems, a similar process unfolded with convergence around two standards by the mid 1990s to late 1990s: pthreads and OpenMP.

In addition, a large number of competing parallel programming models and languages have emerged over the years. For example, in 1993 was born the *Distributed Computing Object Model* (DCOM), a Microsoft technology for communication among software components distributed across networked computers.

From the point of view of the hardware, in the 1990s, the Intel Paragon was released as a series of massively parallel supercomputers based on the Intel i860 RISC microprocessor. Here, up to $2,048$ (later, up to $4,000$) processors were connected in a two-dimensional grid. In 1994, Donald Becker and Thomas Sterling at NASA introduced Beowulf system, a cluster of commodity computers[1]

[1]The term *commodity* indicates a computer hardware which is affordable and easy to obtain.

networked into a small local network with libraries and programs installed which allow processing to be shared among them. The result is a high performance parallel computing cluster from inexpensive personal computer hardware. Beowulf was the first time that an effort was made to use the commodity hardware and then build a cluster of computers that could compete with the top supercomputers. However, a stable and suitable software layer was missing. Beowulf opened the era of parallel and distributed computing made with commodity hardware rather than proprietary machines to assemble a cluster of computers.

With the passage of time, parallel programming shifted to using standard languages, such as Fortran, C, and C++ with standard parallel programming libraries. MPI became the *de facto* standard for parallel programming on cluster computers, and OpenMP became the *de facto* standard for parallel programming on multi-core computers.

Distributed.net [87] was a project founded in 1997 which is considered the first to use the Internet to distribute data for calculation and collect the results. They allowed the users to download the program that would utilize their idle CPU time instead of emailing it to them. Distributed.net completed several cryptology challenges by RSA society.

In 1999 Napster introduced a early form of *Peer-to-Peer* (P2P) system with the purpose to enable sharing of data, such as streaming audio or video. It eliminates requirements of servers and associate infrastructure. The storage can evenly distributed amongst nodes and the costs of bandwidth are spread.

Since 1999 SETI@Home project have analyzed in distributed manner the radio signals that were being collected by the Arecibo Radio Telescope in Puerto Rico. It has gained over 3 million independent users who volunteer their idle computers to search for signals that may not have originated from Earth.

In the late 1990s were spread the Internet-based computing and web services, *e.g.*, the usage of *Uniform Resource Locator* (URL) to call upon to perform the function as a service via the Internet. These systems enabled the communication between client and web services, and they allowed a new way of *application-to-application* communication.

Until early 2000s processor chip manufacturers had exploited *Moore's Law* to increase both the number of transistors on a chip and the chip speed, doubling

the clock frequency about every two years. In 2004 the CPU clock frequencies had gotten fast enough that any further increment would have caused the chips to melt from the heat they generated. Therefore, the vendors started putting more processor cores on the CPU chip. In fact, nowadays there are computers with 2, 4, 8, or more CPU cores. At the same time, memory chip densities continued to increase. In fact, now there are PCs with 4 GB, 8 GB, 16 GB, or more of RAM. In addition, they have Graphics Processing Units (GPUs) with dozens or hundreds of cores. Despite the increase of the computing power and memory size, there still are computational problems too big for a single machine.

Dobre and Xhafa [88] have said that in the 1990s the data volumes generated were sufficiently low that the *Database Management System* (DBMS) itself would figure out the best access path to the data. They have said that the largest databases increased tenfold in size between 2005 and 2008 showing the start of Big Data era.

In the 2000s with the spread of Big Data and mobile technology, new form of distributed systems, such as Cloud Computing, Grid Computing, Mobile and Ubiquitous Computing, were born. Nowadays, cluster and grid architecture are increasingly dominant also using commodity hardware, that is they use a large number of low-cost and low-performance commodity computers working in parallel, instead of using fewer high performance and high cost computers. Google and other companies, having to deal with exponentially growing web traffic and user demands, are doing this to the extreme using clusters of thousands of computational nodes. In the early 2000s Google invented the MapReduce paradigm ([81, 82]), designed to work with distributed processing adopting a massively distributed file system ([115]). This has inspired an open source distributed computing framework called Apache Hadoop ([14]) and its distributed file system ([249]).

Nowadays, parallel programs are written using libraries, such as OpenMP, MPI, CUDA, OpenCL, and so on. However, the parallel applications not only use C, MPI and OpenMP, but they also adopt newer languages (*e.g.*, Java), high-level programming paradigms, such as MapReduce, and distributed computing frameworks, such as Hadoop. This has opened the distributed computing to a much broader range of common applications exploiting Big Data. The modern

"supercomputers" use the same commercial off-the-shelf hardware (*e.g.*, CPU and RAM of desktop PCs) to assemble a cluster of computers. In fact, today, applications exploiting MapReduce running on Big Data and clusters of commodity hardware are emerging. Indeed, most of the world's fastest supercomputers on the Top500 List [259] are clusters of computers.

Nowadays, in database area, standard DBMS systems based on SQL are became inapt for the manage Big Data. In fact, NoSQL database are diffusing. For example, Massively Parallel Processing (MPP) databases allow database loads to be split amongst many nodes.

The future trend of distributed computing could be the *Continuum Computing* and *Smartphone Grids* (or *Smart Grids*). The Continuum Computing is composed by a highly heterogeneous interconnections of systems and/or devices offering different features. It also enables resource sharing and remote control easily, *e.g.*, transferring information from personal computer to tablet. The Smartphone Grids are a set of smartphone or smart devices interconnected into a network. They provide slightly amount of computational power to solve complex problems on smart devices. In fact, *Anwar et al.* in [11] have explored if a cluster comprising of microservers (*e.g.*, Raspberry Pi) can support the popular Hadoop framework. They demonstrate that some applications can yield two orders of magnitude better efficiency than traditional servers.

2.2.1 Scientific High Performance Computing

High Performance Computing (HPC) refers to technologies used by computer clusters to create systems that can provide very high performance in the range of PetaFLOPS[1], exploiting parallel computing. The term HPC is widely adopted primarily for processing systems used in scientific area. For example, the *Partial Differential Equations* are the source of a large fraction of HPC problems. Nowadays, some areas of science are facing an huge increasing in data volumes from satellites, telescopes, high throughput instruments, sensor networks, accelerators, and supercomputers, compared to the volumes generated only a decade ago [36].

Scientific computing is the cross-disciplinary field at the intersection of model-

[1]FLOPS is the acronym for *floating-point operations per second*.

ing scientific processes, and the use of computers to produce quantitative results from these models [93]. Modern architectures used for scientific computation, starting from simple workstations up to parallel supercomputers with huge computational power. An important concern in scientific computing is efficiency.

The computational power of these HPC systems is huge. In fact, several tens of TeraFLOPS are at disposal of researchers for solving their computing problems. This level of performance is possible thanks to the parallel use of thousands nodes connected with high throughput networks [93].

According to *Guest et al.* in [130], HPC is currently undergoing a major change as the next generation of computing systems (named "exascale systems") is being developed for 2020. These new systems pose numerous challenges, such as reduction of energy consumption, development of programming models for computers that host millions of computing units, storage and integration of both observational and simulation or modeling data.

Gray in [127] has defined *data-intensive science* ("*eScience*") as the synthesis of information technology and science that enables challenges on previously unimaginable scales to be tackled.

Bell et al. in [32] have said that in recent decades, computer simulations have become an essential standard tool for scientists to explore domains that are inaccessible to theory and experiments. Whereas simulations and experiments yield ever more data, new techniques and technologies are required to perform data-intensive science. *Bell et al.* refer that new types of computer clusters are emerging, which are targeted for data movement and analysis rather than computing. In astronomy and other sciences, integrated data systems allow local data analysis and local storage, instead of requiring download of large amounts of data.

According to *Juve et al.* in [152], the developers of scientific applications have many options when it comes to choosing a platform to run their applications. One of the advantages of HPC systems over currently deployed commercial distributed computing systems is the availability of high performance I/O devices. HPC systems commonly provide high-speed networks and distributed file systems, while most commercial infrastructures use commodity networking and storage devices. These high performance devices increase workflow performance by making inter-

task communication more efficient.

2.2.2 Explicit and Implicit Parallelism

Parallel and distributed programs are harder to write than sequential ones. In fact, a program that is divided into multiple concurrent tasks is more difficult to write, due to the necessary synchronization and communication that needs to take place between those tasks. In the end of 1980s when the multi-processor machines spread, a user took advantage of this architecture either with heavy work loads which implicitly enable the use of all the resources or by explicitly programming his own applications with parallel algorithms (*explicit parallelism*). In fact, much work was left to the programmer, *e.g.*, synchronization, communication, data partitioning, scheduling, data aggregation, failure management, and so on. Over time high-level libraries and interfaces to facilitate the use of parallel and distributed programming were developed, and more and more functionalities were added.

The interest on parallel and distributed computing is increased thanks to modern computing frameworks and middleware solutions able to facilitate the implementation of parallel programs to be run on a single o more nodes exploiting shared-memory (in a single node) or message-passing. These frameworks are abstraction layers which hide details about hardware devices or other software from an application. They lie between the operating system and applications on each node of a distributed computing system, and they provide services beyond those provided by the operating system to enable the various components of a distributed system to communicate and manage data. In fact, these middleware solutions support and simplify complex distributed applications, and they transparently can provide high-level functionalities, such as: synchronization, communication, data partitioning, scheduling, data aggregation, failure management, and so on.

For example, MapReduce-based frameworks allow to a programmer to write a distributed algorithm in a simple way. In fact, he not needs to be an expert of the distributed and parallel environments, because the programmer can write distributed applications adopting an abstraction layer. Therefore, with the intro-

duction of the moder frameworks was coined the term: *implicit parallelism*.

In this chapter are also reviewed some modern distributed computing middleware solutions that allow to write applications even more easily respects to MapReduce, exploiting very high-level instructions (*e.g.*, SQL-like).

2.3 Storing, Processing and Analyzing Big Data

Big Data problem requires a massive computing power and dedicated storage system. The structured Database Management System (DBMS) successfully employed to store applicative data are inadequate to store and process Big Data. Timely and cost-effective analytics over Big Data is now a key success feature in many businesses and disciplines. Nowadays the large amount of data to be processed requires massively parallel and distributed software running on tens, hundreds, or even thousands of computer nodes. Any given computer has singly a series of absolute and practical limits, for example memory, disk capacity, processor speed, I/O throughput, etc. that can be increased only on the technological evolution basis. In addition, today it is much more cost-effective to purchase commodity computers than to acquire a single *high performance computer*, also showing higher performance than a single server.

The consequence of this tsunami of data is that the traditional relational data storage has reached its limit. In fact, people are aware of the limitations of conventional approaches to storing, managing and processing data, and we need a specific technology for Big Data archiving and to efficiently process them within an short elapsed time.

Therefore, it is not surprising that distributed computing is nowadays the most successful (and adopted) known strategy for storing, processing and analyzing Big Data. Additional technologies being applied to Big Data include: Massively Parallel Processing (MPP) databases, search-based applications, data-mining grids, distributed file systems, distributed databases and cloud-based infrastructure.

Esen Sagynov in [238] classifies the various technology platforms that treat Big Data into *Storage Systems*, *Processing Systems*, and *Analysis Systems*. This categorization is not exclusive, and there are systems that simultaneously address them.

- **Storage Systems**, as for example *Parallel DBMSs* and *NoSQL* systems. Both the Parallel DBMS and NoSQL systems are identical in that they use the scale out expansion approach in order to store large data. In this category also are included the existing storage technologies, for example *Storage Area Network* (SAN), *Network Attached Storage* (NAS), distributed file systems and cloud storage.

- **Processing Systems**. Once a large dataset has been distributed to multiple nodes, however, a huge advantage can be obtained by distributing the processing as well. The key point of parallel processing is *Divide and Conquer* (*D&C*) paradigm, *i.e.*, the dataset is divided in independent parts to be processed in parallel. Big Data processing is performed by dividing a problem into several sub-operations and then combining together the sub-results. In order to be effective systems, the computation must involve as much local data is possible, otherwise the impact introduced by network transfers would be too expensive. The most famous Processing Systems are based on the MapReduce paradigm.

- **Analysis Systems**. In this category there are the systems that analyzes Big Data. The step of finding meaning and information in data is called *Knowledge Discovery in Databases* (KDD). These systems are used to store data, process and analyze the whole or part of interested data in order to infer unknown information. In these systems are applied various technologies as artificial intelligence, machine learning, statistics, and databases. Belong to this class: *On-Line Analytic Processing* (OLAP), Data Cubes, Databases and Statistical Packages.

Khandelwal in [157] outlines another high-level categorization of Big Data platforms to store and process them in a scalable, fault tolerant and efficient manner. The first category includes Massively Parallel Processing (MPP) Data Warehouses that are designed to store huge amount of structured data across a cluster of autonomous nodes, connected via high-speed networks, and perform parallel computations over it. Since they are designed to hold structured data, it is required to extract the structure from the data using an *Extract, Transform,*

Load (ETL) tools and populate these data sources with the structured data. According *Khandelwal*, these systems include:

- **MPP Databases**: these are generally the distributed systems designed to run on a cluster of commodity servers, for example: Aster nCluster, Greenplum, DATAllegro, IBM DB2, Kognitio WX2, Teradata, IBM DB2, Teradata.
- **Appliances**: a purpose-built machine with preconfigured MPP hardware and software designed for analytical processing, for example: Oracle Optimized Warehouse, Teradata machines, Netezza Performance Server and Sun's Data Warehousing Appliance.
- **Columnar Databases**: they store data in columns instead of rows, allowing greater compression and faster query performance, for example: Sybase IQ, Vertica, InfoBrightData Warehouse, ParAccel.

2.4 Emerging Distributed Architectures and Solutions

The amount of computational resources required for applications grows out of proportion. As said in the Section 2.2, Cloud and Grid Computing infrastructures are emerging as new forms of distributed computing. In fact, these terms are becoming more and more popular.

As previously stated, throughout computer history were designed different parallel and distributed architectures and solutions. We cite, as examples, multi-threading[1] computing for multi-core systems, symmetric multiprocessing, cluster computing, massive parallel processing, reconfigurable computing with field-programmable gate arrays (FPGA), general-purpose computing on graphics processing units (GPGPU), application-specific integrated circuits (ASIC), vector

[1] A thread is a basic unit of CPU core utilization that shares with other threads belonging to the same process its code section, data section, and other operating system resources ([113]). A traditional process has a single thread of control, while a multi-threading process has multiple threads of control which can perform more than one task at a time on different CPU cores.

processors, and so on. Each of these solutions has its own pros and cons. One approach that has gained consensus in the recent past is the one based on cluster computing. The rationale of this approach is to build a *virtual supercomputer* by linking together a set of network-connected machines, likely to be assembled using commodity hardware, with the purpose to solve a complex problem by distributing it over different machines. It is possible to deliver virtually unlimited computing power using commodity or recycled calculators, at a fraction of the cost of a multi-processor machine.

In this section we review the concepts of Cloud and Grid Computing, and are also described other modern technologies used in Big Data era.

2.4.1 Cloud Computing

Cloud Computing (CC) is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (*e.g.*, networks, servers, storage, applications, and services) that can rapidly previsioned and resealed with minimal management effort or service provider interaction [135]. Clustering a set of computers provides the scale and performance for cloud. Cloud Computing is TCP/IP-based high-development and integrations of computer technologies, and it is based on several other computing research areas, *e.g.*, HPC, virtualization, utility computing and Grid Computing.

The main features of CC are: resource sharing, service oriented, loose coupling, strong fault tolerant and business model. Advantage for organizations are: flexible response, reliability and cost reduction.

The NIST Cloud Computing Program [208] have emanated some definitions about the Cloud Computing, *e.g.*, the deployment and service models. Figure 2.1 shows some concepts described by NIST cloud computing.

A *deployment model* defines the purpose of the cloud and the nature of how the cloud is located. The NIST definitions for the four deployment models are: public cloud, private cloud, hybrid cloud and community cloud. The public cloud infrastructure is available for public use or for a large industry group, and it is owned by an organization selling cloud services. The private cloud infrastructure is operated for the exclusive use of an company. This cloud may be managed

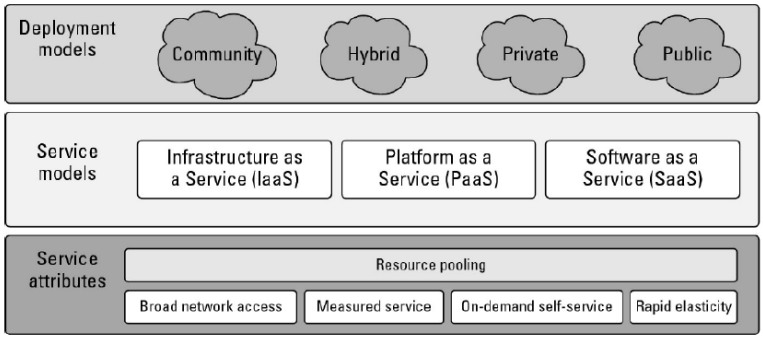

Figure 2.1: The NIST cloud computing definitions. The image was taken from [135].

by that organization or a third party. A hybrid cloud combines multiple types of clouds, such as private or community of public. A community cloud is one where the cloud has been organized to serve a common function or purpose. It may be for one organization or for several organizations, but they share common concerns. A community cloud may be managed by the constituent organization(s) or by a third party.

The NIST has also provided three *service models*, that is: *Infrastructure-as-a-Service* (IaaS), *Platform as-a-Service* (PaaS) and *Software-as-a-Service* (SaaS). IaaS, also called *Hardware-as-a-Service* (HaaS), is the delivery of huge computing resources such as the capacity of processing. Platform-as-a-Service generally abstracts the infrastructures and supports a set of application program interface to cloud applications. Instead, Software-as-a-Service aims at replacing the applications running on PC. In fact, there is no need to install and run the special software on the user's computer if he uses the SaaS.

Cloud computing also is divided into five layers: clients, applications, platform, infrastructure and servers.

Some companies offer CC service, such as: Amazon Elastic Compute Cloud (EC2) [7], Google App Engine [126], Cloudera [69], Sales force's CRM [76] and Microsoft's Azure Services Platform [198]. For example, Cloudera provides Apache Hadoop-based software, and it is increasingly used in cloud computing deployments due to its flexibility with cluster-based, data-intensive queries and other tasks.

Evangelinos and Hill in [97] have described the application of HPC standard benchmarks to Amazon's EC2 cloud computing system, in order to explore the utility of EC2 for modest HPC style applications. They find that this cloud system is emerging as a credible solution for supporting responsive on-demand, small sized, HPC applications.

2.4.2 Grid Computing

According to *Foster et al.* in [106], *Grid Computing* (GC) is composed by hardware and software infrastructure which offer a cheap, distributable, coordinated and reliable access to powerful computational capabilities. Grid computing refers to cooperation of multiple processors on more machines, and its objective is to boost the computational power in the fields which require high capacity of the CPU [135]. Grid computing is a form of distributed computing that involves coordinating and sharing computing, application, data and storage or network resources across dynamic and geographically dispersed organizations [185]. GC is a promising technology for future computing platforms and is expected to provide easier access to remote computational resources that are usually locally limited.

According mainly *Hashemi and Bardsiri* in [135], the main features of GC are: large-scale, geographical distribution, heterogeneity, opening, resource sharing, multiple administrations, concurrency, resource coordination, transparent access, ubiquity, dependable access, consistent access and pervasive access. They are described in following:

Large-scale A grid must be able to deal with a variable number of resources ranging from just a few to millions. In addition, new resources can be added to the infrastructure at any time.

Geographical distribution A GC infrastructure also allows variety of geographically distributed resources to be shared and aggregate.

Heterogeneity of software and hardware resources Different hardware and software solutions can be used.

Opening Each subsystem is a system unto itself and can be accessed directly.

Resource sharing The resources in a grid belong to many different organizations that allow other organizations or users to access them. A GC system simplifies the collaboration between people and resources from different organizations.

Multiple administrations Each organization could establish different access, security and administrative policies.

Concurrency Different processes must be able to work on different nodes at the same time.

Resource coordination The resources in a grid must be coordinated in order to provide aggregated computing capabilities.

Transparent access The users do not need to know the implementation details. In fact, they must have the perception of having a "virtual supercomputer".

Ubiquity All services must be able to be accessed by a user regardless of where it is.

Dependable access A grid must assure the delivery of services under established *Quality of Service* (QoS) requirements.

Consistent access A grid must be built with standard services, protocols and interfaces thus hiding the heterogeneity of the resources.

Pervasive access and fault tolerance The grid should continue to operate even if one or more nodes fail.

A GC is used in many areas, such as financial operations, online multi-player game, weather forecasting, scientific computations, and so on. Some famous examples of grid are: SETI, BOINC, Folding@home, GIMPS.

Grid Computing technologies will promise to change the way organizations tackle complex computational problems. In the next future, organizations and single users will simply plug into a GC infrastructure in a similar fashion to how they now plug into a grid of electric power. Thus, they would only need to pay for what they use, and not buy expense hardware. In this way, the Grid Computing

provides a kind of on-demand access which improves the productivity using extra resources to solve a specific problem.

2.4.3 Comparisons between Cloud and Grid Computing

Hashemi and Bardsiri in [135] have also discussed of the difference between Cloud and Grid Computing. Instead of a few clients running massive, multi-mode jobs, the Cloud Computing services thousands or millions of clients, typically serving multiple clients per node. These clients have small, fleeting tasks (*e.g.*, database queries or HTTP requests) that are often computationally very lightweight but possibly storage or bandwidth intensive. Indeed, in Cloud Computing there are many jobs with short amount of work, while in the Grid Computing there are little jobs with a big amount of work. The goal of GC is the collaborative sharing of resources, while in the CC the goal is the use of service. The grid is focused on computational intensive operations, while the cloud is centered on standard and high-level instances. In the grid there are few users, while in the cloud there are more users. The grid services are not real-time, while the cloud uses real-time applications. In the GC the virtualization is not a commodity, while in the cloud is vital. In fact, the grid infrastructure can use any operating systems (OSs), while a cloud infrastructure uses a hypervisor on which multiple OSs are run. In the grid the security is low, while in Cloud Computing there are high levels of security. Mostly networks with latency and low bandwidth are used in the grids, while the cloud uses dedicated, high-level with low latency and high-bandwidth. The resource management, allocation and scheduling are distributed in the grid, but in the cloud they can be either centralized or distributed. In addition, in GC the failure management is limited (*e.g.*, the failed tasks and applications are restarted), while in the cloud is strong (*e.g.*, virtual machines can be easily migrated from one computer to other).

Figure 2.2 taken from *Foster et al.* [107] shows an overview of the relationship between Clouds Computing, Grid Computing and other subdomains in distributed systems.

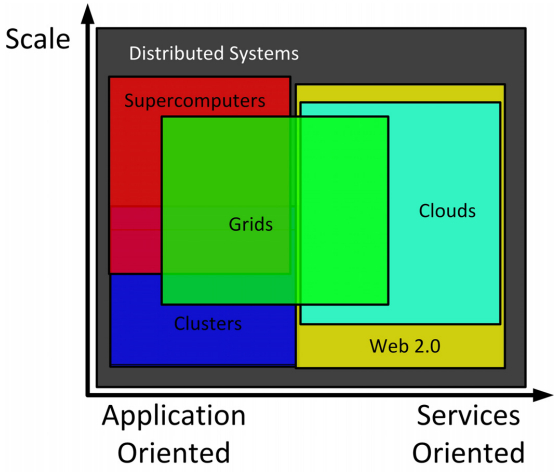

Figure 2.2: Overview of Grids and Clouds Computing in Distributed Computing. The image is taken from [107].

2.4.4 Mobile and Ubiquitous Computing

Nowadays new forms of distributed computing are used, such as: *Mobile Computing* (MC) and *Ubiquitous Computing* (UC).

Mobile Computing, also known as location-aware (or context-aware) computing, enables the use of a computer device even they are moving around. A user can continue to use the resources in their home such as printer, disks, etc., while he is touring the world. Mobile Computing allows the transmission of data, voice and video via a computer or any other wireless enabled device without having to be connected to a fixed physical link.

Ubiquitous computing is a new area of distributed computing that penetrates in life of users enabling devices and computers become helpful. For example, an user can remotely control appliances in home through own smartphone, and an appliance will notify the user when a job is finished.

2.4.5 Current Technologies

The assembling of a cluster of computers is just the first step for solving the problem of processing Big Data. The subsequent problem is the choice of the middleware solution to use for storing and processing data. There are several options to this end. A first distinction is about commercial and free software.

On the commercial side, a popular choice is the MATLAB scientific computing environment, which offers two modules performing parallel computing: *Parallel Computing Toolbox* [189] and *Distributed Computing Server* [188]. Usually commercial software are easier to use, provide a support service and are organized in such a way to minimize the configuration efforts. These advantages may come at a high cost as these tools are usually very expensive. On the free-software side, there are several options available for running a cluster able to process Big Data. These tools may vary according to the adopted computing paradigm and to degree of effort that the researcher has to put in order to deploy the cluster.

Dobre and Xhafa in [88] review various parallel and distributed programming paradigms, analyzing how they fit into Big Data era, and they also present modern emerging paradigms and framework. The authors have said that there is much similarity between parallel and distributed framework, with both supporting message-passing with different properties. The hardware support of parallelism varies from shared-memory multi-core, closely coupled clusters, and higher-latency (possibly lower bandwidth) distributed systems. The coordination, the communication and the synchronization of the different execution units vary from threads with shared-memory on multi-core systems, MPI (between cores or nodes of a cluster), workflow or mash-ups linking services together, and the new generation of data-intensive programming systems based on MapReduce.

Nowadays, both the parallel MPI-based parallelism and the distributed frameworks are implemented by message-passing, in fact this mechanism avoids many errors related to shared-memory threads synchronization. MPI gives excellent performance and ease of programming for MapReduce, in fact it has elegant support for general reductions. However, it does not have the fault tolerance, the flexibility and other high-level features of the current MapReduce-based middleware solutions.

Many of tools operating on Big Data can be thought of as Single Program Multiple Data (SPMD) [78] paradigm or a collection thereof. These programs can be implemented using different parallelization techniques such as threads, MPI, MapReduce, and mash-up or workflow technologies, yielding different performance and usability characteristics.

In the following of the section, are briefly reviewed some modern and popular

Big Data processing frameworks, such as MapReduce and Hadoop, Pig, HBase, Hive and Spark.

MapReduce Paradigm and Hadoop In the recent years, several different architectural and technological solutions have been proposed for processing big amounts of data. An increasingly popular computing paradigm is MapReduce ([81, 82]), which is designed for processing Big Data exploiting distributed resources. MapReduce is appropriate to solve embarrassingly parallel data-intensive problems. Although MapReduce-based frameworks can differ in design and the programming models that they provide, they share similar objectives, such as hiding many problems of parallel programming, providing fault tolerance and execution improvements from the developer. In theory, a programmer can typically continue to write sequential programs. In fact, the processing framework takes care of distributing the program among the available computing units, and it executes each instance of the program on the appropriate chunk of input data.

In particular, a programmer can develop distributed algorithms just defining two functions: *map* and *reduce*. Assuming the input is organized as a set of $<key, value>$ pairs, a generic map function takes as input one of these pairs and returns, as output, a set of intermediate $<key, value>$ pairs. A reduce function is then used to process all the intermediate pairs having the same *key*. Generally, it aggregates all the values with the same *key*, producing a single new value.

The MapReduce paradigm is detailed in Section 3.1, and its most popular and used implementation, Apache Hadoop [14], is deeply described in Chapter 3.

In the MapReduce paradigm, the programmer writes the programs in a "low-level" language in order to perform record-level manipulation on chunk of data. However, application written in higher-level languages (*e.g.*, SQL) are easier to write, modify and understand, also for inexpert programmer (*e.g.*, biologist, chemist, astronomer, etc.). In fact, the MapReduce community is migrating high-level languages on top of the current MapReduce interface to move such functionality into the run time. These domain-specific high-level languages, developed on top of the MapReduce paradigm, hide some of the complexity from the programmer, permitting to focus on the analysis and the application logic. For example, Apache Pig [16] and Apache Hive [12, 266] are two projects oriented

in this direction.

Pig Apache Pig is a composed by high-level dataflow language, called Pig Latin [213], and its execution framework. Pig Latin has the following key properties: ease of programming, optimization opportunities and extensibility. The Pig compiler produces sequences of Hadoop MapReduce jobs starting from a set of high-level instructions. Pig is designed for batch processing of data, and offers SQL-style high-level data manipulation constructs, which can be assembled in an explicit dataflow and interleaved with custom MapReduce functions.

However some Pig applications may suffer for lack of domain-specific and application-specific improvements.

Hive Apache Hive is a data warehouse built on top of Hadoop framework for providing data summarization, query and analysis. They provide querying and managing large datasets residing in distributed storage adopting a SQL-like language called HiveQL. In addition, when it is inconvenient or inefficient to express this logic adopting this language, a programmer can plug map and reduce code using HiveQL.

HBase Apache HBase [15] is an open-source, distributed, versioned and non-relational database modeled after Google's Bigtable [58]. HBase provides Bigtable-like capabilities on top of Hadoop and its file system, with the goal of hosting very large tables atop clusters of commodity hardware. HBase is used when is required random, real time read/write access to Big Data.

Spark Some MapReduce applications are built around an acyclic dataflow model. Apache Spark [17, 304] is a framework that supports the applications adopting this model and in-memory computing, while retaining the scalability and fault tolerance of MapReduce applications. These classes of applications reuse a working set of data across multiple parallel operations (*e.g.*, iterative machine learning algorithms, interactive data analysis tools, clustering, machine learning, computer vision and generic iterative algorithms).

OpenStack OpenStack [215] is an open-source software platform for Cloud Computing, released under the terms of the Apache License, mostly deployed as an Infrastructure-as-a-Service (IaaS). It is useful for creating private and public clouds. In fact, OpenStack software controls large pools of computing, storage, and networking resources throughout a data center, managed through a dashboard or via the OpenStack API.

OpenStack works with popular enterprise and open source technologies making it ideal for heterogeneous infrastructure. OpenStack Data Processing (Sahara) provides a simple means to provision a data-intensive application cluster (Hadoop or Spark) on top of OpenStack. For example, this enables users to set up a multi-node Hadoop cluster using OpenStack and to run Hadoop applications.

Google Cloud Dataflow Nowadays Google is launching a new framework called *Cloud Dataflow* ([147, 125]) to be useful to analyze live data. It allows to create data pipelines for ingesting, transforming and analyzing arbitrary amounts of data in both batch or streaming mode. In MapReduce paradigm, the data analysis is in batch mode, that is all the data must be collected before it can be analyzed, while in Google Cloud Dataflow it is possible to build complex pipelines and analysis. For example, it could also be used to build a ETL system by specifying data pipeline.

Cloud Dataflow is a fully-managed cloud service and programming model for batch and streaming Big Data processing. It has a unified programming model and a managed service for developing and executing a wide range of data processing patterns including ETL, batch computation and continuous computation. This solution provides programming primitives such as powerful windowing and correctness controls. It effectively eliminates programming model switching cost between batch and continuous stream processing by enabling developers to express computational requirements regardless of data source.

2.5 Performance Measurement in Parallel and Distributed Environments

Moret in [201] states that is important to discover and analyze the speed up achieved by parallel algorithms on real machines. In this section are described the most used *performance metrics* used to measure a parallel/distributed implementation respects to sequential (non-parallel) one. In particular, *Kaminsky* in [154] has presented an accurate description of the measure performance in a parallel system according to some major contributions in this area (*e.g.*, [8, 132]).

Let a given program and N be the problem size, *i.e.*, the amount of computation that this program has to do. This application can run on a certain number K of processor units (*e.g.*, cores or computers)[1], therefore, the sequential version of the program run with $K = 1$. The running time T of a program is the wall clock time that it takes to run from start to finish. The running time $T(N, K)$ depends on the problem size[2] N and the number of processors K.

T_{seq} indicates the running time of the sequential version of the program, while T_{par} is the running time of the parallel version. In fact, the sequential version of the program can be different and more efficient respects to the parallel version executed on a single-processor.

Scaling refers to running the program on increasing numbers of processors. There are two ways to scale up a parallel program onto more computing units: *strong scaling* (called *speed up*) and *weak scaling* (called *size up*).

In strong scaling, the number of cores increases while the problem size is fixed, therefore the program should ideally take $1/K$ the amount of time to compute the results for the same problem in sequential setting. However, there are portions of the program that cannot be parallelizable, and overheads related to synchronizations and communication costs.

In weak scaling, as the number of cores increases, the problem size is also increased in direct proportion to the number of processing units. This means that the program should ideally take the same amount of time to compute the

[1] In this section, the terms: cores, computing units, processors, nodes and computers are used interchangeably.

[2] The problem size is supposed to be defined so that T is directly proportional to N.

answer for a K times larger input to the problem.

2.5.1 Speed up

If we add more CPU cores or computers, we should be able to solve faster a problem of a given size. *Amdahl's Law* [8] pointed out the limit on the speed up of a parallel program as it runs on more processors while solving the same problem:

$$Speed\,up(N,K) = \frac{1}{1-p+\frac{p}{K}} \leq \frac{1}{1-p}\,, \tag{2.1}$$

where p is the fraction of the program that can be parallelized, K is the number of processors and N is the input size. The theoretical limit is the reciprocal of the sequential fraction of the program (*i.e.*, $1/1-p$).

The *computation rate* (or *computation speed*), denoted $R(N,K)$, is the amount of computation per second the program performs:

$$R(N,K) = \frac{N}{T(N,K)}\,. \tag{2.2}$$

Speed up is the main metric for measuring strong scaling, and it is the ratio between the computational speed of the parallel program and the computational speed of the sequential program:

$$Speed\,up(N,K) = \frac{R_{par}(N,K)}{R_{seq}(N,1)}\,, \tag{2.3}$$

that is:

$$Speed\,up(N,K) = \frac{T_{seq}(N,1)}{T_{par}(N,K)}\,. \tag{2.4}$$

Note that the numerator of Equation 2.4 is the running time of the sequential version (or *ideal program*) executed on a single-core, not the version of the parallel performed on a single-core. Ideally, the speed up should be equal to K, *i.e.*, number of computing units, but due overheads and non-parallelizable code, this value is less than K. Amdahl's Law places a limit on the speed up that a parallel program can achieve under strong scaling, where the same problem is executed on more processing units. As previously stated, this limit is the reciprocal of

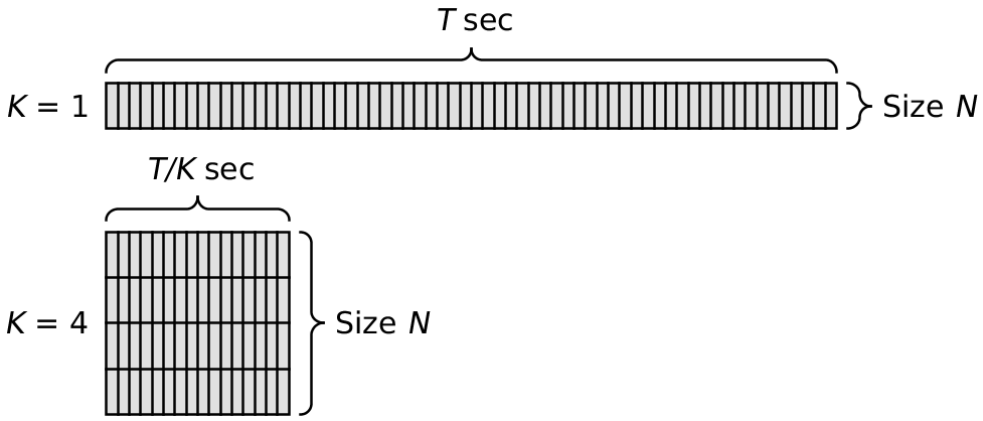

Figure 2.3: An example of Speed up analysis. The image is taken from [154].

the sequential fraction (*i.e.*, non-parallelizable) of the program. In fact, if this fraction is substantial, the performance limit is severe.

Efficiency is a metric that tells how the speed up is close to ideal:

$$Efficiency(N, K) = \frac{Speed\,up(N, K)}{K}\,. \tag{2.5}$$

If the speed up is maximum (*i.e.*, speed up is K), then the efficiency is 1 (regardless of the number of processors). If the speed up is less than ideal, then the efficiency is less than 1.

Figure 2.3 shows an example of Speed up analysis. A fixed sequential program on input of size N was finished in T seconds, while its concurrent version on $K = 4$ computing units was finished in T/K seconds.

During the experimental phase of a parallel/distributed program, it is appropriate to fix the size of the input dataset N and to increase the number of computing units K to perform speed up measurements. This methodology is the most applied in parallel and distributed environments.

In Chapters 4 and 5 we will show some examples of speed up analysis.

2.5.2 Size up

If we add more processors or computers, we should be able to increase the size of a problem that we can solve in a given amount of time. *Gustafson* [132] pointed

out that there is another way to measure the performance of a parallel program. It is appropriate to increase the size of the problem being solved when the number of computing units is increased. In fact, often a user do not want to solve the same problem more quickly increasing the computation units, rather, he want to solve a larger problem in the same amount of time. This methodology is known as weak scaling.

Gustafson asserted that as the problem size increases, the running time of the parallel portion of the program increases, and the running time of the sequential portion of the program typically remains the same. Consequently, as the number of computing units increases and the problem size also increases proportionately, the sequential portion occupies less and less of the total running time of the program. Therefore, the speed up continually increases without hitting the limit imposed by Amdahl's Law (which applies only to strong scaling). However, the running time of the sequential portion of the program does not always stay the same. In particular, the running time of at least some of the sequential portion could also increase as the problem size increases, *e.g.*, the sequential portion uses I/O operations.

Here it is assumed that as the number of computing units K increases then the problem size $N(K)$ will also increase, *i.e.*, N increases in direct proportion to K (*e.g.*, $N(K) = K \times N(1)$).

The *computation rate* is the ratio of problem size to running time:

$$R(N, K) = \frac{N(K)}{T(N(K), K)} \,. \tag{2.6}$$

Size up is the main metric for measuring weak scaling, and it is the ratio between the computation rate of the parallel program and the computation rate of the sequential program:

$$Size\,up(N, K) = \frac{R_{par}(N, K)}{R_{seq}(N, 1)} \,, \tag{2.7}$$

that is:

$$Size\,up(N, K) = \frac{N(K)}{N(1)} \times \frac{T_{seq}(N(1), 1)}{T_{par}(N(K), K)} \,. \tag{2.8}$$

The numerator of Equation 2.8 involves the running time of the sequential version on a single-processor, not the parallel version. If the problem size of the parallel

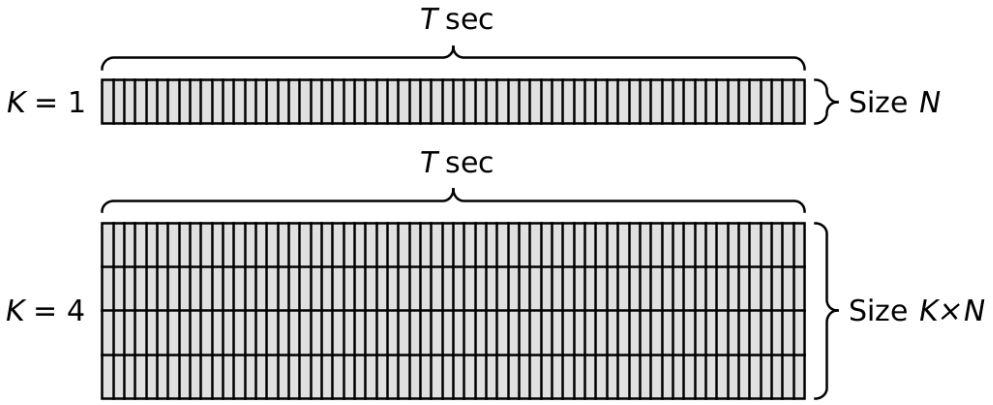

Figure 2.4: An example of Size up analysis. The image is taken from [154].

program is the same as the problem size of the sequential program (which is the case under strong scaling), *i.e.*, $N(K) = N(1) = N$, then the first factor in Equation 2.8 becomes 1, and the size up formula reduces to the speed up formula (see Equation 2.4).

Efficiency is a metric that tell how close to ideal the size up is:

$$Efficiency(N, K) = \frac{Size\,up(N, K)}{K}\,. \tag{2.9}$$

If the size up is ideal, the efficiency is 1, regardless of the number of computing units. If the size up is less than ideal, the efficiency is less than 1.

Figure 2.4 shows an example of size up analysis. A fixed sequential program on input of size N was finished in T seconds, while its concurrent version on $K = 4$ computing units and input of $K \times N$ size large was also finished in T seconds. Size up measures if x-fold larger systems can perform x-fold larger jobs in the same running time as the original system.

In size up analysis, a user must fix the number of computing units and the size of the input dataset is incremented.

Size up and speed up are measuring essentially the same phenomenon, *i.e.*, the ratio of computation rates, but in different contexts. Suppose the problem size of the parallel program running on K computing units is K times larger than the problem size of the sequential program running on a single computing unit. Ideally, then, the parallel and sequential running times should be the same, and

the size up should be equal to K, the number of computing units.

2.5.3 How to Improve the Performance

In the previous we have analyzed the main performance metrics (or indices) used to measure the scalability of a distributed or parallel implementation respects to sequential implementation. Assuming that the sequential program uses efficiently the available resources and its code is "optimal", an user could measure the speed up of the corresponding parallel implementation. Although the related speed up could be approximately linear increasing the number of computing nodes, its *efficiency*, as expressed in Equation 2.5, could be far from the maximum due synchronizations, communications, resource contentions, I/O, sequential code parts, overheads, code problems, memory leaks, memory trashing, and so on.

A parallel or distributed implementation is scalable when increasing the number of computing units, the execution times linearly decrease using the same input. However, a parallel or distributed implementation is all the more "*efficient*" when its efficiency, as expressed in Equation 2.5, approaches to 1. In fact, a parallel/distributed program could be scalable, but not efficient. *Hansen* in [133] has said: "*Efficient programs save time for people waiting for results and reduce to the cost of computation*". Indeed, the goal of an efficient implementation is to reduce resources consumption and completion time as much as possible. For a programmer could be simple to write a distributed code for a given problem, but it is very difficult to write efficient distributed applications.

When a new parallel application is developed, an accurate stage of profiling and tuning is required to valuate its behavior with the purpose to improve the performance. Some bottlenecks in the parallel code are only discovered in running mode. Therefore, it is useful to monitor all the resources and the instructions, trying to discovery potential problems that could limit the efficiency.

In particular, in Chapters 4 and 5 we describe the methodology and the techniques used to improve the scalability and efficiency of some Hadoop distributed applications in digital image forensics and bioinformatics fields. However, before to show these applications, it is appropriate to survey the MapReduce paradigm and Apache Hadoop framework.

Chapter 3

Apache Hadoop Framework

This chapter presents the MapReduce paradigm ([81, 82]) and Apache Hadoop ([14]) framework, that is the most popular and used MapReduce-based distributed computing solution. In particular, Section 3.1 introduces the MapReduce, while in Section 3.2 is provided an overview on Hadoop. The distributed file system used by Hadoop is covered in Section 3.3. The lifetime of a Hadoop MapReduce application is presented in Section 3.4. The main features of Hadoop are provided in Section 3.5, while the difference between Hadoop Combiner versus a custom solution, that can be adopted to improve some applications, is presented in Section 3.6. Section 3.7 addresses the profiling, the tuning and how to improve Hadoop applications.

3.1 MapReduce Paradigm

As previously stated in Chapter 2, in the recent years, several different architectural and technological solutions have been proposed for processing big amount of data. Multi-core processors require parallelism, but many programmers find it uncomfortable to write parallel programs. One computing paradigm that is becoming popular is MapReduce ([81, 82]), which provides easy programming model for a very large set of problems. In fact, a programmer can use it without experience in parallel programming.

The MapReduce paradigm has first been successfully adopted by Google for

creating scalable, fault tolerance and massively-parallel programs that process large amount of data using large commodity computer clusters. Google and other Internet providers also use MapReduce-based distributed computing to convert information gathered from the users (*e.g.*, search queries, visited web pages, emails, posts) into advertisements targeted at the specific user. Nowadays many problems can be solved using a MapReduce-based framework exploiting a cluster of computers. For example, Google uses MapReduce per many tasks, such as: *wordcount*, *adwords*, *pagerank*, indexing data, text-indexing, reverse indexing, and so on. In 2013 Facebook alone used the world's largest Hadoop cluster (see [196]). In fact, just one of several Hadoop clusters operated by Facebook spans more than 4,000 machines, and it has saved over 100 petabytes of data.

MapReduce has now gained a wider audience and it is used in several fields, such as astronomical data processing (*e.g.*, [124, 295, 296]), bioinformatics (*e.g.*, [90, 105, 171, 190, 192, 207, 226, 241, 244, 288]), image and video processing (*e.g.*, [261, 298]), text analysis and document categorization/clustering (*e.g.*, [72, 79, 96, 158, 159, 307, 308]), network traffic measurement and analysis (*e.g.*, [65, 167, 168]), sorting big amount of data (*e.g.*, [214]), data mining (*e.g.*, [162]), computational mathematics, scientific computation, weather prediction, climate modeling, astrophysics, chemistry, geology, engineering computation, computational finance, web indexing, user characterization, targeted advertising, security and cryptography, password cracking, computer games, and so on.

As said in Chapter 2, MapReduce is a computing paradigm designed for processing Big Data exploiting distributed resources. Its main advantage is the possibility to develop distributed algorithms able to scale on large cluster by just defining two functions: *map* and *reduce*[1]. Assuming the input is organized as a set of $<key, value>$ pairs, a generic map function takes as input one of these pairs and returns, as output, a set of intermediate $<key, value>$ pairs. A reduce function is then used to process all the intermediate pairs having the same *key*. Generally, it aggregates all the values with the same *key*, producing a single new value. Map and reduce functions are user-defined, and they are executed, as tasks, in a concurrent way by workers running on the CPU cores

[1]The *map* and *reduce* primitives were present in many programming languages, such as Lisp.

of the nodes of a MapReduce-compliant distributed system. A map task is also called *mapper*, and a reduce task is also named *reducer*. Differently from traditional paradigms, such as explicit parallel constructs based on message-passing, the MapReduce paradigm allows for implicit parallelism. The communication between workers running map functions and workers running reduce functions is accomplished in an automatic and transparent way by the underlying framework. This allows the programmer to focus on the definition of the map and reduce functions, while allowing to define all the aspects related to the execution in a distributed setting (*e.g.*, the number of concurrent map and reduce tasks to issue) through the proper definition of configuration variables. This means that all the operations related to the exchange of data between the tasks involved in a computation are modeled according to a $<key, value>$ file-based approach, and they are transparently accomplished by the underlying MapReduce framework. In particular, many activities are in charge of the MapReduce middleware, such as: synchronization, communications, data distribution, scheduling, parallelization and automatic distribution of the workloads, load balancing, data replication, fault tolerance, redundant execution, data locality computation, status and monitoring of the cluster. In fact, the programmer may only be focused on defining the behavior of the map and reduce functions, and on deciding how data will feed the corresponding map and reduce phases, while, in general case, no particular skill in parallel and distributed systems is required. Figure 3.1 shows an overview of a MapReduce execution.

MapReduce very well works in contexts where there are very large data items which can be processed one by one. In fact, MapReduce was designed for parallel processing massive datasets, that is, data can be broken apart into discrete pieces that can be simultaneously processed. We remember that in parallel computing there is a class of problems, called embarrassingly parallel, where little or no effort is required to split the work into a number of subtasks that can be simultaneously solved. In addition, there also are problems with some subtasks which depend on the results of a few other tasks. However, in the opposite case, there are the non-parallelizable problems, where, for any parallel algorithm resolving a such problem, no speed up may be achieved by utilizing more than one CPU core. In fact, some problems are non-parallelizable at all, while others are very

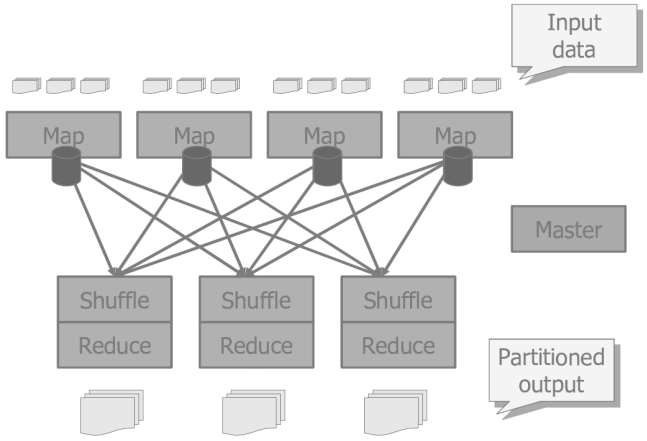

Figure 3.1: An overview of a MapReduce execution.

difficult to efficiently parallelize. In [103] the authors have analyzed which problems can be efficiently modeled using MapReduce and which cannot. Usually, in real scenarios, MapReduce paradigm is used to solve embarrassingly parallel problems.

There are several frameworks implementing the MapReduce paradigm. For example, Disco [86] is a lightweight, open source framework for distributed computing adopting this paradigm. Instead, Dryad [197] implements an extension of the MapReduce paradigm providing an infrastructure that allows a programmer to use the resources of a cluster of Microsoft Windows servers for running data-parallel programs. Dryad combines the MapReduce programming style with dataflow graphs to solve the computation tasks. A Dryad application developer can specify an arbitrary directed acyclic graph to describe the communication patterns of an application, and he expresses the data transport mechanisms between the computation vertices. DryadLINQ is a system and a set of language extensions that enable a programming model for large-scale distributed computing [303]. However, the most popular and used MapReduce framework is Apache Hadoop [14, 18].

Big Data is not necessarily equal to MapReduce, and MapReduce is not needful equal to Hadoop [154]. In fact, *Tudoran et al.* in [270] present *MapIterativeReduce*, an alternative framework which extends the MapReduce programming model to better support reduce-intensive applications, while substantially

improving its efficiency by eliminating the implicit barrier between the map and the reduce phase. The programmers must implement an additional *aggregator* that collects the output data from all reduce jobs and combines them into a single result. However, in applications with a large number of reducers and large amount of data, MapIterativeReduce can prove inefficient. *Yang et al.* in [299] have presented *Map-Reduce-Merge*, a model that adds to MapReduce a *merge* phase that can efficiently merge data already partitioned and sorted (or hashed) by map and reduce modules. This programming model retains the many features of MapReduce, while adding relational algebra (*e.g.*, joins). *Ekanayake et al.* in [94] present *Twister*, an another MapReduce extension, designed to efficiently support iterative jobs. Twister assumes that the intermediate data produced after the map phase of the computation will fit in to the distributed memory.

3.2 Overview on Hadoop

Apache Hadoop ([14, 18, 292]) is currently the most popular and mature framework supporting the MapReduce paradigm. It is a Java-based open source grid computing environment useful for reliable, scalable and distributed computing. From an architectural viewpoint, Hadoop is mainly composed of a data processing framework (called YARN [277] in the second version of Hadoop, *i.e.*, v2.x), plus the *Hadoop Distributed File System* (HDFS) [249]. The data processing framework is mainly based on MapReduce[1] and it organizes a computation as a sequence of user-defined MapReduce operations on datasets of $<key, value>$ pairs. These operations are executed as tasks on the nodes of a cluster. Instead, the HDFS is a distributed file system able to run on commodity hardware and able to provide fault tolerance through replication of data (the HDFS is described in Section 3.3). In addition, the framework provides the Hadoop Common, that is a set of utilities that support the Hadoop and its sub projects. It also includes file system, RPC and serialization libraries.

Apache Hadoop is appropriate to solve embarrassingly parallel data-intensive problems which requires fault tolerance features. It provides scalability, reliability, fault-tolerance, easy deploy-ability, and so on. Hadoop is tailored to manipu-

[1] Hadoop v2.x can use other different paradigms in addition to MapReduce, such as MPI.

late large datasets, in fact, it works more efficiently with large files that requires longer computation time. This framework can be run on a private grid, but the cloud providers already provide easy to install Hadoop services on the cloud environments, *e.g.*, Hadoop on Microsoft Azure [198], Hadoop on Amazon EC2/S3 services and Amazon Elastic MapReduce [6]. In fact, Hadoop is designed to scale from a single node to thousands of nodes, each of which offers local storage and computing resources.

Hadoop is the main framework used to process Big Data. As a matter of fact, it is currently used by many companies in the world, such as IBM, eBay, Twitter, Facebook, Yahoo!, etc. (in [18] is presented the list of institutions that are using Hadoop for educational or production uses). For example, Hadoop is useful to sort big amount of data, to analyze text log, to pre-process raw data and for data mining. However, it is not useful to interactive or on-line processes.

Overview of MapReduce Processing Generally, Hadoop v2.x uses MapReduce paradigm to process data. Coarsely, the input records are split and assigned to map tasks, where each is executed in a concurrent way on a node of the cluster. Each map task processes a subset of input records, and it uses an instance of map function to process a single input record. Therefore, a map function processes an input record $<K, V>$, and it outputs a list of intermediate records *list*$(<K', V'>)$. All the intermediate keys K' are partitioned and allocated to reduce tasks. A reduce task receives a set of intermediate records and it sorts them by key. For each group of keys, the reduce task invokes a reduce function, that synthesizes (or aggregates) groups of records. Therefore, a reduce function processes as input $<K, list(V)>$, and it outputs a new list of records, *i.e.*, *list*$(<K', V'>)$.

Generally, the records with the same key are processed by a single reduce task. The partitioning of the reduce input records is controlled by the Hadoop Partitioner, which establishes the order in which the records in output from the map task (or optionally from Hadoop Combiner), reach a specific reduce task.

In Hadoop the input and output records are files saved in HDFS, while the intermediate records are saved directly on local file system of the nodes. In addition, to save network bandwidth, Hadoop can use the Combiner. It is an

optional component, and it is used as a "local reducer" on the same node where a map task is executed.

3.2.1 The First Hadoop Version

Generally, a v1.x Hadoop cluster consists of a single *master node* and multiple *slave nodes*: the master node runs the `Job Tracker` and the `Name Node` services, while the slave nodes run the `Task Tracker` and the `Data Node` services useful to execute map and reduce tasks. The `Job Tracker` service manages the assignment of map and reduce tasks to the slave nodes, where they will be received and run by `Task Tracker` service. The `Data Node` service manages the HDFS local storage on the node running the `Task Tracker` service. Finally, the `Name Node` service manages the HDFS namespace, by keeping the directory tree of all the files in the distributed file system, and tracking where the file data blocks are kept across the cluster.

3.2.2 The Newer Hadoop Version

The newer version, *i.e.*, v2.x, is mainly composed of two components: a data processing framework called *Yet Another Resource Negotiator* (YARN) [277] and the HDFS. Hadoop YARN replaces the "classic" MapReduce runtime of previous releases, and it is a data processing framework supporting the execution of distributed algorithms through different types of computing paradigms, including MapReduce. So YARN is a framework for job scheduling and cluster resource management, which includes Hadoop MapReduce. The software architectural difference between Hadoop v1.x and v2.x are shown in Figure 3.2.

The `Job Tracker` services has been replaced. In fact, the new architecture separates the two main functions of `Job Tracker` (cluster resource management and job scheduling/monitoring) into two separate components: global `Resource Manager` and per-application `Application Master`. Generally, a v2.x Hadoop simple cluster consists of a single master node and multiple slave nodes: the master node runs the `Resource Manager` and the `Name Node` services, while slave nodes run the `Node Manager` and the `Data Node` services. On the master node, the `Resource Manager` arbitrates the assignment of computational resources to

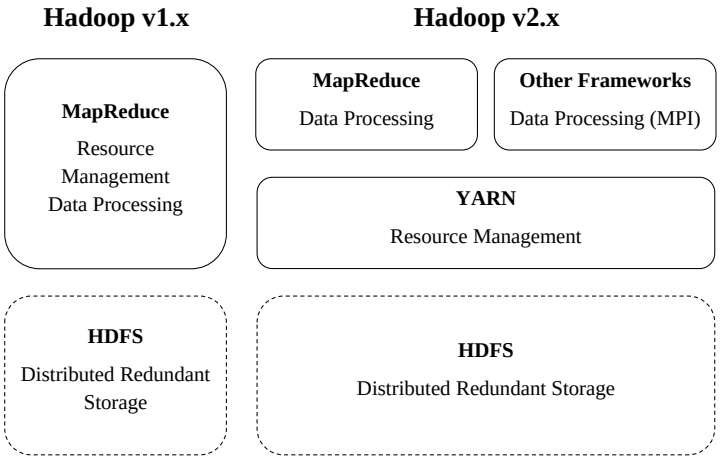

Figure 3.2: The software architectural difference between Hadoop v1.x and v2.x.

applications, and it is a global resource scheduler. On the slave nodes, the `Node Manager` monitors and keeps informed the `Resource Manager` about the the status of the node. Again, on the master node, the `Name Node` service maintains the directory tree of all files existing in the HDFS and keeps tracks of where data blocks are physically placed. On the slave nodes, the `Data Node` service maintains a subset of the HDFS data blocks using the local storage.

A Hadoop *Application* can be a set of MapReduce jobs, in particular, it can be or a single job (MapReduce job classic) or a *Directed Acyclic Graph* (DAG) of such jobs. These applications are run on Hadoop via an `Application Master`, that is a service instantiated by a `Node Manager` on a slave node upon a request coming from the `Resource Manager`. Once created, it asks the Hadoop framework for all the resources required to perform a computation (mainly in terms of CPU and memory). The `Resource Manager` responds by reserving to the application a set of `Containers` (also called workers in this thesis), each being the basic processing unit in Hadoop to execute a map or reduce task. A `Container` owns a number of CPU cores and an amount of RAM, and, on a multi-core slave, more concurrent workers can be run in parallel which execute map/reduce tasks. Therefore, each running Hadoop application has its own `Application Master` that manages application scheduling and executing tasks.

An overview of YARN services in Hadoop is shown in Figure 3.3.

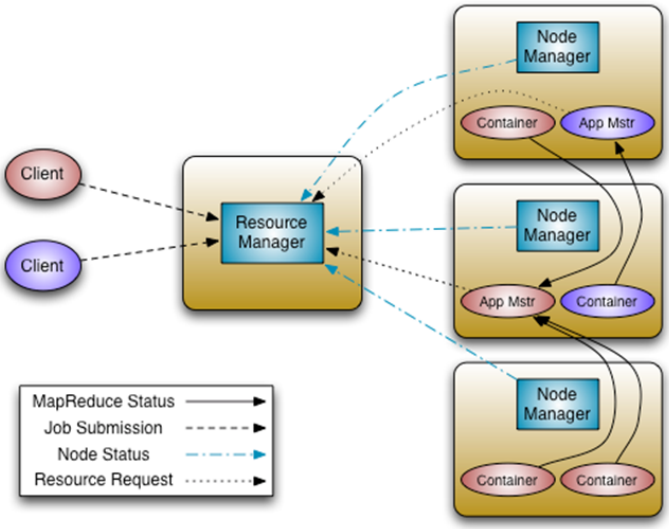

Figure 3.3: An overview of the YARN services in Hadoop. The figure was taken from [13].

3.3 Hadoop Distributed File System

In a distributed system it is often a more efficient approach to run a task on local data, rather than move the data where the task is running. As a matter of fact, one of the main characteristics of Hadoop is its ability to exploit data local computing. In fact, the Hadoop Distributed File System (HDFS) [249] provides functionality to enable applications to move closer to the data, minimizing network congestion and increasing the overall throughput of the system. In other words, it is the ability of the framework to run the computations on the node hosting the data to be processed. This allows to significantly reduce the overhead typically spent in distributed applications to move data over the network.

In particular, the HDFS is a distributed and block-structured file system, able to also run on commodity hardware and able to provide fault tolerance through replication of data. Indeed, it was inspired by *Google File System* (GFS) [115]. The HDFS is able to reliably maintain very large files, in fact, it works by automatically splitting large files in smaller blocks and spreading them across nodes in a cluster. Each file is a sequence of blocks, and all the blocks in the file, except the last, are the same size (referred to as HDFS *block size*). The default HDFS block size (BS) is 64 MB for Hadoop v1.x and 128 MB for Hadoop

v2.x. Fault tolerance is guaranteed by a replication strategy that requires each block to be replicated and stored on different nodes, according to a *replication factor* (RF). In addition, block size and replication factor are configurable for each file. One the one side, increasing the RF will increase as well the chances for a task to be run on a node hosting the data it requires. On the other side, a high replication factor implies a larger performance overhead to be spent initially for writing different replicas of a same file.

3.3.1 HDFS Architecture

The HDFS has a *master/slaves* architecture. A simple cluster consists of a `Name Node`, that is a master server that manages the *file system namespace* and it rules the client access to files. There are many `Data Nodes`, usually one for each slave node in the cluster. Each `Data Node` handles the local storage where such service is running. Moreover, it serves read or write requests, and it creates, deletes, and replicates the blocks on instruction of `Name Node`. For the nature of the HDFS, the relationships between keys can be defined only within the MapReduce application, not by HDFS.

`Name Node` Service The HDFS is a hierarchical file system with files and directories. The `Name Node` maintains the file system namespace. In particular, the `Name Node` keeps in memory the image of the entire namespace of the file system and the *file blockmap*, that is the directory structure of all files in the file system and the reference to the nodes where the file blocks are contained. `Name Node` supports copying, deleting and moving the file operations. It also saves the number of replicas that the file requires (called *file replication factor*).

Any change the file system meta information is recorded by `Name Node`. It uses a *transaction log* called *EditLog* to record every change that occurs to the meta-data in the file system. The EditLog is stored in the local file system of `Name Node`. The entire file system namespace (including the mapping of blocks to files and file system properties) is stored in a file called *FsImage*, saved on the local file system of the `Name Node`. When the `Name Node` starts, it picks up FsImage and EditLog from own local file system. Then, it update FsImage with

information from EditLog. It saves a copy of FsImage on the file system as a checkpoint. Periodic checkpointing is done, so the system can restore everything to the last checkpoint in the event of a crash.

If the `Name Node` goes offline, the whole system goes offline. This problem is solved using a `Secodary Name Node` service and/or *NameNode High Availability* (NN HA) functionality in Hadoop v2.x (see Section 3.3.2).

`Data Node` **Service** A `Data Node` stores in the own local disk the files related to HDFS blocks. It makes reading, writing, movement and replication of files at the request of `Name Node`. Once the `Name Node` has provided the location of a file block, clients can connect directly to `Data Node` for data transferring. The `Data Node` has not information about the HDFS, in fact each `Data Node` stores a HDFS block as an unknown file. When the HDFS services start, a `Data Node` generates a list of all blocks (called *Blockreport*) and it submits this report to the `Name Node`. The `Data Node` provides blocks of data through the interconnection network using a specific block protocol. In addition, they communicate among themselves to balance data, to move copies between them and keep a high data replication.

Block Replication The HDFS blocks are replicated for fault tolerance. The HDFS uses the *replication pipelining*, that is when a client receives responses from `Name Node`, it sends its block into small pieces to the first replication, which in turn copy it at the next replication, and so on. In this way the data are *pipelined* from `Data Node` to the next. The need for "re-replication" may result from: a `Data Node` may be unavailable, a replica can become corrupt or a hard drive on a `Data Node` can fail. The HDFS architecture is compatible with data balancing schemes. A schema could move data from one `Data Node` to another if the free space on a `Data Node` falls below a certain threshold.

3.3.2 HDFS Main Features

The HDFS is designed to be used on low-cost hardware, in fact, one of the main objectives of HDFS is the recovery from the hardware failure. Fault detection

and fast automatic recovery are core architectural goals of the HDFS.

In the following are summarized the main features and goals of the HDFS.

- *High fault-tolerant: detection and error recovery due to failure*
  In a distributed system, the failure is the norm rather than exception. Each `Data Node` sends a heartbeat message periodically at `Name Node` which is used to notify the state of `Data Node`. The failures of the `Data Nodes` are manged through the data replications. Also meta-data disk failure are managed using backup copies. In addition, *snapshots*[1] can restore previous HDFS state if severe malfunctions are experienced. Other functionalities are added in Hadoop v2.x.

- *Data Integrity*
  A checksum for each block is made. If the block is damaged, a replica of the block is used.

- *Large datasets*
  The HDFS is suitable for applications with large datasets ranging from gigabytes to terabytes. In fact, it is not suitable for too many small files (see [291]).

- *Batch processing*
  The HDFS is designed for processing in batch mode than interactive user access. It is preferable a high throughput that low latency. This pattern is MapReduce-compatible.

- *Write-once and read-many model*
  HDFS applications require a file access model of write-once and read-many type. Once a file is created, written and closed, it should not be changed. This assumption simplifies the data *coherency* issues and allows access to data with the same high throughput.

- *High throughput*
  HDFS provides high aggregated data bandwidth.

[1] A *snapshot* is the photography of the state of a system at a particular point in time.

- *"Moving computation is cheaper than moving data"*
  Hadoop can use the *data locality execution*, that is the computation is more efficient if the data are taken from local disk than moving data through the network.

- *Portability across heterogeneous hardware and software platforms*
  The HDFS can be built on heterogeneous commodity hardware.

- *Scalability*
  HDFS provides scalability to hundreds or thousands of nodes in the cluster.

- MapReduce *support*
  MapReduce applications (*e.g.*, web-crawler, text analyzer) fit perfectly to HDFS.

New Features The Hadoop v2.x introduces many new functionalities related to the HDFS, such as NameNode High Availability and HDFS Federation. In the following are summarized these features.

- *NameNode High Availability*
  The `Name Node` may be a single point of failure, in fact, a `Secodary Name Node` can be used as backup. In addition, the newer Hadoop version provides NameNode High Availability (NN HA) that uses a set of `Name Nodes` in standby for failover.

- *HDFS Federation*
  With the aim to scale out the name service, the federation uses more independent `Name Nodes` (*i.e.*, namespaces). The `Name Nodes` are federated and independent, and they require no coordination between them. The federation partitions the HDFS namespace between multiple `Name Nodes` to use cluster with a very large number of files.

3.4 Lifetime of a Hadoop MapReduce Application

As previously stated, the execution of a Hadoop MapReduce application takes place in two consecutive (and potentially overlapping) phases: the map phase

and the reduce phase. During the map phase, one or more map tasks are run by `Containers` on the slave nodes of the Hadoop cluster. In general, each `Container` may run one task a time, while several `Containers` may run in parallel on a same slave node. Several `Containers` may be concurrently run on a same slave if there are many unused resources on that node, such as CPU cores, RAM memory, I/O, and so on. When an application is running, a worker (*i.e.*, `Container`) in the cluster is dedicated to the `Application Master` service.

The execution of a map task goes through four phases. At startup, the task initializes the data structures required for managing the input and the output of the task (*init* phase). Here, an important role is played by setup of the input *record reader*. Then, the `Container` begins the execution of the map functions (*map execution* phase). As soon as output pairs are returned, these are saved in a temporary memory buffer. When the buffer gets almost full or when the map functions execution ends, the output pairs are sorted, partitioned according to the destination reduce task, and written on disk (*spilling* phase). Lastly, data belonging to each partition are merged on disk, and moved to the slave nodes where the reduce tasks will process them (*shuffle* or *copy* phase).

The execution of a reduce task requires three phases. At the beginning, all the pairs produced by map tasks and included in a certain partition are moved on the node where the reduce task assigned to that partition will be run (*shuffle* or *copy* phase). As soon as new pairs are received by a node, they are sorted in order to keep them grouped according to their key (*sort/merge* phase). Finally, for each group of the pairs with the same key, a reduce function will be run by the `Container` on that node (*reduce execution* phase).

The change for an execution to be slowed down by a task taking too much time to run is managed in Hadoop through *speculative execution*. In such a case, the Hadoop framework may decide to run the same task on a different node. Then, as soon as one of the two tasks finishes, the other one is killed.

3.4.1 Splitter and Records Reader

The splitter determines how to divide the input into multiple parts. The access to the input files of a program is managed by Hadoop through the implementation

of a proper `InputFormat` used to read the files. In particular, the splitter is a specification used by Hadoop to virtually organize and manage a data source in smaller parts called input splits, where each split will be processed by a distinct map task. The `InputFormat` is also used by map tasks when processing a split to extract all the $<key, value>$ pairs contained within and to be provided as input to map functions.

In particular, the `InputFormat` defines how to read data from a file split in the map tasks. Hadoop has many implementations of the `InputFormat`, for example, some implementations work with text files and they describe different ways of interpreting these files. Other implementations are built to read binary file formats (*e.g.*, `SequenceFileInputFormat`). In general, the main work of a splitter is to divide data files in fragments (*i.e.*, input splits), that are used by the map tasks. Then these splits are divided further into records, which are used one at a time in the map functions. The Java class hierarchy of `InputFormat` is presented in Figure 3.4.

HDFS blocks have not to be confused with input splits. The former refers to a physical organization of the input data. The latter refers to a logical organization of the input data and do not necessarily corresponds to HDFS blocks (*e.g.*, a split organizing an input file as a set of lines of text may require to access two HDFS blocks to complete a dangling line - see Figure 3.5 for an example). The number of input splits reflects the total size of input file divided by the HDFS block size. As a consequence, a same logical split may include several HDFS blocks or a single HDFS block may contain many logical splits.

3.5 Hadoop Main Features

Usually, when a developer writes an application to be run on Hadoop, he does not have to care about the way the data are spread over and maintained in the different nodes of a cluster or transfered. However, in some cases an explicit control over data locality and data management are required to achieve a better performance. In order to cope with these needs, Hadoop offers several facilities.

In the following, we briefly describe the Hadoop functionalities that have been considered during the next chapters. Some concepts have already been treated

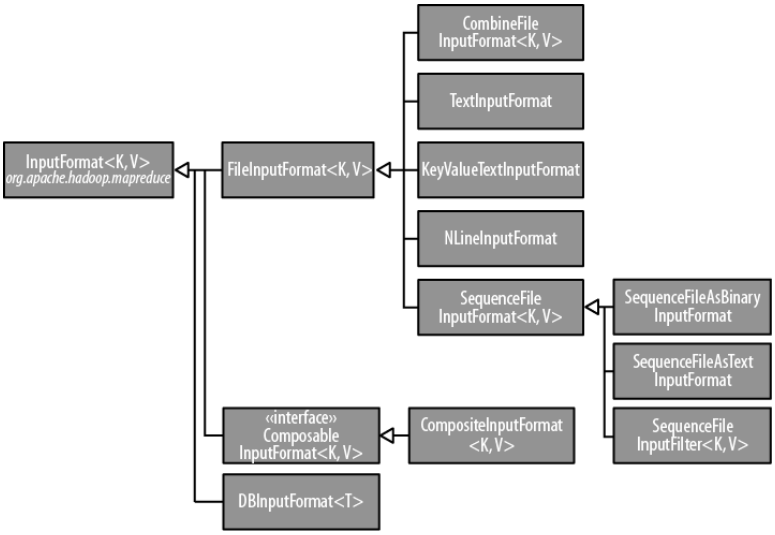

Figure 3.4: Hadoop `InputFormat` class hierarchy. The figure was taken from [292].

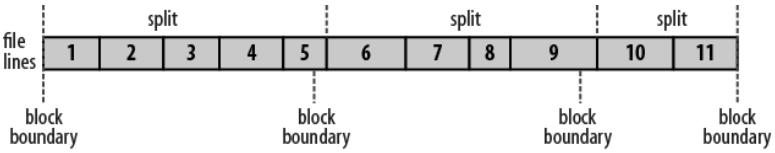

Figure 3.5: Example of logical records and HDFS blocks for a text input file using `TextInputFormat` class. A single file is divided into text lines and the boundaries do not match those of the HDFS block. The split follows the logical boundaries, *i.e.*, the first input split contains the line from 1 to 5, although the line 5 crosses the first and the second HDFS block, while the second input split starts at line 6. This image was taken from [292].

before, but this section gives an emphasis on the possible code improvements exploiting these features.

Data Locality Computation The strategy of moving computation to the data, instead of transferring the data to the computation (slave node) allows to achieve high data locality that translates into high performance. The developer must also write efficient splitter and record reader components to exploit data locality.

Sequence Files The HDFS is known to perform poorly when handling a large number of small files, mainly because of indexing issues (see [291] for details). This problem can be overcome by using the Hadoop *Sequence Files*[1]. These files, implemented through the `SequenceFile` object, store on HDFS sequences of binary $<key, value>$ pairs with arbitrary length, thus working as a container of smaller files. In addition, they can store arbitrary types of data, even compressed. Moreover, they support a variety of serialization frameworks. Finally, they provide a *input split* primitive that breaks their content in several parts to be distributed in different slave nodes. Consequently, data are put on the same nodes where they will be processed so to reduce the amount of data to transmit (*i.e.*, data locality computation).

Cache File This facility is useful for caching read-only HDFS files on slave nodes, before any task is executed on that node. This may help to reduce the I/O network activity during the execution of a task and related to the processing of read-only files (*e.g.*, a library and shared files), at the expense of an initial slower start-up. In Hadoop v1.x this feature is available through a Hadoop object known as `DistributedCache`.

Speculative Execution Some nodes may be much slower than others (a condition called a *straggler*) due to network problems and/or excessive load, and so

[1]The reader should not confuse a Hadoop *sequence file* with a file containing genomic sequences as described in Chapter 5.

on. Therefore, a task map or reduce can be slow and it could delay the completion of processing. Hadoop speculative execution indicates that a same task can be run multiple times on different slave nodes. As soon as a duplicate task (also called backup task) ends, the other ones are killed. In Hadoop, a *killed task* is usually a task duplicate that is killed due to the mirror task termination (this is generally called speculative execution). In addition, the tasks are also killed when a task does not notify its state of progress within a fixed timeout, or the schedulers FairScheduler or CapacityScheduler need some other slot pool (FairScheduler) or queue (CapacityScheduler).

Data Replication HDFS is designed to reliably maintain very large files across nodes in a cluster. This is done by storing each file as a sequence of blocks, where the blocks belonging to a same file can be replicated several times over different nodes for fault tolerance, according to a replication factor (RF)[1]. As mentioned above, a higher replication factor increases the chances for a task to be allocated over a node hosting the data to be processed. However, it puts a heavy burden on the time needed to write files on the HDFS, as writing operations must be propagated to the replicas according to this factor.

File Deleting When a file is deleted, it is not removed immediately from HDFS, but it is moved to a folder called *trash*. The file remains in this directory for some time (6 hours by default), and then it is deleted permanently. After this timeout all blocks associated with it are freed and cleared out the reference in `Name Node`. Cancellation policies can be configured manually. A client can restore files deleted prior to its final disposal, but the directory *trash* contains only the most recent copy of the file.

Combiner In order to save network bandwidth and local disk I/O, Hadoop implements the Combiner, a mechanism able to reduce the data transferred between map and reduce tasks. It is a user-provided function that is invoked before a map task sends its outputs, with the purpose, for example, of batching and

[1]The default HDFS replication factor is 3.

aggregating parts of the output data of a map task just using in-memory operations, before sending the results to the destination reduce tasks. For some problems, a developer could avoid this generic functionality and he could implement a proper improved local in-mapper aggregation to get better speed up. Section 3.6 describes the difference between the Combiner and a such custom solution.

Partitioner This is responsible for *key-space* mapping to reducers, *i.e.*, it establishes the partitioning in which the records in output from the map task (or optionally from Combiner), reach a specific reduce task. In other words, this component is in charge of deciding which reduce task has to process which $<key, value>$ pair outcoming from the map phase.

Hadoop stores the intermediate results of the computations in local disks, where the computation tasks are executed, and it informs the appropriate reducers to retrieve (pull) them for further processing. The partitioning is typically done by running a hash function on the input *key*. The default Partitioner is the `HashPartitioner`, where the reduce task is chosen through a hash function applied to the key, module the number of reduce tasks. The standard hash function can be replaced by a custom one, in order to gain control of the way the partitioning is made.

3.5.1 The Future of Hadoop

Apache Hadoop is a working progress project, in fact, efforts are made to accept the changes proposed by many users to improve the performance. For example, *Appuswamy et al.* in [21] have described several modifications to the Hadoop runtime that target scale up configuration. These changes are transparent, do not require any changes to application code, and do not compromise scale out performance. At the same time their evaluation shows that they do significantly improve scale-up performance of Hadoop.

Nowadays is spreading Apache Spark [17], a fast and general engine for large-scale data processing. It run programs up to 100× faster than Hadoop MapReduce in memory, or 10× faster on disk. Many Spark applications are run on

Hadoop clusters.

3.6 Hadoop Combiner versus In-Mapper Local Aggregation

As explained in Section 3.5, Hadoop allows to use a Combiner function to aggregate the map output pairs. The aggregated pairs of the Combiner are the input to the reduce functions. In fact, if a Combiner function is specified, it will be run by a map task to reduce the amount of data written to disk, and this function may be run repeatedly over the input without affecting the final result. The Combiner function is an "optimization", in fact, Hadoop does not provide a guarantee of how many times it will call it for a particular map output pair, if at all [292]. In fact, even if a programmer explicitly sets a Combiner class, Hadoop may or may not run it at all [274]. When a map task with a Hadoop Combiner emits the $<key, value>$ pair, it is collected in a memory buffer, and then the Combiner may aggregate a batch of these $<key, value>$ pairs before sending them to the reduce tasks. Hadoop may also stores the $<key, value>$ pairs in local file system, and run the Combiner later, which will cause expensive disk I/O [269]. Finally, a Combiner only combines data in the same buffer.

Unlike, an *in-mapper local aggregation* (or *in-mapper local combining*) is a explicit solution written by the programmer, which is much more efficient and resource-frugal than a Hadoop Combiner, because it continually aggregates the data in memory. In fact, as soon as it receives two values with the same key, it combines them and stores the resulting $<key, value>$ pair in a custom data structure, *e.g.*, a hash table. However, this explicit solution requires a lot of programming work, which is completely absent using the Combiner.

In-mapper local aggregation solution is extensively used in the solutions presented in this dissertation to improve the performance (see Chapters 4 and 5).

3.7 Profiling, Tuning and Improving Hadoop Applications

Starting from the stand-alone (or sequential) reference implementation, the first thing is to evaluate it, checking if it efficiently solves the problem. Then we need to check, for example, which operations are more expensive in terms of CPU usage, and how often they are invoked. Then, it must be ascertained where we can parallelize the sequential implementation.

In the previous sections it has been announced that Hadoop could be used to easy develop a distributed version of a non-parallel (sequential) algorithm without particular distributed skills. At first glance, an initial distributed solution obtained from a simple "*mapping/porting*" (or transformation), without the internal knowledge of Hadoop, might suffice. Unfortunately, this is not true for many problems and solutions. In fact, a deep understanding of Hadoop could impact positively on the development of an improved distributed application. In general, a distributed implementation is a necessary but not sufficient condition to obtain very good performance and scalable applications.

As will become clear in the next two chapters, a preliminary stage of profiling of a Hadoop-based naive application is required to understand the potential bottlenecks and for trying to improve its performance. For example, a developer must analyze, use, change or try to improve many components or facilities, called *critical points*, such as: the I/O organization, the input splitter, the record reader, the input split size, the number and the granularity of the map tasks, data locality computation, the local aggregation in the map tasks, the key-space partitioning to reducers, the total number of reducers, the number of transferred data between mappers and reducers, the map or reduce functions, and so on.

Therefore, it is required to identify the internal and external factors that may affect the distributed execution. We need to answers to some questions, such as: *What is the maximum number of map and reduce tasks that need to be allocated to each slave node?; How HDFS replication factors and block sizes affect on running times; Whether to compress or not the data; etc.*

In addition, some strategies must be adopted which allow to monitor the cluster to check for any abnormalities during the execution of the Hadoop jobs.

For analyzing the behavior of a MapReduce application is required the monitor of the status of the job on each node of the cluster, for example: local disk I/O, network I/O, memory and SWAP activities, CPU usage, bottlenecks in the sequential code of map/reduce functions, the numbers of transferred/output $<key, value>$ pairs, data locality computation, and so on.

In fact, we should analyze if we can improve the distributed execution monitoring the allocated resources, checking any wastage of time and resources used to manage the cluster. So automated tools which allow to monitor all these parameters on the cluster are required. Therefore, in addition to the Hadoop statistics, tools as Dstat [293] and Java profilers are useful to gather such information. Once highlighted the problems, they must be solved via code modifications or changing Hadoop configuration parameters, so trying to develop efficient solutions. For example, if a data structure is used as in-mapper local aggregation to get better performance, the number of expected initial elements could impact on the execution times (*e.g.*, a hash table continuously doubled).

Many technical choices will need to reflect the current cluster architecture, in addition to a generic cluster. In fact, many design choices used to improve the execution times could be consequences of cluster architecture. We must answer to questions, such as: *How many physical nodes and CPUs are in the cluster; Which hardware/software configuration is used in the cluster; What are the physical limits of the resources (CPU, memory, network bandwidth, I/O throughput)?*

In addition, a tuning phase related to various parameters is required to search further improvements in the distributed code (*e.g.*, [138, 151, 173]).

Finally, we should analyze as the running times vary changing number of computing units in the cluster, for showing the speed up compared to a standalone execution. In general, fixed an input dataset, it is required to vary the computing units in the cluster (such as 4, 8, 16, 32 and 2^x) to show how the execution times change.

In the following chapters are shown as some Hadoop-based naive (or basic) implementations can be improved to getter a better efficiency and, therefore, a good speed up. In particular, we will propose some naive solutions, obtained according the simple Hadoop *porting*, and we will describe in details as they are made fast and efficient thanks to profiling, tuning and improving activities.

Chapter 4

Processing Big Data in Digital Image Forensics

In this chapter is presented the work done for efficiently engineering and experimenting on Hadoop the algorithm by *Fridrich et al.* [181] used to solve the Source Camera Identification problem (*i.e.*, recognizing the source camera used for acquiring a given digital image).

Initially, in this chapter are introduced the issues about the Digital Forensics and Big Data (see Section 4.1), while Section 4.2 addresses how to analyze massive sets of images. The Source Camera Identification problem is introduced in Section 4.3, while our Hadoop solution for the algorithm by *Fridrich et al.* is presented and analyzed in Section 4.4. Section 4.5 concludes this chapter with some observations.

4.1 Digital Forensics and Big Data

Nowadays, the size of digital memories and the services running on the Web are growing exponentially. In addition, data sources are much more differentiated and heterogeneous than in the past. The data collected could be potential sources of *digital evidences* in legal investigations and processes. In fact, it is customary that a legal case can work on data originating from PCs, servers, cloud services, Online Social Networks (OSNs), phones, tablets, digital cameras, GPS devices,

embedded systems, industrial control systems, audio systems, image repository systems, and so on. In addition, the cyber security requires to manage and analyze a large amount of data (*e.g.*, texts, logs, images or video, network information, etc.) to combat the crimes, such as terrorism.

In 2008 the FBI *Regional Computer Forensics Laboratories* (RCFLs) annual report [99] explained that the RCFLs processed 27% more terabytes of data than they did during the preceding year. These laboratories examined 58,609 pieces of digital media of all kinds for a total of 1,756 terabytes of processed data. To extract information from these data, it also required to use sophisticated analysis tools.

Therefore, a field that is also taking advantage of the possibilities offered by the distributed systems to save and process big amount of data is *Digital Forensics* (or *Computer Forensics*). In general, it is concerned with the acquisition and the analysis of digital media in order to find clues while investigating a crime.

In 2001 the *Digital Forensic Research Workshop* (DFRWS) [85, 218] marked the guidelines for the determination of the science of Digital Forensics: "*Digital Forensic Science: The use of scientifically derived and proven methods toward the preservation, collection, validation, identification, analysis, interpretation, documentation and presentation of digital evidence derived from digital sources for the purpose of facilitating or furthering the reconstruction of events found to be criminal, or helping to anticipate unauthorized actions shown to be disruptive to planned operations*".

One of the first contributions in digital forensics field for distributed environments comes from *Roussev et al.* [234]. In their paper, the authors proposed a novel framework based on the MapReduce paradigm that can be used to implement forensic computing techniques in a distributed fashion.

A more specific issue has been addressed by *Raghava and Shelly* in [227]. The authors used the MapReduce paradigm to significantly speed up the matching process of biometric traits using iris recognition. A different application has been shown in [166] where the authors presented a MapReduce model for the efficient indexing and querying of text documents for digital forensics purposes. *Federici* in [100] discussed the design goals, technical requirements and architecture of *AlmaNebula*, a conceptual framework for the analysis of digital evidences built

on top of a cloud infrastructure, and aimed to embody the concept of "*Forensics as a service*".

Guarino in [129] explores the challenges in Digital Forensics and Data Science. He proposes how techniques and algorithms used in Big Data analysis can be adapted to the digital forensics, ranging from the managing of evidence through MapReduce paradigm to machine learning techniques for analysis of big disk images and network traffic dumps. A common task is the attribution of a file fragment (coming from a file system image or from unallocated space) to specific file type.

In [247] is presented a *Network Forensics* system, called *ForNet*. It is a distributed network logging mechanism to aid digital forensics over wide area networks, and it aims to address the lack of effective tools for aiding investigation of malicious activity on the Internet. ForNet builds and stores summaries of network events, based on which queries are answered.

Instead, *Lu et al.* in [180] have proposed a heavy network fingerprint discriminant algorithm. They have implemented it on the top of Apache Hadoop.

In addition to textual information and traffic networks, nowadays, photos and videos accompany us in our daily and personal life. For instance, it has been estimated that about 1.1 billion digital still cameras were shipped worldwide in 2013 [42], and about a billion of cameras were shipped in 2014 [43]. From a common viewpoint, photos and videos have become part of a new communication language, that mixes together the spoken language with multimedia digital contents. This is shown by the enormous amount of digital images exchanged through Online Social Networks (OSNs) and photo sharing websites. In February 2015, for example, the total number of photos uploaded to Facebook was about 400 billion (they were 250 billion in September 2013), while the average number of photos uploaded per user was about 217 images (September 2013) [253].

In fact, in the recent years, a new discipline, called *Digital Image Forensics* (DIF), is born and it is one of the application fields where the problem of processing Big Data is arising. The digital image forensics is focused on the acquisition and the analysis of images (or videos) found on digital devices or on the Web for investigation purposes. This research field is very active, as witnessed by the many contributions proposed in this area. It may be useful, for example,

for establishing if a digital image has been altered after it has been captured (*i.e.*, *Digital Image Integrity*) (*e.g.*, [51, 53, 54, 60, 98, 301]), if it contains hidden data (*i.e.*, *Digital Image Steganography*) (*e.g.*, [59, 110]) or what camera has been used to capture that image (*i.e.*, *Source Camera Identification*) (*e.g.*, [28, 46, 48, 122, 181]).

Today, the manipulation of digital images is simpler than ever thanks to solutions like sophisticated photo-editing software or photo-sharing social networks. Indeed, one of the key characteristics of digital images is their pliability to manipulation. As a consequence, we can no longer take the authenticity of digital photos for granted. This can be a serious problem in situations where the reliability of the images plays a crucial role, such as when conducting criminal investigations. This is often referred to as *Tampered Image Detection* or *Image Integrity* problem. Given an input image, *is it possible to establish whether it has been tampered with or not? That is, can we prove that the image has been modified by any kind of operation or that it exactly corresponds to the camera output?* In fact, there are plenty of tools that allows even an inexperienced user to modify the content of a digital image without leaving a visible trace of alternation. This practice may be harmful if used, *e.g.*, to alter the digital evidences in a criminal trial or to support the spread of false news for political propaganda. In such a scenario, it becomes often important to ensure that a digital image is authentic and has not been subject to any form of manipulation, especially in some application fields such as journalism, criminal investigations and legal matters. This risk is today higher than in the past, thanks to the flourishing of applications and online services for editing and tampering digital images.

Another important problem in digital image forensics is the recognition of the camera that has shot an image, also distinguished between camera of the same brand and model. This problem is well-studied in literature, *e.g.*, [28, 46, 48, 122, 181]. In addition, classification and clustering algorithms are useful to automatically review sets of photos, for instance, to separate suspect images (or parts of a photo) from the rest. These problems are widespread in the investigations to the fight against online child pornography. Indeed, it is required to evaluate if a suspicious image has been taken by the same digital camera used to shoot the photos which appear on an user's album published at a OSN.

Most of these problems have already been discussed in the scientific literature and valuable solutions are usually available. However, with the growth of digital photography, there is required to assess how these solutions scale when dealing with Big Data and/or how their performance can be improved to respond faster to investigators.

For example, *Fridrich et al.* [181] were the pioneers solving the Source Camera Identification (SCI) problem exploiting the camera sensor noise as fingerprint. In fact, many solutions have been developed over the years to solve it. However, with the spread of the images published OSNs, solutions for fast and efficient SCI are required. It is a big task not only to perform these computations, but also to save images and their meta-data.

4.2 Analyzing Massive Datasets of Images

As previously stated, the explosion of images and videos through the OSNs could limit the applicability of the existing algorithms and tools used to verify whether photos (or videos) were forged (*e.g.*, [51, 61, 111]), to semantically cluster and organize images and videos (*e.g.*, [27, 229]) or to identity the source camera or its model (*e.g.*, [120, 121, 181, 264]).

In fact, the way by which performing Source Camera Identification on large datasets, has not received much attention in the scientific literature. One of the few contributions in this area, *i.e.*, [122], presents a large-scale test of SCI from sensor fingerprints. The authors tested over one million images spanning $6,896$ individual cameras covering 150 models, and used an improved version of the *Fridrich et al.* algorithm. The only piece of information available about the experimental setting they chose concerns the usage of a cluster of 40 2-core AMD Opteron processors, where 50 cores were devoted to this application. Nothing is said about the changes necessary to run the algorithm in a distributed environment and about its performance compared to its non-distributed counterpart.

Another contribution describing a large-scale experimentation of a Source Camera Identification algorithm is presented in [123]. The authors describe a fast searching algorithm based on the usage of a collection of *fingerprint digests*, so to easily identify the origin camera of a given image if its fingerprint is in

the database. The authors performed their experimentation with the help of the Matlab software and a database of 2,000 iPhones, proving the feasibility of the approach proposed. Even in this case, no details are provided about the way the experimentation has been conducted.

The authors in [29] represent camera fingerprints in binary-quantized form to store them more efficiently and to speed up the identification process. They showed through both analytical study and simulations that the reduction in matching accuracy due to quantization is insignificant if compared to conventional approaches. The authors created a compact representation of fingerprints through the most severe form of quantization, *i.e.*, by quantizing every element of sensor fingerprints into a single bit. They conducted experiments to determine the change in the performance due to the loss of information resulting from binarization.

4.2.1 Our Contribution

In this chapter we focus on the development of a fast and efficient distributed solution for the Source Camera Identification (SCI) problem (*i.e.*, recognizing the camera used for acquiring a given digital image) based on the algorithm by *Fridrich et al.* [181]. A common approach adopted by many SCI algorithms requires the extraction of a set of features, usually *sensor fingerprints* of a camera, from an image under scrutiny and their matching with a set of features of previously known cameras in order to identify the originating camera. As observed in [29], these operations may be very computational expensive when dealing with a large set of images, for two reasons. The first one is that the large dimensionality and high precision representation of sensor fingerprints put a heavy burden on all the operations related to storing and loading fingerprint data. The second one concerns the high time complexity of some of the operations required by the identification algorithms, further increased by the high dimensionality of data to process, both in terms of number of images and number of pixels. Therefore, the process can be very time consuming. The operations are individually very CPU and I/O-intensive, as they involve the processing of images containing millions of pixels. Moreover, they are expensive as a whole, as this cost has to be multiplied

for the number of images (which may be thousands or millions) used to build, to test and, finally, to use a recognition system.

The SCI problem has already been dealt with in the recent past, as proved by the contributions available in the scientific literature, such as [28, 29, 122, 123, 142, 181]. It may be approached either from an algorithmic viewpoint or from a computational resources viewpoint. In the first case, for example, it is possible to speed up the retrieval of a fingerprint and its matching with the fingerprint of an input image by storing camera fingerprints in a compressed way through a binary quantized form ([29]). In the second case, it is possible to speed up the operations by using faster processors or by spreading these activities over several computing cores of a same calculator. In this second case, however, there could be severe performance bottlenecks, precisely because the use of multiple CPU cores on the same computer would require them to share and to contend memory and storage resources at the same time.

These limitations may be overcome by resorting to a distributed approach based on data local computation. In a few words, images are processed in parallel by several computing nodes of a same cluster. All the images to be processed are not concentrated on a same machine (or disk) but are distributed in the local storage of the machines of the computer cluster. When a computation involving a certain image has to take place, it is convenient to run it on the machine hosting that image rather than moving the image to a different computing machine. This reduces the amount of data transferred over the network while allowing for a virtually unlimited scalability.

In particular, in this chapter, we present the work done for efficiently engineering, on Hadoop, a reference algorithm for the Source Camera Identification problem. The selected algorithm is the *Fridrich et al.* algorithm [181], that is the most popular and cited SCI solution. In particular, in Section 4.3 are provided more information about the problem of Source Camera Identification, followed by a description of the algorithm by *Fridrich et al.* The first implementation has been developed in a straightforward way, by adapting our stand-alone Java implementation of the traditional *Fridrich et al.* algorithm and with the help of the standard facilities available with Hadoop (see Section 4.4.1).

The resulting code, when run on our cluster system of commodity PCs, exhib-

ited much shorter execution times than the original one, by concurrently running multiple instances of the algorithm on different computing nodes. However, its performance produced discouraging effects, in fact, the performance of the distributed version of the algorithm was quite below expectations. In particular, in Section 4.4.2, are shown these experimental results obtained when running the distributed implementation of the *Fridrich et al.* algorithm. In this same section, are also described the datasets and the experimental settings employed for our tests.

A closer investigation revealed the existence of several performance issues due to the inability of our distributed implementation to take full advantage of the underlying cluster resources. In Section 4.4.3 we focus on some serious performance issues exhibited by our distributed implementation of the *Fridrich et al.* algorithm, and we investigate their causes through an in-depth profiling activity. Thus, we decided to monitor the status of each cluster node during the execution of our experiments.

From this point on, we put in practice an engineering methodology aiming, first, at pinpointing the causes behind the performance issues we observed, through a careful profiling activity, and, second, at solving them through the introduction of several theoretical and practical improvements. Through this profiling activities we were able to pinpoint some serious performance bottlenecks that heavily affecting the performance of the distributed algorithm. Then, several variants of the original code have been implemented in order to overcome these problems and improve the overall performance. In fact, several improvements were then tried, and their effects were measured by accurate experimentations. This allowed for the development of alternative implementations that, while leaving unaltered the original algorithm, were able to improve the usage of the underlying cluster resources as well as of the Hadoop framework, thus resulting in a much better performance than our original naive (or *vanilla*) implementation. The impact of these improvements (see Section 4.4.4) on the performance and on the scalability of the distributed version of the *Fridrich et al.* algorithm is analyzed by conducting some more experiments. The resulting implementations succeed in delivering a performance much better than the original distributed implementation (see Section 4.4.5 for details).

A preliminary version of this work was presented in [55].

4.3 Source Camera Identification Problem

Source Camera Identification (SCI) problem concerns the identification of the digital camera used for capturing a given input digital image.

A very common identification strategy consists in analyzing the *noise* in a digital image to find clues about the digital sensor that originated it. In a digital image, the noise can be defined as color distortion in a pixel in comparison with the original picture. These distortions may be due to the *Shot Noise*, a random component, and/or to the *Pattern Noise*, a deterministic component. The Pattern Noise, in turn, can be divided into two main components: the *Fixed Pattern* noise (FP) and the *Photo-Response Non-Uniformity* noise (PRNU). The FP noise is caused by *dark currents*, that is the information returned by the pixel detectors of a digital sensor when they are not exposed to light. The PRNU noise is caused mainly by the *Pixel Non-Uniformity* noise (PNU), resulted from the different sensitivity of the pixel detectors to light. This difference is due to the inhomogeneity of the wafers of silicon and the imperfections derived from the manufacturing process of the sensor. Figure 4.1 shows the different components of the noise in a digital image.

Thanks to their deterministic and systematic nature, the PNU noise and the FP noise are the ideal candidates for providing a sort of fingerprint of digital cameras. For example, in [160] the authors used the dark current noise to identify a camcorder from videotaped images. The idea of using the PNU noise for camera identification, instead, has been initially explored by *Fridrich et al.* in [181]. The authors observed that this method was successful in identifying the source camera used to take the considered picture, even distinguishing between cameras of the same brand and model. Satisfactory results were also obtained with images subjected to post-processing operations such as JPEG compression [286], gamma correction, and a combination of JPEG compression and in-camera resampling. The effectiveness of this method has been further confirmed by an experimental evaluation whose results are available in [122]. The authors downloaded from the Flickr image database a set of pictures taken by 6,896 individual cameras

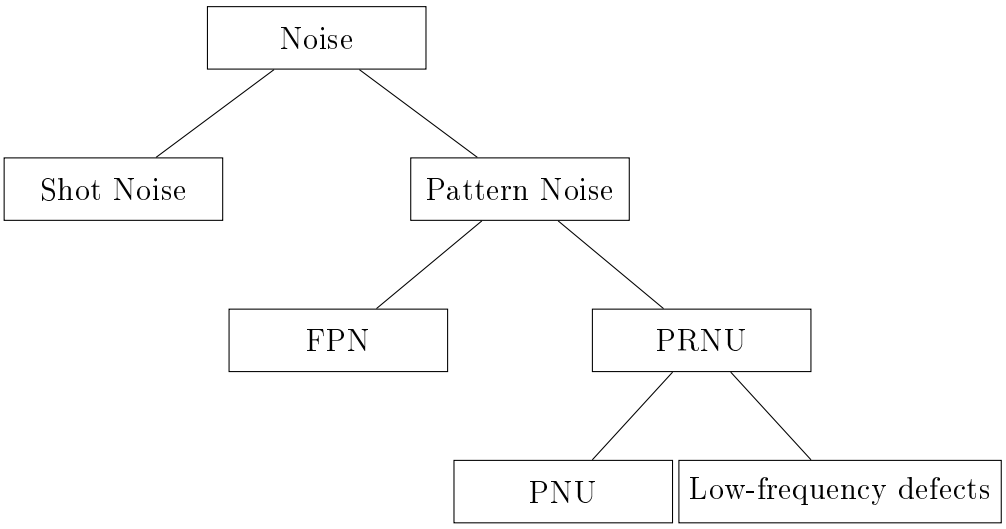

Figure 4.1: The components of the noise in a digital image.

(covering 150 camera models), for an overall number of more than one million pictures. According to their results, the algorithm they used was able to exhibit, in that setting, a *False Rejection Rate* (FRR, *i.e.*, the rate of images not attributed to their originating camera) smaller than 0.0238, and a *False Acceptance Rate* (FAR, *i.e.*, the rate of images attributed to the wrong camera) set to a very small value (*i.e.*, 2.4×10^{-5}).

4.3.1 The Algorithm by *Fridrich et al.*

In this section we describe the original version of the SCI algorithm by *Fridrich et al.* [181], which is the basis of our reference implementation. All the operations described hereafter have to be repeated either three times, if we choose to work on the red, green and blue color channels (RGB), or just one time, if we consider the grayscale representation of the input images. In our case, we decided to work in the RGB space, following the instructions provided in [181].

Let I be the image under scrutiny and $CamSet = \{C_1, C_2, \ldots, C_n\}$ the set of candidate origin cameras for I. The algorithm operates in four steps.

The first step (Step I) is the calculation of the *Reference Pattern* (*i.e.*, the camera sensor fingerprint) RP_C for each camera C belonging to $CamSet$. The approach proposed by *Fridrich et al.* in [181] consists in estimating RP_C by

extracting the *residual noises* (RNs) from a set of pictures taken by using C and, then, combining these noises together, as an approximation of the PNU noise. The residual noise RN of an image I can be defined as:

$$RN_I = I - F(I) \tag{4.1}$$

where $F(I)$ is a filter function that returns the noise-free variant of the image I. The filter F simulates the behavior of the Wiener filter in the wavelet domain, following the approach suggested by *Mihcak et al.* in [199]. The operation described above is applied pixel-by-pixel and is iterated over a group of images with the same spatial resolution, here named *enrollment* images, taken by using C. This returns a group of residual noises, including both a random noise component and the PNU noise estimation of C. The sum of the residual noises is then averaged to obtain a tight approximation of the camera C fingerprint, as follows:

$$RP_C = \frac{\sum_{k=1}^{m} RN_k}{m}\,. \tag{4.2}$$

The average operation reduces the contribution of the random noise components while highlighting the contribution of the deterministic noise components.

Let n be the number of different cameras, and let m be the number of enrollment images for each camera, we must compute $n \times m$ residual noises.

The second step (Step II) is propaedeutic to the calculation of the decision thresholds carried out during the third step of the algorithm. We first introduced a set of *calibration* images (also called *training* images) using each of the cameras belonging to $CamSet$. We then calculated the *correlation* between the fingerprint of each camera C (*i.e.*, RP_C) and the residual noise of each image T (*i.e.*, RN_T) taken from the calibration set. The calculation is accomplished using the *Bravais-Pearson correlation* index as follows:

$$corr(RN_T, RP_C) = \frac{(RN_T - \overline{RN_T})(RP_C - \overline{RP_C})}{\left\|RN_T - \overline{RN_T}\right\| \left\|RP_C - \overline{RP_C}\right\|}\,, \tag{4.3}$$

where the bar above a symbol denotes the mean value. This index returns a value included in the interval $[-1, +1]$, where higher values imply a higher probability

that an image T has been taken by using a camera C. In our case, it is evaluated for each of the three RGB color channels and for each pixel of the input images. Notice that if the spatial resolution of T does not match the resolution of the images used for determining RP_C, they could be adapted using a cropping or resizing operation. The same step can be repeated for a second set of images, called *testing* set, to be used for validation the recognition system during the third step of the algorithm.

Let n be the number of cameras, and let k be the number of calibration/testing images for each camera, then we must compute $k \times n^2$ correlation indices.

The third step (Step III) is about the calibration or training of the recognition system used to recognize the source camera of a given image. The identification is based on the definition of a set of three acceptance thresholds (one for each color channel) to be associated to each of the cameras under scrutiny. If the correlation between the residual noise of I and the Reference Pattern of a camera C, on each color channel, exceeds the corresponding acceptance threshold, then C is assumed to be the camera that originated I. The thresholds are chosen so to minimize the False Rejection Rate (FRR) for images taken by using C, given an upper bound on the False Acceptance Rate (FAR) for images taken by using a camera different than C (Neyman-Pearson approach). For the definition of these thresholds are used the correlation indices of the calibration images, while the correlation indices of the testing images can be used to compute the *recognition rate* (RR).

The last step (Step IV) concerns the identification of the camera that captured I. Here the algorithm first extracts the residual noise from I, RN_I, then correlates it with the Reference Patterns (RPs) of all the cameras under scrutiny using the system calibrated in the third step. If the correlation exceeds the decision threshold of a certain camera, on each of the three color channels, a matching camera is found.

4.3.2 Reference Implementation

Our stand-alone (non-parallel) reference implementation of the *Fridrich et al.* algorithm has been coded entirely in Java, and it is called *Camera Hardware*

Identification (`CHI`). An equivalent implementation of PNU filter F was also written in Java. `CHI` closely follows the original algorithm, but we also provide two types of decision modules based on correlation indices: the first type is based on Neyman-Pearson (like in *Fridrich et al.*), and the second one uses a multi-class *Support Vector Machine* (SVM) [275] classifier. We also have chosen to use a multi-class SVM classifier instead of using the original algorithm based on the Neyman-Pearson approach, because of the better performance exhibited by this classifier in our experiments. However, these issues are outside the scope of the dissertation because they not impact on the complexity of our distributed algorithm.

A SVM classifier belongs to the class of supervised learning classifiers. These classifiers are able to estimate a function from labeled training data, with the purpose of using it for mapping unknown instances (not labeled). In the classification problem, the training data consist of a set of instances, where each is a pair consisting of a vector of features and the desired group (class). In our case, the features are the values extracted from correlating each RN of an image under scrutiny with each Reference Pattern (RP). In particular, `CHI` uses the correlation values of the calibration images to train SVM classifier, while the correlation values of the testing images are adopted to validate the trained SVM (see Step III).

4.4 Source Camera Identification on Hadoop

In this section is shown as *Fridrich et al.* algorithm is engineered as a distributed solution according MapReduce paradigm. Initially is discussed a naive (or vanilla) implementation of the algorithm, called `HSCI`, then, after a careful profiling, are provided some improvements to speed up the performance.

4.4.1 The Algorithm by *Fridrich et al.* on Hadoop

It has been developed in Java a MapReduce-based implementation of the *Fridrich et al.* algorithm, which it was split in four different modules, each corresponding to the four processing steps of the *Fridrich et al.* algorithm, plus a fifth module

Table 4.1: Overview of our Hadoop-based implementation of the *Fridrich et al.* algorithm.

Step	**Hadoop Role**	**Input**	**Output**
(Step I) Calculating Reference Patterns	MapReduce Job	Enrollment Images	Reference Patterns
(Step II) Calculating Correlation Indices	Map-only Job	Calibration and Testing Images, Reference Patterns	Correlation Indices
(Step III) Recognition System Calibration	-	Correlation Indices	Thresholds, FAR, FRR, Recognition Rate
(Step IV) Source Camera Identification	Map-only Job	Input Image, Reference Patterns, Thresholds	Camera Id

related to the preliminary image loading activity on the Hadoop Distributed File System (HDFS). In the following, we describe in details these modules. Table 4.1 show an overview of the basic implementation.

4.4.1.1 Setup: Loading Images

With this preliminary step, the images to be used during the algorithm execution are loaded on HDFS storage. In our implementation, this task was accomplished by copying and keeping the images as separate files.

4.4.1.2 Step I: Calculating Reference Patterns

The aim of this step is to calculate the Reference Pattern (RP) of a generic camera C, by analyzing a set of enrollment images with the same spatial resolution and taken by using C. In the map phase, each processing task receives a set of images, extracts their corresponding residual noises and outputs them. In the reduce phase, the processing function (one for each camera C) uses the set of residual noises of C produced in the previous tasks and combines them, thus

generating the RP_C. This operation is repeated for each camera under scrutiny.

The pseudo-codes of the map and reduce functions of this step are illustrated in Algorithm 1. As said in Chapter 3, in Hadoop, each input record is structured as a $<key, value>$ pair. During this step, *key* is derived from the image meta-data and *value* stores the Uniform Resource Locator (URL) of the image on HDFS. When the map function is invoked, it receives this record, loads the corresponding image in memory from HDFS and, finally, extracts the residual noise (RN) from the image. As an output, the function produces a new $<key, value>$ pair, where *key* is the *camera id* and *value* is the URL of RN directly saved on HDFS. During the reduce phase, a function receives a tuple in the $<key, value>$ format, where *key* is the id of a camera, *e.g.*, C, and *values* is a set of the URLs to RNs (saved on HDFS) for that camera, as calculated during the map task. All the RNs of the same camera are summed and then averaged to form the Reference Pattern for C as described in Section 4.3.1 (see Equation 4.2). As an output, the function generates a new $<key, value>$ pair, where *key* is the id of C, and *value* is RP_C.

The Figure 4.2 shows the overall and conceptual view of our distributed algorithm when running the Step I on a cluster of 4 slave nodes.

4.4.1.3 Step II: Calculating Correlation Indices

During this step, the algorithm extracts the RN of each calibration image and correlates it with the RPs of all the input cameras. The same operation is repeated for the testing images.

In the map phase, each processing task receives a list of input images to be correlated as $<key, value>$ records, where *key* is derived from the image meta-data and *value* stores the image URL on HDFS. For each URL, the corresponding image is (possibly) transferred to the slave node, and the RN is extracted and correlated with the RPs of all the input cameras calculated in the previous step.

If the resolution of the input image does not match the resolution of the RP of a given camera, cropping and/or scaling techniques are used in order to correct the issue. For each correlation, the map function generates a $<key, value>$ pair, where *key* is the keyword *"Correlation"* and *value* consists of: the *image identifier*, the *camera identifier* used for shooting the image, the *RP identifier*, a value

Algorithm 1 Pseudo-code of `HSCI` for Calculating Reference Patterns (Step I)

function MAP(Key_I, $Path_I$)
▷ It returns the RN_I of an enrollment image I taken from camera C with the purpose to compute the RP_C in the reduce function. Key_I is a set of meta-data used for identifying the enrollment image I, including the id of the camera C used for shooting I, id_C; $Path_I$ is the HDFS path of I.

$I \leftarrow$ LOAD($Path_I$)
$F_I \leftarrow$ APPLYPNUFILTER(I)
$RN_I \leftarrow$ SUBTRACT(I, F_I)
EMIT(id_C, RN_I)

end function

function REDUCE(id_C, list(RN_C))
▷ It returns RP_C by averaging the RNs of the enrollment images of the camera C. The variable id_C is the id of the camera C; list(RN_C) is the list of the residual noises extracted from enrollment images taken by using C.

$RP_C \leftarrow$ NEW(Zeros)

for each $Path_{RP_i}$ in list(RN_C) **do**
    $RN_i \leftarrow$ LOAD($Path_{RP_i}$)
    $RP_C \leftarrow$ SUM(RP_C, RN_i)
end for

$RP_C \leftarrow$ AVERAGE(RP_C)
EMIT(id_C, RP_C)

end function

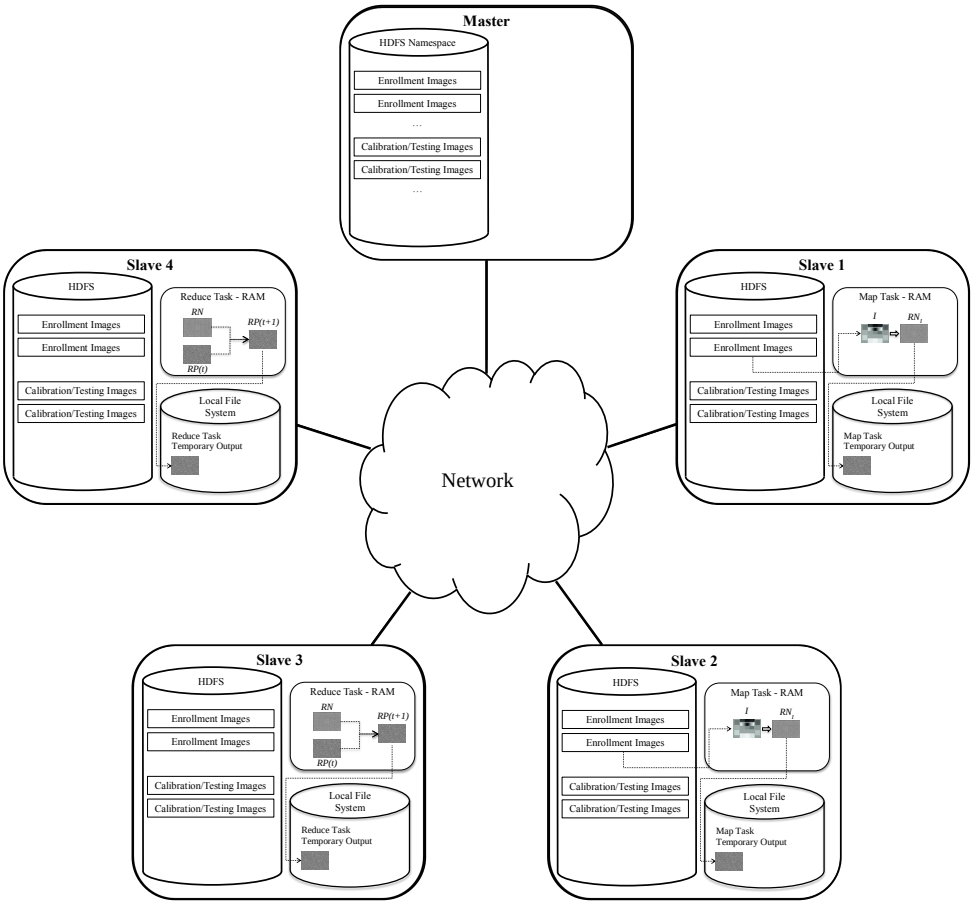

Figure 4.2: Conceptual view of our distributed algorithm for SCI when running the Step I on a cluster of 4 slave nodes. Images are not centralized but they are saved on HDFS. Slave 1 and Slave 2 are extracting RNs from enrollment images available on HDFS (map task). The result of this extraction can be buffered on local file system before being sent to HDFS. Slave 3 and Slave 4 are summing and averaging all RNs of images shot using a same camera to compute the RP (reduce task).

indicating the correlation preprocessing *type* (*e.g.*, none, crop, resize, etc.), plus the three correlation indices (one for each color channel). The output of all map tasks is collected in a plain text file, *CORRs*. Since each processing node has to load the *RP* of all the input cameras, we used the Hadoop *Cache File* mechanism to make each node transfer to its local file system a copy of these files, before starting the Hadoop job (see Section 3.5 for details). In particular, it was used the object `DistributedCache`, a facility provided by the Hadoop framework v1.x to cache files (text, archives, jars etc.) needed by applications. In this step, a reduce task is not required. The pseudo-code of this step is illustrated in Algorithm 2.

Notice that the strategy of partitioning the input images while replicating a copy of the *RP*s in all the nodes of a cluster is particularly advantageous when the number of images is much larger than the number of input cameras. On the other hand, if the number of cameras is much larger, then it is likely to be convenient to do the opposite (*i.e.*, partition the *RP*s over all the nodes while replicating the input images).

Algorithm 2 Pseudo-code of `HSCI` for Calculating Correlation Indices (Step II)

function MAP(Key_I, $Path_I$)
▷ It returns the correlation indices between RN_I and all RPs. Key_I is a set of meta-data used for identifying the calibration or the testing image I, including the id of the camera C used for shooting I, id_C; $Path_I$ is the HDFS path of I.

$I \leftarrow$ LOAD($Path_I$)
$F_I \leftarrow$ APPLYPNUFILTER(I)
$RN_I \leftarrow$ SUBTRACT(I, F_I)

for each $Path_{RP_i}$ in list(RPs) **do**
$RP \leftarrow$ LOAD($Path_{RP_i}$)
$corr \leftarrow$ CORRELATE(RN_I, RP)
EMIT("Correlation",{id_I, id_C, *calibration/testing*, id_{RP}, *type*, $corr$})
end for

end function

4.4.1.4 Step III: Recognition System Calibration

According *Fridrich et al.* algorithm, in this step a set of three acceptance thresholds (one for each color channel) is calculated for each of the cameras under scrutiny. The thresholds are determined using the Neyman-Pearson approach and exploiting the correlation values of the calibration images computed in the previous step. The correlation values of a set of testing images, calculated during Step II, are then used to validate the recognition system according the recognition rate (RR) by comparing them to the aforementioned acceptance thresholds. Since this step is computationally cheap, it is run directly on the only Hadoop master node, without using any form of parallelization.

In particular, according to *Fridrich et al.* we adopt the Neyman-Pearson approach and we determine the value of a threshold t for a camera C by maximizing the probability of detection (or, equivalently, minimizing the FRR) given an upper bound on the FAR, FAR $< \alpha_{FAR}$. The pseudo-code used for computing the threshold for a camera C is illustrated in Algorithm 3. The function `computeThreshold` is iterated for each camera and color channel.

Optionally, in this step we can use a SVM classifier instead of Neyman-Pearson approach. During this step, the SVM-based classifier is trained and tested using the correlation indices calculated in the previous step. In our case, we decided to perform this operation on a single node (*i.e.*, the master node) because, in our setting, it features very short execution times when executed in a sequential way. The SVM implementation we have used is the one available with the Java Machine Learning Library (Java-ML) [1], including the LIBSVM module [57]. At the end of this step, the classifier has been trained and it is ready to be used for the identification step. Moreover, an estimation of the accuracy of the training phase is returned to the user, organized as the number of successful matches (recognition rate) between the testing source images and their corresponding reference cameras.

4.4.1.5 Step IV: Performing Source Camera Identification

The aim of this step is to establish which camera has been used for capturing an image I. The input of the Hadoop job is the directory where the RPs have been

Algorithm 3 Pseudo-code of `HSCI` for Recognition System Calibration (Step III)

function COMPUTETHRESHOLD(id_{RP_C}, id_C, $CORRs$, α_{FAR}, T, chn)
▷ It returns a threshold t for the camera C using chn as a color channel, in addition also FAR and FRR are returned. The variable id_{RP_C} is the id of the RP_C, id_C is the id of camera C, $CORRs$ is the file containing the correlations computed in Step II, α_{FAR} is the FAR upper bound, T is a set of candidate thresholds, chn is the color channel to use.

for each t in T **do**
$FAR \leftarrow$ COMPUTEFAR(id_{RP_C}, id_C, $CORRs$, t, chn)
if $FAR \leq \alpha_{FAR}$ **then**
$FRR \leftarrow$ COMPUTEFRR(id_{RP_C}, id_C, $CORRs$, t, chn)
return (FAR, FRR, t)
end if
end for

end function

function COMPUTEFAR(id_{RP_C}, id_C, $CORRs$, t, chn)
▷ It returns the FAR using images taken by using cameras different from C, but classified as taken from C (exploiting id_{RP_C}).

return $1 - \prod_{\substack{id_{C'} \in CamSet, \\ id_{C'} \neq id_C}}$ COMPUTEFRR(id_{RP_C}, $id_{C'}$, $CORRs$, t, chn)

end function

function COMPUTEFRR(id_{RP_C}, $id_{C'}$, $CORRs$, t, chn)
▷ It returns the FRR using the correlations between RP_C and the residual noises of the calibration images of C'.

$tot \leftarrow 0$, $count \leftarrow 0$
for each correlation $corr$ in $CORRs$ between the residual noise of a calibration image taken by using camera C' and RP_C **do**
$tot \leftarrow tot + 1$
if GETCORR($corr$, chn)$< t$ **then**
$count \leftarrow count + 1$
end if
end for
return $count/tot$

end function

stored. The output is the identifier of the camera recognized as the originating camera for the input image. For each input RP, a new map function is invoked. This function uses a copy of the residual noise of I, *i.e.*, RN_I, for calculating its correlation with the input RP. The pseudo-code of the map function of Step IV is illustrated in Algorithm 4. Then, the job returns a file containing the list of the correlation values needed to perform the recognition phase exploiting the results of the previous step (*i.e.*, camera thresholds or trained SVM). Finally, the predicted *camera id* is returned.

Algorithm 4 Pseudo-code of `HSCI` for Source Camera Identification (Step IV)

function MAP(Key_{RP}, RP)
▷ It returns the correlation indices between the RN_I of a fixed preloaded input image I and the input RP. Key_{RP} are the meta-data of RP.

$corr \leftarrow$ CORRELATE(RN_I, RP)
EMIT("Correlation",{id_I, id_{RP}, $type$, $corr$})

end function

This step works on a single image to perform the camera identification. Alternatively, when there are many unknown input images, the distributed functionalities of Step II can be exploited to compute the correlation values between each input image and each RP.

4.4.2 Experimental Analysis

In this section we discuss the results of a preliminary experimental analysis we have conducted. We compared the performance of our Hadoop-based implementation of the *Fridrich et al.* algorithm with its non-distributed counterpart. The discussion also includes a description of the performance metrics, the datasets and the experimental settings used in our analysis.

4.4.2.1 Performance Metrics

The benchmarking methodology that have been instrumented to measure the performance of Source Camera Identification on Hadoop is here described. The

following *Key Performance Indicators* (KPIs) have been selected and collected during the experiments, such as: real time elapsed to complete each step, overall number of images processed for time unit on a cluster and recognition rate.

- *RT*: The real time (RT) elapsed to complete each step can be naturally observed using the clock of the master node as a stopwatch with a pretty good resolution (seconds) if we are considering total run times longer than few hours.

- *Imgs2Min*: A more interesting *KPI*, more useful for end users, is the overall number of images processed for time unit (*e.g.*, minutes) on a cluster with a fixed number of slave nodes. This is strictly related to the run time, but is affected by several external factors, such as the image resolutions.

- *RR*: To measure the reliability of the final result of SCI, at the end of the Step III, we observe another *KPI* called recognition rate (RR), which is representative of the quality of the results of the previous steps. This is an index that will show whether our Hadoop application returns results comparable to the reference (sequential) implementation.

- *PCsStat*: It is important to monitor the overall health status of the computer cluster used for distributed applications. In fact, for each slave node, it is important to supervise indicators such as: CPU activities, RAM and SWAP[1] usages, I/O activities on local disk and network, and so on. The Hadoop logs can register very useful systems counters and debug information, but the operating system utilities can also detect node-level performance statistics and possible bottlenecks. For gathering information, we used Dstat tool [293] with data sampling every 10 seconds (to lower costs and to obtain a good sampling). Dstat is a versatile resource statistics tool and it is a replacement for *vmstat*, *iostat*, *netstat* and *ifstat* utilities. Dstat is useful for monitoring systems during performance tuning tests, benchmarks or troubleshooting. Adopting this tool, individually, for each node

[1] A process can be swapped temporarily out of main memory to a backing store (*i.e.*, local disk), and then brought back into memory for continued execution. *Swapping* makes it possible for the total physical address space of all processes to exceed the real physical memory of the system [113].

of the cluster, we can monitor CPU usage, RAM and SWAP used, bytes written/read on local disk, number of send/received network packets and incoming/outgoing network throughput.

4.4.2.2 Dataset

The dataset used in our experiments was created in conjunction with the CNCPO[1]. It consists of 5,160 JPEG images, shot using 20 different Nikon D90 digital cameras. This model has a CMOS image sensor ($23.6 \times 15.8mm$) and maximum image size of $4,288 \times 2,848$ pixels. 258 JPEG images were taken for each camera at the maximum resolution and with a very low JPEG compression. The images were organized in 130 enrollment images, 64 calibration images and 64 testing images for each camera.

As for the enrollment images, we first made a preliminary experiment where we tried different numbers of images, ranging from 50 to 300. According to our results, the value 130 offered a good trade-off between the cameras recognition rate and the performance overhead required for processing these images. We also observed that it is possible to save computation time by choosing a smaller number of enrollment images at the expense of a degradation of the identification results.

Enrollment images were taken from a *ISO Noise Chart 15739* [223], as shown in Figure 4.3. Calibration and testing images, instead, portray different types of subjects. Figure 4.4 shows an example of training and testing images from our dataset.

The images were taken by using each of the 20 cameras used for our tests. The overall dataset is about 20 GB large, and about 40% of that size is due to enrollment images.

4.4.2.3 Experimental Settings

All the experiments shown in this chapter were conducted on a homogeneous cluster of 33 commodity and commercial off-the-shelf PCs equipped with 4 GB

[1]National Center for the Fight against Online Child Pornography (Centro Nazionale per il Contrasto alla Pedopornografia Online, CNCPO) part of the "Dipartimento della Pubblica Sicurezza" within the Italian Ministry of Interior.

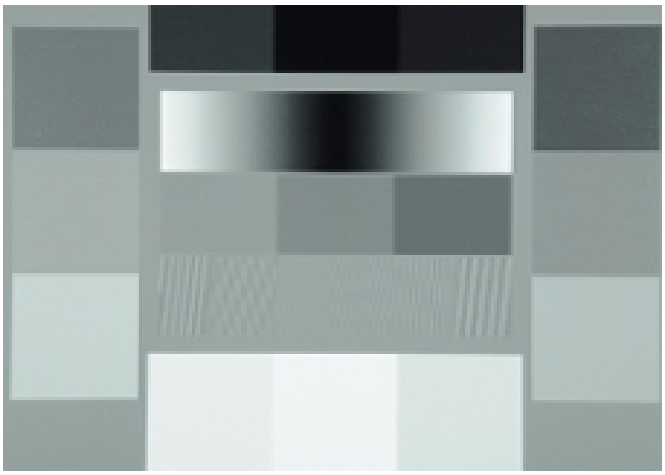

Figure 4.3: An example of enrollment image. The scene is from a test chart used for noise measurements, ISO 15739 [223].

Figure 4.4: An example of training and testing images from the dataset of Nikon D90 cameras.

of RAM, an *Intel Celeron G530 @ 2.40 GHz x2* processor (*i.e.*, dual-core CPU), *Windows 7* host operating system and a 100 Mbps Ethernet card. In this environment, we installed on each computer a virtual machine (VM) running the *Ubuntu 12.10 64-bit (Kernel 3.5)* guest operating system through the VMware Player software, and equipped with 3,100 MB of RAM, 2 CPUs and 117 GB of virtual disk storage (file system type *ext4*)[1].

The computing unit of a common cluster is mostly based on virtual machines, therefore it is feasible to experiment with the MapReduce paradigm using a virtualized data center, like in previous experimentations conducted with Hadoop (see, *e.g.*, [143]).

Our cluster included 32 slave nodes and a master node, and the Hadoop version was 1.0.4[2]. On each slave node, at most one map or reduce task was run due to memory limits. In addition, on each slave node, we set the properties that allow the framework to wait the end of all map tasks, before starting the reduce tasks, due to memory limits. In our cluster, these choices were made after noticing that the simultaneous presence of two map tasks or a map task and reduce task on the same machine cause some problems related to the main memory. This occurs because the filtering operation requires a lot of memory. In our preliminary experiments, we tried several different combinations of HDFS replication factors (*i.e.*, 2, 4, 8) and HDFS block size (*i.e.*, 32 MB, 64 MB, 128 MB). According to our results, the best performance and trade-offs were achieved when a replication factor set to 2 and a block size set to 64 MB were used. Thus, in all the forthcoming experiments we always used these settings.

4.4.2.4 Preliminary Experimental Results

During our experiments, we developed several Hadoop-based variants of the original *Fridrich et al.* algorithm. The first variant, here denoted `HSCI`, is the naive (or vanilla) implementation of the algorithm described in Section 4.4.1. In this implementation, all the image files to be processed are initially loaded on the Hadoop Distributed File System (HDFS). The files containing the residual noises

[1] We used VMs because there are restriction policies in the laboratory where the experiments were conducted.

[2] Our experimentation was conducted between end of 2013 and 2014.

Table 4.2: Execution times, rounded in minutes, of different distributed variants of the *Fridrich et al.* algorithm on a Hadoop cluster of 32 slave nodes, compared to the sequential counterpart, that is `SCI`, run on a single node. `SCI` has no setup cost, as the input images are already loaded on the machine running the algorithm.

Variant	Setup	Step I	Step II
`SCI`	0	888	5,257
`HSCI`	43	750	334
`HSCI_Tar`	88	581	322
`HSCI_Seq`	55	290	304

(RNs) obtained during the execution of the algorithm are also loaded on HDFS, as soon as they become available. As a consequence, map and reduce tasks take as an input (or provide as an output) not a copy of these images/RNs but a URL pointing at them.

We made a preliminary and coarse comparison between the performance of `HSCI` and the implementation running as a stand-alone application (*i.e.*, sequential or non-parallel on a single node), here named `SCI`, by measuring the overall execution time of the different steps of the algorithm in both settings. `SCI` was executed on our single slave node as a single-thread Java program. The results, available in Table 4.2, show that, when processing the second step, `HSCI` exhibits approximately a 16× speed up ($Speed\,up_{StepII} = Time_{StepII}(\texttt{SCI})/Time_{StepII}(\texttt{HSCI}) \approx 16$), thus providing a performance that is about one half of the maximum theoretical speed up achievable using a cluster of 32 slave nodes. This is even more evident in Table 4.3, where we show the (approximate) average number of images processed in a minute by each implementation in each step. On the contrary, the performance gain on the first step of the algorithm is almost negligible. Such a result is due to the reduce phase of this step. Each reduce task, in fact, has to collect from HDFS all the RNs generated during the map phase (in our experiments, 130 RN files for each RP file to generate, with the average size of a RN file of about 140 MB). This activity puts a heavy burden on the running time of the first step, as implemented by `HSCI`. Notice that through all the experimentations, we will focus only on Step I and Step II of the *Fridrich et al.* algorithm, because they are, by far, the most computationally expensive.

Table 4.3: Average number of images processed in a minute of the different variants of the *Fridrich et al.* algorithm on a Hadoop cluster of 32 slave nodes, compared against the sequential counterpart executed on a single node.

Variant	Step I	Step II
SCI	3	1
HSCI	3	8
HSCI_Tar	4	8
HSCI_Seq	9	8

In order to investigate the poor performance of HSCI during the first step of the *Fridrich et al.* algorithm, we traced the CPU and the network usage of slave nodes when running this step. The results, available in Figure 4.5 and Figure 4.6[1], show that the CPU is mostly unused (*i.e.*, the node runs on a dual-core processor, hence a CPU usage of 50% stands for a single-core used at the 100%). Conversely, the network activity dominates both the map and reduce phases: the map phase, because of the time required to download from HDFS the input images and to write on HDFS the resulting *RN* files; the reduce phase, because of the time required to collect all the *RN* files produced in the map phase. A possible explanation for such long times is related to the problems faced by Hadoop when managing a very large number of small files, as documented in [291].

A first attempt we tried for solving this problem has been to pack together group of images. In the resulting implementation, here referred to as HSCI_Tar, images are grouped in uncompressed archives stored on HDFS, with each archive containing 10 images. Here, each map task takes, as input, the URL of an archive and uses it to download and, then, unpack the corresponding archive file from HDFS. We expect this implementation to be faster than HSCI in the first step of the algorithm, because of the much smaller number of files to handle and because of their larger size. This expectation has been confirmed by the experimental results, with HSCI_Tar exhibiting a 20% performance gain over HSCI during Step I (see Table 4.2). This implies, in turn, the ability to process a higher number of images in a fixed time window (see Table 4.3). Differently, the second step of the

[1]We used the tool of performance monitoring Dstat [293] to generate these figures, doing a sequence of samples ranging from before the beginning of the Step until after its conclusion.

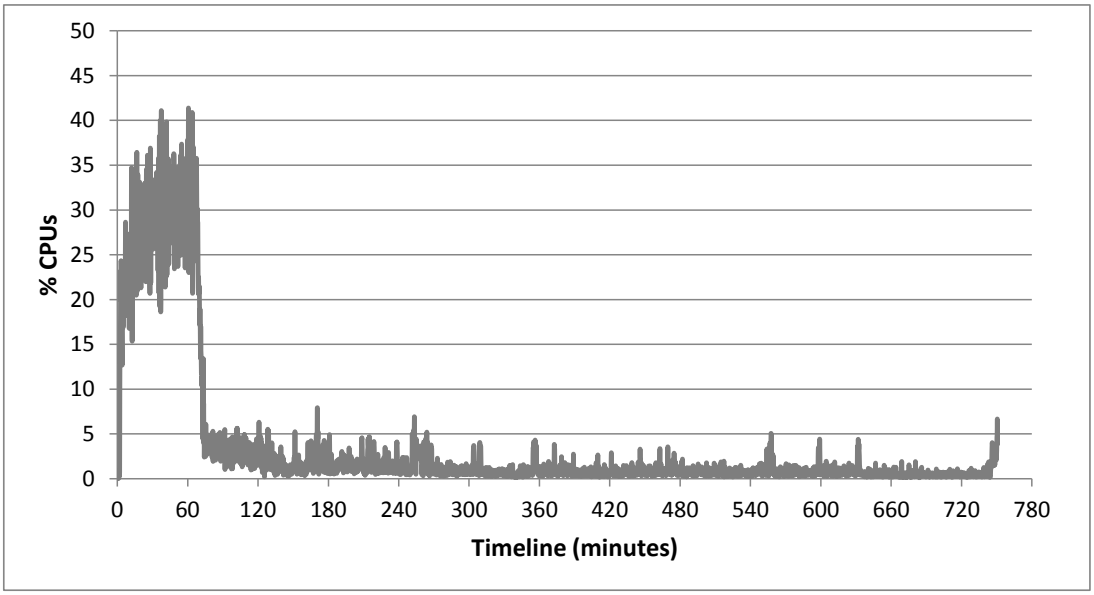

Figure 4.5: Average CPU usage of slave nodes, in percentage, when running Step I of `HSCI`.

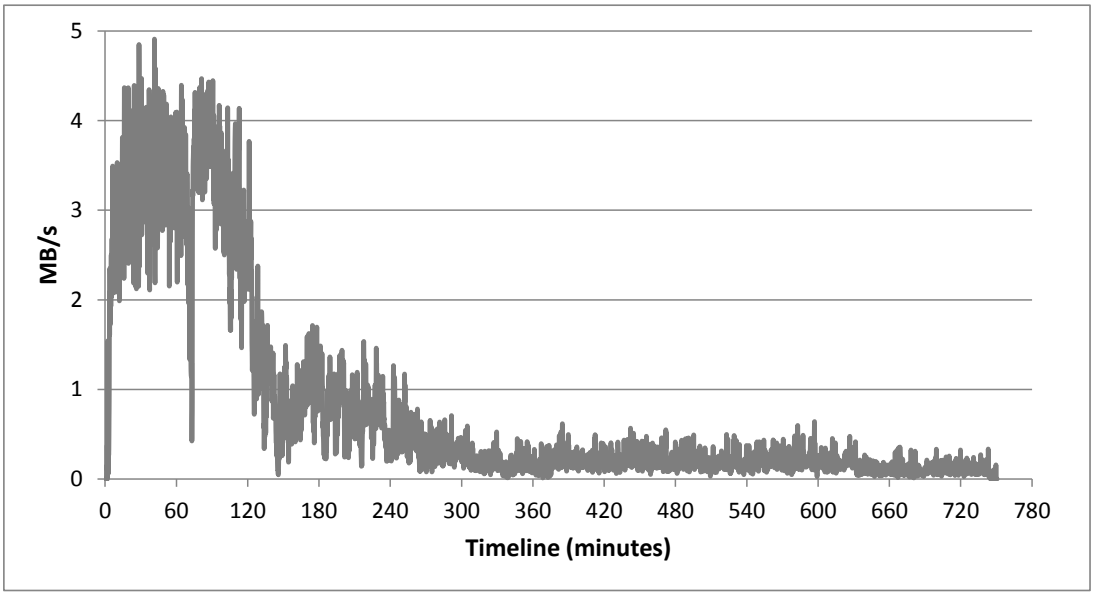

Figure 4.6: Average incoming network throughput, in MB/s, when running Step I of `HSCI`.

algorithm does not take any advantage from this improvement.

Despite of this improvement, the performance of MapReduce version of the *Fridrich et al.* algorithm are still unsatisfying. The network congestion due to the need of locating on HDFS the files to process and moving them on the slave running a task is still a major drawback, even if reduced thanks to the usage of archive files.

A more efficient solution would be to fully exploit data local computation by further reducing the number of files to be processed and by placing the data on the nodes running the tasks in charge to process them, and to avoid the *large number of small files* problem [291]. The solution we found consists in maintaining only two very large files containing all the image files. They have been coded as Hadoop `SequenceFile` objects and are: `EnrSeq`, used for storing a set of enrollment images, and `TTSeq`, used for storing a set of training (calibration) and testing images. In both files, the images are ordered according to their originating *camera id*. Then, we used the input split capability available with Hadoop sequence files for partitioning these two files among the different computing nodes, with the aim of promoting data local execution. Notice that the residual noises calculated during the first step of the algorithm are still written as separate files on HDFS as they become available, while the images are no longer directly downloaded from HDFS as individual files. In this case, in the Hadoop sequence file input, the *key* of each pair is the meta-data of an image, while the *value* of that pair stores a binary copy of that image. The outcoming Hadoop implementation is labeled as `HSCI_Seq` (where `Seq` stands for Hadoop `SequenceFile`s, not for stand-alone or sequential implementation). The experimental performance of `HSCI_Seq` implementation, when running the first step of the *Fridrich et al.* algorithm, is much better than `HSCI` variant, with an execution time that is approximately 2.6× faster than `HSCI`[1]. Also the second step of the algorithm seems to take advantage of this solution, as it is slightly faster than the `HSCI` solution.

These preliminary results seem to confirm, on a side, that is possible to drastically reduce the execution time of the *Fridrich et al.* algorithm by using the MapReduce paradigm. On the other side, it is clear the `HSCI_Seq` implementation of the algorithm still offers much room for improvement.

[1] $Speed\,up_{StepI} = Time_{StepI}(\texttt{HSCI})/Time_{StepI}(\texttt{HSCI\_Seq}) \approx 2.6$.

4.4.3 Profiling Activities for Detecting Bottlenecks

As already discussed in Section 4.4.2.4, a preliminary round of experimentations led us to develop a Hadoop-based variant of the *Fridrich et al.* algorithm, named `HSCI_Seq`, whose performance met enough our expectations. The same experiments revealed that the performance of this algorithm in a distributed setting is strongly influenced by the network activity required to load and/or to save files (RNs) on the underlying distributed file system.

In this section, we further analyze these phenomenons. The results of a thorough profiling activity aimed at charactering the behavior of the `HSCI_Seq` implementation will be presented, in order to improve our understanding about the way an algorithm, such as the one by *Fridrich et al.*, performs when adapted to run on Hadoop framework. We also assess the possibility of achieving further performance improvements.

We recall from the previous section that the input dataset for our tests contains two Hadoop sequence files: `EnrSeq` and `TTSeq`. The first one contains $2,600$ enrollment images to be used for the Step I of the *Fridrich et al.* algorithm (calculation of RPs). The second one contains $2,560$ calibration/testing images to be used during the Step II (calculation of correlations). The images are equally distributed over 20 cameras and the size of these files is about 20 GB.

During Step I, the processing of the `EnrSeq` file requires the creation of 130 map tasks, *i.e.*, one for each HDFS block of input file, where the size of `EnrSeq` is about 8 GB and the HDFS block size is set to 64 MB. The average amount of data exchanged between map tasks and reduce tasks is approximately 355 GB (without considering tasks and data replicas). In our experimental analysis run of `HSCI_Seq`, the framework executed, in the average, 141 map tasks: 130 completed successfully, the remaining 11 killed by the framework. The existence of these additional tasks is due to the Hadoop speculative execution[1]. Of these tasks, 122 were data local map tasks. In details, the `EnrSeq` file contains $2,600$ enrollment images, therefore we have $2,600$ map input records. For every such image, the map function extracts the residual noise (RN) and it write it on HDFS, thus

[1]Hadoop speculative execution indicates that a same task can be run multiple times on different slave nodes. As soon as the duplicate tasks end, the other ones are killed. See Section 3.5 for details.

Table 4.4: Map and reduce timing in minutes during Step I of `HSCI_Seq`.

Step I	**Running Time**
Map	72
Reduce	217

generating 2,600 RN files. The average size of a RN file is of about 140 MB. In the reduce phase, we set the number of reduce tasks to 20, that is the number of RPs to be calculated. In our profiling experiment, 29 reduce tasks were launched by the Hadoop framework, with only 20 completing their execution and the other ones being killed by the framework.

Table 4.4 shows that about 75% of the running time of `HSCI_Seq` during Step I is spent in the reduce phase. On one side, we suppose that this overhead is due to the time consumed by each reduce function to retrieve from the HDFS the corresponding RN files to sum, *i.e.*, 130 RNs for each RP to be calculated. On the other side, we expected that this second phase would have lasted lesser as it performed a simple operation from the computational viewpoint (*i.e.*, multiple sums of matrices).

In order to clarify this behavior, we traced the start and the end execution time of each task, both map and reduce phase, ran in our experiment. In Figure 4.7, we show an overview of the map and reduce tasks used by Hadoop when running the Step I of the `HSCI_Seq` algorithm. In some cases, the Hadoop framework may decide to issue a same task a second time (*e.g.*, for recovering a task that has been assigned to a free slave node, without being completed). These cases are highlighted in the figure by coloring black the tasks that are killed when their twin tasks complete their executions.

As it can be seen in the figure, the overall time spent by each slave node for processing map tasks is almost the same, because, as soon as a slave node finishes processing a map task, a new one is allocated to it by the system, until all map tasks are executed. When turning to the reduce phase, we observe that some reduce tasks end as soon as they start or are killed immediately. That can be explained by the fact that these tasks have not been assigned a RP to be calculated, in fact, the overall number of slave nodes completing a reduce task that computes at least one RP is 12 against a total number of 20 RPs to be

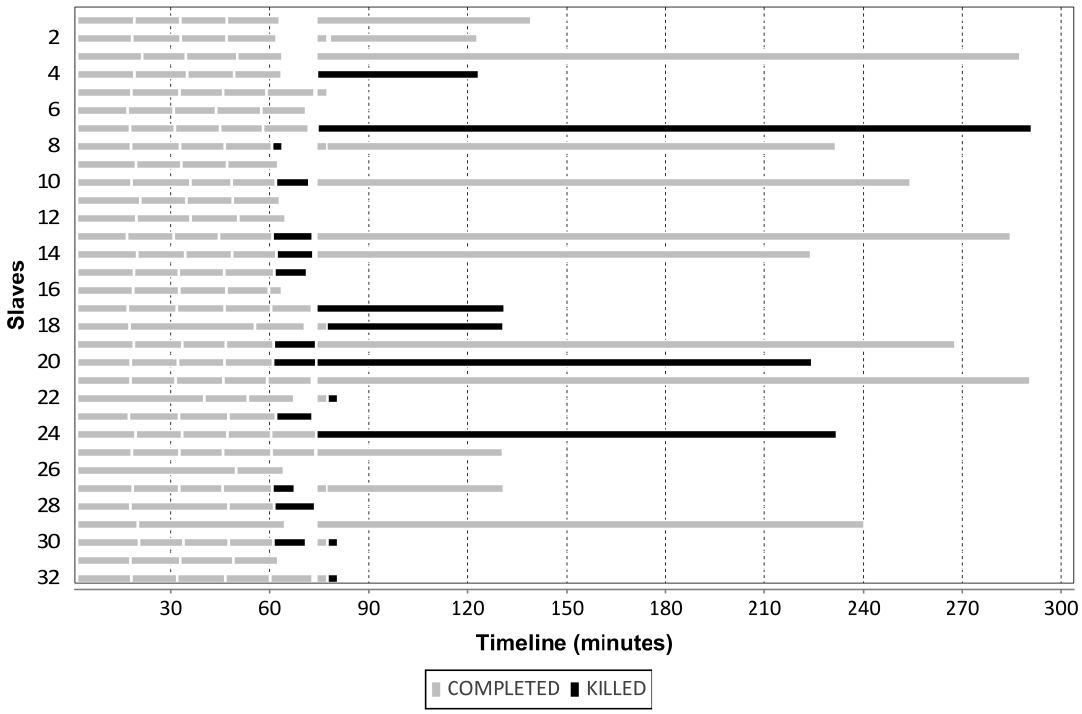

Figure 4.7: `HSCI_Seq` variant - An overview of map and reduce tasks launched during an execution of Step I. Notice that reduce tasks start only after the termination of all map tasks as described in Section 4.4.2.3.

calculated. Of these 12 nodes, 4 have a running time significantly faster than the other 8, because the 8 slowest nodes were calculating the *RP*s for 2 cameras and the remaining 4 nodes (the fastest) were calculating the *RP*s for just one camera. This unbalanced assignment is due to the standard hash function used by the Hadoop Partitioner service (see Section 3.5) for the distribution of the keys (in our case, the id of the cameras) to be processed in the reduce tasks.

We further analyzed the behavior of these tasks by profiling their CPU usage and their network activity during the same first step of the algorithm. In Figure 4.8 we report, for example, the CPU activity of *slave1*. During the first 60 minutes, spent processing map tasks, a single-core of the node was used almost at its maximum. Notice that, in our case, it is not possible to run two distinct map tasks on the same node because the amount of memory in it would not be enough. Therefore, we cannot fully exploit the two CPU cores available in each node. Instead, the second significant activity, *i.e.*, that related to the execution

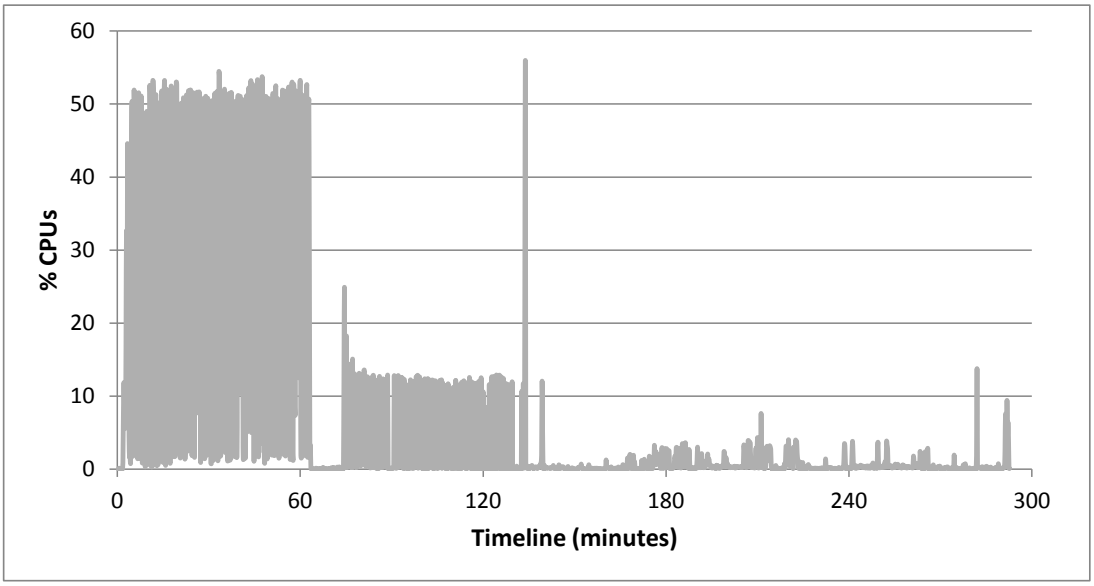

Figure 4.8: CPU usage of *slave1*, in percentage, when running Step I of `HSCI_Seq`.

of a reduce task covering about 65 minutes, featured a 10% average CPU usage.

This seems to confirm that, during the reduce phase, the CPU of the involved slave nodes is nearly unused, as this phase is dominated by the network activity related to the retrieval from HDFS of the RN files to sum. This observation is also supported by the analysis of the incoming network throughput for *slave1* node during Step I, as demonstrated in Figure 4.9. The figure shows that there is an intense network activity for *slave1* along all the map phase and the reduce phase. The remaining CPU activity in Figure 4.8 is due to the overhead required by Hadoop to keep alive the *slave1* node services.

During Step II, at least 194 map tasks are created using the testing and calibration images available in the `TTSeq` file (*i.e.*, one for each HDFS block of input file, with the size of `TTSeq` of about 12 GB). Considering only the map tasks successfully completed, the framework invokes $2,560$ map functions, *i.e.*, one for each calibration/testing image. Therefore, $2,560$ RNs are calculated and each of them correlated with the 20 RPs created in the previous step, thus requiring $2,560 \times 20 = 51,200$ correlations. In this run of the experiment, the framework ran 210 map tasks: 194 completed successfully, the remaining 16 killed by the

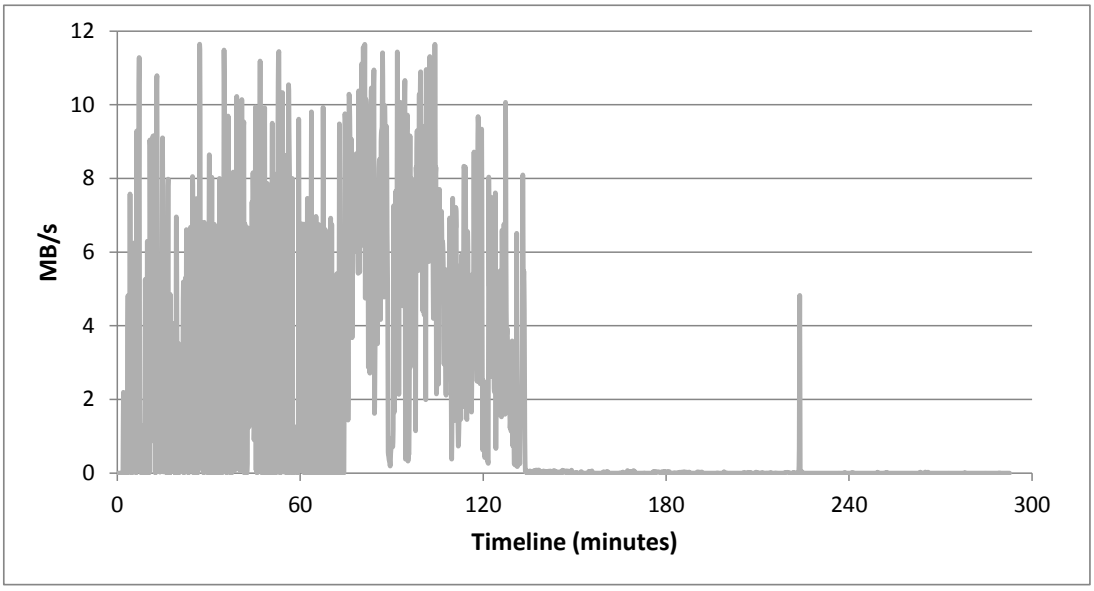

Figure 4.9: Incoming network throughput of *slave1*, in MB/s, when running Step I of `HSCI_Seq`.

framework. Of all these tasks, 187 were data local map tasks.

We recall that the Step II of the `HSCI_Seq` does not make use of the reduce phase, thus its execution time is approximately equal to the execution time of the map phase. An in-depth analysis of the map tasks revealed that they are characterized by an intense I/O activity, needed to load the Reference Patterns. However, these tasks also feature a very intense CPU activity, due to the work required to perform the correlations on big input files, as shown in Figure 4.10, where the average CPU usage stays around 40% during all the execution of the map phase. That indicates, on one hand, that the CPU does not suffer much from delays due to I/O activity, and, on the other hand, that there is a margin for improving by taking advantage of the second core of the CPU, actually unused. As already stated above, in fact, the available memory in each node is likely to be insufficient to run two tasks at the same time. In addition, Figure 4.10 shows that the CPU of *slave1* remains unused while waiting for the framework to copy the *RP*s from HDFS to the local file system.

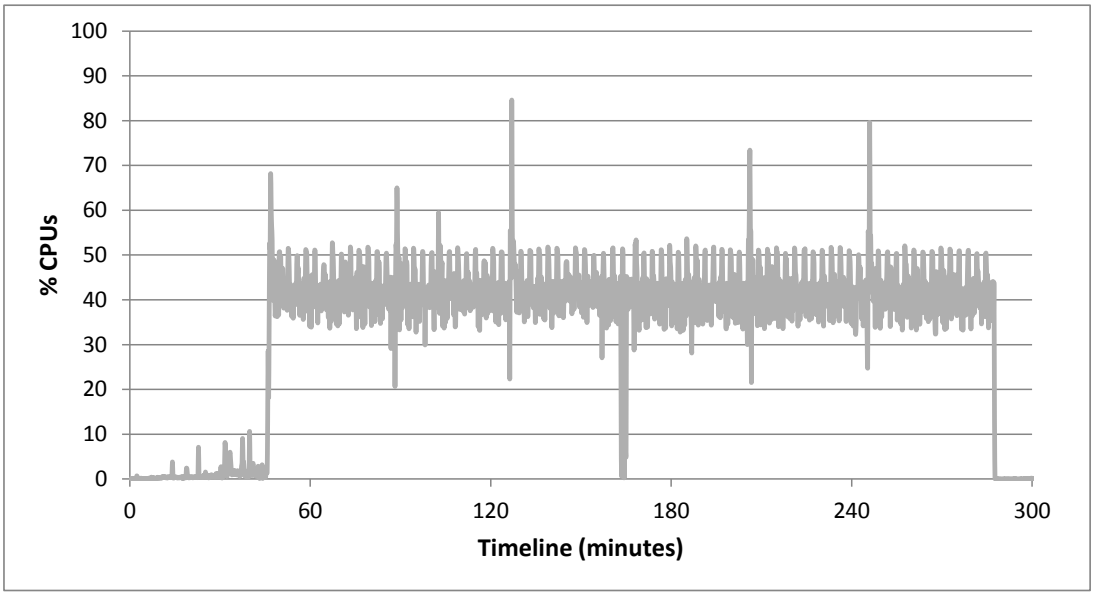

Figure 4.10: CPU usage of *slave1*, in percentage, when running Step II of `HSCI_Seq`.

4.4.4 Code Improvements

Following the profiling activity presented in the previous section, we pinpointed three issues affecting the performance of `HSCI_Seq`:

- **Excessive network traffic.** The big amount of data exchanged between map tasks and reduce tasks during Step I gives rise to network congestions that slow down in a significant way the data gathering phase of reduce tasks.
- **Poor CPU usage.** The map phase during Step II is characterized by an intense CPU activity, but it is not able to take advantage of the availability of an additional CPU core.
- **Bad intermediate-data partitioning strategy.** The standard reduce task partitioning strategy implemented by Hadoop does not guarantee during Step I of the algorithm a fair and balanced assignment to the slave nodes of the RPs to be calculated.

Starting from here, we developed and tried several algorithmic and/or practical improvements to solve these issues. The most significant ones are discussed in the following.

4.4.4.1 Excessive Network Traffic

The excessive network traffic arising in Step I is mostly due to the transfer of a large number of *RN* files from map tasks to reduce tasks. Consequently, we required each map task to aggregate all the *RN*s generated for a same camera into one *RN* file, before sending it to the corresponding reduce task. This allows us to transmit the residual noise of a group of images at the same cost as that of one of them. The aggregation is done by summing all the *RN* files produced by a same node for a same camera during a map task. To facilitate this operation, the enrollment images are ordered by the *camera id* and the partial sum of the *RN* files is kept in memory by the node, without involving any I/O disk operation.

From a technical viewpoint, this aggregation would have not been possible using the standard Hadoop Combiner, this facility requiring all the objects to aggregate (*i.e.*, the *RN* files) to be stored in memory (see Sections 3.5 and 3.6 for additional details). Such a strategy is not adequate in our case, because the size of all the *RN* files exceeds the physical memory of the computing nodes. As a workaround, we implemented an ad-hoc solution, by means of a code to run during the map task. It does not require to store all the *RN* files in memory, but just their sum (this solution is denoted *in-mapper local aggregation*). In addition, we use Hadoop implicit mechanism for directly passing this sum as *value* to the pair output by the node, rather than saving it on HDFS. We named `HSCI_Sum` the variant of `HSCI_Seq` featuring this improvement.

The Figure 4.11 focuses on the behavior of a slave node when running a map task on a cluster during the Step I of `HSCI_Sum`.

A possible further refinement of this improvement consists of compressing the objects containing the residual noise sums and the Reference Patterns, before sending them over the network. The expectation is that the time spent by each node to compress and decompress a file would be repaid by the smaller transmission times required to exchange the compressed files over the network. In

addition, the transmission of smaller files would also reduce the probability of a network congestion due to several nodes exchanging large files at the same time. The compression algorithm we have used is the Lempel-Ziv coding [309]. We labeled `HSCI_Zip` this refinement of `HSCI_Sum`.

4.4.4.2 Poor CPU Usage

The standard behavior of the map task during Step II requires the loading from the local file system of a camera RP, followed by the calculation of its correlation with an input RN file. While carrying out the first activity, the CPU is almost unused, as it is essentially an I/O-intensive operation. The second activity, instead, is CPU-intensive and makes no use of the file system.

A possible intra-parallelization of this task, allowing for the usage of a second CPU core, consists in modeling the loading and the correlation activities on the *producer-consumer* paradigm, then to be implemented as a multi-threaded application. A first thread would be in charge of loading RP files from the local file system and adding them to an in-memory shared queue. In the meanwhile, the second thread would load RP files from the shared queue and would use them to calculate the correlation with an input RN. Both threads are executed concurrently, so that, while one thread is calculating the correlation between the input RN file and the RP of a given camera, the other thread is loading in memory the RP of the next camera. Notice that it is not possible to maintain in memory the RP of all the cameras because of their large size. The implementation of this strategy, here denoted `HSCI_PC`[1], also includes the improvements introduced by `HSCI_Sum`.

Figure 4.12 focuses on the behavior of a slave node when running a map task on a cluster during Step II of `HSCI_PC`.

4.4.4.3 Bad Intermediate-data Partitioning Strategy

The standard partitioning strategy implemented by Hadoop, when used for allocating reduce functions according to the *camera id* (as required by Step I of `HSCI_Seq`), may assign multiple functions to a same slave node while leaving other

[1] `PC` in `HSCI_PC` stands for producer-consumer paradigm.

slave nodes without functions to process. This may occur because the standard hash function employed by Hadoop does not prevent the possibility of collisions. We overcome this problem by introducing a custom partitioner featuring a perfect hash function, so that wherever the number of slave nodes is higher than the number of cameras, no single node would be assigned to more than one reduce function at time. This function maps distinct keys (*i.e.*, *camera id*) on a set of integers so to guarantee a more balanced partitions. For instance, in our case, the adopted function guarantees that each node will process either none or one RP. The implementation of this strategy, here denoted `HSCI_All`, also includes the improvements introduced by `HSCI_Sum` and `HSCI_PC`.

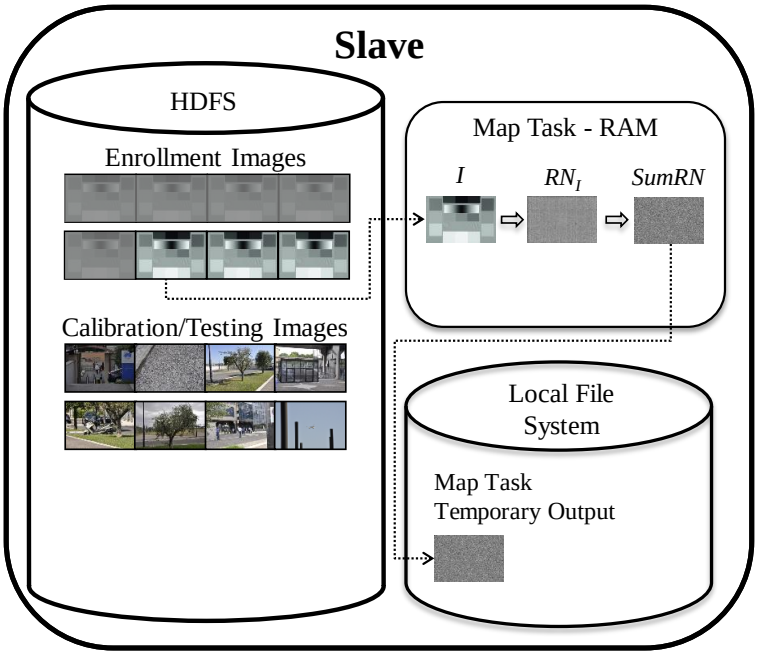

Figure 4.11: Focus on the behavior of a slave node when running a map task during Step I of `HSCI_Sum` on a cluster. The task is processing a block of pictures taken by using a same camera C and available in its local HDFS partition. For each image I in this block, the map function extracts its residual noise RN_I and adds it to $SumRN$. When no more images for C are available, the task emits, as an output, a new pair $<id_C, SumRN>$. This pair is saved on the local file system, while waiting to be transmitted to a reduce task.

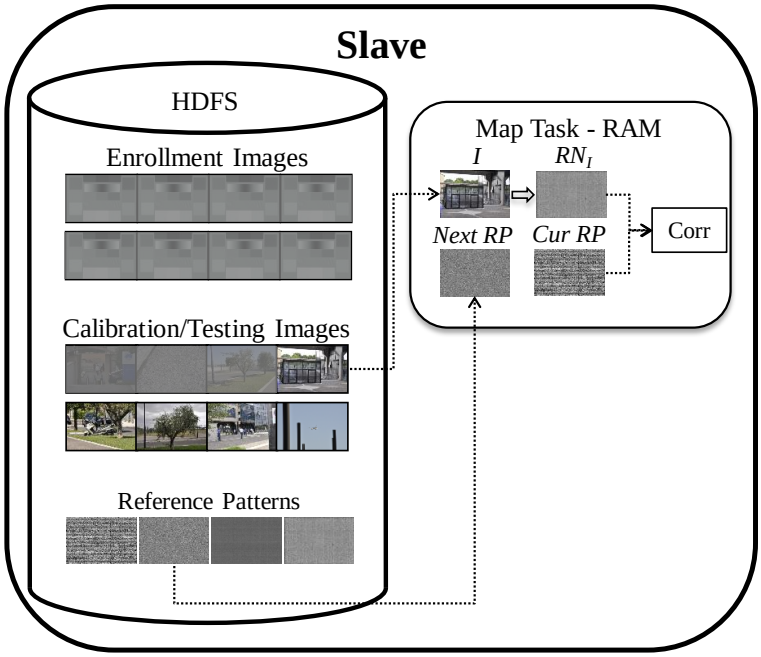

Figure 4.12: Focus on the behavior of a slave node when running a map task during Step II of `HSCI_PC` on a cluster. The task is processing a block of calibration/testing pictures available in its local HDFS partition. The same HDFS partition is used for storing a copy of all the RPs calculated during the previous step. For each image I in the input block, the map function extracts its residual noise RN_I and correlates it with all the RPs. The output of the correlation is saved on HDFS. Due to memory limits, only two RPs can be maintained in memory at the same time. So, in order to amortize RP loading times, the map task loads the next RP in an asynchronous way and using a different thread, while processing the current RP. The RPs are preloaded from HDFS to local file system at the start of the Step II.

4.4.5 Advanced Experimental Analysis

After developing the improvements presented in the previous section, we performed another round of experiments in order to compare the improved codes to `HSCI`. The results, available in Figure 4.13, report a significant performance improvement on `HSCI_Seq`. The first improved code we consider is `HSCI_Sum`. This algorithm differs from `HSCI_Seq` in the way RNs are transmitted from map tasks to reduce tasks. Namely, it implements an aggregation strategy that drastically reduces the amounts of data exchanged between map and reduce tasks. For instance, in our experiments, the amount of data exchanged during Step I by `HSCI_Sum` is about 6% of that exchanged by `HSCI_Seq` in the same phase. This led to a consistent performance improvement in our experiments, since the Step I phase of `HSCI_Sum` required 49 minutes, in the average, to be accomplished against the 290 minutes required for the same step by `HSCI_Seq`. It is interesting to note that smaller amounts of data to exchange not only imply faster communications but could also result in a much smaller number of tasks being replicated and re-run by the Hadoop framework, thanks to shorter network congestions. In addition, an aggregated RN is not directly saved on HDFS, but we use a copy of it as *value* in the $<key, value>$ map output pair. Therefore, the HDFS is not congested of residual noises, and its performance are improved, especially in Step II.

Differently from `HSCI_Sum`, the performance of `HSCI_Zip` are more contrasting. This algorithm requires the intermediate files produced by map tasks to be compressed before being transmitted, in order to reduce their size and shorten transmission times. This strategy brought a very small advantage during the execution of Step I, while heavily affecting the performance of Step II. The reason of such a bad behavior is the overhead to be paid by map tasks for decompressing RPs before correlating them with input images, during Step II.

We now turn to `HSCI_PC`. This algorithm uses the producer-consumer paradigm to evaluate correlations during the map phase of Step II, by means of a multithreaded architecture. This approach brought a consistent performance gain compared to `HSCI_Seq` and `HSCI_Sum`, as the overall execution time of Step II dropped from 304 (`HSCI_Seq`) and 276 (`HSCI_Sum`) to 236 minutes (`HSCI_PC`).

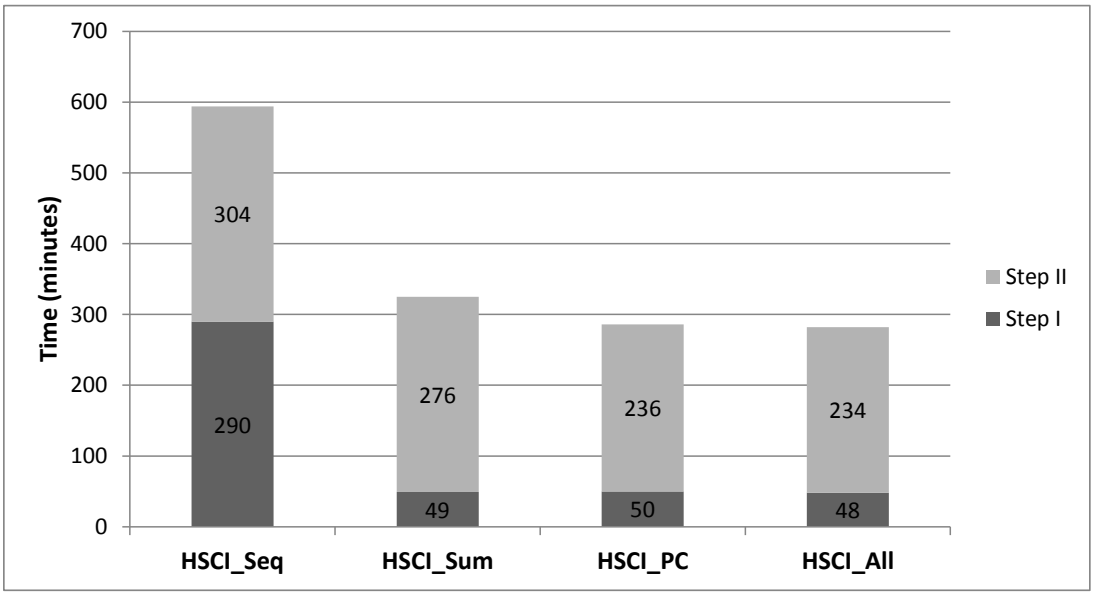

Figure 4.13: Execution times, in minutes, of the different steps of the variants of the *Fridrich et al.* algorithm on a Hadoop cluster of 32 slave nodes. `SCI` is left out from the chart because its performance is out of scale. We also exclude the performance of `HSCI`, `HSCI_Tar` and `HSCI_Zip`.

The result also includes a consistent increasing in the CPU usage, exhibited by `HSCI_PC` when processing the map phase of Step II and shown in Figure 4.14 (for a comparison see Figure 4.10).

Finally, we consider `HSCI_All`. This algorithm uses a custom partitioner to ensure that, in our setting, two reduce functions cannot be assigned to a same slave node during Step I, while leaving other nodes unused (unless duplicate tasks). Even in this case, we noticed a slight performance improvement on `HSCI_Sum` during Step I (48 minutes against 49 minutes), though smaller than we expected. A closer investigation revealed that, on one side, the custom Partitioner was able to avoid the assignment of two different reduce functions to a same node (see Figure 4.15), and that, on the other side, the stack of improvements decreased the average execution time of the reduce functions so much that the effects of this last improvement were quite negligible.

Table 4.5 shows the execution times, in minutes rounded, of the different variants of the *Fridrich et al.* algorithm on a Hadoop cluster of 32 slave nodes

Table 4.5: Execution times, in minutes, of the different variants of the *Fridrich et al.* algorithm on a Hadoop cluster of 32 slave nodes compared with the sequential counterpart run on a single node. The HDFS replication factor is 2 while HDFS block size is 64 MB.

Variant	Step I	Step II
`SCI`	888	5,257
`HSCI_Seq`	290	304
`HSCI_Sum`	49	276
`HSCI_Zip`	47	457
`HSCI_PC`	50	236
`HSCI_All`	48	234

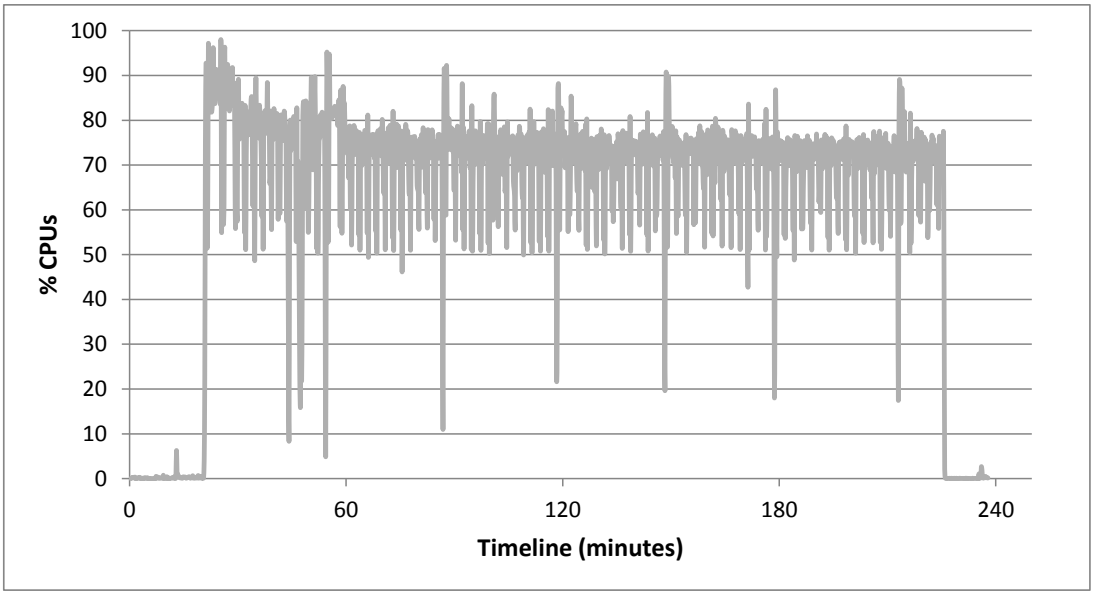

Figure 4.14: CPU usage of *slave1*, in percentage, when running Step II of `HSCI_PC`.

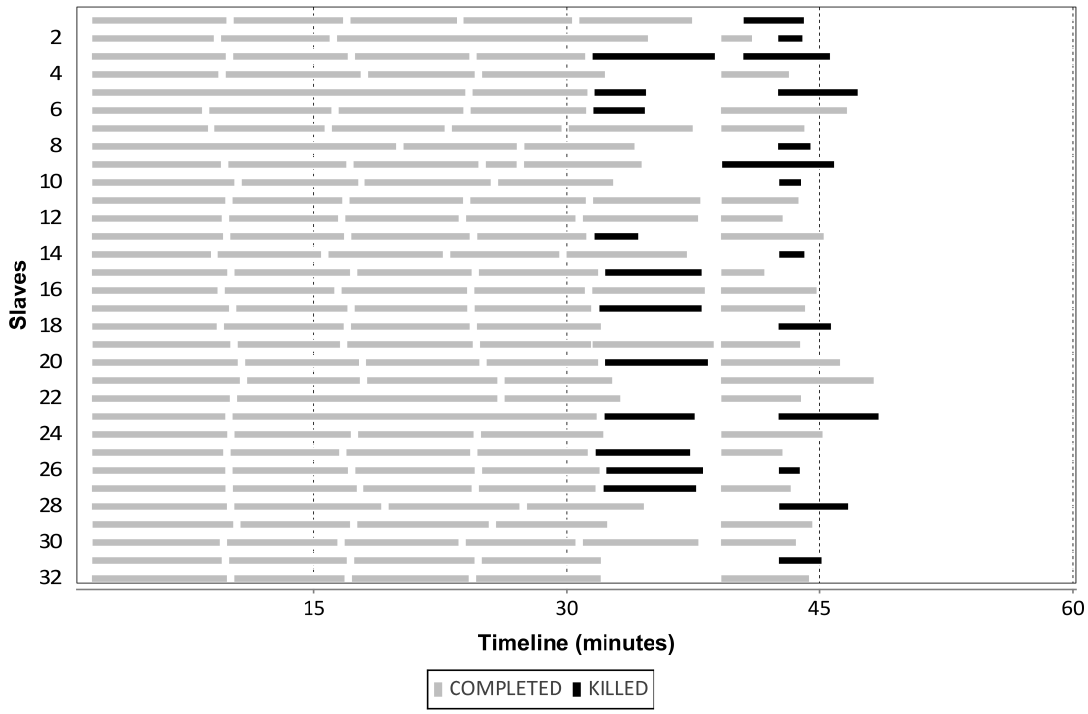

Figure 4.15: `HSCI_All` variant - An overview of map and reduce tasks launched during Step I. Notice that reduce tasks start only after the termination of all map tasks as described in Section 4.4.2.3.

compared against the sequential counterpart run on a single node. Despite our expectations, the usage of compression in order to reduce the exchange times of the *RP* files did not produce any advantage on the overall execution time, as shown in Table 4.5. On the contrary, we observed a bad increasing of the Step II execution time, likely to be due to the time spent uncompressing these files. Instead, the adoption of the producer-consumer pattern in the computation of the correlation indices improved the execution time of Step II of `HSCI_All` (or `HSCI_PC`) by about a 15% over the performance of `HSCI_Sum`.

4.4.5.1 Speed up Analysis

In this last round of experiments, we investigated the scalability of `HSCI_All` compared to its sequential counterpart, that is `SCI`.

We focused our attention on the two more intensive computational steps of

the *Fridrich et al.* algorithm: the calculation of the *RP*s (*i.e.*, Step I) and the calculation of the correlation indices (*i.e.*, Step II). We increased the size of the cluster from 4 up to 32 slave nodes, and we measured the efficiency of `HSCI_All` compared to that of `SCI` according to the following formula:

$$E(n) = \frac{T_{\texttt{SCI}}}{n \cdot T_{\texttt{HSCI\_All}}(n)}\,. \tag{4.4}$$

See Section 2.5.1 for the explanation of speed up and efficiency. In Equation 4.4, n is the number of slave nodes of the cluster, $T_{\texttt{SCI}}$ and $T_{\texttt{HSCI\_All}}(n)$ are the execution times of `SCI` and `HSCI_All`, respectively, when run on a cluster of size n. The results are available in Table 4.6, Figures 4.16 and 4.17. We observe that, as the cluster size increases, the performance improvement for Step I gets smaller than the one achieved by Step II. This drawback is due to the fact that, when processing the reduce phase of Step I using 32 slave nodes, only 20 of these are employed (excluding duplicated tasks), since 20 is the number of *RP*s to be calculated.

These results are likely to hold even when considering images shot using a much larger number of cameras. Input images, independently of their overall number, are always organized in blocks of fixed size, and each block is assembled so to contain images shot using a same camera or two cameras at most. All images of a block are stored on a same node and are likely to be processed by that node. This implies that each node is able to calculate the sum of the residual noises for a block of images without any interaction with the other nodes. An increase in the number of images per node will result in a larger number of map tasks to be executed sequentially on that node, and, again, the performance of the other nodes will not be affected. Similarly, the number of reduce tasks is proportional to the number of Reference Patterns to calculate and to the number of the nodes in the cluster. Increasing the number of cameras to process would increase proportionally the number of reduce tasks to run. These could be run either in a sequential way, when running on a small cluster, or in a completely parallel way, if executed on a much larger cluster.

In all our tests, the SVM classifier was able to correctly identify the source camera used to shoot $1,277$ images, thus achieving $\approx 99.8\%$ of recognition rate.

Table 4.6: Running times rounded to minute of the `HSCI_All` algorithm on a Hadoop cluster of increasing size.

Number of Slaves	Step I	Step II
4	288	1,725
8	146	858
16	78	451
32	48	234

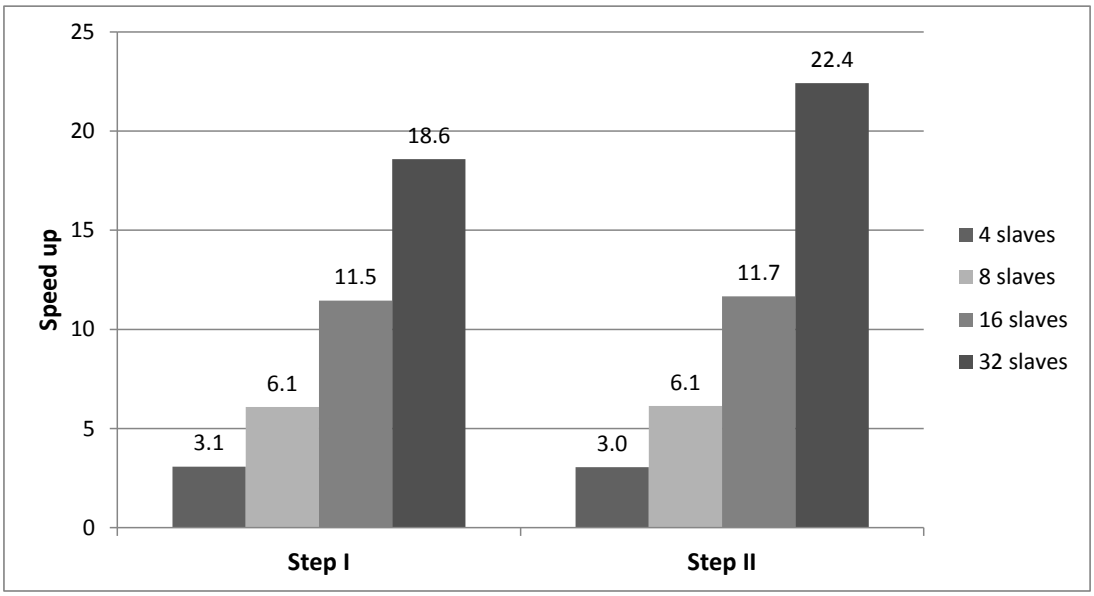

Figure 4.16: Speed up of `HSCI_All` compared to `SCI` when running on a cluster of increasing size.

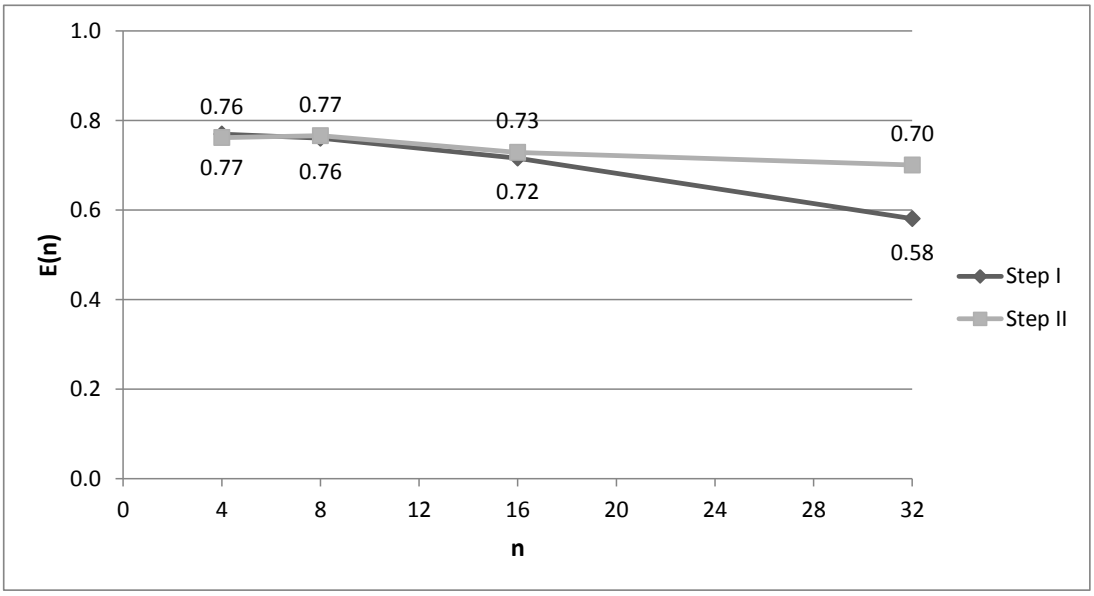

Figure 4.17: Efficiency of `HSCI_All` compared to `SCI` when running on a cluster of increasing size (n is the number of slave nodes).

4.5 Final Remarks

In this chapter, we discussed the engineering of an efficient Hadoop-based implementation of the *Fridrich et al.* algorithm, in order to solving the Source Camera Identification problem. We were able to quickly obtain a running distributed implementation for this algorithm, by leveraging the standard facilities available with the Hadoop framework to wrap up an existing implementation of the algorithm and make distributed. However, this vanilla/naive distributed implementation exhibited a very poor performance. This motivated us to perform a thorough profiling activity which led, first, to pinpoint several performance issues and, then, to develop several both theoretical and practical improvements. It is interesting to note that none of these improvements required the modification of either the original *Fridrich et al.* algorithm or its implementation. They were just geared toward a better usage of the underlying computing cluster as well as more efficient data communication and elaboration patterns. The resulting improved code exhibited a much better performance than the vanilla distributed implementation.

Our goal was to show that when porting Source Camera Identification algorithms to Hadoop, a careful profiling and engineering activity is often needed to fully exploit the real potential of this distributed computing system.

Despite the focus on the algorithm by *Fridrich et al.*, many of the improvements we developed can be trivially used to improve the performance of other Source Camera Identification algorithms when run on Hadoop (*e.g.*, [30, 120, 121, 264]). It is also possible to use these improvements to develop efficient Hadoop-based variants of algorithms belonging to different digital image forensics domains (*e.g.*, [61, 111]). In particular, in Chapter 6 are provided some insights on how our activities, *e.g.*, engineering, methodology, profiling, improvements and results, can be extended to other Source Camera Identification algorithms and to other digital image forensics problems.

Finally, an interesting future direction for our work would be the formalization of this methodology and its experimentation with other case studies. In fact, in the following chapter is addressed another research area where the Big Data problem is increasing, that is bioinformatics.

Chapter 5

Processing Big Data in Bioinformatics

Initially, in Section 5.1, the problems and some solutions about Biology and Big Data are introduced. A brief overview of the area is also presented. Section 5.2 introduces two benchmark problems studied in this chapter, and, then, our solutions to speed up biological analyses in Big Data era for these problems are presented. In particular, in Section 5.3 is described our distributed framework for the development of *Alignment-free Sequence Comparison* methods, while in Section 5.4 is discussed our fast distributed algorithm for the extraction of k-mer local and cumulative statistics. Finally, the Section 5.5 concludes this chapter with some remarks and comments.

5.1 Biology and Big Data

Another area where the problem of Big Data is emerging consists of the Computational Biology and the Bioinformatics. Bioinformatics is an interdisciplinary field that consists the collection, classification, storage, and analysis of biological data exploiting software tools. Nowadays, Next-generation DNA sequencing (NGS) machines are generating an enormous amount of sequence data, placing unprecedented demands on traditional single-processor read mapping [241]. NGS technologies feature with low cost and high throughput, therefore, they generate

an enormous amount of sequence data, called *reads*.

Here the term *read* means a short sequence of DNA (typically 25-400 base pairs long) defined on the alphabet $\{A, C, G, T, N\}$, where the N symbol denotes a not informative character. The reads are raw sequences that come off a sequencing machine, and a read is related a single genome. The reads containing one or more occurrences of N are called *untrimmed reads*, while a *trimmed read* contains no N characters. A *base* indicates a nucleotide's nucleobase in the context of a DNA or RNA, and the primary DNA nucleobases are: A, C, G and T.

As said in [145], NGS machines are generating an enormous amount of genomic data due to the decrease of genomic sequencing costs. The data output of NGS has outpaced Moore's Law (more than doubling each year). In fact, in 2005 a single sequencing run produced roughly one gigabase of data, while, in 2014 the rate climbed to a 1.8 terabases of data in a single sequencing run. At the start of 2000s, the first human genome required 15 years to sequence and cost nearly 3 billion dollars. In contrast, in 2014 modern sequencing machines sequenced over 45 human genomes in a single day for approximately $1,000$ dollars each ([145]).

Whereas the advancement of NGS and shotgun sequencing technologies produced massive amounts of genomics data, therefore, the bioinformatics researchers must now face the analysis of large-scale datasets. Indeed, biological datasets are more expensive to store, process and analyze than to generate them [212]. Nowadays all this determines the Big Data era in the computational biology, in fact, genomic experiments have to process the so-called "*Biological Big Data*" [211].

Kahn [153] in his perspective on the future of genomic data, clearly states that many of the challenges that lay ahead of genomics have a computational nature, either in terms of data management or analysis or both. Those challenges seem to arise by the now evident mismatch between sequencing capability versus storage and CPU power. In order to tackle those challenges effectively, one needs to use hardware efficiently in conjunction with good algorithm. While this second aspect has gained the prominence it deserves only recently [34], the use of hardware in the form of High Performance Computing (HPC) is a classic and even the object of courses dedicated to biologists with little background on the subject, *e.g.*, [38]. However, it is not even clear that the hardware capabilities we have now are really used to their full potential. The researchers could await weeks or months if they

use their own PCs or workstations to process this huge amount of biological data. In fact, parallel and distributed implementations can be developed for reducing the total execution time, and to ease the management, treatment and analyses of NGS data [211]. Several large-scale bioinformatics projects already benefit from parallelism techniques in HPC infrastructures as clusters, grids, graphics processing units, and clouds (*e.g.*, [22, 41, 285]).

Therefore, the dramatic fall in the cost of genomic sequencing and the increasing convenience of distributed computing resources require to develop parallel and scalable bioinformatics algorithms to analyze these data.

Masseroli et al. in [186] illustrated that the data generated by NGS technologies has not been matched by corresponding progress in data query, integration, search and analysis, thus creating a gap in the potential use of NGS data.

Berger et al. in [34] have described that the big volume of biological data makes the arising problems computationally infeasible. The widening gap between data generation and computing power implies that many of the established ways of analyzing smaller datasets simply cannot scale, not even with faster computers or with cloud computing. For example, popular search algorithms, such as BLAST [205], are becoming too slow. Adopting the paradigm of *compressive genomics* (*e.g.*, [67, 116]), data are compressed in such a way that they can be efficiently and accurately searched without decompressing first. Several software platforms have been developed for basic data analysis and integration. In [34] are also described methods and tools to solve some problems, such as DNA assembly and biologic data mining.

Giancarlo et al. in [116] have highlighted that the the Big Data era requires the design of efficient and effective methodologies for both their compression and storage. In fact, in their paper, are surveyed methods and tools used to compress DNA sequences, and in addition, some methods that use compressed data are also presented (*compressive sequence analysis*). Another area in which data compression has played a key role is alignment-free comparison of biological sequences. For example, in [67] is presented a method for clustering based on compression.

In this chapter we choose to work on two simple and most pervasive problems in computational biology, and study the solutions available for their in terms of

how effectively the computing power we have is used. Following [179], we concentrate on *lab-scale* hardware, and we use methods and techniques proper of algorithm engineering [52]. It is appropriate to point out that algorithm engineering have boomed in Computer Science in the past 15 years but seems to be mostly undetected in bioinformatics.

The problem of comparing large collections of genomic sequences has been only recently considered. Counting the number of occurrences of every k-mer[1] in a sequence is a central problem in many applications, such as genome assembly, error correction of sequencing reads, fast multiple sequence alignment and repeat detection (*e.g.*, [44, 92, 137, 148, 156, 161, 169, 184, 200, 203, 252]). K-mer counting is conceptually and programmatically one of the simplest jobs, if we do not care about the efficiency. In fact, the number of existing contributions on this problem advises that an efficient solution, with reasonable memory use, is far from trivial. Many of the algorithms formulated so far perform efficiently when run on short sequence data, but they do not work well when run on much longer sequences. Taking as an example the simplest problem of the k-mer counting on very long sequences, the analysis of these sequences is slowed by the inability of the currently available k-mer counting tools to process these sequences in an time and memory efficient way. In fact, the data generated by NGS technologies have caused the growth of the sequence to be analyzed, whereby the current stand-alone (non-parallel) k-mer counting tools too slow and memory-intensive. These operations can be significantly accelerated by reformulating the algorithms as distributed algorithms and taking advantage from several computers at the same time.

In the following we present the sequence analysis problem with a brief overview of the area. In addition, a summary of the main contributions in bioinformatics to process Big Data are also presented (see Section 5.1.2).

5.1.1 A Brief Overview about the Sequence Analysis

One of the main goals of Biology and, more in general, the *Life Sciences* in the study of biological sequences is to assess either homology or function or both.

[1] A k-mer is one of the all the possible substrings of length k that are contained in a string.

The first consists of establishing the evolution of a biological sequence, which can be an entire genome or even a single gene. The second consists of discovering what is the function of a biological sequence, usually newly discovered.

Both homology and function are nearly impossible to formalize in mathematical terms since they are inherently related to the evolution of living species, which is a process that can be described only in part by Mathematics. Yet, it has been observed and experimentally validated that "similarity" among a set of biological sequences gives, in most cases, good indications about common ancestry and function [131]. Therefore, in order to perform investigations about homology and function with the use of computational methods, one fundamental step is the design of good mathematical functions that can quantify how "similar" are a set of sequences. A good similarity function must satisfy two criteria: be informative in terms of biological research, be fast to compute and be frugal in terms of space usage.

The *Sequence Analysis* is the major field of search in bioinformatics. It is a way of arranging the sequences of DNA, RNA, or protein to identify regions of similarity that may be a consequence of functional, structural, or evolutionary relationships between the sequences [202]. Therefore, in bioinformatics particularly relevant is the assessment of how similar to each other biological sequences in a set are. Such an information may then be used for various further investigations, *e.g.*, phylogenetic studies. For brevity, we refer to this area with the classic term of *Sequence Comparison*, which it is a central part of Sequence Analysis (see [91, 131]).

Sequence Comparison methods can be broadly divided into two main branches: methods alignment-based or not. The first, which can be considered to be the *Holy Graal* of Sequence Analysis [131], consists of methods that assess the similarity among sequences via *Alignments* (see [20] for details). The traditional approaches for Sequence Alignment fall into two categories: *Global Sequence Alignment* and *Local Sequence Alignment*. The first attempts to align every residue in every sequence, and it is useful when the sequences in the query set are similar and of roughly equal size. Local sequence alignment is useful for dissimilar sequences that are suspected to contain regions of similarity within their larger sequence context. In fact, local alignments can identify regions of similarity

within long sequences. Many types of algorithms are been used to global or local alignment of the sequence, in fact, there are slow but formally correct methods based on dynamic programming. A general global sequence alignment technique is the *Needleman-Wunsch* algorithm [206], which is based on dynamic programming. Instead, the *Smith-Waterman* algorithm [254] is a general local alignment method also based on dynamic programming.

There are also efficient, probabilistic, heuristic algorithms that do not guarantee to find best solutions. For example, *word-based* algorithms are heuristic methods that are not guaranteed to find an optimal alignment solution, but are significantly more efficient than dynamic programming. Word-based methods identify a series of short, non-overlapping subsequences (called "*words*") in the query sequence that are then matched to candidate database sequences. Famous programs word-based are FASTA tool [273] and *Basic Local Alignment Search Tool* (BLAST) [5, 205]. The FASTA programs find regions of local or global similarity between protein or DNA sequences, either by searching protein or DNA databases, or by identifying local duplications within a sequence. The BLAST program, instead, finds regions of local similarity between sequences. This program compares nucleotide or protein sequences to sequence databases and calculates the statistical significance of matches.

In addition, there are two types of alignment analysis: *Pairwise Sequence Alignment* (PSA) and *Multiple Sequence Alignment* (MSA). Pairwise sequence alignment methods are used to find the best-matching piecewise (local) or global alignments of two query sequences, whereas multiple sequence alignment is an extension of pairwise alignment to incorporate more than two sequences at a time. Multiple alignment methods try to align all of the sequences in a given query set, and alignments are also used to aid in establishing evolutionary relationships by constructing *phylogenetic trees*[1]. A phylogenetic tree is a tree showing the inferred evolutionary relationships among various biological species using similarities and differences in their physical or genetic characteristics. In Phylogenetic is important to create a phylogenetic tree (or distance tree) from the genomic sequences. In general, these trees are derived by clustering the sequences through their distances (or similarity). The clustering can be made using algorithms such

[1]The *Phylogeny* is concerned with the evolution of species and higher taxonomic order.

as *Unweighted Pair Group Method with Arithmetic Mean* (UPGMA) [256, 255] or *Neighbor Joining* (NJ) [239].

Song et al. in [257] said that the dominant approaches for sequence comparison are alignment-based including the Smith-Waterman algorithm and BLAST. Although alignment-based approaches generally yield excellent results when the molecular sequences of interest can be reliably aligned, their applications are limited when the sequences are divergent or come from different regions of various genomes, and a reliable alignment cannot be obtained.

Unfortunately, most of the alignment methods result to be slower and slower to use, due to their intrinsic time complexity that compounds with the growing quantity of sequence data they have to process in each run. In fact, Next Generation Sequencing (NGS) has led to the generation of billions of sequence data, making it increasingly infeasible for sequence alignment to be performed on stand-alone machines (*e.g.*, single computer). In order to address, at least in part, such a drawback, a second branch of Sequence Comparison methods has emerged, named to as *Alignment-free*. Although fairly recent [280], it has boomed [279], becoming very quickly populated with methods that are particularly appealing because their running time is proportional to the length of the input sequences, even if they are usually less accurate than traditional alignment-based approaches. Moreover, they have been proven to be effective and significant for biological investigations (*e.g.*, [56, 102]). A recent account of the impact and future developments of this area is presented by *Vinga* in [279].

Therefore, a limit of alignment-based approaches is the computational complexity. In fact, very short sequences can be aligned in a short time, but most interesting problems require the alignment of lengthy sequences that cannot be aligned in a short period of time. Thus, algorithms based on *Alignment-free Sequence Comparison* provide an attractive alternative compared to traditional methods. Most of the alignment-free methods use *word frequencies* (or k-mer counting), where the "*words*" are small fragments of sequence called k-mers (or n-grams) in the literature, in which k (or n) is the fixed length of the oligonucleotide[1] to represent a sequence. In theory, these methods are not computa-

[1]The oligonucleotides are short (*oligo*) sequences of nucleotides (RNA or DNA), typically with 20 or fewer base pairs. The *nucleic acid notation* uses: A, C, G and T, to represent the

tionally expensive, but the large number of very long sequences implies long execution times. Alignment-free methods, in which shared properties of subsequences (*e.g.*, identity or match length) are extracted and used to compute a distance matrix, have recently been explored for phylogenetic inference (see [136] for details). Therefore, an alternative to MSA in phylogenetic inference is the so-called alignment-free approach, in which pairwise similarity is computed from subsequences, *e.g.*, counts of exact (or inexact) subsequences of defined length, or by extension, of conserved sequence patterns, or alternatively of match lengths.

Chan et al. in [56], using simulated sequence sets of various sizes in both nucleotides and amino acids, systematically assess the accuracy of phylogenetic inference using an alignment-free approach, based on D_2 statistics (see Section 5.3.1.1), under different evolutionary scenarios. They have found strong evidence for the scalability and the potential use of alignment-free methods in large-scale phylogenomics.

In addition, genetic recombination and, in particular, genetic shuffling are at odds with sequence comparison by alignment, which assumes conservation of contiguity between homologous segments. A variety of theoretical foundations are being used to derive alignment-free methods that overcome this limitation [280].

To better understand how the computer science can aid the biology, an example may be of help in illustrating the impact of similarity functions on biological research. Assume one is given a set of species for which one is interested in knowing their common ancestry, *i.e.*, an evolutionary taxonomy, which is usually represented via a phylogenetic tree. Usually a reliable taxonomy requires many years of investigation and deep biological knowledge. Figure 5.1 provides a taxonomy of 15 species that has been obtained solely with the use of biological knowledge, with very little computational work. It would be certainly of great benefit to the biologists to start from a working hypothesis for their classification. Here clustering, in particular Hierarchical, can be of great help: for two species, we can use as distance function the similarity between their genomes. Figure 5.2 provides an example of Hierarchical Clustering with the same 15 species as in Figure 5.1. The two trees are remarkably close, therefore the tree built with com-

four nucleotides commonly found in DNA.

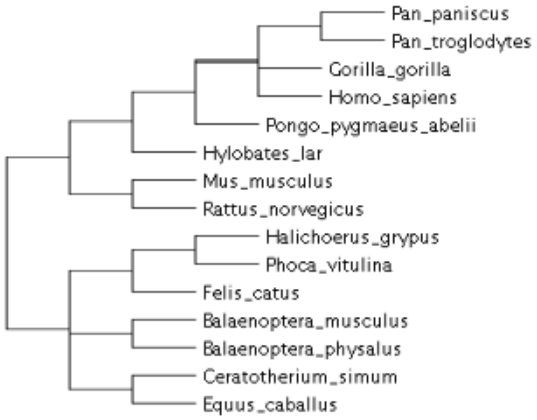

Figure 5.1: The taxonomy of 15 species obtained by the National Center for Biotechnology Information (NCBI).

putational techniques is a good starting point for a more refined taxonomy. Its construction took only a few seconds on a conventional computer.

5.1.2 Applications for Big Data Analysis in Bioinformatics

Parallel and distributed solutions for processing Big Data in bioinformatics has been explored in several scientific contributions so far. In fact, this is confirmed by the flourishing of software, experimentations and applications in the field of computational biology developed using distributed framework, *e.g.*, MapReduce paradigm.

High Performance Computing (HPC) in bioinformatics has a long history of successes, and it has been classically associated with "Big Science Bioinformatics" tasks, such as protein structure prediction (*e.g.*, [3]), or genome assembly (*e.g.*, [203]). However, the growing amount of data that the NGS technologies provide poses unprecedented computational challenges that even the most basic and routine bioinformatics applications become amenable for HPC implementations.

Due to their routine use in Sequence Analysis, in particular for large database searches, a considerable effort to parallelize the reference alignment methods has taken place, with some degree of success (*e.g.*, [172, 178, 284]). On the other hand, alignment-free methods have not enjoyed all that attention.

The MapReduce paradigm is gaining consensus as a viable approach to the

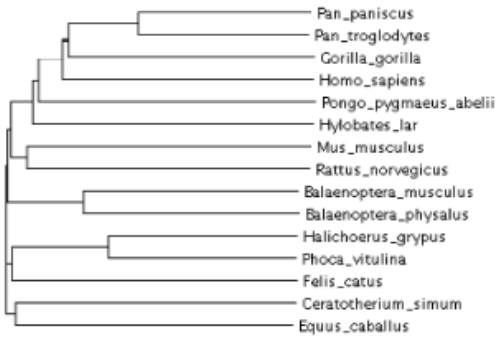

Figure 5.2: Hierarchical Clustering of the same 15 species as in Figure 5.1. It is based on a quantification of the similarity between each pair of mitochondrial genomes of the species listed at the leaves of the tree. The sequence similarity function is based on Kolmogorov Complexity and Data Compression [102].

processing of large amount of data in a scalable and efficient way. This is confirmed also by several contributions existing in bioinformatics and based on the usage of Apache Hadoop.

In the following are reviewed some popular applications used for processing Big Data in bioinformatics. These contributions are ordered by year.

Folding@home [258] is a distributed computing project for disease research that simulates protein folding, computational drug design, and other types of molecular dynamics. The project uses the idle processing resources of thousands of PCs owned by volunteer users who have installed the software on their machines.

Pairwise distances are often used to construct multiple sequence alignments. For example, the multiple sequence aligner *ClustalW* [265] is used for aligning multiple protein or nucleotide sequences, computing all pairwise distances between the input sequences. The alignment is achieved via three steps: pairwise alignment, guide-tree generation and progressive alignment. *ClustalW-MPI* [172] is a distributed and parallel implementation of ClustalW where all three steps have been parallelized to reduce the execution time. The software uses Message Passing Interface (MPI) library and runs on distributed workstation clusters as well as on traditional parallel computers.

Grid-K [128] is a *Grid Aware service for Compression-based Classification of Biological Sequences and Structures*. The application is a Grid tool for the classi-

fication of biological sequences and structures, based on Kolmogorov Complexity, Universal Similarity Measures and Data Compression.

Gaggero et al. in [112] have parallelized BLAST ([5, 205]) application and a gene expression analysis tool, called *Gene Set Enrichment Analysis* (GSEA) [260], on Hadoop.

BlastReduce [240] is a parallel read mapping algorithm for aligning sequence data from those machines to reference genomes, for use in a variety of biological analyses, such as *Single Nucleotide Polymorphism* (SNP) discovery, genotyping, and personal genomics. It is modeled after the widely used BLAST sequence alignment algorithm, but it uses Hadoop to parallelize execution to multiple compute nodes.

Kepler is a scientific workflow management systems. In [289] has been integrated Hadoop with Kepler, and it is provided an easy-to-use architecture that facilitates users to compose and execute MapReduce applications in Kepler scientific workflows.

Liu et al. in [177] have designed a pattern finding algorithm for motif based on MapReduce to improve the efficiency. In particular, it is described a MapReduce-based finding algorithm for analyzing the complex network.

Crossbow [77, 164] uses Hadoop for its calculations for whole genome resequencing analysis and SNP genotyping from short reads. It combines the aligner Bowtie [165] and the SNP caller SOAPsnp [174].

Biodoop [171] is a suite of parallel bioinformatics applications based on Hadoop consisting of three qualitatively different algorithms: BLAST, GSEA and GRAMMAR [25].

Qiu et al. in [225] have presented the experience in applying two Microsoft technologies Dryad and Azure to three bioinformatics applications. They also compare with traditional MPI and Hadoop implementations. The selected applications are an *Expressed Sequence Tag* (EST) sequence assembly program, PhyloD statistical package to identify HLA-associated viral evolution, and a pairwise Alu gene alignment application.

In [241] is presented *CloudBurst*, a parallel read-mapping algorithm for mapping next-generation sequence data to the human genome and other reference genomes. It is modeled after the short read-mapping program RMAP, and re-

ports either all alignments or the unambiguous best alignment for each read with any number of mismatches or differences. CloudBurst uses the Hadoop framework to parallelize execution using more computers. In addition, in [190] is described *CloudBLAST*, an implementation which integrates Hadoop, Virtual Workspaces, and ViNe as MapReduce, virtual machine and virtual network technologies, respectively, to deploy the commonly used bioinformatics tool BLAST on a WAN-ased test bed.

In [108] two use cases are described, one the analysis of gene sequence data (35,339 Alu sequences) and other a study of medical information (over 100,000 patient records). Here the performance of MapReduce computing model with MPI are compared. The authors look at initial processing (such as Smith-Waterman dissimilarities), clustering (using robust deterministic annealing) and *Multi Dimensional Scaling* to map high dimension data to 3D for convenient visualization.

Myrna [163] is a Hadoop pipeline for calculating differential gene expression in large RNA-Seq datasets. The authors apply Myrna to the analysis of publicly available datasets and they assess the goodness of fit of standard statistical mode.

Hydra middleware was proposed to bridge the gap between the *Current Scientific Workflow Management Systems* (SWfMS) and the HPC environment, by providing a transparent way for scientists to parallelize workflow executions while capturing distributed provenance. In [75] is presented an extension to Hydra middleware through a specific cartridge that promotes data parallelism in bioinformatics workflows.

In [285] the authors have redesigned a typical comparative genomics algorithm, the *Reciprocal Smallest Distance* (RSD) to run on Hadoop. They then employed the RSD-cloud for ortholog calculations across a wide selection of fully sequenced genomes.

In [237] is proposed an approach that combines the dynamic programming algorithm with the computational parallelism of Hadoop to improve accuracy and to accelerate of multiple sequence alignment. In addition, MapReduce is used in mapping and assembly sequence reads, and also in gene expression analysis and SNP analysis.

McKenna et al. in [192] have described the *Genome Analysis Toolkit* (GATK),

a structured Java programming framework designed to easy the development of efficient and robust analysis tools for NGS using the MapReduce paradigm. The MapReduce architecture in GATK separates the complex infrastructure needed to access the massive NGS data from logic specific to each analysis tool. In fact, GATK provides a set of data access patterns that encompass the majority of analysis tool needs.

Matthews and Williams in [191] have evaluated the viability of the MapReduce framework for designing phylogenetic applications. The problem of interest is generating the all-to-all *Robinson-Foulds* distance matrix, which has many applications for visualizing and clustering large collections of evolutionary trees. It is introduced *MapReduce Speeds up Robinson-Foulds* (MrsRF), a multi-core algorithm to generate a $t \times t$ Robinson-Foulds distance matrix between t trees using the MapReduce paradigm.

Kelley et al. in [156] have proposed *Quake*, a program to detect and correct errors in DNA sequencing reads using k-mer counting. Here Hadoop is used as proof of principle in k-mer counting.

Wang et al. in [288] have presented *mrClust*, an k-mer MapReduce-based algorithm for EST clustering, while *Taylor* in [263] has presented an overview of the current usage within the bioinformatics community of Hadoop and of associated open source software projects.

PeakRanger [101] describes a Hadoop-based framework with supports for splitting the job by chromosomes to take advantage of the *Chromosome-Level Independence* (CLI) of ChIP-seq datasets. In the CLI case, MapReduce becomes "split-by-chromosome-then-call-peaks" where chromosomes are used as keys.

Niemenmaa et al. in [207] have presented *Hadoop-BAM*, a library between *Binary Alignment/Map* (BAM) files[1] and Hadoop-based analysis applications. The alignment data is commonly stored in the standardized, compact and indexed BAM format. Hadoop-BAM acts as an integration layer between analysis applications and BAM files stored in the Hadoop Distributed File System (HDFS) that are processed using Hadoop. Hadoop-BAM solves the issues related to BAM data access by presenting a convenient API for implementing map and reduce functions that can directly operate on BAM records.

[1]The BAM format is a binary format for storing sequence data.

Almeida et al. in [4] have described a solution to sequence comparison that can be thoroughly decomposed into multiple rounds of MapReduce operations. The taken route makes use of iterated maps, a fractal analysis technique, that has been found to provide an alignment-free solution to sequence analysis and comparison. This solution not requires dynamic programming, but it uses a numeric *Chaos Game Representation* (CGR) data structure.

Contrail [73, 242] uses Hadoop for *de novo* assembly from short sequencing reads (without using a reference genome), scaling up de Brujin graph construction, while *Zou et al.* in [310] have presented MapReduce frame-based applications that can be employed in NGS and other biological domains.

Rasheed and Rangwala in [228] have described *MrMC-MinH*, a distributed algorithm for clustering metagenome sequence reads. The algorithm is implemented within Hadoop, and it approximates the computation of pairwise sequence similarity with a minwise hashing approach. The algorithm is capable of performing agglomerative hierarchical clustering or a greedy clustering approach.

Ekanayake et al. in [224] have described a wide range of topics using Dryad MapReduce framework, including iterative MapReduce programming model to analyses the metagenomics data.

Nordberg et al. in [209] have introduced the *BioPig* sequence analysis toolkit as one of the solutions that scale to data and computation. It is built on Hadoop and Pig dataflow language, and it runs different types of analysis. For example, given a set of sequences, the *pigKmer* module computes the frequencies of each k-mer and it outputs a histogram of the k-mer counts. A number of variations of k-mer counting are available, *e.g.*, count only the number of unique reads that contain the k-mers or group k-mers within one or two hamming distance.

Masseroli et al. in [187] have presented an application, called *Bio Search Computing* (Bio-SeCo), to explorative search of distributed biomedical-molecular data and the integration of the search results to answer complex biomedical questions. The authors use services from existing applications, and they support explorative integrated search and ranking-aware combination of distributed biomedical-molecular data. In fact, they have registered in Bio-SeCo a set of bioinformatics services and their semantics connections. Bio-SeCo offers an integrated environment where to perform data exploration, which automatically

saves intermediate results, combines them taking account their partial order and supplies ordered global results.

Forer et al. in [105] have summarized a number of software solutions that exist in the domain of bioinformatics that utilize the MapReduce paradigm. In order to facilitate their utilization and integration, they then have described *Cloudgene*, a graphical workflow engine that allows these existing solutions to be easily chained together.

Drew and Hahsler in [90] have presented the *Super Threaded Reference-Free Alignment-Free N-sequence Decoder* (Strand), a highly parallel technique for the learning and classification of gene sequence data into any number of associated categories or gene sequence taxonomies. Strand uses a much longer word length with respect to RDP[1], and it does so efficiently by implementing a *Divide and Conquer* algorithm leveraging MapReduce style processing and locality sensitive hashing. Strand performs word extraction using lock-free data structures to identify unique gene sequence words. It is able to learn gene sequence taxonomies and classify new sequences faster than the RDP classifier while still achieving comparable accuracy results.

De Witte et al. in [80] have presented a parallel framework for comparative motif[2] discovery. The framework is word-based and gene-centric, and the authors have implemented two methodologies for phylogenetic footprinting: an alignment-based approach, where conservation is scored based on pregenerated multiple sequence alignments, and an alignment-free approach, where conservation does not depend on the relative position or orientation of the candidate motif. The framework was implemented in C/C++ and the MPI was used to handle the inter-node communication.

Karimi et al. in [155] have proposed a scalable method in which are used optimization techniques borrowed from database technology, namely bitmap indexes. They are used to speed up searching and matching of billions of DNA signatures[3]

[1] A word-based method for alignment-free is a naive Bayesian classifier called *Ribosomal Database Project* (RDP) Classifier [290].

[2] A sequence motif is DNA or amino-acid sequence pattern that is widespread and it has a biological significance.

[3] A DNA signature is a short nucleotide sequence fragment which is used to distinguish species across all other species.

in the short reads of thousands of different microorganisms, using Hadoop, Hive and HBase.

Hill et al. in [139] have presented *K-mulus*, an application that performs distributed BLAST queries via Hadoop using a collection of established parallelization strategies. In addition, it is provided a method to speed up BLAST by clustering the sequence database to reduce the search space for a given query.

Radenski and Ehwerhemuepha in [226] have proposed a Hadoop application to codon[1] counting using Hadoop API Streaming and local aggregation.

Schumacher et al. in [244] have presented *SeqPig*, a collection of tools to manipulate, analyze and query sequencing datasets in a scalable and simple manner, using Apache Hadoop and Apache Pig.

Wiewiórka et al. in [294] have presented *SparkSeq*, a general-purpose, flexible and easily extensible library for genomic cloud computing adopting Apache Spark. This tool can be used to build genomic analysis pipelines in *Scala* programming language and run them in an interactive way.

Zhao et al. in [306] have presented *SparkSW*, a system that implements the Smith-Waterman algorithm on Apache Spark distributed computing framework, with a couple of off-the-shelf workstations.

5.2 Selected Benchmark Problems

In the biological sciences, the collection of k-mer statistics, *i.e.*, how many times each k-mer occurs in a given set of genomic or proteomic sequences, is one of the earliest and still most valuable sequence analysis tools since those statistics can be used to infer information about function, structure and evolution of biological sequences (*e.g.*, [44, 118, 119, 137, 161, 169]). Since the chosen problem is at the start of many bioinformatics pipelines, we set the foundation for the development of efficient distributed pipelines that use k-mer statistics.

Nowadays, as said in Section 5.1.1, there is a big growth of the alignment-free methods, mostly word-based, such as [56, 136, 141, 251, 257, 267, 287]. Although these methods are computationally lighter than those alignment-based, in the

[1] A *codon* is a sequence of 3 nucleotides along the mRNA.

NGS era becomes important to speed up the running times of such comparisons when very large sequences are used.

We have seen in Section 5.1.2 that there already have been several proposals in the past about the possibility of analyzing genomic sequences in a parallel or distributed way. Many of these contributions, such as ClustalW-MPI and the Genome Analysis Toolkit, focus on problems different than the word-based alignment-free sequence comparison.

In this chapter are separately treated these two problems, *i.e.*, Alignment-free Sequence Comparison (based on word counts or word features) and K-mer Statistics (or Counting).

5.2.1 Alignment-free Sequence Comparison Problem

As said in Section 5.1.1, Sequence Comparison *i.e.*, the assessment of how similar two biological sequences are to each other, is a fundamental and routine task in computational biology and bioinformatics. Classically, alignment methods are the *de facto* standard for such an assessment. Due to the growing amount of sequence data being produced, a new class of methods has emerged: Alignment-free methods. Research in this ares has become very intense in the past few years, stimulated by the advent of NGS technologies, since those new methods are very appealing in terms of computational resources needed. Despite such an effort and in contrast with sequence alignment methods, no systematic investigation of how to take advantage of distributed architectures to speed up alignment-free methods, has taken place. Another issue that has not received many attention is related to the possibility of using a distributed architecture to solve problem instances that are hard to solve on a stand-alone setting (or single machine, non-parallel) because of memory constraints. In Section 5.3 is provided a contribution of that kind, by evaluating the possibility of using the Hadoop distributed framework to speed up the running times of alignment-free methods based on word counts, compared to their original stand-alone (non-parallel) formulation. It is also explored the possibility of running alignment-free sequence methods on very long sequences, by using a proper MapReduce formulation able to spread on the several nodes of a Hadoop cluster the data structures required to run them.

5.2.2 K-mer Statistics Problem

Due to its fundamental nature, many algorithms computing those statistics have been developed, supported by various architectures, such as [24, 37, 83, 84, 161, 175, 184, 200, 209, 231, 244, 288]. In fact, although the problem is algorithmically very simple, the sheer amount of data that has to be processed in a typical application has motivated the development of many algorithms and software systems that try to take advantage either of parallelism or of sophisticated algorithmic techniques or both. Some of the current tools only work on short sequence data, in fact, some not work or scale well when run on much longer sequences. See Section A.2 for additional details about the state of the art on algorithms collecting K-mer statistics.

Let $\mathcal{S}$ be a set of genomic sequences, we are interested in collecting *Local Statistics* (LS), *i.e.*, how many times each of the k-mers appears exactly and separately in each of the sequences in $\mathcal{S}$, or *Cumulative Statistics* (CS), *i.e.*, how many times each of the k-mers appears exactly and cumulatively (globally) in sequences in $\mathcal{S}$.

In Section 5.4 is presented a highly engineered, scalable and efficient Hadoop solution to compute k-mer exact statistics for both LS and CS on short or long sequences. The proposed solution can exceed in performance the current faster implementations (Hadoop-based or not), in addition, a careful profiling of some of the most successful methods that have been developed for CS, from which it is evident that they do not scale well with computational resources, is also presented in Section A.3.

5.3 A Distributed Framework for the Development of Alignment-free Sequence Comparison Methods

Alignment-free methods are an alternative to traditional alignment-based algorithms able to process efficiently very long sequences. They are interesting from this viewpoint because their running time is proportional to the length of the in-

put sequences, although they are usually less accurate than traditional alignment-based approaches. Despite this, many of these methods and the corresponding programs are likely to perform very poorly or may not work at all, when run on very long sequences in NGS era. Following a trend that has been established in the last decades, the efficient processing of big amount of sequence data does not necessarily require very expensive super computing facilities. Instead, it is often more convenient to distribute the processing activity on a large number of commodity computers. Many of the alignment-free sequencing algorithms have been originally conceived as stand-alone (sequential) algorithms and, thus, they must be reformulated as distributed algorithms. Moreover, it is also required to rethink in a distributed fashion all the support activities related to the management of the input sequences to analyze. Starting from this premise, we experimented with the possibility of reformulating several of the alignment-free sequence analysis algorithms proposed so far in the literature as distributed algorithms by means of the MapReduce paradigm. This paradigm seems to be very well suited for activities like those required by alignment-free sequencing algorithms, usually requiring the independent processing of several (potentially very long) sequences.

In this section we investigate systematically how alignment-free methods can be designed and engineered to take full advantage of computer architectures, exploiting Hadoop framework. The choice of a distributed architecture is due to the fact that, although neglected in the past in bioinformatics, it is now being considered as a viable framework for the fast solution of computational biology problems, *e.g.*, [244]. Moreover, we also concentrate on a representative sample of the word-based alignment-free methods available, presented in Section 5.3.1, focusing on how one can achieve effective gains using a distributed environment by starting with rather simple implementations. The selected methods are based on *exact-word counts* algorithms (*e.g.*, Euclidean [136, 300], D_2 Statistics [56, 257, 267, 287], Feature Frequency Profile [251]) and *inexact-word counts* algorithms (*e.g.*, Spaced-Words [141], Co-phylog [302]).

In order to obtain a systematic study, a software framework has been designed and developed with the intent to simplify the implementation and the experimentation with alignment-free sequence comparison algorithms based on word counts. The stand-alone framework is described in Section 5.3.2 and it has been used for

implementing several of the algorithms presented in Section 5.3.1.

A first round of experiments has been conducted on a single CPU core, where we measured the performance of these algorithms on a reference dataset. The aim of these experiments was not to compare neither to evaluate the quality of the solutions found by the different algorithms. Instead, we were interested in characterizing the maximum size of the problems that could be practically solved by these algorithms on a stand-alone setting and, also, to pinpoint the issues that would prevent these algorithms to work efficiently, or to work at all, on larger problems.

Starting from these considerations, in Section 5.3.3, it is presented a reformulation of the most promising traditional alignment-free sequence analysis according to the MapReduce, and their consequent implementations. Then, we developed an implementation for these algorithms on the top of the Apache Hadoop distributed framework. By taking advantage of careful profiling analysis of the algorithms, we engineer very fast implementations of them. The proposed framework is called *Alignment-free Sequence Comparison on Hadoop* (`HAFS`).

Finally, in Section 5.3.4, we repeated the same experiments performed on the stand-alone setting (*i.e.*, single-core) on a cluster of 5 multi-processor workstations running Hadoop, and we have compared the corresponding results. We also present a performance analysis and profiling of the implementations acquired during the previous activities.

A preliminary version of the research work presented in this section was published in [49].

5.3.1 Alignment-free Sequence Comparison Methods

In the following, we briefly review the alignment-free methods used in this study. They are a representative set of the myriads available, and they have been chosen either because considered fundamental in the literature or because particularly innovative.

As already stated, generally, the similarity of biological sequences is established via alignments. However, the scales of the problems have drastically changed. For instance, one could easily face taxonomic classification tasks that

involve millions of species. Indeed, the study of entire bacterial populations is becoming a standard in metagenomics [243].

Fortunately, the need for similarity functions substantially different than the ones based on sequence alignment was readily realized, although it goes to the merit of *Vinga and Almeida* [280] to have given to this area a very fortunate name: *Alignment-free Sequence Comparison*. The area has boomed in the past 10 years, going from a handful of papers to hundreds of them (see [279]).

Algorithms in this area can be broadly classified into two main categories: *explicit collection of (sub)-sequence statistics* and *implicit collection of (sub)-sequence statistics*.

- **Explicit Collection of (Sub)-Sequence Statistics**
  Consider a sequence S, and let D_S be the number of subsequences of length k that appear in S, together with their number of occurrences. In fact, D_S is an explicit collection of a sequence statistics about S. Letting D_Q be for a sequence Q as D_S for S, how "close" are D_Q and D_S is certainly an indication of how "close" Q and S are. All methods described in this section belong to this category.

- **Implicit Collection of (Sub)-Sequence Statistics**
  It is well known that classic data compression algorithms, *e.g.*, Lempel-Ziv [309], reduce the dimension of a text by eliminating redundancy, which in many cases consists of repeated parts of the text. In a sense, in doing their job, those algorithms implicitly collect and use sequence statistics about the text to compress. Although not entirely obvious, such a feature has deep connections with Kolmogorov Complexity which, in turn, is at the base of universal similarity metrics for sequences. Additional information is presented in *Ferragina et al.* [102].

Alignment-free methods in biology and the life sciences have been applied in order to extract universal *genomic*, *proteomic* and *epigenomic* features. That is, sequence features that characterize a class of biological processes in the mentioned areas. For example in genomics and proteomics there are the following studies: composition of amino acidic sequences: common rules, even for structurally

and evolutionarily diverse sequences [19]; k-mers distribution across species: unimodality and additional common features [66]; informational genome analysis: how different are genomes? [45]; mammalian enhancers comparison [70]. Instead, in epigenomics, for example, there are the following studies: motif-free sequence specificity detection in epigenomics [221] and epigenomic dictionaries for nucleosome positioning [117].

As mentioned above, alignment-free methods became popular in recent years, since their run time is usually proportional to the total sequence length, but it is known that alignment-free methods are generally less accurate than alignment-based approaches. Efficient distance computation is the major contribution of alignment-free methods. In fact, as defined by *Haubold* in [136], there are many methods used to compute the distance (or the similarity) between two sequences. These methods are mainly based on frequencies of words of some fixed length (*i.e.*, *word counts* or k-mer counts), or on the lengths of exact matches (*i.e.*, *match lengths*) between pairs of sequences. When counting frequencies of words, we can count the *exact-word* or *inexact-word*. An inexact-word (or *approximate-word*) pattern contains *do not care* (wildcard) characters, where a *do not care* character is denoted by 0, while 1 denotes a match. For example, if a word pattern is defined as 101, then word ATA matches AAA and ATA, but not TAA. In general, all of these methods are used to give a data representation, *i.e.*, phylogenetic tree (or distance tree), from a set of input sequences using pairwise distances between the sequences (distance matrix). A distance matrix can be used to construct phylogenetic tree using clustering algorithms.

In this section we only use alignment-free methods based on exact-word counts or inexact-word counts.

5.3.1.1 Methods based on Exact-Word Counts

Consider an alphabet Σ of n symbols and an integer $k \geq 1$ (sometimes called *word pattern*). Formally, Σ^k is the set of k-mers and here it is assumed to be sorted lexicographically, so that the integer i in $[1, n^k]$ can represent the i-th k-mer in the list. A k-mer[1] is a substring of exactly k characters. This term, typically,

[1] K-mers are also called k-tuples, k-grams, k-words or k continuous words. Therefore, throughout this chapter, we use these terms interchangeably.

refers to all the possible substrings, of length k, that are contained in a string.

Comparison of the similarities between two segments of biological sequences using k-mers arises from the need for rapid sequence comparison. This very simple method computes the difference of overlapping k-mer frequencies between sets of sequences. The amount of k-mers possible given a string of length L, is $L - k + 1$, whereas the amount of possible k-mers given n possibilities[1] is n^k.

In particular, in the methods based on k-mers, for alignment-free sequence comparison of two input sequences S and Q, the first step is to count the number of occurrences of every k-mer in the sequences separately and then it needs to record the k-mer frequencies for each sequence (this counting can be carried out in linear time assuming k as a constant). In addition, a measure d of "difference" between the two sequences is defined based on the two frequencies vectors. If the measure satisfies distance constraints (*i.e.*, $d(S,Q) \geq 0$; $d(S,Q) = 0 \iff S = Q$; $d(S,Q) = d(Q,S)$; and $\forall\, S,Q,T : d(S,T) \leq d(S,Q) + d(Q,T)$), then the measure is a *distance* (or *metric*), otherwise we called it *dissimilarity measure*[2]. A dissimilarity measure indicates how two sequences are different.

In literature a large number of measures are been calculated using k-mer frequencies (*e.g.*, [63, 140, 230]). A simple example is the *Squared Euclidean Dissimilarity Measure* [136]. The next three methods are representative of the ones collecting the number of exact occurrences of each k-mer in the two sequences to be compared. In fact, we present some popular methods based on word frequencies, that is: Squared Euclidean dissimilarity measure, D_2 Statistics (or Scores) and *Feature Frequency Profile* (FFP).

Squared Euclidean Dissimilarity Measure It is a dissimilarity measure, proposed in [300], and defined as:

$$d_{SE}(S,Q) = \sum_{i=1}^{n^k} (s_i - q_i)^2 \tag{5.1}$$

[1] In the case of DNA sequences, n is 4, *i.e.*, $\Sigma = \{A, C, G, T\}$.

[2] In this section, generally, the term *dissimilarity measure* is also used for referring to a *distance measure*.

where n is the number of characters in the input alphabet (*e.g.*, $n = 4$ with alphabet $\Sigma = \{A, C, G, T\}$); S and Q are two sequences; s_i is the number of occurrences of the i-th k-mer in S, while q_i is its analogous in Q. The Squared Euclidean measure is not a metric as it does not satisfy the triangle inequality. Typically k is set to 5 in [300].

D_2 Statistics Another exact-word counts approach for alignment-free sequence comparison uses the D_2 statistics (see [56, 257, 267, 287, 310] for details). A D_2 score is calculated based on the exact count of shared k-mers between any two sequences, thus representing the extent of similarity they share. Formally:

$$D_2(S, Q) = \sum_{i=1}^{n^k} s_i \times q_i \tag{5.2}$$

where s_i and q_i are the same as in the Squared Euclidean dissimilarity.

It was pointed out in [176] that D_2 is not appropriate for the comparison of two sequences because it may have biases. In fact, D_2 statistic tends to be dominated by single-sequence noise. In particular, one can normalize the D_2 score, *e.g.*, via the use of the a priori probability of occurrence for each k-mer observed in a sequence (D_2^S Statistic) or via an a priori estimate of the mean and variance of k-mer occurrences (D_2^* Statistic) in a sequence.

D_2^S (see [230, 287]) is a self-standardized statistic and it is based on Shepp's statistic, in which a D_2 score is normalized based on probability of occurrences of specific k-mer in the sequence[1]. For a k-mer $\mathbf{w} = (w_1, ..., w_k)$, $p_{\mathbf{w}} = \prod_{i=1}^{k} p_{w_i}$ is the probability of occurrence of $\mathbf{w}$. In particular, $p_{\mathbf{w}}$ is the probability of word $\mathbf{w}$ under the null model. Let $\bar{s}$ and $\bar{q}$ be the number of all possible $\mathbf{w}$ (*i.e.*, k-mers) respectively in sequences S and Q (*i.e.*, $\bar{s} = s-k+1$ and $\bar{q} = q-k+1$, where s and q are the lengths of S and Q respectively), and that $p_{\mathbf{w}}^S$ and $p_{\mathbf{w}}^Q$ the probability of a specific k-mer $\mathbf{w}$ respectively in sequences S and Q. $S_{\mathbf{w}}$ counts the number of occurrences of $\mathbf{w}$ in S, and similarly, $Q_{\mathbf{w}}$ counts the number of occurrences of $\mathbf{w}$ in Q. $S_{\mathbf{w}}$ and $Q_{\mathbf{w}}$ can be normalized as:

$$\tilde{S}_{\mathbf{w}} = S_{\mathbf{w}} - \bar{s}p_{\mathbf{w}}^S \quad \text{and} \quad \tilde{Q}_{\mathbf{w}} = Q_{\mathbf{w}} - \bar{q}p_{\mathbf{w}}^Q \,. \tag{5.3}$$

[1] In D_2^S the superscript "S" stands for "*Shepp*" and also for "*self-standardized*".

The probabilities $p^S_{\mathbf{w}}$ and $p^Q_{\mathbf{w}}$ are the probability of k-tuple $\mathbf{w}$ under the background model for the two input sequences.

D_2^S statistic is defined as:

$$D_2^S = \sum_{\mathbf{w}} \frac{\tilde{S}_{\mathbf{w}} \tilde{Q}_{\mathbf{w}}}{\sqrt{\tilde{S}^2_{\mathbf{w}} + \tilde{Q}^2_{\mathbf{w}}}} \,. \tag{5.4}$$

Reinert et al. [230] set $\frac{0}{0} = 0$ in Equation 5.4, and they said that, under reasonable assumptions, the D_2^S statistic is approximately normally distributed, when sequence lengths tend to infinity, and not dominated by the noise in the individual sequences.

The statistic D_2^* is based on centered counts, divided by the square root of their means. Similarly, D_2^* is based on the postulation that number of occurrences of word $\mathbf{w}$ (*i.e.*, k-mer) is approximately Poisson, therefore its mean and variance are approximately the same for long word $\mathbf{w}$ (see [230, 287]). In *Song et al.* [257] D_2^* is defined as:

$$D_2^* = \sum_{\mathbf{w}} \frac{\tilde{S}_{\mathbf{w}} \tilde{Q}_{\mathbf{w}}}{\sqrt{\bar{q}\bar{s} p^S_{\mathbf{w}} p^Q_{\mathbf{w}}}} \,. \tag{5.5}$$

In [230] the authors replaced p_a, the (unobserved) letter probabilities, by $\widehat{p}(a)$, that is the relative count of letter a in the concatenation of the two sequences, based on the null hypothesis that the two sequences are independent and both are generated by *i.i.d.* (*independent and identically distributed*) letters from the same distribution. Then it is estimated the probability of occurrence of $\mathbf{w} = w_1, ...w_k$ by $\widehat{p}_{\mathbf{w}} = \prod_{i=1}^{k} \widehat{p}_{w_i}$. In *Reinert et al.* [230] are estimated the letter probabilities, even when it is assumed that all letters are equally likely.

$$D_2^* = \sum_{\mathbf{w}} \frac{\tilde{S}_{\mathbf{w}} \tilde{Q}_{\mathbf{w}}}{\sqrt{\bar{s}\bar{q}\widehat{p}_{\mathbf{w}}}} \,. \tag{5.6}$$

The authors in [230] set $\frac{0}{0} = 0$ in Equation 5.6; and they have shown that D_2^* outperforms D_2^S in terms of power for detecting the relatedness between the two sequences.

D_2^* is based on the intuitive idea that the number of occurrences of k-mer $\mathbf{w}$

is approximately the same for relatively long tuples.

Feature Frequency Profile (FFP) This technique, proposed by *Sims and Kim* in [251], always computes the count of each possible feature (*i.e.*, k-mer) in an input sequence. Each word count in each sequence is normalized by dividing it by the total number of features existing in that sequence. Then, the resulting features with associated normalized count are grouped together to form the *feature profile* of that sequence. It has been shown that similar sequences exhibit similar profiles. Thus, it is possible to estimate the distance between two sequences by measuring the dissimilarity between their respective FFPs. This can be calculated by measuring the similarity between two probability distributions, using the *Jensen-Shannon Divergence* (JSD) method ([250]).

The JSD is a popular method of measuring the similarity between two probability distributions, and it is a symmetric and smoothed version of the *Kullback-Leibler Divergence* (KLD).

JSD of Q from S, denoted $\mathrm{D_{JS}}(S \parallel Q)$, is defined as:

$$d_{FFP}(S,Q) = \mathrm{D_{JS}}(S \parallel Q) = \frac{1}{2}D_{KL}(S \parallel M) + \frac{1}{2}D_{KL}(Q \parallel M) \tag{5.7}$$

where

$$M = \frac{1}{2}(S+Q) \tag{5.8}$$

and the KLD of Q from S, denoted $D_{KL}(S \parallel Q)$, is defined as:

$$D_{KL}(S\|Q) = \sum_{i=1}^{n^k} s_i \ln \frac{s_i}{q_i}\,. \tag{5.9}$$

KLD is a non-symmetric measure of the difference between two probability distributions S and Q. Specifically, the KLD of Q from S, denoted $D_{KL}(S\|Q)$, is a measure of the information lost when Q is used to approximate S. KLD measures the expected number of extra bits required to code samples from S when using a code optimized for Q, rather than using the true code optimized for S.

5.3.1.2 Methods based on Inexact-Word Counts

The previous word count methods are designed to recover the topology of a phylogeny rather than its branch lengths [136]. A well-known drawback of using exact-word counts in sequence comparison is that word matches at neighbouring sequence positions are statistically far from independent. In view of how biological sequences evolve, *i.e.*, via insertions, deletions and substitutions of symbols, it is quite natural to consider alignment-free methods that account for occurrences of inexact-words in sequences. We consider here some very recent proposals.

Spaced-Word Frequencies A spaced-word over an alphabet Σ can be seen as a word composed of symbols from Σ and wild-card symbols, *e.g.*, $T{*}{*}AG{*}T$. The basic version of this spaced-word approach uses one single fixed *pattern* P of *match* and *do not care* positions, represented as a sequence of 1 and 0, respectively, and calculates the relative frequencies of spaced-words with respect to this pattern (see [40, 141, 272] for details). The first and last characters in P must be 1. For example, the pattern $P = 101$ reports a central *do not care* position and two lateral *match* positions. In this example, the word ATA matches with AAA and ATA, but not TAA. We call this type of pattern P as *spaced pattern*[1]. Having calculated the frequencies of spaced-words in the input sequences, their similarity/dissimilarity can be determined using a proper measure, *e.g.*, the ones described in Section 5.3.1.1, such as Euclidean or JSD. The dissimilarity measure between two sequences S and Q using a pattern P, $d_P(S, Q)$, is the distance between the corresponding frequency vectors.

Multiple Patterns Spaced-Words The spaced-word technique has been further extended by *Leimeister et al.* in [170] with the replacement of the single pattern P with a set of patterns $\mathcal{P} = \{P_1, .., P_m\}$. To be more precise, given two sequences S and Q, this technique averages the dissimilarity measures calculated with respect to all individual patterns in the set $\mathcal{P}$. Therefore, the dissimilarity measure $d_{\mathcal{P}}$ is defined as:

[1] We call the matching k-mers of a *spaced pattern* as *spaced-words* or *spaced-k-mers*.

$$d_{\mathcal{P}}(S,Q) = \frac{1}{m}\sum_{P\in\mathcal{P}} d_P(S,Q)\,. \tag{5.10}$$

In the results presented in [170], *Leimeister et al.* have shown that spaced-word frequencies based on a single pattern with a small number of *do not care* positions lead to better phylogenetic trees than contiguous word[1] frequencies (although the improvement achieved with this first approach is limited). But a significant improvement is obtained by using the multiple patterns approach: the resulting phylogenetic trees are superior to the trees constructed with contiguous word frequencies or single-pattern spaced-words, and the results are less sensitive to the number of *do not care* positions. In particular, the results of spaced-words are improved if the number of patterns is increased, but this also increases the run time. On simulated DNA and protein sequences, the authors have observed that the quality of the results converges to an optimum between 60 and 70 patterns, in fact, a further increase does not lead to a significant improvement of tree quality. Under an *i.i.d.* sequence model, the expected number of occurrences of a spaced-word is approximately the same as for the corresponding contiguous word (obtained by removing the *do not care* positions), and spaced-word matches at neighboring sequence positions are less dependent on each other if a non-periodic pattern P is used. A main advantage of spaced-word frequencies is that occurrences of spaced-words at different sequence positions are statistically less dependent on each other. Therefore, dissimilarity measures using spaced-words can be expected to be more stable.

Co-phylog *Yi and Jin* in [302] have presented Co-phylog, an assembly-free phylogenomic approach that creates a "micro-alignment" at each *object* in the sequence using the *context* of the object. It uses these objects to calculate pairwise distances. Therefore, it is not only as efficient as the existing alignment-free approaches but also as accurate as the alignment-based methods.

We define a *structure* of a pattern P to match by using a formula:

$$C_{a_1,a_2,\ldots,a_m}O_{b_1,b_2,\ldots,b_{m-1}} \tag{5.11}$$

[1] A *contiguous word* indicates a k-mer without *do not care* positions, *i.e.*, an exact-word.

where a_i $(i = 1, ..., m)$ and b_i $(i = 1, ..., m-1)$ are the lengths of the i-th consecutive 1s segment (*i.e.*, the *context*) and the i-th consecutive 0s segment (*i.e.*, the *objects*) respectively. Here, 1 always denotes a *match/care* position and 0 denotes a *do not care* position. For example, $P = 1110111$ has a structure $C_{3,3}O_1$, *i.e.*, a seed with length $k = 7$ and a wildcard character in the middle position. In the structure, C is called context and O is called object. Fixed P, the technique works by converting an input sequence in the set containing context-object pairs.

In particular, given the structure of a pattern P, for each input sequence S, we index each O-gram in S (*i.e.*, the consecutive *do not care* characters) by its respective C-gram (*i.e.*, the consecutive *match* characters). If different O-grams with the same C-gram occur while indexing the genome, the C-gram is flagged (*i.e.*, marked). After all of the O-grams are indexed, the unmarked C-grams and their respective O-grams, *i.e.*, the context-object pairs, are output.

After calculating the previous step for each input sequence, for each pair of sequences, *e.g.*, S and Q, we define d_{co} as:

$$d_{co}(S, Q) = \frac{\sum_{i=1}^{|R|} I_i}{|R|} \tag{5.12}$$

where R is the intersection of the context sets of S and Q. Moreover, $I_i = 0$ if $object_{S,P}(c_i) = object_{Q,P}(c_i)$, otherwise $I_i = 1$, where c_i is the i-th context of R and $object_{S,P}(c_i)$ indicates the object associated to the context c_i in the sequence S with the structure of the pattern P. The same explanation applies to object $object_{Q,P}(c_i)$.

Haubold in [136] has concluded that when it comes to choosing an alignment-free dissimilarity measure, d_{co} is a strong candidate, especially when analyzing large genomes where the time and/or memory consumption of other methods is often prohibitive. However, d_{co} did not return the correct primate phylogeny, so the jury is still out on which method is best.

5.3.2 Alignment-free Sequence Comparison on a Single-Core

A first round of experiments was performed to evaluate the scale of the problems that can be conveniently solved with each of the implementations considered in this study, on a Linux machine equipped with 32 GB of RAM and 2 *AMD Opteron @ 2.10 GHz* processors (16 total cores). The maximum amount of RAM memory allocated to these experiments has been set to 4,096 MB, so as to reflect the availability of RAM memory for each task of our subsequent Hadoop experiments. Here we only use a single-core, in fact, only a stand-alone (non-parallel) implementation is experimented. In these experiments, we measured the CPU and the memory usage of the algorithms, when processing different types of sequences. Performance measurement has been done by instrumenting the Java source code of the implementations.

5.3.2.1 Stand-alone Implementation

A set of Java classes has been implemented, featuring the general implementation template that can be extracted from the techniques reviewed in Sections 5.3.1.1 and 5.3.1.2.

Let $\mathcal{D}_S$ be the set of input sequences to be compared and let P be the pattern (spaced or not) to be taken into account, where $|P| = k$. The implementation pattern we consider consists of two steps:

- **Indexing:** Each sequence $S \in \mathcal{D}_S$ is processed individually in order to extract a set of features (*e.g.*, the k-mer counts) which are then stored using a Java hash map (*i.e.*, hash table[1]) data structure. Currently, the features extracted are the ones needed by the algorithms described in Sections 5.3.1.1 and 5.3.1.2. That is, exact and inexact k-mer counts and the context-object information used by the Co-phylog technique.

- **Pairwise Comparisons:** For each pair of distinct sequences $S_i, S_j \in \mathcal{D}_S$, a measure of their dissimilarity is computed, based on the features collected during the first step. The framework supports many of the dissimilarity

[1] In this chapter the terms *hash table* or *hash map* are used interchangeably.

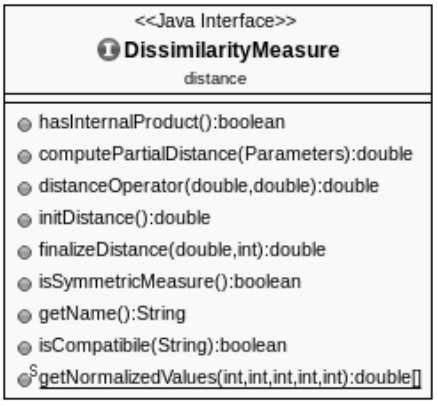

Figure 5.3: The Java Interface `DissimilarityMeasure` used in our framework to manage a dissimilarity measure for alignment-free sequence comparison.

measures, such as those presented in Section 5.3.1. In particular, it supports the dissimilarity measures: Euclidean, Squared Euclidean, KLD, JSD with absolute and relative frequencies of counts. Moreover, several variants of D_2, D_2^S and D_2^* are also available, using different methods for calculating the probability of a k-mer or of a single character of the alphabet. Finally, there is an implementation of the dissimilarity measure used by the Co-phylog method described in [302].

This implementation is sequential[1], *i.e.*, no parallel or distributed tasks are executed concurrently. In addition, the implementation is extensible to new dissimilarity measures. In fact, a developer can create a new dissimilarity measure, simply writing a Java Class that implement the Interface `DissimilarityMeasure`, as reported in Figure 5.3. The Class Diagram related to the dissimilarity measures initially implemented in our framework is presented in Figure 5.4.

5.3.2.2 Datasets

Our experiments have been conducted on a randomly-generated dataset $\mathcal{D}$ consisting of several sequences defined on the $\{A, C, G, T\}$ alphabet. The dimensions we considered for this purpose are: the length *len* of each sequence, the overall

[1] We remember that a sequential or stand-alone implementation only uses a single CPU core at a fixed time instant, *i.e.*, it is not parallel.

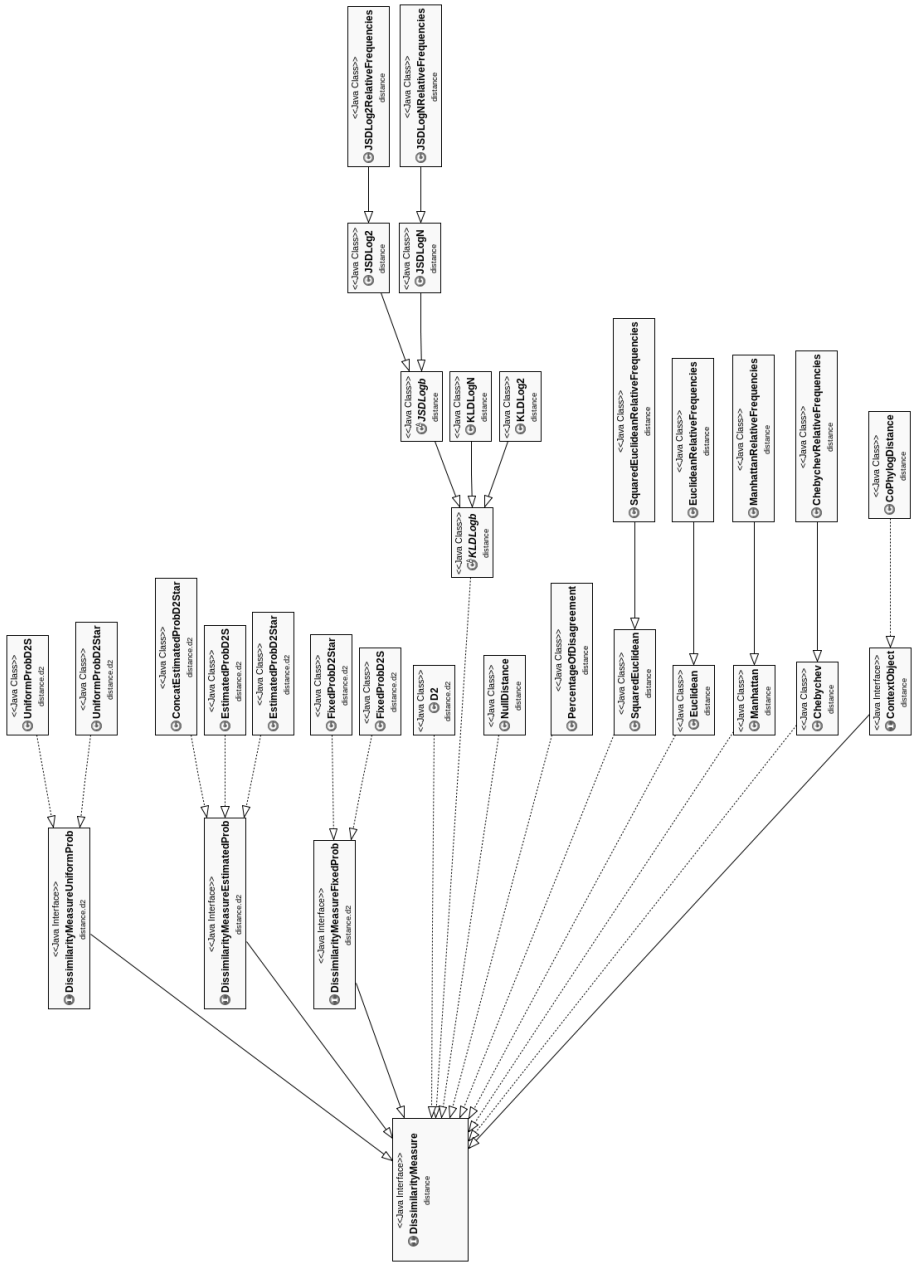

Figure 5.4: The Class Diagram related to the dissimilarity measures initially implemented in our framework for alignment-free sequence comparison.

number *numb* of sequences to be compared and the length k of the k-mers to be extracted.

We tested several different numeric assignments to these variables. Here, are reported the settings that were more challenging for the considered algorithms on our experimental platform: ($len \in \{52,428,800;\ 524,288,000;\ 1,620,000,000\}$, $numb \in \{5,\ 10,\ 15,\ 20,\ 25,\ 30\}$, $k \in \{1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9,\ 10\}$). Smaller values for those variables were handled by the input algorithms considered in this study with no problems at all, while it was not possible to experiment with larger values due to the resulting excessive memory usage and/or to the very long execution time. All the sequences have been created with the help of the Java standard pseudorandom number generator assuming a uniform distribution and have been saved individually as FASTA[1] files. See Section A.1 for additional information about FASTA file format.

5.3.2.3 Preliminary Experimental Results

In our first round of experiments, we measured the overall CPU time spent for evaluating the dissimilarity between collections of sequences having the same size and belonging to $\mathcal{D}$. For example, in Figure 5.5, we report the time spent for processing an increasing number of sequences, each having a size of $52,428,800$ characters, while using several different types of dissimilarities. We set k to 6 (*i.e.*, $P = 111111$), when using dissimilarities based on k-mers, and the pattern 1110111, when using dissimilarities based on context-object extraction. According to those results, most of the considered measures based on k-mers exhibit very similar execution time, except for the ones based on the D_2^S and D_2^* statistics with estimated probabilities. This is probably due to the overhead required by these methods for estimating the k-mer probabilities on the input sequences. Moreover, the Co-phylog measure always exhibits a longer execution time than the other dissimilarities. This is probably due to the fact that the algorithm for the extraction of the context-object information from an input sequence is more complex than the one for k-mers extraction. We were unable to run tests with values of k higher than 11 because the available memory (*i.e.*, 4 GB) was not

[1] A FASTA file is a file format, which should not be confused with FASTA software [273].

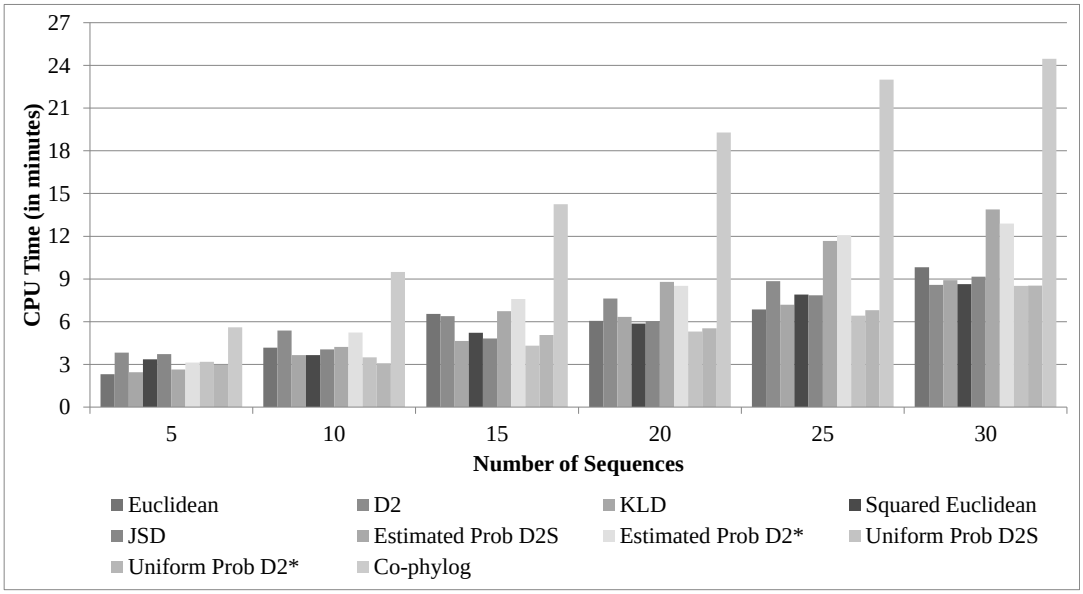

Figure 5.5: Overall CPU time required to evaluate the dissimilarities between randomly-generated sequences in collections of increasing size, with each sequence having a size of $52,428,800$ characters, while using several different types of dissimilarity measures. The parameter k is set to 6 for dissimilarities based on k-mers (*i.e.*, $P = 111111$). The Co-phylog measure uses the pattern $P = 1110111$ as its parameter.

enough to run the algorithms without loss of performance.

We then profiled the considered algorithms to better understand their internal behavior and to explain the performance we measured in our tests. We first noticed that the execution time of these algorithms is dominated by the time spent interacting with the hash map data structure used to store the k-mers or the context-object information. This explains why most of the algorithms based on k-mers exhibit approximately the same execution time. In addition, the memory required by this data structure grows exponentially with k (when counting the k-mers) or with the size of the pattern (when extracting the context-object information). As a consequence, increasing the value of k may easily lead to the creation of a hash map too big to fit in the available memory. The size of this data structure grows also when we increase the number of sequences

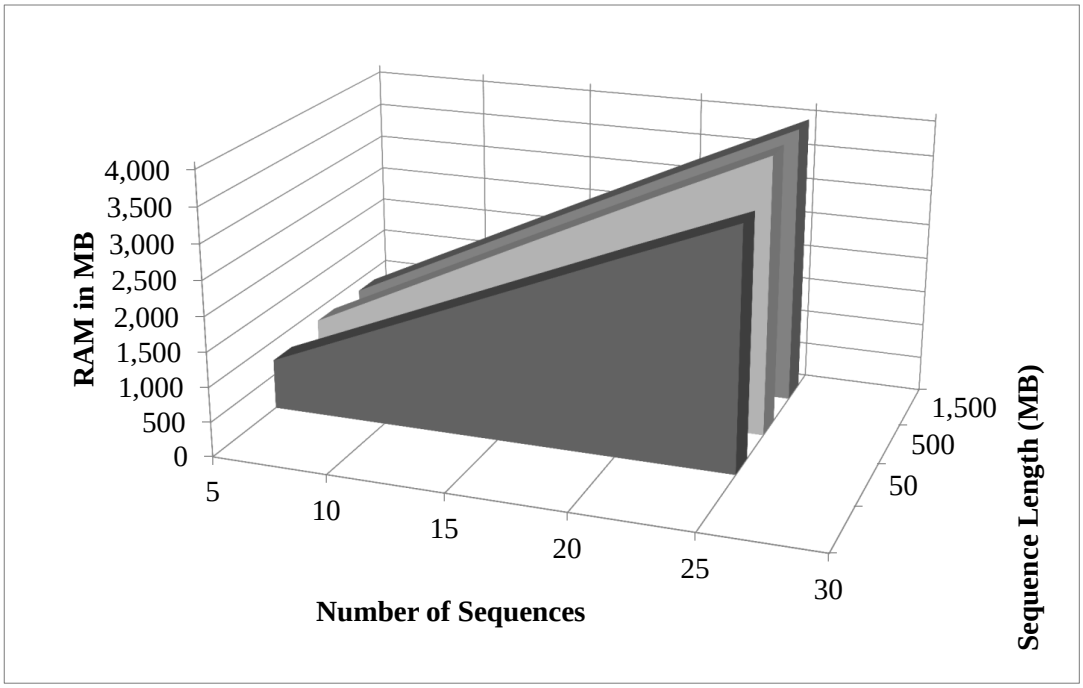

Figure 5.6: Total memory used when running algorithms for evaluating the dissimilarities between different collection of randomly-generated sequences with increasing size (in MB), using k-mer counts, with k set to 11.

to compare, because we have to maintain the k-mer counts (or of the context-object information) for each of the input sequences. Instead, if we keep fixed the number of sequences to be compared while increasing their size, we experience a much smaller expansion of the required memory. This phenomenon is visible in Figure 5.6, where we plot the memory usage of algorithms based on k-mer counts when comparing different collections of sequences of increasing size, with k fixed at 11.

Our stand-alone implementation only works in main memory. It is known that there are hybrid algorithms that periodically flush a hash table on the disk, but the I/O bottlenecks can reduce the performance. This aspect is studied in Section 5.4.

5.3.3 Alignment-free Sequence Comparison on Hadoop

In our first round of experiments, we pinpointed one important performance issue affecting the implementations being evaluated. Their performance is dominated by the time required to interact with the hash maps used to store information about the input sequences. Similarly, most of the memory they use is required again by this hash map. Consequently, when the number of sequences to be compared and/or the length of the pattern to be extracted increases, the size of these data structures becomes so large to drain the physical memory of the computer, thus preventing the execution of the algorithms. In addition, these tasks could be executed in parallel manner.

This issue can be addressed by resorting to a distributed approach. First, it is possible to speed up the extraction operation by processing several sequences at the same time and in a distributed fashion. Second, it may be possible to extend the range of problems that could be solved efficiently by virtually spreading over several machines the hash maps used to store the information of the sequences. We investigated such a possibility and, correspondingly, we developed this idea by reformulating the original algorithms according to the MapReduce paradigm. This has been done by implementing the algorithms on top of the Apache Hadoop framework and by running them as distributed algorithms on the same datasets used for our preliminary tests. It is worth pointing out that, although the implementation of the algorithms under Hadoop was quite simple and straightforward to achieve, the performance of these implementations was pretty below our expectations. A careful profiling activity allowed us to design and develop several improvements that led to a much better performance.

When developing the MapReduce-based formulation of the algorithms chosen in Section 5.3.1, we adapted the decomposition strategy proposed by *Elsayed et al.* in [96] for computing pairwise document similarity in large document collections. In their work, the authors used two types of MapReduce jobs to compute the similarity between each pair of documents. The first type of job determines the occurrences of each word in each of the documents under analysis. The second type of job establishes the similarity between pairs of documents by comparing the occurrences of the words therein contained. We adopted this approach

and further developed it. In our case, the first type of job is able to process an input sequence using one or more indexing strategies (*e.g.*, k-mer counting, context-object extraction) at the same time according to the input configuration. Similarly, the second type of job is able to determine the dissimilarity between two sequences according to a user-provided dissimilarity measure. It is possible to process several sequences at the same time or to establish the dissimilarity among the sequences of a collection by running multiple instances of the two types of jobs. In the following, we provide more details about this approach.

Step 1 - Indexing Hadoop Job This job is used to extract, for each genomic sequence and for each (spaced or not) pattern P of length $|P| = k$, the k-mers or the context-object information that will be later used to compute the dissimilarity between sequences.

- **Mapper** The map function takes as an input a pair $<idSeq, S>$, where *idSeq* is a unique identifier for the input sequence and S can be either the entire genomic sequence or part of it (in case of very long sequences exceeding the HDFS block size). Then, for each k-mer it finds in the input sequence, it outputs either the pair $<kmer, (idSeq, 1)>$ or $<context, (idSeq, object)>$. In addition, each map function outputs the pair $<idSeq, |S|>$. Notice that if the input sequence is split initially in several parts, the size of the original sequence is established at the end of this step by summing the size of all the sequence splits/parts having the same *idSeq*.

- **Reducer** The reduce function receives, as an input, a set of pairs $<K, L>$. K can be either a k-mer or a context. In the first case, L reports the list of $(idSeq, 1)$ values generated by the map functions for that k-mer. In the second case, it reports the list of $(idSeq, object)$ values generated by the map functions for that context. If the input genomic sequences are split before being processed by the map function, in the reduce function all the pairs pointing to the same sequences are summed or all the contexts having at least two different objects are marked. A new record $<K, L'>$ is provided as an output for each input key K, where L' is a list of pairs $(idSeq, count)$ or $(idSeq, object)$ (the marked contexts are excluded). In other words, for

each k-mer, a reduce function returns the frequencies in each input sequence. After that, each reduce task will save the records it produced in a distinct Hadoop `SequenceFile` F to be processed in the second step. Finally, some *support files*, shared through the HDFS cache mechanism, contain the sequence identifiers and their lengths, the probability for each symbol of the alphabet (for each sequence or for all sequences) and/or the absolute frequencies of each symbol of the alphabet in each sequence (*i.e.*, $k = 1$).

Step 2 - Dissimilarity Measurement Hadoop Job In this step we evaluate the pairwise dissimilarity for each pair of input genomic sequences via dissimilarity measures. In order to speed up this operation, each map task is provided with a local copy of the support files generated at the end of the previous step.

- **Mapper** A map function reads a pair $<K, L'>$ generated in the previous step. Then, for each pair of distinct sequences and for each dissimilarity measure compatible with the related pattern P (associated to k-mer or context), this function computes the partial dissimilarity measure according to the chosen method (*e.g.*, symmetric or asymmetric measure for k-mer counts, or context-object information). As an output, the map function produces a $<(idSeq_A, idSeq_B, P, D), (pdiss, 1)>$ pair, where $idSeq_A$ and $idSeq_B$ are the identifier of two input sequences, P is the pattern of the k-mers or the context, D is a dissimilarity measure, $pdiss$ is the partial dissimilarity, and 1 indicates the number of computed $pdiss$ (*i.e.*, the number of k-mers or shared contexts used to compute the dissimilarity value).

- **Reducer** The partial dissimilarities are used by reducers to compute the final dissimilarities. In particular, a reduce function receives, as an input, the pairs $<(idSeq_A, idSeq_B, P, D), list\{(pdiss', 1)\}>$ and produces, as an output, a $<(idSeq_A, idSeq_B, P, D), diss>$ pair, where $diss$ reports the final dissimilarity measure between $idSeq_A$ and $idSeq_B$ using D as a dissimilarity measure and P as a pattern. Therefore, we can compute more dissimilarity measures also using different patterns on each pair of sequences.

5.3.3.1 Improvements

We developed in a pretty straightforward way a first MapReduce formulation of the implementations presented, by using the facilities provided with the Hadoop framework. However, the performance of these implementations was bad. A careful profiling activity allowed us to isolate some performance bottlenecks that prevented our implementations from fully exploiting the computational capabilities of the underlying Hadoop cluster. Then, we developed two improvements able to partially solve these problems.

In-Mapper Local Aggregation According to our preliminary experimentations, the choice of having a map task output a pair for each k-mer it finds, while analyzing the input sequence, is very space and time expensive. Indeed, it would be better for each map task to use a local data structure to maintain the statistics about the k-mers found during its analysis and, then, return these statistics at the end of its execution. This approach could be adopted with no effort by using the Combiner facility available with Hadoop (Section 3.5). This facility allows a map task to buffer all of its output pairs and to summarize them, through the execution of a user-defined Combiner function. In our case, the usage of the Hadoop Combiner would allow a map task to sum, on its own, the frequencies of the k-mers or aggregate the context-object information found while scanning the input sequences. The aggregated information would be returned at the end of the map task. However, according to our results, this solution has an important drawback: the aggregation is not incremental but, generally, takes place at the end of the task. This implies that the map task could keep in memory all of its output pairs before combining them (see Section 3.6 for details). As a consequence, a map task would likely run out of memory when processing long sequences and multiple patterns, therefore, buffering the pairs on local disk or partial aggregations can be provided. We significantly improved this operation by not using the standard function used by Hadoop to combine these results. Instead, we introduced in each map task of Step 1 a persistent Java hash map data structure that is used to progressively index and sum the frequencies of the k-mers, or to progressively index and update the context-object information. This improvement resembles

to the aggregation of RNs presented in Section 4.4.4.1. Conversely, in Step 2 we have used the Combiner to aggregate the partial dissimilarities in each map task.

Input Split Strategies One of the most challenging aspects to face when analyzing very long genomic sequences is about the strategy used to read these sequences from input and keep them in memory. A naive solution would be to feed the map tasks carrying out the analysis with a copy of the whole sequences to be analyzed. This solution works well when dealing with short sequences, but it is doomed to fail when processing very long sequences: they are likely to be too big to fit in the physical RAM of a single node. In addition, we would like to take advantage of the case where the number of nodes of the Hadoop cluster is higher than the number of sequences to analyze. Finally, breaking long sequences into smaller parts while increasing, at the same time, the number of tasks, would allow the nodes of the Hadoop cluster to better interleave CPU-bound, disk-bound and network-bound activities.

Thus, a more sophisticated approach is required, able to feed all the computing nodes of a cluster while exploiting the data local computation capability of Hadoop.

We developed two different strategies for managing the input of the sequences. The first strategy assumes that the sequence to process is short enough to fit in the physical memory of the calculator that will be used to analyze it. Thus, for each input sequence, it works by creating one single record $<Key, Value>$, where Key is the *sequence identifier* (*idSeq*) and $Value$ is the *entire genomic sequence*. The second strategy, to be used with very long sequences, works by splitting the input sequence in several records $<Key, (V_1, V_2)>$, where Key is a unique sequence identifier, V_1 contains the characters of the j-th row of the input FASTA file of the genomic sequence and V_2 contains the first $k-1$ characters of the $j+1$-th row (V_2 is empty if the j-th row is the last row of the input file). A FASTA line consists of few tens of characters (see Section A.1 for details).

5.3.4 Experimental Analysis on Hadoop

5.3.4.1 Experimental Settings

All our experiments with Hadoop have been conducted on a cluster of 5 nodes equipped each with 32 GB of RAM, 2 *AMD Opteron @ 2.10 GHz* processors (16 total cores), *Linux CentOS 6* operating system, approximately a TB of disk drive and a Giga-Ethernet network card. Our Hadoop cluster includes 4 slave nodes and a master node. The master node runs the `Resource Manager` and the `Name Node` services, while the slave nodes run the `Node Manager` and the `Data Node` services. The Hadoop version is 2.7.1[1]. On each slave node, up to 8 concurrent map/reduce tasks were allowed. We used a HDFS replication factor set to 2 and a block size set to 128 MB.

5.3.4.2 Experimental Results

We recall that when developing the MapReduce version of the algorithms presented in Section 5.3.3 we had two objectives. First, we were interested in obtaining an efficient distributed implementation able to keep pace with the performance of the sequential one while being able to scale well with the size of the underlying computing cluster. Second, we were interested in increasing the size of the problems that could be managed, thus overcoming the memory limits experienced with the stand-alone implementations. Along this track, we repeated the same experiments presented in Section 5.3.2.3.

In particular, for the sake of brevity, we firstly evaluated the scalability of our distributed implementation by evaluating the Squared Euclidean dissimilarity (see Equation 5.1) between different sequences using an increasing number of concurrent map/reduce tasks in execution at the same time, *i.e.*, workers/`Containers` (see Section 3.2.2). We report in Figure 5.7 the result for this experiment when considering 20 different sequences of $\approx 1,600,000,000$ characters each and with $k = 10$. The size of this datasets is approximately 30 GB. For the execution times, we distinguish among the time spent extracting the k-mers from the input sequences (*i.e.*, Step 1) and the time spent evaluating the Squared Euclidean

[1]The experimentations were conducted between July and August 2015.

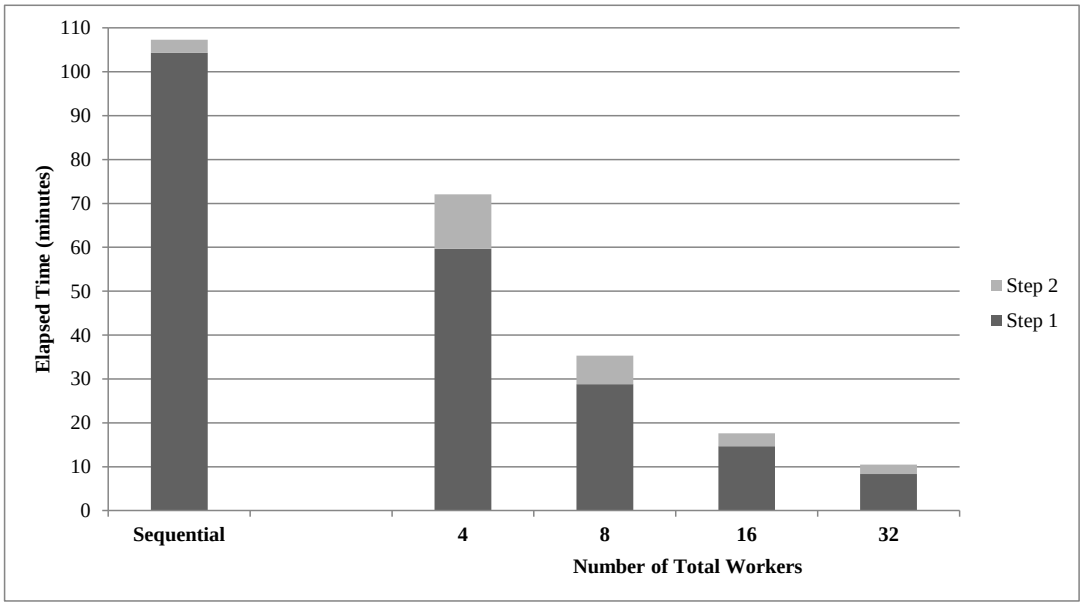

Figure 5.7: Elapsed times for evaluating the Squared Euclidean dissimilarity measure between 20 different sequences of $\approx 1,600,000,000$ characters each, with $k = 10$ and an increasing number of concurrent map/reduce tasks at the same time (*i.e.*, workers or Hadoop `Containers`).

dissimilarity between the k-mer frequency vectors (*i.e.*, Step 2). The outcoming results are compared with the executions times required by the sequential (non-parallel) version of the same algorithm. The first thing we notice is that the distributed implementation using 4 workers has just $\approx 1.5\times$ speed up with respect to its sequential counterpart. Indeed, the distributed implementation incurs in a performance overhead that is related to the need of saving on file and, then, transmitting over the network the k-mer counts. This overhead is completely absent in the sequential implementation where there is no need of transferring data since Step 1 and Step 2 of the algorithm are carried out by the same process using its own main memory space. There is also a performance overhead due to the stack of network, file system and job scheduling protocols required by Hadoop. The effect of this overhead is visible in Figure 5.8, where we show the CPU usage of one of the nodes running the map/reduce tasks. The load spikes are due to the

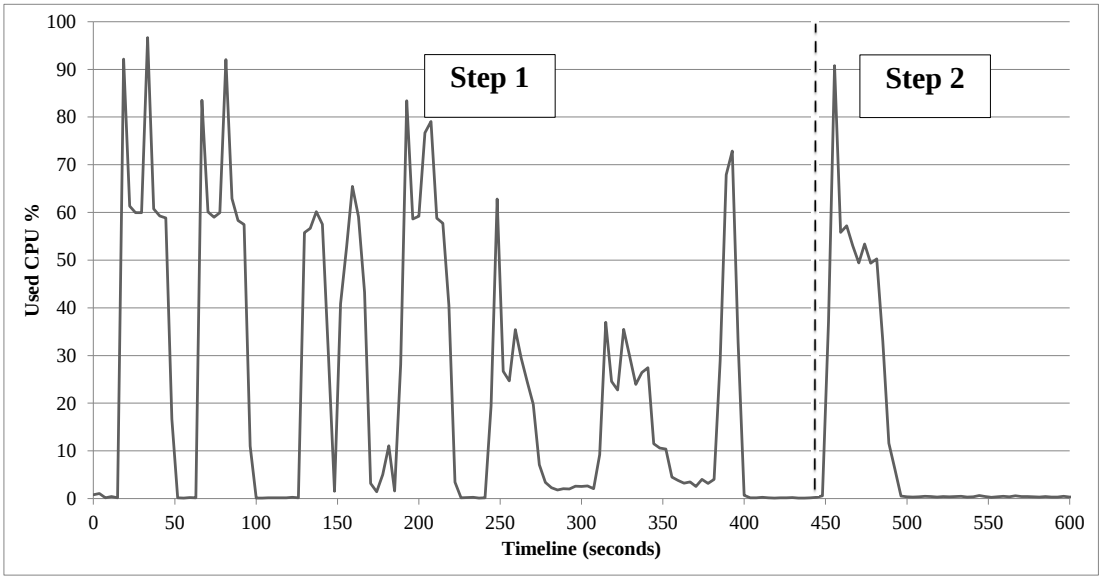

Figure 5.8: CPU usage profile of a slave node of the Hadoop cluster used for evaluating the Squared Euclidean dissimilarity between 20 different sequences of $\approx 1,600,000,000$ characters each, with $k = 10$ and 8 concurrent map/reduce tasks on each slave node (*i.e.*, 32 total workers). For gathering information, we used Dstat tool [293].

extraction of the k-mer counts and to the evaluation of the distance between pairs of sequences. The remaining intervals, where the CPU is almost unused, are due to the activity carried our by the Hadoop framework for saving data on disk, for transferring it from map tasks to reduce tasks and for partitioning it before start feeding reducer. If we focus only on the distributed implementation, we notice that our solution is able to scale fairly. These results are also maintained using different dissimilarity measures and patterns.

In second experiment, we were interested in assessing whether the adoption of a distributed approach would allow us to (efficiently) solve larger problems than the ones solvable in our stand-alone setting. We recall, to this end, that the main memory issues we found in our tests were related to the experimentations with large values of k. In such a setting, the size of the hash maps used to store the k-mer frequency counts tends to exceed the available physical memory. We expect our distributed approach to implicitly solve this problem, as map tasks running on separate nodes would be able to calculate and maintain, each, the

frequency counts for just a subset of the input sequences (*i.e.*, at most, 128 MB). The subsequent operation of merging the partial frequency counts would be run in a distributed way as well thus avoiding again the excessive memory usage problem.

The results of this second experiment, reported in Figure 5.9, are not completely in line with these results presented in previous. In these tests we measured the time required for evaluating the Squared Euclidean dissimilarity between 20 different sequences of $\approx 1,600,000,000$ characters each, using 32 concurrent workers, and increasing values of k. On a side we notice that, differently from the sequential case, we have been able to run our distributed implementation with $k = 15$. On the other side, we observe that the outcoming execution times for $k = 15$ are one order of magnitude longer than the ones measured with $k = 10$. Moreover, we notice that differently from previous experiments, the execution cost paid to evaluate the dissimilarity between different frequency vectors is now larger than the one spent for extracting the frequency vectors. To explain such a difference, we profiled the overall number of distinct k-mers extracted during the execution of each of these tests as well as the amount of data produced at the end of Step 1 and used to feed Step 2.

As shown in Table 5.1, the number of k-mers found with $k = 15$ is about $1,000$-fold larger than the number of k-mers found with $k = 10$. Moreover, the size of the data produced at the end of Step 1 grows from ≈ 200 MB to ≈ 200 GB. Indeed, the much increased number of k-mers to handle puts pressure on the Step 2 of the algorithm, thus magnifying its execution time. Moreover, there is an additional performance overhead due to the need of saving twice the output of the map tasks on disk. This occurs because the output is first saved on the local file systems of the nodes running the map tasks. Then, it is transmitted to the nodes running the reduce tasks. Here, it is saved again on disk before being processed. This overhead, that is characteristic for Hadoop, becomes problematic when the size of the output of map tasks is very big, like in our case.

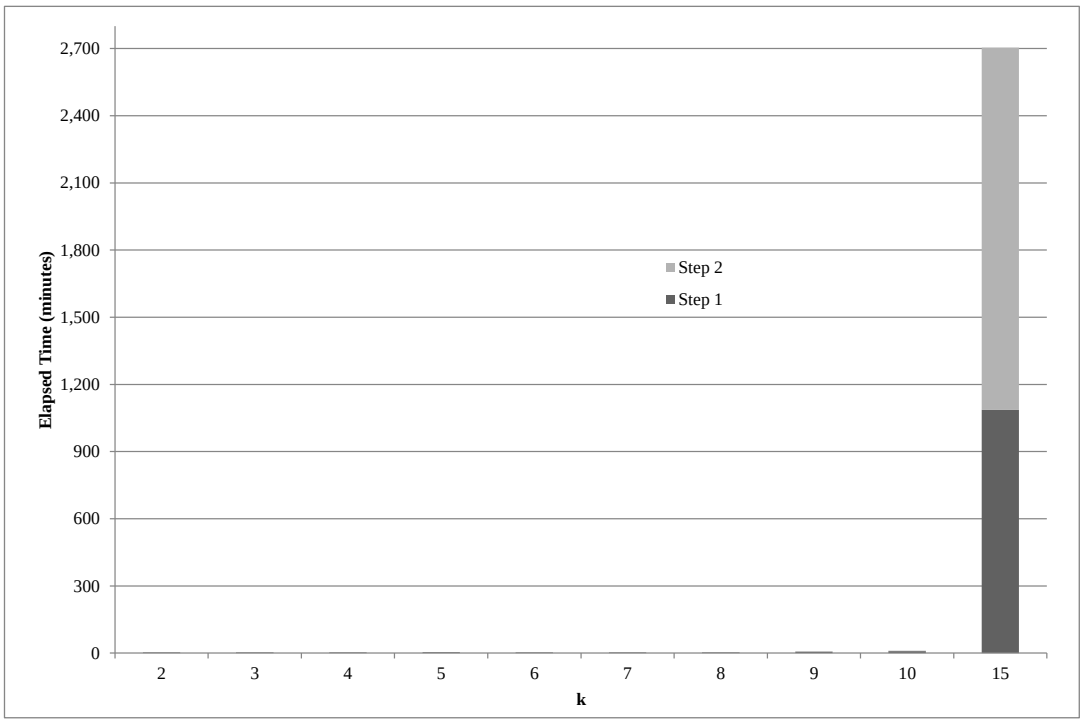

Figure 5.9: Elapsed times for evaluating the Squared Euclidean dissimilarity between 20 different sequences of $\approx 1,600,000,000$ characters each, using 32 concurrent workers, and values of k increasing up to 15.

5.3.5 Remarks

In this section was presented a first systematic study of word-based alignment-free sequence comparison methods on Hadoop framework, yielding valuable information both for the programmers and the users of those methods, in terms of usage of computational resources. In details, we have been able to develop a MapReduce formulation of a generic alignment-free sequence analysis algorithm that is able to scale well with the number of used concurrent tasks. Moreover, our solution allows to conveniently process problem instances that are usually hard to solve in a stand-alone setting because of memory limitations.

Our engineering, tuning and profiling activities were allowed to create `HAFS`, a distributed framework useful to easy develop word-based alignment-free sequence comparison methods. In fact, new word-based dissimilarity measures can be easy integrated in `HAFS`, further than those presented in Figure 5.4.

Table 5.1: Profiling information collected when evaluating the dissimilarities between 20 different sequences of $\approx 1,600,000,000$ characters each, using values of k increasing up to 15.

k	Number of Distinct k-mers	Step 1 Output Size (KB)
2	16	3
3	64	14
4	256	54
5	1,024	217
6	4,096	873
7	16,384	3,506
8	65,536	14,090
9	262,144	56,623
10	1,048,576	227,541
15	1,073,741,824	196,681,472

However, our results are only an initial step and there are still some open performance issues. In fact, there is to analyze in detail the related problem of the computation of k-mer statistics on a Hadoop cluster, which has already been mentioned in the previous. In particular, in the Section 5.4 will be presented an more efficient and scalable Hadoop solution for counting large value of k (*e.g.*, $k = 31$), which is also compared with the state of art of parallel k-mer counting tools.

5.4 K-mer Statistics on Hadoop

As an important case study, we concentrate on a very simple paradigmatic problem that has become data-intensive due to the growing sequence capabilities mentioned earlier: the collection of k-mer statistics (or counting) for sequences defined over a finite alphabet Σ. We recall that k-mer typically refers to all the possible subsequences of length k that are contained in a sequence. Formally, let $\mathcal{S}$ be a set of sequences, each coming from Σ^*, we are interested in collecting two types of statistics: local and cumulative.

- **Local Statistics (LS)**: how many times each of the k-mers in Σ^k appears separately in each of the sequences in $\mathcal{S}$.

- **Cumulative Statistics (CS)**: how many times each of the k-mers in Σ^k appears cumulatively (globally) in sequences in $\mathcal{S}$.

Throughout this section, we assume that $\Sigma = \{A, C, G, T\}$, therefore, when experimenting with untrimmed reads, the subsequences of length k which contain at least a character N will be discarded and they will not affect the statistics.

The *canonical* representation of a k-mer is by definition itself or the *reverse complement*[1] of it, whichever comes first lexicographically. When we extract canonical k-mer statistics, the count of a k-mer is the number of occurrences of both the k-mer and its reverse complement (see [134] for details about the reverse and/or the complement DNA sequences).

The interesting scenario in which to study LS is given by long sequences, modeling a preprocessing step typical of genome-scale alignment-free classification and compositional analysis of species (*e.g.*, [66, 118, 119]). It is also worth to mentioning that k-mer statistics are becoming increasingly important in epigenomics (*e.g.*, [117, 222]) and metagenomics (*e.g.*, [297]). The interesting scenario in which to study CS is given by a large set of short sequences, *i.e.*, reads, modeling the preprocessing step that reads go through before assembly (*e.g.*, [71]).

The k-mers extraction is a simple task, but counting them in NGS era can easily pass the memory capacity of a single traditional computer. In fact, usually each k-mer is stored in a hash map and the exact memory requirements depends on the length of k and the type of hash map that is used. Therefore, although both versions of the problem are algorithmically very simple, the sheer amount of data that has to be processed in a typical application has motivated the development of many algorithms and software systems that try to take advantage either of parallelism or of sophisticated algorithmic techniques or both. Section A.2 presents the state of the art on algorithms collecting k-mer statistics. To the best of our knowledge, attention has been given mainly to CS, evidently due to its foundational role in sequence assembly. Those algorithms can be broadly divided into the ones that compute the statistics exactly and those that provide only estimates. It is very unfortunate that the at least equally important LS has been given very little consideration with respect to CS. In fact, to the best of our

[1] The reverse complement of the k-mer ATCG is CGAT.

knowledge, none of the methods working on CS has been designed to explicitly support also the local statistics.

Among the algorithms designed for cumulative statistics and reported in Section A.2, we have selected only the ones that provide exact statistics. In fact, we have chosen the most representative according to the literature, for each type of computer architecture we are interested in. The algorithms so selected are as follows. First, the most representative of the algorithms that are disk-based and that work in a shared-memory environment using multi-threading: *KAnalyze* [24], *Jellyfish2*, an evolution of Jellyfish [184], *KMC2* [84] and *DSK* [231]. Then, we selected *BioPig* (k-mer counting module) [37, 209] as a representative of algorithms supported by distributed architectures. Between these algorithm, the most popular and faster is KMC2, as is testified in literature (see Section A.2). It is to be pointed out that all these algorithms work by skipping all the k-mers of an input sequence containing at least one N character. We also observe that BioPig and KMC2 have been designed to work on short sequences only, *e.g.*, reads. This could imply that may exhibit very bad performance when processing long sequences or fail at all to work. A careful profiling of some of the most successful methods that have been developed for CS is presented in Section A.3, where it shows that these methods do not scale well with computational resources.

The k-mer counting is a fundamentally I/O-bound problem, so we could expect that one major performance bottleneck to face, when solving it, is about the amount of time required to physically load in memory from disk and process the (potentially very large) input sequences. This is a problem that cannot be solved by using only a multi-threaded approach as the different concurrent threads would have to share and compete for the same I/O devices. We overcome this problem by adopting a completely distributed approach.

In literature there are some solutions for k-mer cumulative counting on Hadoop, such as those presented in a Hadoop textbook [220], in k-mer counting module of BioPig [37, 209], or in *Pahadia et al.* [216, 217]. However, these solutions are very simple, and they have many problems as it will be explained in Section 5.4.1 and experimented in Section 5.4.3.4.

In this section is presented a highly engineered distributed algorithm on Hadoop for both LS and CS cases. The solution proposed here is called *K-mer*

Counting on Hadoop (KCH). It is efficient, with respect to other previous solutions based on distributed architectures, and fully scalable in terms of processing units. Since the chosen problem is at the start of many bioinformatics pipelines, we set the foundation for the development of efficient distributed pipelines that use k-mer statistics.

Initially, in Section 5.4.1 is described a simple and inefficient solution for k-mer local and cumulative statistics. The related problems are also highlighted. Subsequently, in Section 5.4.2 is outlined our fast and efficient solution, called KCH, for computing these statistics.

5.4.1 A Naive Solution for K-mer Statistics on Hadoop

We present here a very simple MapReduce-based k-mer counting algorithm. We assume that the input data are initially available in a distributed form, in general execution occurs in a data local way (*i.e.*, computation can take place where data are available) and output data are saved, again, in a distributed form. This means that, in our case, input data are partitioned a priori on a cluster of computers, and each slave node of the cluster initially processes only its own partition of the data.

A first naive algorithm was designed to exploit this approach, and able to compute k-mer statistics for both Local Statistics (LS) and Cumulative Statistics (CS) with Hadoop. It is worth pointing out that an analogous algorithm for cumulative statistics is given in a Hadoop textbook [220] and BioPig [37, 209] (see Section 5.4.3.4 for a comparison about BioPig).

Differently to other Hadoop solutions, our statistics are also computed on very long input sequences (*e.g.*, a sequence that crosses many HDFS blocks), in addition to short sequences (*e.g.*, reads).

Our naive approach consists of a map phase and a reduce phase.

- **Mapper** The map function takes as input a genomic sequence Seq (or part of it) and its identifier $idSeq$ and returns, as output, the list of k-mers it contains. Each k-mer is returned as soon as it is found. In particular, for each k-mer occurrence found in its input (sub)-sequence, a map function emits a pair $<kmer, 1>$ in the case of CS, or $<(idSeq, kmer), 1>$ in the

case of LS. Therefore, multiple occurrences of the same k-mer are reported as distinct pairs.

- **Reducer** The reduce function aggregates all the pairs returned by map functions and related to the same k-mer. In the case of local statistics, they are also aggregated according to the sequence they belong to. Therefore, as output for LS, the reduce function also returns the identifier of each sequence containing the k-mer with associated frequency. Otherwise, in the case of cumulative statistics, the reduce function only returns the k-mer and its frequency.

See Figure 5.10 for details about an example of map/reduce input/output pairs.

Local Statistics
Map: $<idSeq, Seq> \rightarrow \text{list}(<(idSeq, kmer), 1>)$
Reduce: $<(idSeq, kmer), list\{1\}> \rightarrow <(idSeq, kmer), frequency>$

Cumulative Statistics
Map: $<idSeq, Seq> \rightarrow \text{list}(<kmer, 1>)$
Reduce: $<kmer, list\{1\}> \rightarrow <kmer, frequency>$

Figure 5.10: Input and output pairs of a MapReduce naive algorithm designed to compute k-mer statistics for both Local Statistics (LS) and Cumulative Statistics (CS) with Hadoop.

The strategy just outlined is inefficient when processing many large sequences because the Hadoop middleware has to manage and process even billions or trillions of pairs emitted from map tasks essentially equal to the size of the collection (see Section 5.4.3.4 for experiments). In detail, all these pairs have to be sorted, partitioned and saved first on the local disk of the worker node during the spilling phase of the map function (see Section 3.4), then they are moved on the local disk of a (possibly) distinct node where they will be processed by a reduce function. Those operations become particularly expensive when applied to a very large number of pairs. In this case, the presence of several workers running concurrently on a same slave node makes the I/O bus congested when they are trying to

save at the same time on the local disk a huge number of output pairs. Therefore, the executions times can be deteriorated. Similarly, the huge amount of items that have to be moved, through the shared network, from map tasks to reduce tasks produces a similar congestion effect on the shared network connections.

As a result, when implemented on Hadoop, this strategy exhibits very disappointing execution times. Indeed, a careful profiling of it reveals that the I/O-bound nature of the problem is one of its major performance bottlenecks.

A first solution to elude these problems could be the adoption of the Hadoop Combiner, that aggregates the output pairs of a map task. However, this facility could be use more memory or it could not efficiently work [292]. In fact, the execution of Combiner is not guaranteed, and Hadoop may temporarily store the $<key, value>$ pairs in local file system. Therefore, running the Combiner later which will cause expensive operations on disk. See Sections 3.5 and 3.6 for additional details.

In alternative, a map task could use a hash map for in-mapper local aggregation, as used in Section 5.3.3. An in-mapper aggregation is much more efficient and resource-frugal than the Hadoop Combiner, because it continually aggregates the data in memory. In fact, as soon as it receives two values with the same key, it combines them and stores (or updates) the resulting $<key, value>$ pair in the hash map.

In particular, in the next subsection, we outline how the problems of this naive implementation have been addressed to obtain good performance.

5.4.2 KCH: Fast and Efficient Solution for K-mer Statistics on Hadoop

In this section is described our algorithm for k-mer statistics on Hadoop, called `KCH`. Starting from our previous experience, we developed a refined k-mer counting algorithm improving the previous one in several respects, both in terms of the design and of the implementation, such as: efficient input management, fast local k-mers extraction, two-levels k-mers aggregation with explicit partitioning and memory-frugal requirements.

We next provide a high-level description of our MapReduce algorithm for CS

case, followed by additional details of its most relevant parts. The modifications for the case LS are also detailed in following.

Mapper Each map task uses a set of r local hash tables Hts to maintain the frequency counts of the k-mers found while scanning its own input sequences or subsequence. These hash tables represent a partitioning of the universe of possible k-mers (*i.e.*, Σ^k) and they are used to perform a first level of aggregation. With reference to Algorithm 5 that gives the pseudo-code for CS, we next provide details about the various functions composing the algorithm. Explanatory comments are indicated with the symbol "▷" in the pseudo-code.

At the start of a map task (*setup* function), it creates r local hash tables each with C_{map} entries.

The input of a map function is the identifier of the sequence $idSeq$ and the sequence (or part of it, in case of very long sequences) Seq. Our algorithm uses the standard binary encoding of a letter of the alphabet $\{A, C, G, T\}$ to pack a k-mer into an integer number. Since now each character of a k-mer needs two bits rather than eight, we have a saving in memory usage but also an additional one in the transmission of partial statistics from the tasks performing map to the ones performing reduce. Moreover, it makes possible a significant advantage also in the scanning strategy used to extract k-mers from an input sequence, which is organized as follows. Initially, a new k-mer $kmer'$ is extracted by looking at the first k characters of the input sequence and packed into a single integer. The same can be done for its canonical representation. From this point on, new k-mers (or their canonical representations) are extracted by processing the last k-mer found by means of binary shift and AND operations. The process goes on until the end of the sequence is reached or a N character is found. In this last case, the algorithm skips all the subsequences of length k containing the N character and starts over the scanning strategy. In addition, when a *newline* character is found, it is ignored. Whenever a new k-mer $kmer'$ is found, its local partial frequency count is updated accordingly but no output is provided. In particular, the hash table for which one needs to increment the counter or start a new one is identified as follows: $kmer'$ is placed in the hash table having id obtained by taking the numerical representation of k-mer, *i.e.*, $kmer'$, mod r. Once that its input has

been scanned, the map task proceeds by emitting as output the copy of each hash table, together with an identifier, by executing the *endupFlush* function.

It is worth pointing out that, once fixed an initial size for the hash tables, some of them may need to be expanded at run time and that, in turn, may cause a map task to run out of memory. In order for that to be avoided, the algorithm follows a *flushing strategy* by means of which a hash table can be output even if the task has not completed yet its execution. In fact, a function, named *intermediateFlush*, is executed to check if the number of elements in a hash table ht exceeds a certain threshold t. If this is true, the algorithm emits the id of the table as key, *i.e.*, $idHt$, and its binary copy as value, *i.e.*, ht. Then, this local table is replaced with one empty. Therefore, after analyzing the whole input, the map task proceeds by emitting as output the copy of each hash table, accompanied by identifier executing a function called *endupFlush*.

Reducer At the end of the map phase, all hash tables related to a same partition of Σ^k are sent for aggregation to the same distinct reduce task. Therefore, a reduce function receives as input all the hash tables of the same partition to compute their aggregated statistics. In particular, it merges the hash tables using a new hash table, called ht_{merge}, that aggregates their counters. In addition, the reduce function will dump on a distinct textual file on HDFS the counts of all k-mers found in its corresponding aggregated hash table. The pseudo-code of a reduce task for CS is presented in Algorithm 6.

The interesting scenario in which to study LS is given by long sequences. In fact, the formulation of our algorithm for the case of LS on long sequences is simple starting from the one presented for CS. In fact, each map task processes the input of a same sequence, therefore, all the frequencies in the hash tables are related to the same genomic sequence. When a copy of a hash table is emitted as value, the related key is the combination of the identifier of the sequence and the id of the hash table. In this way, a reduce function receives all the hash tables of a certain partition belonging to a same sequence. For saving output space, a distinct HDFS directory will be created for each sequence, with each directory containing a number of text files equal to the number of reducers. Each file will

be named after the reducer it refers to and will contain the statistics for all k-mers aggregated from the reduce task (without the sequence identifier).

Figure 5.11 gives an example of the map/reduce input/output pairs.

Local Statistics
Map: $<idSeq, Seq> \rightarrow \text{list}(<(idSeq, idHt), ht>)$
(ht is a list of $<kmer', frequency>$ elements)
Reduce: $<(idSeq, idHt), list\{ht\}> \rightarrow \text{list}(<(idSeq, kmer), frequency>)$

Cumulative Statistics
Map: $<idSeq, Seq> \rightarrow \text{list}(<idHt, ht>)$
(ht is a list of $<kmer', frequency>$ elements)
Reduce: $<idHt, list\{ht\}> \rightarrow \text{list}(<kmer, frequency>)$

Figure 5.11: Input and output pairs of `KCH` algorithm designed to compute k-mer statistics for both Local Statistics (LS) and Cumulative Statistics (CS) with Hadoop.

Algorithm 5 Pseudo-code of `KCH` Mapper for the case of CS.

function SETUP(r, C_{map})
▷ r is the number of local hash tables in a map task; C_{map} is the number of entries of each hash table. This function is executed at the start of the map task.

$Hts \leftarrow$ GETHTLIST(r, C_{map}) ▷ It creates r hash tables each with C_{map} as the initial number of entries. Each hash table ht in Hts has an identifier $idHt \in [0, r-1]$.

end function

function MAP($idSeq$, Seq)
▷ $idSeq$ represents the header of the sequence while Seq is a short sequence (*e.g.*, reads) or a part of very large genomic sequence.

for each k-mer $kmer'$ in Seq **do** ▷ $kmer'$ is the integer encoding of a k-mer.
$idHt \leftarrow kmer' \mod r$
ADDTO($kmer'$, 1, $idHt$) ▷ It increments by 1 the counter of a k-mer $kmer'$ if it exists in the hash table $idHt$ (*i.e.*, $<kmer', frequency_{old}+1>$), otherwise a new entry is added with counter 1 for $kmer'$ (*i.e.*, $<kmer', 1>$).
INTERMEDIATEFLUSH(Hts, t)
end for

end function

function INTERMEDIATEFLUSH(Hts, t)
▷ If the number of elements in a hash table ht (in Hts list) exceeds a custom threshold t, this hash table is flushed and a new table is used.

for each ht (with id $idHt$) in Hts **do**
if SIZE(ht) $\geq t$ **then**
EMIT($idHt$, ht) ▷ It emits the pairs consisting $idHt$ as key and a binary copy of ht as value.
EMPTY(ht) ▷ The hash table ht is emptied.
end if
end for

end function

Algorithm 5 *Continued.* Pseudo-code of `KCH` Mapper for the case of CS.

function ENDUPFLUSH
▷ It emits the pairs consisting $idHt$ as key and ht as value, where $idHt$ is the index of a hash table, and ht is a binary copy of this table. This function is executed at the end of the map task.

for each ht (with id $idHt$) in Hts **do**
EMIT($idHt$, ht)
end for

end function

Algorithm 6 Pseudo-code of `KCH` Reducer for the case of CS.

function REDUCE($idHt$, list(ht))
▷ It performs the second level of counter aggregation exploiting all k-mer counters emitted in hash tables with id $idHt$.

$ht_{merge} \leftarrow$ EMPTYHT(C_{red}) ▷ It creates a empty hash table with C_{red} entries and id $idHt_{merge}$. It is used to perform the second stage of aggregation.

for each ht_{curr} in list(ht) **do**
for each $<kmer'$, $frequency>$ in ht_{curr} **do**
ADDTO($kmer'$, $frequency$, $idHt_{merge}$) ▷ The counter of $kmer'$ is incremented by $frequency$ in the hash map $idHt_{merge}$.
end for
end for

for each $<kmer'$, $frequency>$ in ht_{merge} **do**
$textKmer \leftarrow$ NUM2TEXT($kmer'$)
EMIT($textKmer$, $frequency$) ▷ It emits the pairs consisting of the textual representation of the k-mer $kmer'$ as key and its frequency as value.
end for

end function

In the following, are detailed the main features of `KCH`, such as the use of the efficient FASTA input management, the fast local k-mers aggregation, and the two-levels k-mer counts aggregation with explicit partitioning.

5.4.2.1 Efficient FASTA Input Management

The access to the input files of an application is managed by Hadoop through the implementation of a proper `InputFormat` used to read the files. The Hadoop splitter organizes a data source in smaller parts called input splits, where each split is processed by a distinct map task. The `InputFormat` mechanism is also adopted to extract the $<key, value>$ pairs from an input split for being used as input to the map functions. As said in Section 3.4.1, HDFS blocks have not to be confused with input splits. The former refers to a physical organization of the input data, while the latter refers to its logical organization. For instance, a text file would be divided by HDFS in several blocks having the same size, with the possibility for a line of text to fall across two different blocks. Instead, a split would be able to provide a more abstract view of the input file, where each line is contained in only one split. To make this possible, a split may need to access two HDFS blocks to complete a dangling line.

We have developed `FASTAshortInputFileFormat` and `FASTAlongInputFile Format` Java classes specifically designed to efficiently handle large FASTA files and that can be used by any Hadoop-based bioinformatics application requiring that type of input. The first class handles short sequences, such as reads, while the second sequences of arbitrary length, such as a single sequence of tens of gigabytes (crossing several HDFS blocks).

The `FASTAshortInputFileFormat` class handles short sequences, and it works by initially reading into a memory buffer the whole content of a split to be processed, with the help of some low-level byte-oriented functions provided by Hadoop. This acquisition includes also all the (potential) characters that are found at the beginning of the subsequent split and that may be the terminal part of a short sequence starting in the current split. Once loaded in memory, this buffer is directly processed by map functions using our input routine. That is, whenever a map function of an application using this input routine is executed,

a reference to the buffer is passed rather than a copy of the sequence to be processed. Moreover, the indices marking, in the buffer, the beginning and the end of the sequence to be processed are also provided.

`FASTAlongInputFileFormat` follows a similar approach, but it manages a very large sequence in a FASTA file. In particular, it reads in memory all the bytes of the current input split plus at most $k-1$ characters (different from newline characters) belonging at the next input split. Those additional characters are used to extract the k-mers which start in the current split, but they fall in the next split.

Therefore, `FASTAshortInputFileFormat` and `FASTAlongInputFileFormat` are designed according to maximize data locality computation.

In Section 5.4.3.4 are reported some experimental evaluations between our readers and some solutions presented in literature.

5.4.2.2 Fast Local K-mers Aggregation

The choice of the hash table implementation to use for maintaining the k-mer counts has an important impact on the performance of our algorithm, both in terms of CPU time and memory requirements. For this reason, we had to look at an alternative to the standard hash table implementation available with Java, since we found that it is not memory-frugal. Our solution uses the `OpenHashMap` classes included in the `fastutil` library [278] to maintain k-mer counts. Indeed, it is among the most efficient implementations available in Java [283]. In addition, it provides an efficient implementation of hash tables both in terms of memory and CPU time. Memory efficiency is achieved by using a very compact internal representation that avoids to store any supplementary information apart from the inserted keys and their corresponding values. Time efficiency is achieved in several ways like by dropping any support for synchronization (*i.e.*, having two or more threads manipulate the same hash table simultaneously may lead to errors). We did not need this feature as, in our case, each map function uses its own hash tables to maintain k-mer counts while the aggregation of these statistics is done in parallel by different reduce functions, each on its own set of hash tables. Another important feature available with the `OpenHashMap` classes is the availability of

an an `addTo` operation, that allows to increase the value associated to a key without first fetching it. By using this operation, our algorithm is able to update in place the frequency count associated to a k-mer whereas other hash table implementations typically require two operations to this end (*i.e.*, first retrieve the count from the hash table using a key, then write the updated count in the hash table using the same key).

5.4.2.3 Two-levels K-mer Counts Aggregation

A serious performance issue of the naive approach to k-mer counting on Hadoop is related to the huge amount of k-mer statistics returned by map functions, especially when working with large values of k. This bulk of data has to be first saved on the worker nodes running the map functions and, then, has to be transmitted to worker nodes running the reduce functions to be aggregated and counted. To solve this problem we have introduced a two-levels aggregation strategy with explicit partitioning.

Initially, we have tried to use a single hash table in each map task with the purpose to aggregate the counters of k-mers extracted. These aggregations are not written to local disk, but they occur in-memory in the mapper itself. A similar approach was also adopted in Section 5.3.3 and it is called *in-mapper local aggregation*. Here this solution is named `Preliminary KCH`.

However, the number of exchanged pairs between map and reduce tasks was still very high, and, for CS, a reduce function is started for each distinct k-mer. This is a serious problem with large value of k. Therefore, we have thought to emit as map output a binary copy of the hash table used as local aggregation. Unfortunately, the aggregation is made in a single reduce function. Consequently, we have adopted the strategy to divide the hash table in a map task in a fixed number of bins (partitions or hash tables), so the reduce phase could be parallelized. This strategy, used in `KCH`, is called *in-mapper local aggregation with explicit partitioning*.

Therefore, in `KCH`, at a first level, each map function uses a vector of hash tables to maintain partial statistics about the k-mers found when scanning its own input. At a second level, each reduce function will now get in input a list

of hash tables containing the partial statistics for a certain family/partition of k-mers. These statistics will be aggregated in one single hash table containing the final statistics. In the map phase, the contents of the hash tables are serialized and saved using a low-level byte array encoding. This ensures optimal performance with respect to other encoding solutions offered by Hadoop. This strategy has two important advantages. First, the number of output pairs returned by a map function when processing the sequences using large values of k scales down by several orders of magnitude (*e.g.*, few hash tables against millions or billions k-mers occurrences). This has the important side effect of greatly speeding up the Hadoop shuffle and the sort phase, as the number of involved records is significantly smaller. Second, the overall amount of data to transmit between map and reduce functions is shrank as well because most part of the aggregation is performed by map functions.

Compressing the copy of a hash table not brings any advantage between map and reduce phases.

The proposed aggregation strategy with partitioning is useful when the number of distinct k-mers is high. In fact, with low value of k, such 3 and 7, this partitioning could not bring no advantage compared to simple in-mapper local aggregation where the map output pairs are aggregated in a single hash map.

A comparison between `KCH` and our Hadoop-based naive solution for k-mer counting is presented in Section 5.4.3.4. In addition, is also shown a comparison with an implementation of `KCH` that uses the in-mapper local aggregation without partitioning strategy (*i.e.*, `Preliminary KCH`).

5.4.3 Experimental Analysis

5.4.3.1 Datasets

Following [231], we use the Illumina human genome dataset [144] in order to obtain datasets for our experiments. It contains short sequences, referred to as reads, which are defined on the alphabet $\{A, C, G, T, N\}$, where N corresponds to an indefinite base.

This dataset contains 111 compressed FASTQ files. The compressed size of the dataset is $\approx$ 109 GB, while an estimation of the size of the FASTQ files is

$\approx$ 480 GB as total. Each FASTQ file contains a large collection of reads, where each read is composed by 4 lines:

1. The first line starts with a @ character, and it is followed by a *read identifier* and an optional description. Reads from the Illumina software use a fixed format for the identifier, while FASTQ files from the *NCBI/EBI Sequence Read Archive* [204] often can include a description.

2. The second line contains the raw sequence letters (*i.e.*, a read).

3. The third line starts with a + character, and is optionally followed by the same sequence identifier and any description.

4. The last line encodes the quality values for the sequence in the second line, and it must contain the same number of symbols as letters in the sequence.

When used in [231], all these files have been transformed and merged into a single very large FASTA file without performing any trimming [232].

Notice that, although a typical dataset for LS would consist of entire genomes while a typical one for CS of reads, for uniformity and ease of comparison, it is best to construct artificial datasets for LS based on a real dataset (the Illumina one) for CS.

Let l denote the number of bytes, *i.e.*, characters, that a dataset to be generated must have. The generation procedure for CS is very simple: pick as many reads (and their headers) from the Illumina dataset as needed to obtain the required size. The reads are selected in the order in which they appear in the original dataset. This grants that, as the sizes of the generated datasets grow, the smaller ones are contained in the larger ones, ensuring consistency of experimentation. Each generated dataset is stored in a FASTA file. In particular, we have generated the datasets of 2, 8, 32 and 128 GB, *i.e.*, *CS_2GB*, *CS_8GB*, *CS_32GB* and *CS_128GB*.

The procedure to generate a dataset of l bytes for LS is slightly more complicated. First, a random number n in the interval $[1, \lfloor \log l \rfloor]$ is picked. That gives the number of sequences in the dataset. The choice of the interval assures that we get sequences whose lengths are much larger than the number of sequences

contained in the dataset, as it is typical in genomic studies. Then, the interval $[1, l]$ is partitioned uniformly and at random in n segments. The length of each segment gives the number of characters in a sequence to be included in the dataset. Those segments and the Illumina dataset are swept from left to right, concatenating reads until a sequence of the length corresponding to the current segment length is obtained. Such a process may require rounding of the segment length values. Each of the sequences so obtained is stored in a separate FASTA file with a single header line. In particular, we have generated the datasets of 2, 8, 32 and 128 GB, *i.e.*, *LS_ 2GB*, *LS_ 8GB*, *LS_ 32GB* and *LS_ 128GB*.

In what follows, we provide details about the datasets used for the LS experiments. Each sequence is coded in a separate multi-lines FASTA file.

1. *LS_ 2GB* dataset consists of 8 sequences, with lengths approximately: 377 MB, 234 MB, 517 MB, 129 MB, 15 MB, 9 MB, 75 MB, 692 MB, respectively.

2. *LS_ 8GB* dataset consists of 30 sequences, with lengths approximately: 302 MB, 242 MB, 647 MB, 315 MB, 528 MB, 30 MB, 114 MB, 6 MB, 20 MB, 611 MB, 247 MB, 217 MB, 358 MB, 149 MB, 354 MB, 4 MB, 174 MB, 13 MB, 720 MB, 9 MB, 47 MB, 586 MB, 205 MB, 16 MB, 1138 MB, 613 MB, 180 MB, 43 MB, 204 MB, 100 MB, respectively.

3. *LS_ 32GB* dataset consists of 26 sequences, with lengths approximately: 6112 MB, 927 MB, 1103 MB, 3100 MB, 2147 MB, 1265 MB, 591 MB, 183 MB, 1272 MB, 1361 MB, 454 MB, 699 MB, 552 MB, 1085 MB, 797 MB, 359 MB, 593 MB, 740 MB, 50 MB, 164 MB, 1295 MB, 3387 MB, 2544 MB, 618 MB, 728 MB, 642 MB, respectively.

4. *LS_ 128GB* dataset consists of 16 sequences, with lengths approximately: 1223 MB, 19029 MB, 6936 MB, 2626 MB, 5801 MB, 3894 MB, 11932 MB, 3175 MB, 16900 MB, 7554 MB, 1524 MB, 18257 MB, 2844 MB, 10510 MB, 1565 MB, 17302 MB, respectively.

5.4.3.2 Experimental Settings

All the experiments described in the next sections have been performed on a supercomputing cluster with the following configuration: a server, acting as *master*

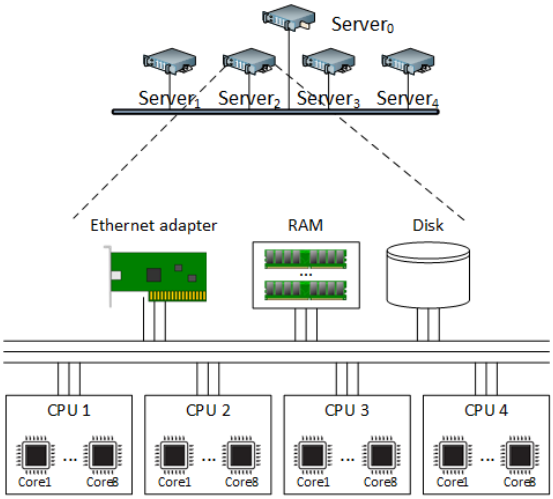

Figure 5.12: Physical cluster hardware used in `KCH` experiments.

node, with 2 processors *quad core Intel Xeon* 1.6 *Ghz CPU family* 6 *model* 26 *stepping* 5, and 32 GB of RAM; and 4 servers, acting as *slave* nodes, equipped each with 64 GB of RAM and 4 *AMD Opteron* 6272 @ 2.10 *GHz* processors with 32 total cores. Each server runs *CentOS* 6.7 *Linux* 64 *bit* operating system (kernel version 2.6.32) and it owns a local 791 GB disk drive (197 GB disk drive on master node) and a Gigabit Ethernet connection. The experiments to estimate the performance of `KCH` and other distributed algorithms (*e.g.*, BioPig) use all these nodes, whereas the other selected tools only use a single slave node. The Hadoop version is 2.7.1[1].

A schematic rendering of the cluster hardware, with one master and four slave nodes is provided in Figure 5.12, while a schematic representation of the cluster configured to run `KCH` is presented in Figure 5.13. At start-up, input files are available in HDFS, where they have been originally split in several parts of equal size (except for the last one) and scattered among the slave nodes of the cluster. Then, a map task (*i.e.*, a set of map functions) for each file part will be run by a `Container` (worker) on a slave. As soon as are to be completed all the map tasks, the framework Hadoop starts the running of reduce tasks on the output of the previous phase. The computation ends when all tasks have been executed.

[1]The experimentations were conducted between September and December 2015.

5.4.3.3 Tuning Phase

The setup of a Hadoop cluster for `KCH` requires the definition and setting of several configuration parameters that will determine the final performance of the entire implementation. Here we have identified three of those parameters and we have experimentally determined their best setting. They are highly correlated and they must be carefully set up depending on the hardware available.

How Many Workers per Node With reference to Figure 5.12, that schematically depicts the hardware, we have described in Section 5.4.3.2, one obvious decision is to have the master node to run the Hadoop cluster management services, typically task scheduling and distributed file system management. Next task is to decide how many workers to assign to each slave node. Indeed, as already mentioned, and schematically illustrated in Figure 5.13, the degree of parallelism that can be achieved by a node of a Hadoop cluster is defined by the number of workers (*i.e.*, Hadoop `Containers`) that can run concurrently on that node. The number of workers per node should be equal to the number of cores available on that node. However, this rule may not work with multi-core systems, where each node may be equipped with tens or hundreds of CPU cores and it has a single local disk.

In our case, each slave node is equipped with 32 cores and 64 GB of RAM. This would allow us to run up to 32 workers in parallel on a same slave node. However, this would lead to a severe performance degradation for two main reasons. First, running 32 workers on the same node would leave each worker with less than 2 GB of RAM, an amount of memory that would prevent the worker from efficiently processing long genomic sequences for large values of k. Second, the 32 workers would share the same local disk and the same network connection. For the nature of our problem, this would imply several performance bottlenecks due to two or more workers trying to access the same resource (either the network or the local disk) at the same time. Therefore, the advantage of scaling a computation over a larger number of cores could be canceled out by the computational overhead due to virtual memory thrashing and I/O bus congestions. A rule of thumb is to analyze te performance of the algorithm as a function of workers per node up to

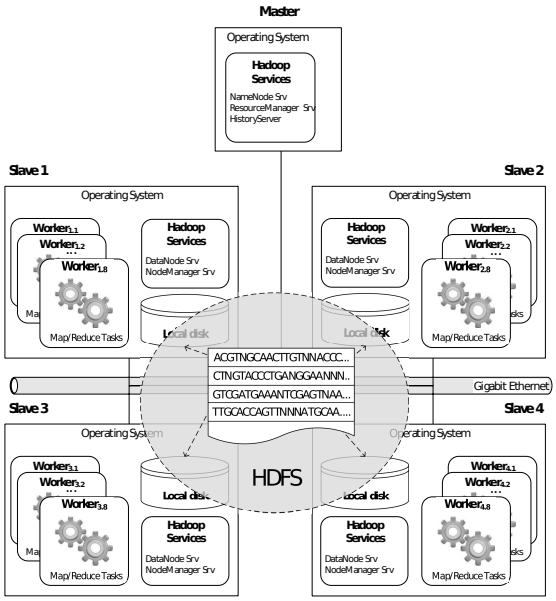

Figure 5.13: A schematic representation of a cluster configured to run KCH algorithm designed for Hadoop. The communication among nodes uses the Gigabit Ethernet, while the storage of input and output files is distributed among the local disks present in each slave node.

the physical number of cores available on that node, exploiting all the available RAM. Then, one choose the number of workers resulting in the best performance.

Input Partitioning Another important parameter is the size of the pieces in which the input files are split (*i.e.*, HDFS block size), where each piece (called input split) is then processed by a distinct map task. The input block size defines the overall number of total map tasks and the workload of each of them (*i.e.*, *task granularity*).

A small block size would require the execution of many short-lasting map tasks, each counting a relatively small number of k-mers. Consequently, the map tasks lifetime would be dominated by the time required to manage them. Splitting an input file in many small pieces would improve the parallelism in the execution of the map tasks, but would increase the amount of intermediate data to be transferred. This happens because the two-levels aggregation strategy employed by KCH is less effective when the size of the input sequences for each map task gets smaller.

Conversely, splitting an input file in few very large pieces would significantly help the data compression performed by the two-levels aggregation strategy, but at the expense of reducing the degree of parallelism. A block size too large would likely increase proportionally the number of distinct k-mers to count with the side effect of consuming all the memory of a map task, thus causing its interruption.

In order to exploit parallelism, a general strategy is to choose the block size so to have a number of map tasks that is, at least, equal to the overall number of workers in the cluster. Moreover, the block size should be as large as possible, provided that each map task would have enough memory to process input blocks with that size. Thus, the performance of `KCH` in regard to this parameter must be analyzed as a function of the number of workers. That is, by experimenting with block sizes resulting in a number of map tasks that is an increasing multiple of the number of workers.

Partitioning Σ^k Another configuration parameter is the number of partitions (*i.e.*, hash tables or bins) of the universe of possible k-mers to be used during the two-levels aggregation strategy. Notice that all the k-mers found while scanning the input sequences and falling in each partition are processed by a distinct reduce function. Thus, as for the case of splitting the input, keeping this number low implies the execution of a small number of long-lasting reduce functions. This decreases the overhead related to the execution of reduce functions but prevents from fully exploiting the parallelism of the cluster (few long tasks are more difficult to be scheduled in an efficient way than many short tasks) and it requires the usage of more memory for maintaining the k-mer counts. Instead, using many short reduce functions would improve parallelism, but would increase as well the performance overhead due to the execution of all these functions. In our case, the number r of reduce tasks (*i.e.*, the number of hash tables) must be analyzed as a function of k in order to obtain an appropriate value r_k, for each k.

Tuning Results Given k, one should tune the previous parameters by varying them while running `KCH` on datasets of different sizes chosen for the tuning. This would return, for each dataset, a 3-dimensional grid reporting the execution times measured while running `KCH` with each of the considered assignments to the tuning

parameters. Then, one would choose, for each dataset size, the parameters-assignment yielding the lower execution time. However, such an approach would be cumbersome and very time-consuming. Here, we propose a heuristic that is able to produce a good calibration of these parameters in a smaller amount of time. We just fix two tuning datasets that are representative, in size, for large datasets and small datasets, *i.e.*, *CS_8GB* and *CS_128GB*. Then, we proceed by running the tuning procedure on these two datasets and use the outcoming parameters assignments as standard configurations for the more general case. Moreover, we also experiment with a single "large" value of k, *i.e.*, $k = 31$, that is a worst case scenario. In all our experiments the HDFS replication factor is set to 2.

The results of this tuning are reported in Tables 5.2 and 5.3, and they are summarized in the following:

- **Number of workers per node**. The number of workers per node returning the best execution times for `KCH` is 16, when processing a light workload, and 8, when processing a heavy workload.
- **HDFS block size**. The HDFS block size m returning the best execution times for `KCH` is either 64 MB or 128 MB, when processing a light workload, and 256 MB, when processing a heavy workload.
- **Number of reduce tasks** The number r of reduce tasks returning the best execution times for `KCH` is 279 both for light and heavy workloads.

Due to the small timing differences between the configurations arising from the tuning on a light workload and on a heavy workload, and because of the need of conducting the rest of our experiments at scale, we decided to use for our tests only the configuration emerging from the tuning on a heavy workload (*i.e.*, 8 workers per node, 279 reduce tasks, and a block size set to 256 MB). In particular, adopting 8 workers for each slave node, each worker has 8 GB of RAM. In the experiments presented in following we alway use these parameters unless otherwise stated in the text.

Table 5.2: Execution times of KCH in minutes when run on dataset of size *CS_ 8GB* with $k = 31$, while using an increasing number of workers per node and of reduce tasks r. In this experiment, the HDFS block size m is set to 64 MB, 128 MB, 256 MB or 512 MB. The shortest execution times are marked in bold. Empty cells report failed executions.

	Dataset *CS_ 8GB*															
	$m =$ **64 MB**				$m =$ **128 MB**				$m =$ **256 MB**				$m =$ **512 MB**			
	# Workers per Node				**# Workers per Node**				**# Workers per Node**				**# Workers per Node**			
r	**4**	**8**	**16**	**32**	**4**	**8**	**16**	**32**	**4**	**8**	**16**	**32**	**4**	**8**	**16**	**32**
15	9	8	-	-	8	12	-	-	11	11	-	-	-	-	-	-
31	9	6	8	-	9	7	8	-	9	8	13	-	26	30	-	-
62	9	7	6	-	8	6	6	-	8	6	-	-	9	10	-	-
93	10	7	6	6	9	6	6	-	8	6	10	-	9	10	-	-
124	10	7	6	-	9	6	6	-	8	7	9	-	9	14	-	-
155	10	6	6	6	9	6	5	-	8	6	11	-	9	10	-	-
186	10	7	6	6	9	6	6	-	9	6	9	-	9	10	-	-
217	10	6	**5**	6	9	6	6	-	8	7	-	-	9	-	-	-
248	10	7	6	6	9	6	6	-	9	7	9	-	9	9	-	-
279	10	7	6	6	9	6	**5**	-	9	6	9	-	9	9	-	-
310	11	7	6	7	9	6	6	-	9	7	-	-	9	11	-	-

Table 5.3: Execution times of KCH in minutes when run on dataset of size *CS_ 128GB* with $k = 31$, while using an increasing number of workers per node and of reduce tasks r. In this experiment, the HDFS block size m is set to 64 MB, 128 MB, 256 MB or 512 MB. The shortest execution times are marked in bold. Empty cells report failed executions.

	Dataset *CS_ 128GB*															
	$m =$ **64 MB**				$m =$ **128 MB**				$m =$ **256 MB**				$m =$ **512 MB**			
	# Workers per Node				**# Workers per Node**				**# Workers per Node**				**# Workers per Node**			
r	**4**	**8**	**16**	**32**	**4**	**8**	**16**	**32**	**4**	**8**	**16**	**32**	**4**	**8**	**16**	**32**
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
31	282	-	-	-	196	-	-	-	177	-	-	-	-	-	-	-
62	161	-	-	-	133	-	-	-	117	-	-	-	126	-	-	-
93	159	125	-	-	133	119	-	-	112	110	-	-	104	139	-	-
124	149	151	-	-	118	125	-	-	104	-	-	-	106	145	-	-
155	147	109	-	-	118	91	-	-	101	83	-	-	93	110	-	-
186	155	109	-	-	117	91	-	-	102	82	-	-	94	103	-	-
217	139	101	-	-	118	85	-	-	103	88	-	-	100	-	-	-
248	145	106	-	-	118	87	-	-	104	82	-	-	98	-	-	-
279	135	87	94	-	115	81	86	-	103	**73**	-	-	93	96	-	-
310	142	97	96	-	118	81	94	-	104	79	-	-	96	96	-	-

5.4.3.4 Experimental Results

We have conducted a sets of experiments to evaluate KCH on different scenario. It is important to emphasize that a stand-alone implementation, that only works in memory for extracting k-mers, fails with large datasets and k due to RAM problems. Therefore, many solutions in literature are designed to use the disk as auxiliary memory. Section A.2 presents the state of the art on algorithms collecting k-mer statistics.

Initially, we compare KCH with other Hadoop-based solutions, then is shown the scalability of KCH increasing the number of total workers in LS and CS case. Finally, the scalability of KCH for cumulative statistics is compared with the most popular and fast k-mer counting tool in multi-threading environments, *i.e.*, KMC2 (see Section A.2.1.6 for details about KMC2).

Our datasets of 2 GB, 8 GB, 32 GB and 128 GB are loaded on HDFS in approximately 20 seconds, 80 seconds, 5 minutes and 20 minutes, respectively.

KCH versus Other Hadoop-based Solutions Initially, we have compared KCH, which uses the in-mapper local aggregation with explicit partitioning, with the naive solution for k-mer cumulative statistics as described in Section 5.4.1. Here the naive solution also uses the Hadoop Combiner to aggregate the partial counters for improving the performance. In addition, we have also compared these solutions with a variant of KCH that adopts in-mapper local aggregation without partitioning called `Preliminary KCH` (*i.e.*, a map task uses a single hash table for in-mapper aggregation, and it simply emits the pairs at the end). Section 5.4.2.3 describes the difference between `Preliminary KCH` and KCH.

Figures 5.14 and 5.15 show the results of these comparisons for $k = 15$ and 31, respectively, extracting canonical k-mers. We have used 4 slave nodes with 32 total workers. For the case $k = 15$ (see Figure 5.14) with the dataset *CS_128GB*, KCH is $\approx 30\times$ faster than the naive solution, while `Preliminary KCH` is $\approx 6\times$ faster than the naive implementation. In other words, KCH is approximately $5\times$ faster than `Preliminary KCH` solution. For the case $k = 31$ with the dataset *CS_128GB* (see Figure 5.15), the naive solution has failed due to memory problems after about 24 hours, while KCH is approximately $3.4\times$ faster than the other

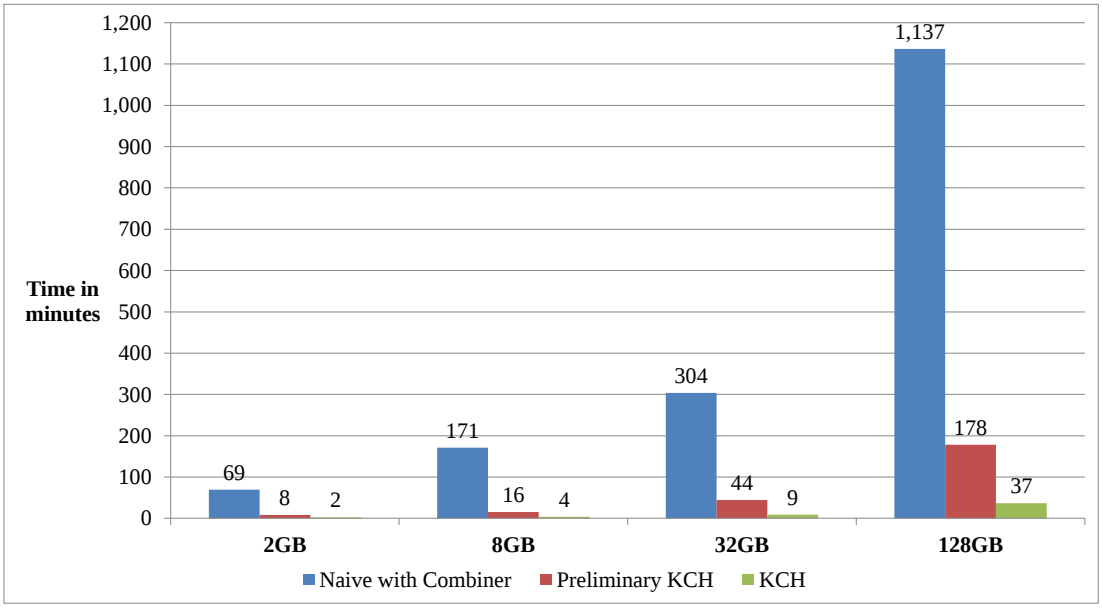

Figure 5.14: *Cumulative Statistics*. Comparison of `KCH` versus the Hadoop-based naive solution with Combiner and `Preliminary KCH` using $k = 15$ for all datasets.

solution. Incrementing the k, *e.g.*, $k = 31$, there are many unique k-mers due to errors in reads (*e.g.*, [194]), therefore, the aggregation strategy of `KCH` brings less advantages with respect to case $k = 15$.

In addition, we have compared these three solutions with very small value of k, such as 3 and 7. In this scenario, the performance of `KCH` (in-mapper local aggregation with explicit partitioning) are similar to `Preliminary KCH` that uses in-mapper local aggregation without partitioning. However, these two solutions are much faster than the naive solution with Combiner.

Another Hadoop-based solution for k-mer counting is contained in the tool BioPig ([37, 209]), which is build on Hadoop and Pig dataflow language [213]. BioPig is a collection of cloud computing tools to scale data analysis and management, and it also includes a tool for k-mer counting (*kmerCount.pig*) [150]. Section A.2.1.8 presents additional details about BioPig.

Figure 5.16 reports the results related to cumulative statistics on 4 slave nodes and 32 total workers between `KCH` and BioPig. In BioPig experiments we have selected our CS datasets of 32 GB and 128 GB (*CS_32GB* and *CS_128GB*)

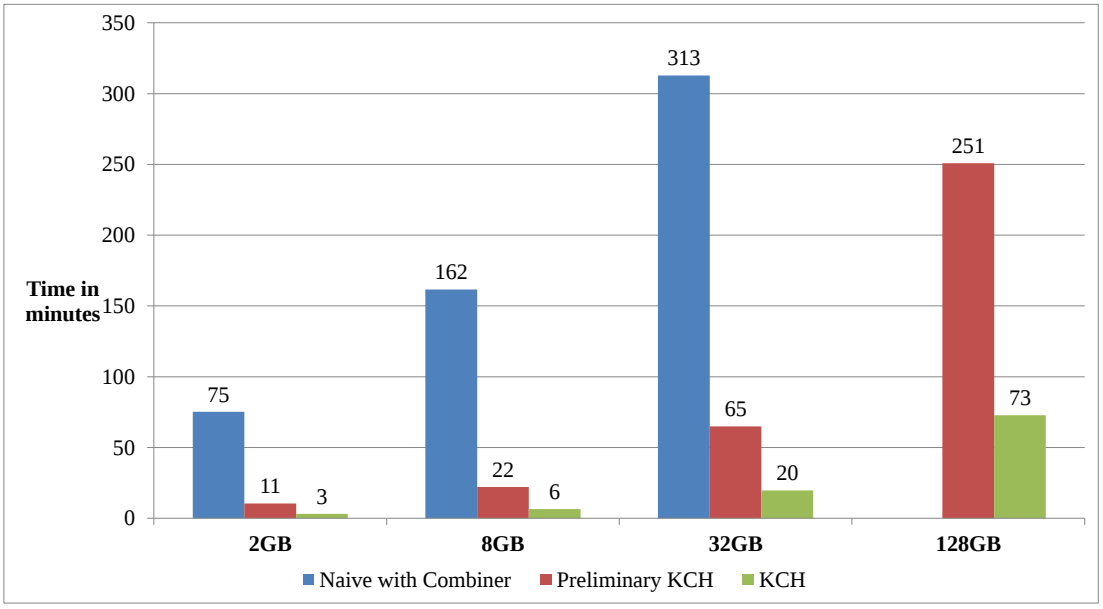

Figure 5.15: *Cumulative Statistics*. Comparison of `KCH` versus the Hadoop-based naive solution with Combiner and `Preliminary KCH` using $k = 31$ for all datasets.

using non-canonical[1] k-mers for $k = 3$ and $k = 7$. BioPig only extracts non-canonical k-mers on short sequences, using the query *kmerCount.pig* provided in [150]. We have used the same number of reducers (*i.e.*, 64 for $k = 3$ and 279 for $k = 7$) and block size (*i.e.*, 256 MB) for `KCH` and BioPig. We have only used small k values because the execution times of BioPig are very slow. In fact, `KCH` is on average approximately 40× faster than BioPig in these settings[2]. BioPig is not at all competitive with respect to `KCH` since it suffers of exactly the kind of problems outlined when presenting the naive implementation of k-mer counting for Hadoop (see Section 5.4.1).

Comparison about FASTA Input Management We have performed a simple test to compare the experimental performance of our FASTA reader classes (as discussed in Section 5.4.2.1), against the two provided by BioPig (*i.e.*,`FASTAInputFormat` and `FASTABlockInputFormat`). We also included in our test the Hadoop

[1]The term *non-canonical k-mers* indicates the standard/traditional k-mers.
[2]The average speed up of `KCH` with respect to BioPig is ≈ 40.

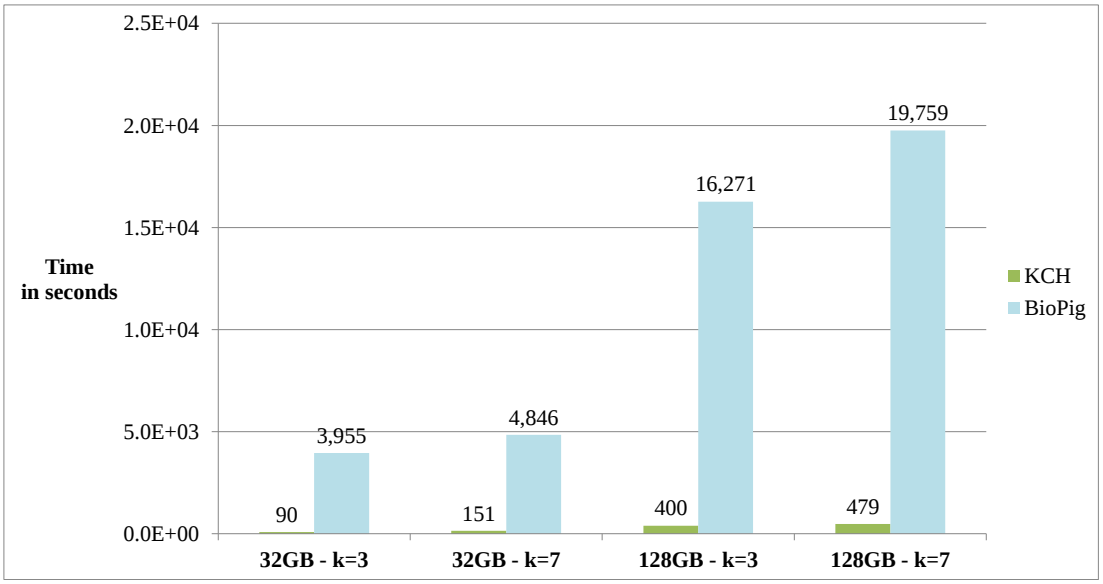

Figure 5.16: *Cumulative Statistics*. Results of `KCH` with respect to BioPig, for the case $k = 3, 7$ (non-canonical k-mers), for *CS_32GB* and *CS_128GB* datasets.

default `TextInputFormat` class. The BioPig readers are only able to deal with short sequences, whereas the `TextInputFormat` class does not support sequences spanning multiple lines. For this reason, we used, as reference datasets, those containing a set of short sequences and used for cumulative statistics (see Section 5.4.3.1 for details). Since we were interested just in benchmarking the time required to read input sequences, we considered a very simple Hadoop job with no reduce function. The map function of this job just counts the occurrences of each character in an input sequence (*i.e.*, $k = 1$) without any form of output. We have used for this test a single slave node with 2 workers (only a worker runs map functions because one executes the `Application Master` service). The HDFS block size was always set to 256 MB.

Our experimental results are visible in Figure 5.17. Notice the very bad performance of the two BioPig reader classes. Although being more engineered than the standard `TextInputFormat`, these two classes pay the penalty of being more complex because of the support for multi-lines sequences. Instead, the naive line-based approach of `TextInputFormat` pays off in such a simple scenario. Despite this, we notice that our reader class `FASTAshortInputFileFormat` is

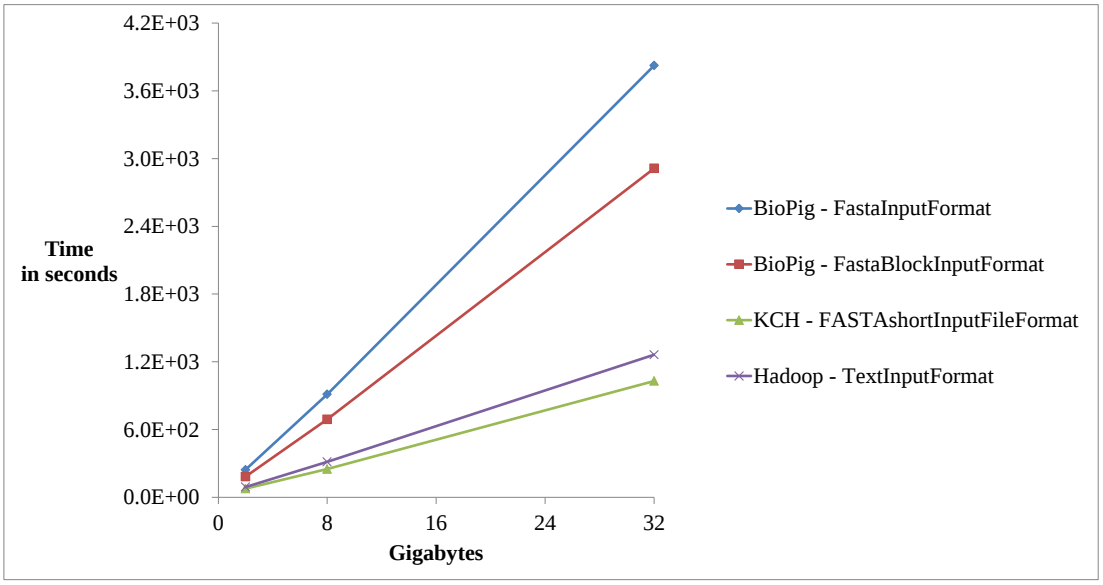

Figure 5.17: Comparison between different Hadoop-based `InputFormat` solutions to process FASTA files. The benchmark counts the number of different characters in each short sequence (*i.e.*, read) without any output, while using datasets of increasing size.

always consistently faster than the other ones.

Local Statistics Since none of the algorithms available in the literature for k-mer statistics (summarized in Section A.2.1) can be used in this setting, we take as a measure of performance only the scalability of our algorithm `KCH`. The speed up in time is obtained as a function of the number of total workers that it has available from the Hadoop cluster.

The experiments are performed with varying dataset sizes and values of k. For the former, we have used *LS_2GB*, *LS_8GB*, *LS_32GB* and *LS_128GB*, respectively, since that range of sizes well represents possible input sizes of datasets coming from genomic and metagenomic studies. Likewise, the chosen values of k, *i.e.*, 3, 7 and 15, are representative of the ones that are expected to be used in applications such as alignment-free sequence comparison and compositional analysis of biological sequence (*e.g.*, [66, 116, 118]), where values of k substantially above 10 are hardly found. In this case, we are interested in extracting non-canonical

k-mer counts as we are using entire genomes as input sequences.

Figures 5.18, 5.19, 5.20 and 5.21 report the results of the LS experiments for `KCH`. Here we have used 4 slave nodes varying the number of workers (*i.e.*, Hadoop `Containers`) for each slave (*i.e.*, 1, 2, 4 and 8). We have used our datasets for LS extracting local statistics about non-canonical k-mers for $k = 3, 7, 15$. For the case $k = 3$ and 7 we have used a single hash map for each mapper, and the number of the reduce tasks was 279. In this case the map function simply emits the pairs $<(idSeq, kmer), frequency>$. For $k = 15$ we used 279 hash maps for each map function and 279 reducers. In this case the map function emits the pairs $<(idSeq, idHt), ht>$. In our tests the hash tables sizes are right according datasets and k values to prevent intermediate hash tables flush.

As it is evident from Figures 5.18, 5.19 and 5.20, the advantage of using more and more workers, *i.e.*, the scalability of the algorithm, becomes more and more evident as the dataset size increases. Indeed, when processing relatively small datasets (*i.e.*, *LS_ 2GB*), there is almost no benefit from scaling the processing of our algorithm on a high number of workers. This holds mainly because Hadoop, based on the block size $m = 256$ MB, splits the input sequences in a small number of parts that are distributed to the map tasks, thus preventing the parallelism to be exploited on the map side. The opposite case occurs when processing relatively large sizes (*i.e.*, *LS_ 128GB*). Here, the input sequences are split in a much larger number of parts (519 parts in the *LS_ 128GB* case) thus allowing to fully exploit the intrinsic parallelism of Hadoop.

Cumulative Statistics Here the experiments are performed varying dataset sizes and values of k. The selected datasets are: *CS_ 2GB*, *CS_ 8GB*, *CS_ 32GB* and *CS_ 128GB*. In addition to the values of k used in the previous, we have also used $k = 31$, which is a value that finds use in k-mer statistics for sequence assembly [71]. Moreover, since we are dealing with reads that have yet to be assembled, we consider canonical k-mer counts.

The results for `KCH` are reported in Figures 5.22, 5.23, 5.24, 5.25 and 5.26. Here we have used 4 slave nodes varying the number of workers (*i.e.*, Hadoop `Containers`) for each slave (*i.e.*, 1, 2, 4 and 8). We have used our datasets extracting cumulative statistics about canonical k-mers for $k = 3, 7, 15, 31$. For

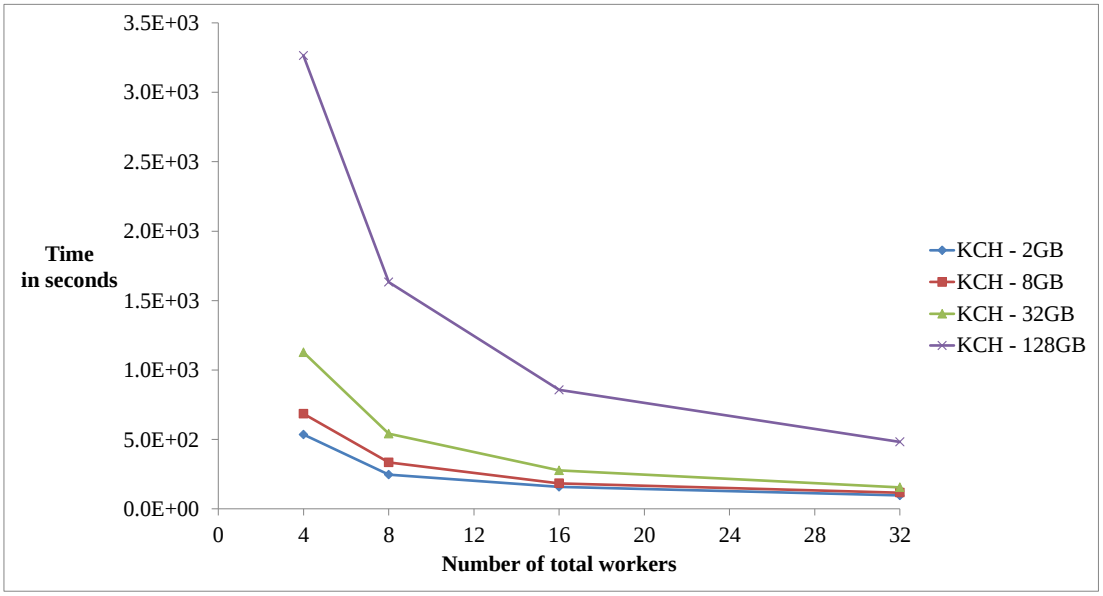

Figure 5.18: *KCH Local Statistics.* Scalability of KCH, for the case $k = 3$, for all datasets, which are indicated in the figure according to the legend to the right. The abscissa gives the number of workers used, while the ordinate gives the corresponding time.

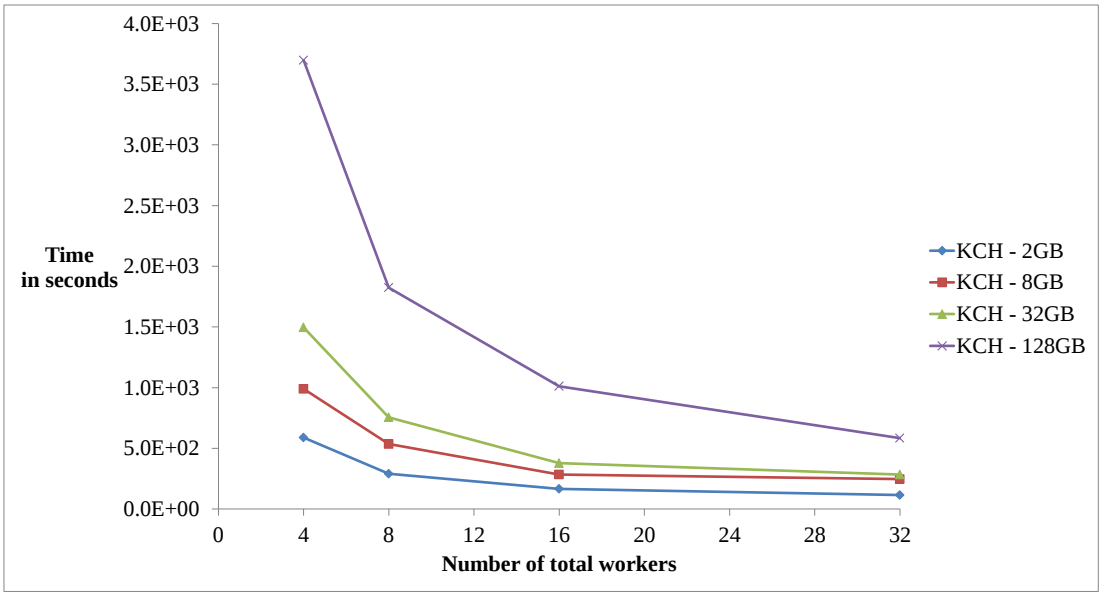

Figure 5.19: *KCH Local Statistics.* Scalability of KCH, for the case $k = 7$, for all datasets, which are indicated in the figure according to the legend to the right. The abscissa gives the number of workers used, while the ordinate gives the corresponding time.

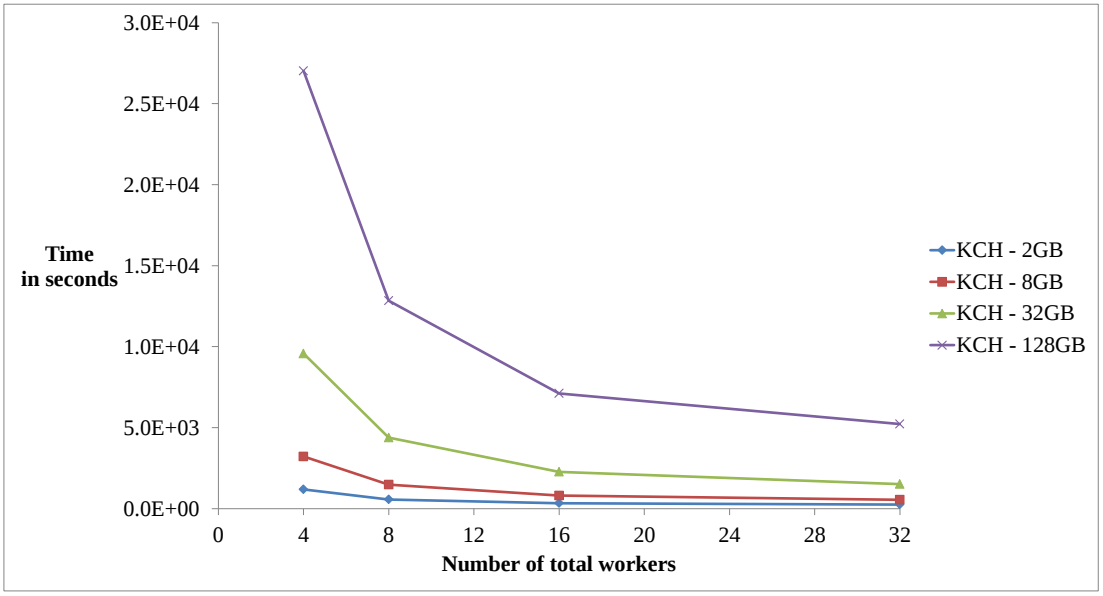

Figure 5.20: *KCH Local Statistics.* Scalability of KCH, for the case $k = 15$, for all datasets, which are indicated in the figure according to the legend to the right. The abscissa gives the number of workers used, while the ordinate gives the corresponding time.

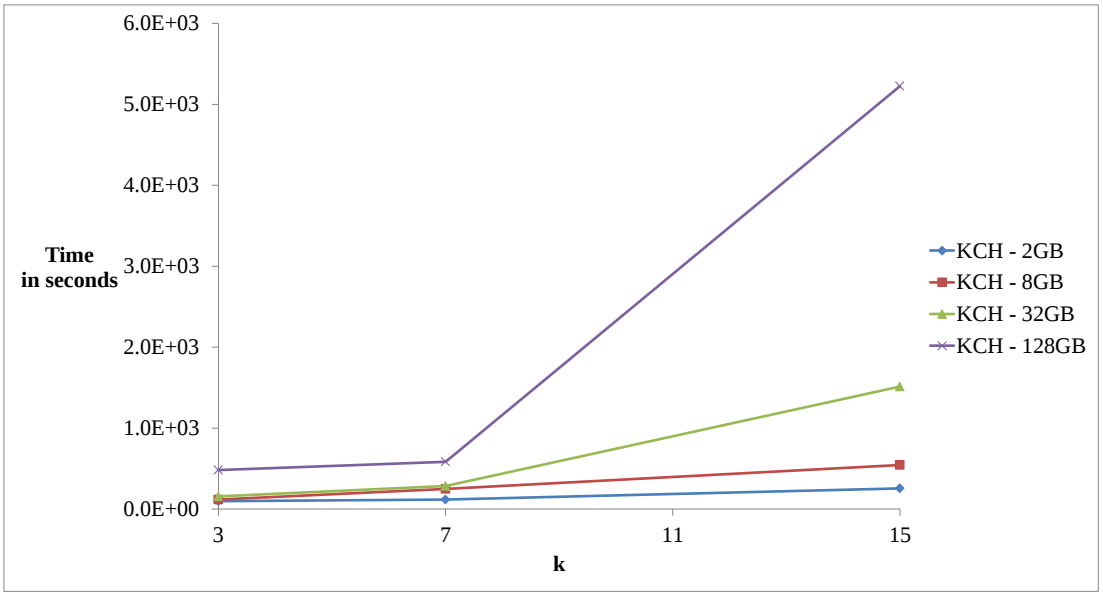

Figure 5.21: *KCH Local Statistics.* Execution times of KCH for the all datasets and for $k = 3, 7, 15$ using 32 total workers.

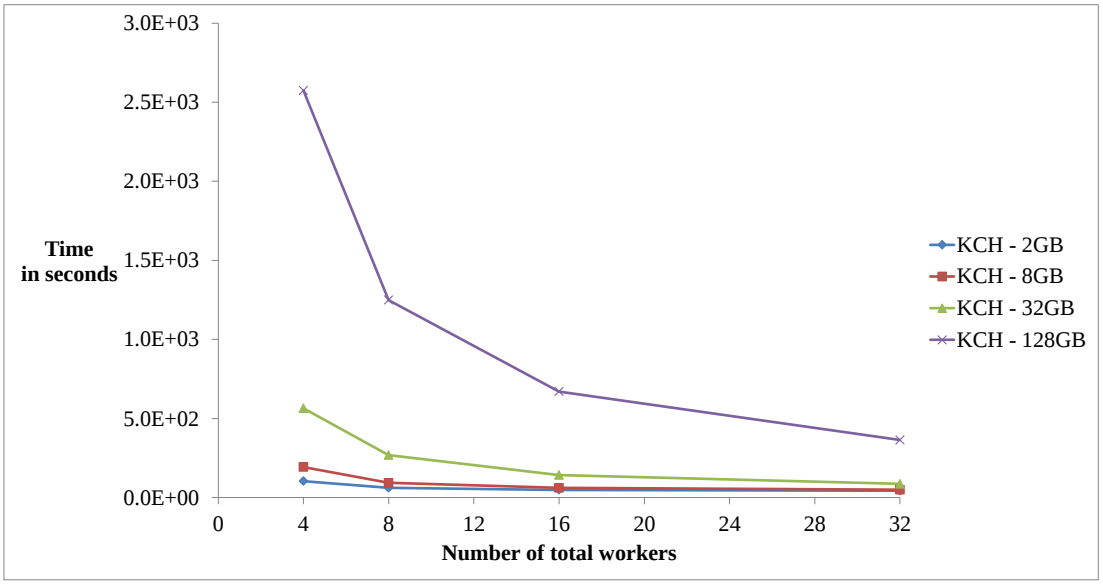

Figure 5.22: *KCH Cumulative Statistics*. Scalability of KCH, for the case $k = 3$, for all datasets, which are indicated in the figure according to the legend to the right. The abscissa gives the number of workers used, while the ordinate gives the corresponding time.

the case $k = 3$ and 7 we have used a single hash map for each mapper, and the number of the reducers was 32 and 279, respectively. In this case the map function simply emits the pairs $<(idSeq, kmer), frequency>$. For $k = 15$ and 31 we used 279 hash maps for each map task and the number of the reducers was 279. In this case the map function emits the pairs $<(idSeq, idHt), ht>$. In our tests the hash tables sizes are right according datasets and k values to prevent intermediate hash tables flush. It is evident from Figures 5.22, 5.23, 5.24 and 5.25 that the performance of KCH for CS is as that for LS, in particular scalability is preserved. Therefore, doubling the number of total workers the execution times are approximately halved for *CS_ 8GB*, *CS_ 32GB* and *CS_ 128GB*.

Figure 5.27 shows the comparison between KCH using 4 slave nodes varying the number of workers per slave and KCH using a variable number of slave nodes with always 8 workers per slave. Cumulative statistics for $k = 31$ are extracted. The solid lines indicates experiments running on 4 slaves varying the number of workers for slave (*i.e.*, 2, 4 and 8, respectively). The dashed lines indicates the experiments running on 1, 2 and 4 slave nodes, respectively, always using

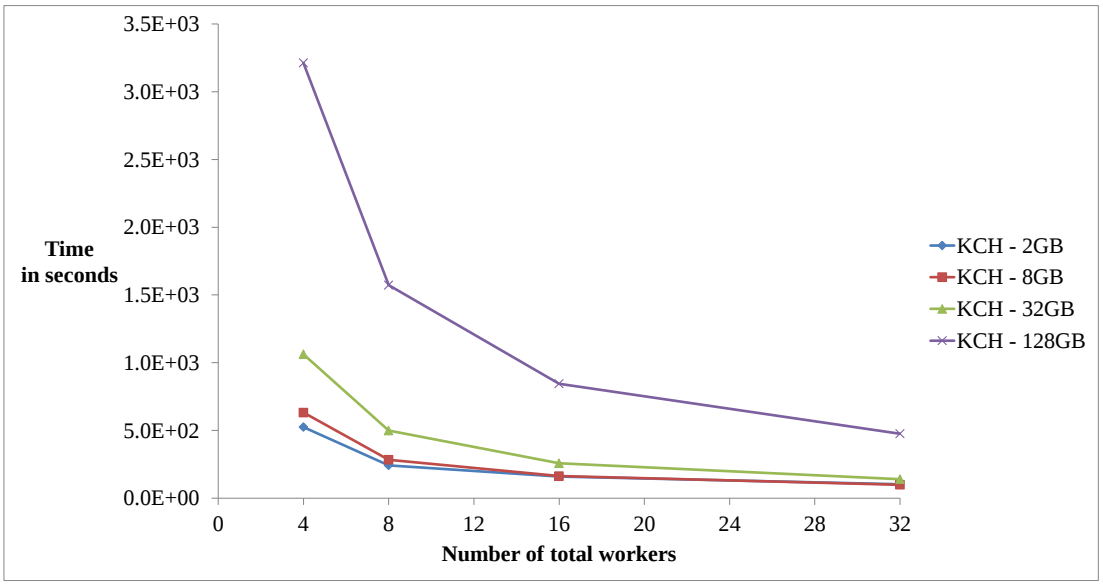

Figure 5.23: *KCH Cumulative Statistics.* Scalability of KCH, for the case $k = 7$, for all datasets, which are indicated in the figure according to the legend to the right. The abscissa gives the number of workers used, while the ordinate gives the corresponding time.

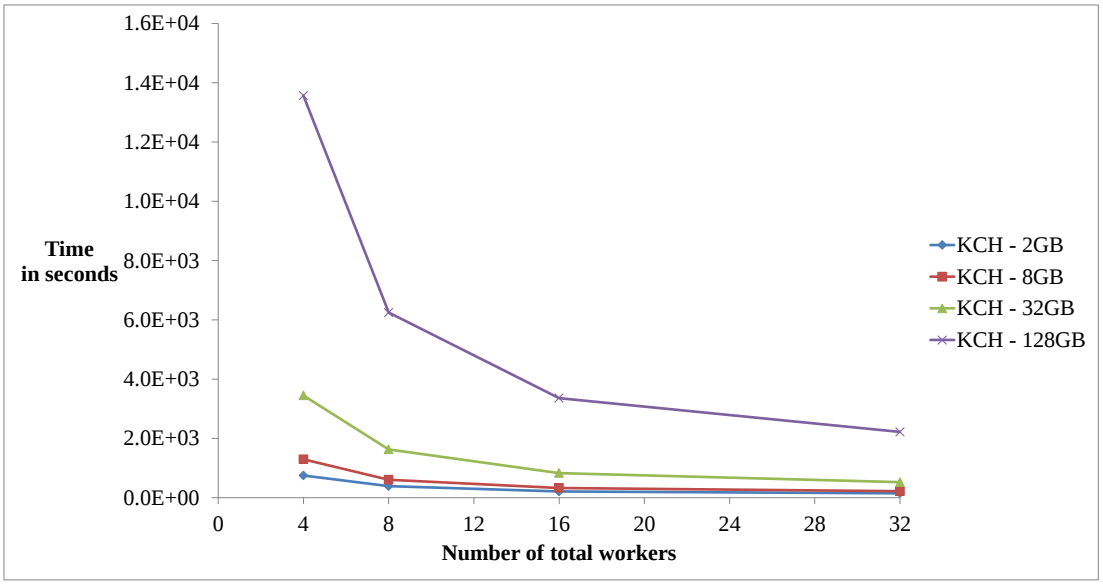

Figure 5.24: *KCH Cumulative Statistics.* Scalability of KCH, for the case $k = 15$, for all datasets, which are indicated in the figure according to the legend to the right. The abscissa gives the number of workers used, while the ordinate gives the corresponding time.

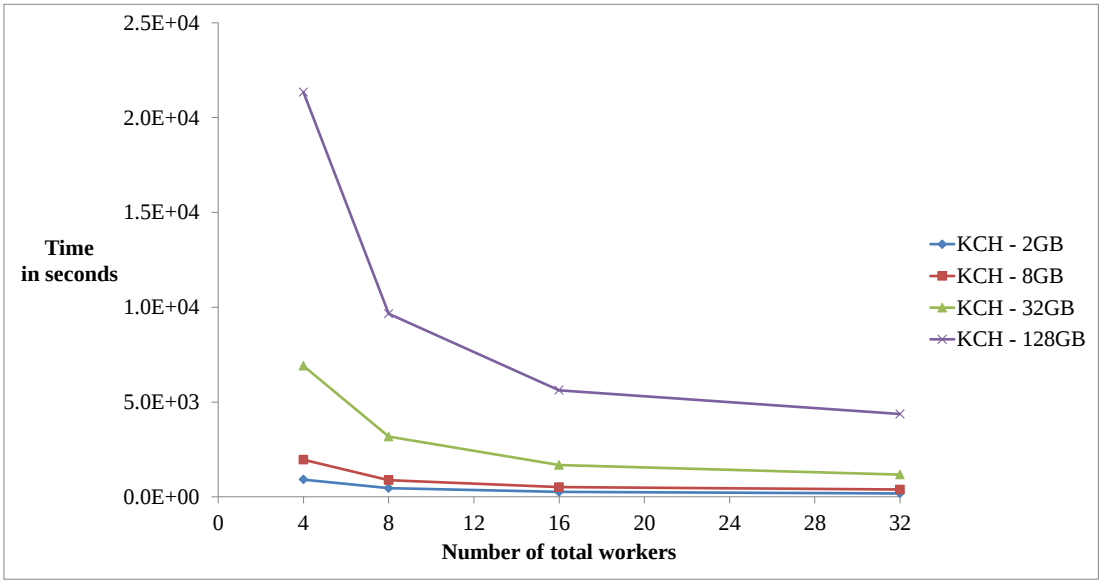

Figure 5.25: *KCH Cumulative Statistics.* Scalability of KCH, for the case $k = 31$, for all datasets, which are indicated in the figure according to the legend to the right. The abscissa gives the number of workers used, while the ordinate gives the corresponding time.

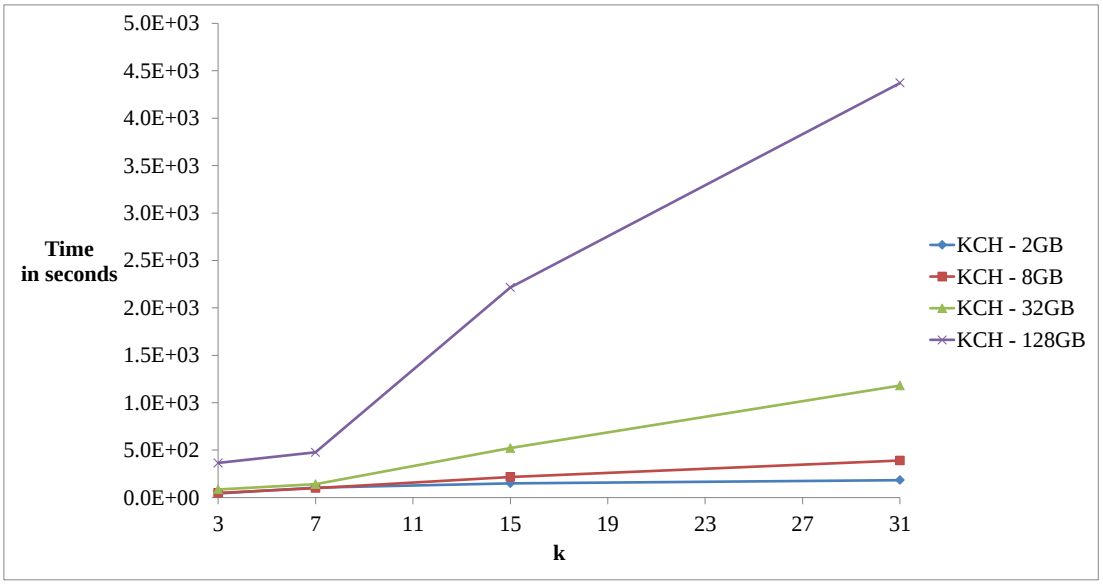

Figure 5.26: *KCH Cumulative Statistics.* Execution times of KCH for the all datasets and for $k = 3, 7, 15, 31$ using 32 total workers.

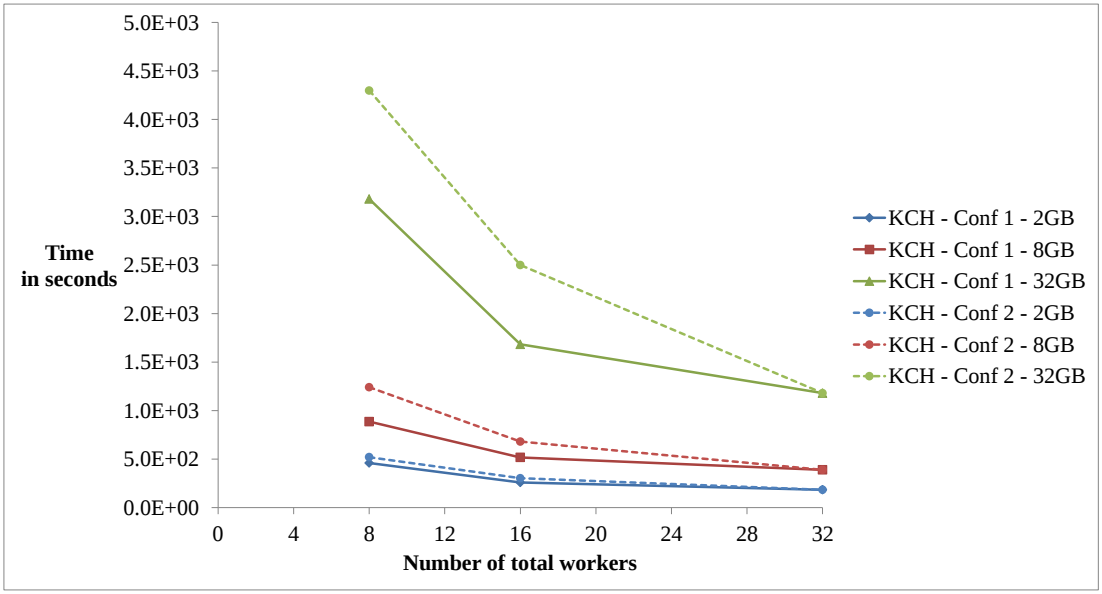

Figure 5.27: *KCH Cumulative Statistics.* Scalability of KCH, for the case $k = 31$ using a different number of slave nodes, for all datasets, which are indicated in the figure according to the legend to the right. The solid lines indicates experiments running on 4 slaves varying the number of workers for slave (*i.e.*, 2, 4 and 8, respectively), *i.e.*, *Configuration 1* (Conf 1). The dashed lines indicates the experiments running on 1, 2 and 4 slave nodes, respectively, using 8 workers for slave node, *i.e.*, *Configuration 2* (Conf 2). The abscissa gives the number of total workers used, while the ordinate gives the corresponding time.

8 workers for slave node. In particular, the figure shows that the scalability is preserved both when the number of slave nodes varies and when is varied only the number of workers for slave. The running times adopting 1 and 2 slaves, with 8 and 16 total workers, are major than the corresponding case (4 slaves with 8 and 16 total workers), because there always are 8 workers for slave which compete on the same resources, *e.g.*, I/O bus, memory and disk.

Figure 5.28 shows the speed up between our stand-alone (sequential) Java implementation executed on a single-core and KCH varying the number of total workers on 4 slave nodes. It is used the dataset *CS_32GB* with $k = 15$. In particular, the figure shows that the speed up is very close to the maximum for the cases 4, 8 and 16 workers.

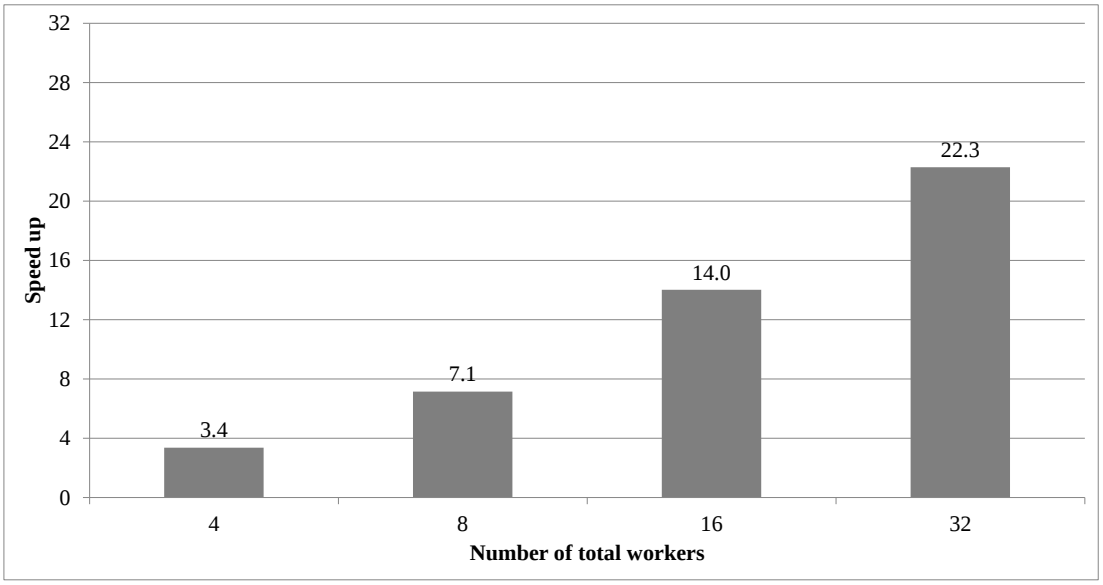

Figure 5.28: *KCH Cumulative Statistics.* Speed up of KCH for the case $k = 15$ and dataset *CS_32GB*. The abscissa gives the number of workers used, while the ordinate gives the corresponding speed up than our stand-alone implementation executed on a single-core.

Comparison between KCH and KMC2 Here we compare KCH with an efficient and fast solution for CS in a multi-threading system, *i.e.*, KMC2. Many contributions in literature have said that KMC2 is a fast and efficient tool (see Section A.2).

In [83] *Deorowicz et al.* have proposed *K-mer Counter* (KMC), a simple, efficient, parallel disk-based algorithm for counting k-mers. The authors in [84] have presented a new version for KMC, *i.e.*, KMC2, that borrows from the efficient architecture of preliminary version of KMC, but reduces the disk usage several times and improves the speed usually about twice. KMC2 has been designed and highly engineered on a shared-memory parallel architecture having in mind NGS genome assembly problems and, in fact, it only works with short reads as input. Therefore, such an algorithm is not as general as KCH.

KMC2 provides, as output, the k-mer counts in a binary file. In fact, in our performance report, we also include the time required to convert these binary files in a human-readable format, an operation that can be usually carried out using the support tool provided with KMC2.

When considering in our experiments a multi-threading shared-memory algorithm, tests are carried out using only one of the available slave nodes. Therefore, in order to obtain results as fair as possible, a thread is considered as a worker in our setting.

Figure 5.29 shows the results of `KCH` compared with KMC2, for $k = 31$. In particular, we notice that KMC2 hardly scales with the number of threads. This seems to be an inherent limitation of that algorithm. It is clear that the advantage granted by the specialization of KMC2 to short reads versus the generality of `KCH` disappears as the degree of parallelism and the dataset sizes increase. However, the performance of `KCH` can be improved on cluster bigger than the one we have used, since `KCH` is scalable. In fact, if the number of the slave node is doubled, the execution times of `KCH` can be approximately halved (see Figure 5.27 for a comparison).

It is methodologically important to highlight that here we see an excellent example of the differences in performance between two algorithms that use different architectures. Indeed, KMC2 adopts a multi-threaded shared-memory architecture, as it works by running multiple threads on the same machine that share the same memory space. This implies that there is no communication overhead due to data being transferred from one thread to another, while the small number of threads prevents the occurrence of performance bottlenecks. Instead, `KCH` runs in a fully distributed environment and it does incur in a significant performance overhead that it has to pay whenever two workers have to communicate via the network connection. Such an overhead is offset by the gain of parallelism only when the number of worker increases. This is clearly visible, again, in Figure 5.29 where we see that, for the *CS_32GB* case, `KCH` exhibits better execution times than KMC2, when using at least 16 workers. Such an analysis is largely confirmed for other values of k, although they may also influence the point in which `KCH` outperforms KMC2. The results for other values of k are reported in Section A.3. However, low values of k are hardly of any use in k-mer counting for sequence assembly.

Section A.3 also presents the comparison between `KCH` and other solutions for cumulative statistics in multi-threading environments. In general, the considerations about KMC2 can be also extended for this tools.

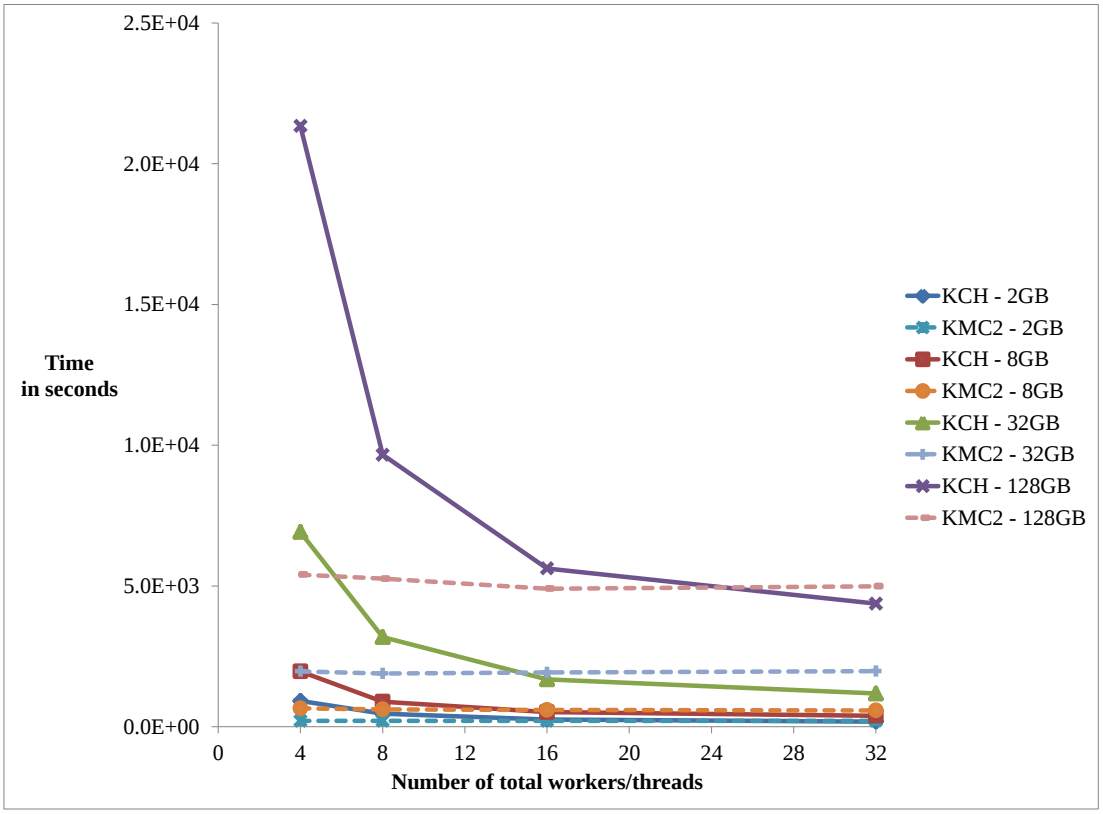

Figure 5.29: *Cumulative Statistics.* Scalability of our algorithm (`KCH`) with respect to KMC2, for the case $k = 31$ and for all datasets, which are indicated in the figure according to the legend to the right. The abscissa indicates the number of workers/threads used, while the ordinate gives the corresponding time in seconds.

5.5 Final Remarks

Although MapReduce and Hadoop are very powerful computational paradigms and programming environments, respectively, to exploit parallelism, relying on the top level functions and macros offered by Hadoop does not allow to take full advantage of the computational power of the hardware resources available. For that to happen, a solution must be entirely conceived in a distributed way, it must ensure data locality and it must be based on the low-level I/O primitives that Hadoop offers. Therefore, it is not trivial to "transform" a stand-alone (sequential) program into a scalable and efficient distributed implementation, although the problem is simple in nature. On one side, the *implicit* distributed programming

can reduce the implementation work. On the other side, a programmer must evaluate how much performance loss exists. In fact, designing, engineering, tuning and profiling activities in Hadoop are vital to obtain fast and scalable solutions.

The simple k-mer counting is extremely used in bioinformatics, therefore, scalable, efficient, fast and distributed solutions must be taken into account. The Hadoop-based naive solution for k-mer counting is conceived as a toy for beginners, therefore, in Big Data era it cannot be adopted. On the other hand, `KCH` is much more fast, although there is much work of design and engineering to improve the performance.

The improvements achieved by `KCH` (see Section 5.4) could be also adopted in the framework `HAFS` presented in Section 5.3 during the indexing phase (*i.e.*, first step). In addition, in `HAFS`, it is required to develop more efficient and fast solutions for computation of distances. In fact, the objective is to use `KCH` as the first line of a distributed pipeline in bioinformatics operations.

The algorithmic engineering methodology [52] is also being recognized as useful for the development of the fast and efficient implementations. In fact, the interactions and beneficial feedback among algorithm design, experimental evaluation and software tuning are becoming part of the development of algorithms for bioinformatics.

Chapter 6

Conclusion and Future Works

In this chapter we review the work presented in this thesis with emphasis on the obtained results, the lessons learned, which could help other researchers, and the future works. In particular, Section 6.1 summarizes the main obtained results, and in Section 6.2 are presented the future directions of our research.

6.1 Outcomes

Nowadays storing and processing Big Data are important tasks to be addressed exploiting the distributed systems. Initially, in Chapter 1, we have presented the Big Data problem, and the motivation and objectives of our thesis. Then, in Chapter 2 was presented the parallel and distributed computing with emphasis on emerging paradigms and software solutions to process Big Data, such as MapReduce ([81, 82]) and Apache Hadoop ([14]), and how to measure the performance in parallel and distributed environments. The MapReduce paradigm and Hadoop are deeply treated in Chapter 3.

In this dissertation, we have experimentally evaluated as the distributed computing, in the present case Hadoop, can be used to speed up a heavy computation using a set of computers. We have discussed as the algorithm engineering, the tuning, the profiling, the code improvements, and the experimental activities on a Hadoop application can improve its performance, *i.e.*, to reduce the execution times, obtaining scalable, fast and efficient solutions. Our goal was to show that, when porting algorithms to Hadoop, a careful profiling and engineering activity

is often required to fully exploit the real potential of the distributed computing system. In addition, we also provided an experimental methodology that can be used to validate or to improve the distributed implementations.

In particular, in this thesis, we have taken as benchmarks some problems of two research areas where the Big Data problem is increasing: Digital Image Forensics and Bioinformatics. In the first field, we have analyzed the algorithm by *Fridrich et al.* [181] for the Source Camera Identification (SCI), *i.e.*, recognizing the digital camera used for acquiring a given digital image. In Chapter 4 are shown the details of this algorithm, and then is described our work done to efficiently speed up the running times of *Fridrich et al.* approach using a Hadoop application executing on a commodity cluster. Our results are summarized in Section 6.1.1.

Nowadays, there is still a lot gap between the available applications and Big Data in bioinformatics. We have tried to reduce this gap using distributed computing on two main problems: Alignment-free Sequence Comparison and extraction of K-mer Statistics. These problems were introduced in Sections 1.3 and 5.2. In Chapter 5 are presented our Hadoop-based distributed implementations to solve these problems: Alignment-free Sequence Comparison on Hadoop (`HAFS`) and K-mer Counting on Hadoop (`KCH`). The obtained results are briefly summarized in Sections 6.1.2 and 6.1.3, respectively.

In the following we briefly review our contributions solving these problems, and the general lessons useful to other researchers. The general remarks are shown in Section 6.1.4.

6.1.1 Results about Source Camera Identification on Hadoop

In Chapter 4, we discussed the engineering of an efficient Hadoop-based implementation of the algorithm by *Fridrich et al.*, in order to solving the Source Camera Identification problem. We were able to quickly obtain a naive distributed implementation for this algorithm, by leveraging the standard facilities available with the Hadoop framework to wrap up an existing Java implementation and to

make it distributed. However, this naive distributed implementation exhibited a very poor performance. This motivated us to perform a thorough profiling activity which led, first, to pinpoint several performance issues and, then, to develop several improvements. The enhanced distributed implementation exhibited a much better performance than the initial distributed implementation exploiting a commodity cluster of 33 PCs. In addition, as pointed out by experimental results, the proposed implementation is also scalable.

Despite the focus on the identification algorithm by *Fridrich et al.*, many of the developed improvements can be trivially used to improve the performance of other Source Camera Identification algorithms, such as [30, 120, 121, 264], when run on Hadoop. Most of these algorithms, in fact, share a same execution pattern, where a set of features is extracted from each image of a given input set, in order to build a digital fingerprint for the camera used for shotting it. This fingerprint is compared to the fingerprint of an image under scrutiny to determine whether the photo has been taken by using that camera or not. For instance, the usage of Hadoop `SequenceFile`s with an associated custom splitter would allow for a more efficient processing of a large number of images. The usage of an in-mapper local aggregation would allow a map task to aggregate together the features extracted from a set of images before sending them to the reduce task, so to save network bandwidth and reduce the overall execution time. The usage of the producer-consumer pattern would allow a map task or a reduce task (in a multi-core setting) to drastically reduce the waiting time due to the input of the images or fingerprints to be processed.

6.1.2 Results about Alignment-free Sequence Comparison on Hadoop

In Section 5.3 we have presented a first systematic study of word-based alignment-free sequence comparison methods on Hadoop. We have evaluated the possibility of using Hadoop to speed up the running times of these methods, compared to their original stand-alone (non-parallel) formulation. In details, we have been able to develop a MapReduce formulation of a generic alignment-free sequence com-

parison algorithm that is able to scale well with the number of used concurrent processing units. In fact, `HAFS` is a distributed framework for the development of word-based alignment-free sequence comparison methods, and it is extensible to new dissimilarity measures between sequences. Indeed, our framework includes many measures of dissimilarity to compare two sequences, and, it is easy extensible, that is a new word-based dissimilarity measure can be included only writing a Java Class to manage it.

Moreover, our solution allows to conveniently process problem instances that are usually hard to solve in a stand-alone setting because of memory limitations. It is also explored the possibility of running alignment-free sequence methods on very long sequences, by using a proper MapReduce formulation able to spread on the several computers of a Hadoop cluster the data structures needed to run them.

In details, a first round of experiments has been conducted on a single machine adopting a Java stand-alone implementation, where we measured the performance of these algorithms on a reference dataset. Then, we developed an implementation for these algorithms on the top of the Apache Hadoop. By taking advantage of careful profiling analysis of the algorithms, we engineer very fast implementations of them. We repeated the same experiments performed on the stand-alone setting on a homogeneous cluster of 5 multi-processor workstations running Hadoop, and we have compared the corresponding results. We have also presented a performance analysis and profiling of the implementations acquired during the previous activities.

6.1.3 Results about K-mers Statistics on Hadoop

The k-mers extraction is a simple task, but counting them in Next-generation DNA sequencing (NGS) era can easily pass the memory capacity of a single PC. This counting is extremely used in bioinformatics, therefore, scalable, efficient, fast and distributed solutions must be taken into account in Big Data era.

In Section 5.4 we have provided a well-designed and properly engineered Hadoop algorithm, called `KCH`, for this fundamental task in bioinformatics. It works extracting cumulative statistics (CS) or local statistics (LS) on short or

long sequences. Although both versions of the problem are algorithmically very simple, the big amount of data to be processed has motivated the development of many algorithms and tools that try to take advantage either of parallelism or of sophisticated algorithmic techniques or both.

In fact, for example, in literature there are some solutions for k-mers cumulative counting on Hadoop, such as [37, 209, 216, 217, 220]. However these solutions are very simple and they have many problems as extensively discussed and experimented in Sections 5.4.1 and 5.4.3.4. Indeed, a naive Hadoop solution for k-mers counting is considered as a toy for beginner, therefore, in Big Data era it cannot be adopted. On the other hand, `KCH` is much more fast, although there is much work of design and engineering to improve its performance.

Therefore, `KCH` is a highly engineered, scalable and efficient Hadoop solution to compute k-mers statistics for both LS and CS, and these features are experimentally evaluated in Section 5.4.3.4. In addition, our analyses show that `KCH` can be very competitive with respect to algorithms whose parallelism is supported by computer architectures other than distributed. In fact, `KCH` is efficient, with respect to some solutions based on parallel and distributed architectures, and fully scalable in terms of processing units.

The k-mer counting problem is at the start of many bioinformatics pipelines, and `KCH`, thanks to its performance and generality with respect to algorithms for k-mer counting, is the first example clearly showing that competitive bioinformatics pipelines based on Hadoop can be built for pervasive tasks, *e.g.*, sequence analysis.

6.1.4 General Remarks

In the era of Big Data only the distributed storage and computing can be used to analyze these data. Nowadays, it is well known that the distributed middleware solutions (*e.g.*, Apache Hadoop) could be adopted to easy develop a distributed version of a sequential algorithm without particular distributed skills. Indeed, frameworks like Hadoop are attractive because they offer the possibility of coding full-fledged distributed applications with very low efforts and, in some cases, by just wrapping up in a proper way some existing applications. Indeed, a novice

developer could use Hadoop to easy transform a stand-alone application in one distributed with little effort using the general and common utilities of Hadoop.

Although these middleware solutions can facilitate a "*painless transformation*" of an sequential implementation into one distributed, without any engineering and profiling activities, some deployed solutions are not entirely fast and efficient. In fact, in this dissertation we have seen that some Hadoop-based naive implementations may suffer from some problems, as shown in Chapters 4 and 5. We have also evaluated that some Hadoop-based applications on textbooks (*e.g.*, those based on *word counts*) may have many problems, such as: they are not resource-frugal; they are very simple (few lines of codes which delegate to standard and general Hadoop facilities); they are very general. In fact, also a novice programmer can easy use and understanding these implementations.

Although MapReduce and Hadoop are very powerful computational paradigms and programming environments, respectively, to exploit parallelism, relying on the top level functions and macros offered by Hadoop does not allow to take full advantage of the computational power of the hardware resources available. For example, a solution must be entirely conceived in a distributed way, it must ensure data locality and it must be based on the low-level I/O primitives that Hadoop offers. In fact, a simple-minded use of such a powerful tool is not enough to "cash in" all the advantages of distributed architectures. The easiness of use implies a cost, that is the resulting implementations may not be able to fully exploit the potential of a distributed system and they can have pitfalls.

Hadoop hides most of technicalities and transparently addresses some issues of the distributed systems. In fact, this framework very well does many distributed activities, such as: tasks scheduling, fault tolerance management, distributed file system management, coordinations, data input partitioning, data local computing, and so on. However, engineering activities and improvements in records input readers, in map and/or reduce functions, and between map and reduce phases, must be explicitly done by the developer.

In some case, the initial distributed solution obtained from simple mapping (without understanding the internal details of Hadoop) might suffice, but this is not true for many problems and solutions. A deep knowledge of Hadoop could impact positively on the development of an improved distributed application.

We cannot afford to write inefficient code in map/reduce functions, just because we have multiple computers together, and, therefore, we can take the luxury of wasting resources.

Therefore, from our experience, we have seen as the Hadoop facilities are targeted to the general case, and according the problem to solve, accurate engineering, tuning, profiling activities can improve greatly the performance with respect to a naive MapReduce solution. In these cases, an engineering methodology based on the implementation of smart improvements driven by a careful profiling activity, like in our dissertation, may lead to a much better experimental performance. In our experience we have seen as using algorithm engineering can be reduced disk I/O activities and network traffic, maximized the CPU usage, reduced the intermediate-data or improved the data partitioning strategy.

It is easy to write distributed applications with Hadoop, but it is very difficult to make them efficient, fast and scalable without a deep knowledge of the framework itself, of distributed computing and hardware available. On the one side, the *implicit* distributed programming can reduce the implementation work. On the other side, a developer must evaluate how much performance loss exists. An accurate design and engineering phase in Hadoop is vital to obtain fast and scalable solutions. It is not enough to be a simple expert of the application domain (*e.g.*, bioinformatics) to build programs on Hadoop, a programmer must also be an expert of distributed systems, hardware, algorithm engineering, etc. (*e.g.*, [212]). In fact, in the era of implicit parallelism is yet required the expert of distributed environments and algorithm engineering, which must have new skills compared to the past. Therefore, the domain expert must be accompanied by a person with these capabilities, or he must acquire these skills.

6.2 Future Directions

Our distributed implementation of algorithm by *Fridrich et al.*, proposed in Chapter 4, could be experimented on a very large number of images exploiting a bigger cluster. In addition, it is also required to analyze techniques which reduce the dimension in bytes of a Reference Pattern without a significant loss of information

(*e.g.*, [29]). In this way, it will be possible to efficiently manage more fingerprints. Then, it is also suggested to exploit the data locality of the Reference Patterns to compute the correlation values for each image. In addition, other massive search methods for matching fingerprints could also be evaluated. In fact, more advanced research techniques such as *composite-based fingerprint search* or *short digest fingerprint search* could be used, instead of the standard linear search. Clustering the images in a dataset according the corresponding fingerprint could also be considered.

It is also possible to use the proposed improvements in Chapter 4 to develop efficient Hadoop-based variants of algorithms belonging to different digital image forensics domains. The image forgery detection algorithms, for example, are based on the analysis of the PNU noise, like those discussed in [51, 61, 111]. These algorithms use a notion of camera fingerprint and an execution pattern similar to the one introduced by the *Fridrich et al.*, but for a different purpose. Here, the absence of the camera fingerprint in a region of the image under scrutiny, taken by that camera, is used as a clue to determine whether the image has been forged or not.

We also remark that our findings for SCI can be also helpful to develop efficient Hadoop-based solutions for completely different application domains. We may consider the case of astronomical observations, where a set of exposures portraying a same region of the sky and taken in different moments of time, are "stacked" and then combined to produce a single high-quality image that, in turn, will be used to assemble a mosaic of images portraying a wider region of the sky (see, *e.g.*, [183]). In this case, we have both a huge amount of data to process and an execution pattern that resembles the one used by the *Fridrich et al.* algorithm.

The improvements achieved by `KCH` (see Section 5.4) could be also adopted in the framework `HAFS` (presented in Section 5.3) during the indexing phase. In fact, `KCH` can be used as a pipeline for other bioinformatics applications. In addition, `HAFS` could use more efficient and fast solutions for computation of dissimilarity measures. A Hadoop-based toolbox for bioinformatics applications could be arranged starting from `KCH` and `HAFS`, extending our work to other bioinformatics problems.

Some current experimentations of the algorithms for alignment-free sequence

comparison use still "toy sequences" (*i.e.*, few sequences, each of few KB or MB), so these methods must be experimental evaluated on very large real sequences to assess the real performance of these algorithms. The next step could be to exploit the framework `HAFS` to speed up large-scale experimentations on real sequences adopting large clusters (*e.g.*, Amazon EC2 or Microsoft Azure). In addition, `HAFS` could also be adapted and applied to metagenomics studies (*e.g.*, [243, 246, 268]).

It could be interesting to use different distributed middleware solutions to solve the problems studied in this dissertation. In fact, as future direction, `HAFS` and `KCH` could be transformed to use Apache Spark ([17, 304]) on the top of Hadoop. For example, in-memory operations in Spark could speed up the pairwise comparisons in `HAFS`.

As we have seen, in the Big Data era, only the distributed computing can be used to manage and process large amounts of data. In any case, Hadoop and MapReduce are not always the cure for all problems in Big Data era. In fact, there could be other distributed middleware solutions (also customized for a specific problem) to speed up a computation in Big Data world, and the same middleware on different architectural clusters could have different performance. Indeed, *Appuswamy et al.* in [21] said that map-intensive Hadoop jobs to do relatively well for scale out, and shuffle-intensive jobs to do well on scale up. They have find that scale out works better for CPU-intensive tasks since there are more cores and more aggregate memory bandwidth. Instead, scale up works better for shuffle-intensive tasks since it has fast intermediate storage and no network bottleneck. An evaluation on such topics could use the Hadoop applications proposed in this dissertation.

In addition, as we seen in Chapter 3, some solutions have been proposed to improve Hadoop framework. In fact, Hadoop is a vital and evolving project, therefore, some improvements experimented in this dissertation, such as the in-mapper local aggregation proposed in Chapters 4 and 5, could be incorporated in Hadoop in the future.

Finally, as an important future direction for this thesis, we believe that our activities, such as algorithm engineering, tuning, profiling, experimental methodology and analyses could be adopted in other different case studies with respect to digital image forensics and bioinformatics.

Appendices

Appendix A

Bioinformatics

In this Appendix additional information about the Chapter 5 is gathered. In particular, Section A.1 presents some information about FASTA file format. Section A.2 describes the state of the art on algorithms collecting k-mer statistics, while in Section A.3 is illustrated the comparison between our Hadoop-based solution for k-mer statistics, *i.e.*, `KCH`, and other solutions for Cumulative Statistics (CS).

A.1 FASTA File Format

FASTA format is a textual file type for representing either nucleotide sequences or peptide sequences, in which nucleotides or amino acids are depicted in the standard IUB/IUPAC amino acid and nucleic acid codes. The FASTA format also allows to use sequence names and other comments to precede the sequences. In fact, a sequence in FASTA format begins with a single-line description, followed by one or more lines of sequence characters. The description line is distinguished from the sequence data by a greater-than (">") symbol in the first position of the line. Other optional characters following this symbol are the identifier of the sequence and its possible description. A sequence ends when another description line is starting. It is recommended that all lines of file should be shorter than 80 characters.

```
>SEQUENCE_1
TATAACCTCTTTGGTCCATACACGGTGATGCTGTACCTCACATGTTCCTCAAGCCAACAATGTTCCATTCCTCCTTACAT
AGTAATTTTCATTCTATTACATTCCATTCCATTCGAGTACATTGCATTCCGTTCCATTCCATTGCATTCCATTCGAATCC
TGTAAAGGTGTTTCTCCAAGGCTGGCTGCTGGCTGGAGACAGAAAACTTTTTTGTTTTAAGNTTTTTAGCAAACTCCTTC
GGTTCTGAGTGTTTTGAACGAAGACCTTGGAAGCTCTGGGTGGGATTCAGAAATGCTATACNCAGAGCATTCAGATGAGA
>SEQUENCE_2
TCCAGGTNTGTTTCTAACTCCTGCATTACTGATAATATGAAACCTAAACATATAAGCAAAACATATACTACATTCAAGGC
GTGGCTGGTATTGCCCTGTTATCAGGATTTCCTCTGGAGTTTCGGTGTGTGTTTGTTTGATGGCATCTTCGAGGACCCCT
TTTACTAAGTTTGCTTTCAAAATGGTCTCATAATTGTAATAGTACATAGAGGTAACATTATCATGCAGCTTTGGTCCATT
TTCTAAGATAGCTTTAGATTGGTCTTAAATTGAATTTCAAATTATAAAATGATGACCGTAAGTCACTCATGAATTTCTTT
TAGCATTAGGAGATATACCTAATGTAAATGATGAGTTAATGTGTGCAGCACACCAATATGGCACATGTATACATATGTAA
```

Figure A.1: An example of FASTA file with two multi-lines sequences.

An example of a FASTA file with multi-lines sequences is illustrated in Figure A.1.

A.2 State of the Art on Algorithms Collecting K-mer Statistics

In this section we review the state of the art on algorithms collecting k-mer statistics.

In the biological sciences, there are two basic versions of the general problem of collecting k-mer statistics, as already discussed in Section 5.2.2, *i.e.*, Local Statistics (LS) and Cumulative Statistics (CS). The Local Statistics are very useful in genomic and proteomic studies since such a statistics can be used to infer information about function, structure and evolution of biological sequences (*e.g.*, [44, 118, 119, 137, 161, 169]). In addition, LS is also rapidly assuming a central role in epigenomic studies (*e.g.*, [117, 222]). On the other side, Cumulative Statistics are essential for *de novo* genome assembly, certainly with the use of *Sanger* sequencing technology (*e.g.*, [148, 203]), and even more so with the use of most of the up-to-date technologies in which assembly seems to be intimately related to the construction of suitable de Bruijn graphs [71].

From the algorithmic point of view, both LS and CS can hardly be considered as a difficult problem. In addition, the amount of sequence data being produced

nowadays makes both tasks challenging, both in terms of time and space. Due to the pressing needs of the novel sequencing technologies, there has been a considerable activity regarding mainly efficient algorithms for CS, with LS receiving virtually no attention. In general, for extracting LS, one could independently run a CS tool for each sequence. However, this solution could not benefit of the all advantages of the algorithm and it could require an input file for each sequence.

The corresponding algorithms and software systems that have been developed, for the purposes of this research are best classified as follows:

- The ones that compute CS exactly and that take advantage of computer architectures to be resource-efficient. They can be further subdivided according to the architectural features that they exploit.
  - *Sequential systems* or (*stand-alone/non-parallel systems*), *e.g.*, Meryl [200] and Tallymer [161].
  - *Multi-threading shared-memory systems*, *e.g.*, DSK [231], Jellyfish [184], KMC [83, 84], MSPKmerCounter [175]. Here many solutions are *disk-based*, *i.e.*, they either fall completely in the *External Memory* paradigm ([2, 282]) for the design of algorithms or use essential aspects of it.
  - *Distributed systems*, *e.g.*, BioPig (k-mer counting module) [37, 209].
- The ones that *estimate* k-mer statistics, possibly filtering out rare k-mers. That is, k-mers that occur only once or a few times are typically not counted, and in all other cases the count is a good approximation of the real value. Here the resource-efficiency is achieved by using sophisticated algorithmic techniques, some of which have been devised recently to deal with Big Data (see [26]). This category includes: BFCounter [194], Khmer [305], KmerGenie [64], KmerStream [193] and Turtle [236]. These tools calculate approximate statistics.

Algorithms for exacts statistics have a much broader use with respect to the ones for approximate statistics, since those latter seem to be specifically dedicated to assembly problems. For this reason, algorithms and software in the first

category are the most relevant for this research. Therefore, in what follows, we adhere as much as possible to the common features of the experimental methodology already used in the mentioned systems.

Table A.1 presents the main aspects of each of the algorithms available in the literature for k-mer statistics, while Table A.2 presents a synopsis of the main experiments that have been performed to assess the competitiveness of each of the mentioned tools. A selection of algorithms for exact cumulative statistics is presented in Section A.2.1.

Table A.1: Summary of the main features of each of the algorithms designed for the collection of k-mer statistics. In view of the classification given in Section A.2, the first two columns of the table are self-explanatory. The column "Range of K" indicates the operating range of algorithms with respect to k-mer length. Two types of input come-up in applications: the first consisting of a collection of sequences each up to a given length (called short sequences, *e.g.*, read size) and the second consisting of collection of sequences of arbitrary length, including reads. The last column indicates the type of statistics that the algorithm computes, while column 4 indicates whether it is exact or approximate.

Algorithm	**Class**	**Range of K**	**Exact and/or Approximate Statistics**	**Input Type**	**Statistics Type**
KMC (*Deorowicz et al.* [83, 84])	Multi-threading and disk-based	$k \leq 256$	Exact	Short sequences, *e.g.*, reads	CS
KAnalyze (*Audano and Vannberg* [24])	Multi-threading and disk-based	$k \leq 31$	Exact	Sequences of arbitrary length, including reads	CS
Jellyfish (*Marçais and Kingsford* [184])	Multi-threading and disk-based	$k > 1$	Exact and approximate	Sequences of arbitrary length, including reads	CS
DSK (*Rizk et al.* [231])	Multi-threading and disk-based	$k \leq 31$	Exact	Sequences of arbitrary length, including reads	CS

Table A.1: *Continued.* Summary of the main features of each of the algorithms designed for the collection of k-mer statistics. In view of the classification given in Section A.2, the first two columns of the table are self-explanatory. The column "Range of K" indicates the operating range of algorithms with respect to k-mer length. Two types of input come-up in applications: the first consisting of a collection of sequences each up to a given length (called short sequences, *e.g.*, read size) and the second consisting of collection of sequences of arbitrary length, including reads. The last column indicates the type of statistics that the algorithm computes, while column 4 indicates whether it is exact or approximate.

Algorithm	**Class**	**Range of K**	**Exact and/or Approximate Statistics**	**Input Type**	**Statistics Type**
BioPig - k-mer Counting (*Nordberg et al.* [209])	Distributed System	$k \geq 1$	Exact	Short sequences, *e.g.*, reads	CS
MSPKC (*Li and Yan* [175])	Multi-threading and disk-based	$k \leq 64$	Exact	Short sequences, *e.g.*, reads	CS
BFCounter (*Melsted and Pritchard* [194])	Multi-threading	$k \leq 31$	Exact and approximate	Short sequences, *e.g.*, reads	CS
Tallymer (*Kurtz et al.* [161])	Sequential	$k \leq 4,961$	Exact	Sequences of arbitrary length, including reads	CS
Meryl (*Miller et al.* [200])	Multi-threading	$k \leq 32$	Exact	Sequences of arbitrary length, including reads	CS
Turtle (*Roy et al.* [236])	Multi-threading	$k \leq 64$	Approximate	Short sequences, *e.g.*, reads	CS
Khmer (*Zhang et al.* [305])	Sequential and multi-threading	$k \leq 255$	Approximate	Short sequences, *e.g.*, reads	CS

Table A.1: *Continued.* Summary of the main features of each of the algorithms designed for the collection of k-mer statistics. In view of the classification given in Section A.2, the first two columns of the table are self-explanatory. The column "Range of K" indicates the operating range of algorithms with respect to k-mer length. Two types of input come-up in applications: the first consisting of a collection of sequences each up to a given length (called short sequences, *e.g.*, read size) and the second consisting of collection of sequences of arbitrary length, including reads. The last column indicates the type of statistics that the algorithm computes, while column 4 indicates whether it is exact or approximate.

Algorithm	**Class**	**Range of K**	**Exact and/or Approximate Statistics**	**Input Type**	**Statistics Type**
KmerGenie (*Chikhi and Medvedev* [64])	Multi-threading	$k \leq 121$	Approximate	Short sequences, *e.g.*, reads	CS
KmerStream (*Melsted and Halldórsson* [193])	Sequential	$k \geq 1$	Approximate	Short sequences, *e.g.*, reads	CS

Table A.2: A synopsis of the experimental setup used to evaluate the algorithms listed in Table A.1. The new columns, with respect to that table, indicate the datasets used in the experimentation, the relevant parameters for the main experiments, the algorithms used for comparison and the top performers, respectively.

Algorithm	**Datasets**	**Main Experiments**	**Comparison with**	**Top Algorithms**
Tallymer (*Kurtz et al.* [161])	Whole genome shotgun sequences from maize (B73) chromosome 8 (total size 10^9 bp)	$k = 20$		Tallymer
Jellyfish 1 (*Marçais and Kingsford* [184])	M.gallopavo from Turkey genome (≈ 24 GB), Homo sapiens (3 GB), Drosophila ananassae (3.5 GB), Coxiella burnetii (35.6 GB), Zea mays (33 GB)	$k = 22$; $k = 5$, 10, 15, 20, 25, 30	Tallymer (serial) 1.3.4, meryl 5.4, meryl 6.1	Jellyfish 1
BFCounter (*Melsted and Pritchard* [194])	Human genomic DNA of 7.5 M 100 bp from NA19240; genome-wide sequence data from the 1000 Genomes Project Pilot II	$k = 25$, 31	Jellyfish 1 and naive k-mer counting	Jellyfish 1 and BFCounter

Table A.2: *Continued.* A synopsis of the experimental setup used to evaluate the algorithms listed in Table A.1. The new columns, with respect to that table, indicate the datasets used in the experimentation, the relevant parameters for the main experiments, the algorithms used for comparison and the top performers, respectively.

Algorithm	**Datasets**	**Main Experiments**	**Comparison with**	**Top Algorithms**
DSK (*Rizk et al.* [231])	Human genome Illumia dataset NA18507 (SRX016231); Escherichia coli ($\approx$ 20.8 million of reads of average length 36 bp) and Drosophila ananassae ($\approx$ 9.1 million of reads of average length 150 bp) datasets	$k = 27$ for comparisons with other algorithms on the NA18507 dataset; $k = 21$ for E.coli DNA and Drosophila RNA	BFCounter 0.2, Jellyfish 1.1.5	DSK-SSD, Jellyfish 1
KmerGenie (*Chikhi and Medvedev* [64])	Staphylococcus aureus (size 2.8 MB), human chromosome 14 (size 88 MB) and Bombus impatiens (size 250 MB)	$k = 21, 31, 41, 51, 61, 71, 81$	DSK	KmerGenie
KMC 1 (*Deorowicz et al.* [83])	Homo sapiens NA19238 (353 GB) and HG02057 (208 GB); Caenorhabditis elegans (16.4 GB)	NA19238 with $k = 22, 25, 28, 31$. HG02057 and Caenorhabditis elegans with $k = 22, 28, 40, 55$	Jellyfish, BFCounter, DSK	KMC 1, Jellyfish, DSK
BioPig - pigKmer (*Nordberg et al.* [209])	Cow rumen Metagenomic data (100 MB- 500 GB)	$k = 20$	Tallymer (serial), Kmernator (MPI-version)	Kmernator, BioPig

Table A.2: *Continued.* A synopsis of the experimental setup used to evaluate the algorithms listed in Table A.1. The new columns, with respect to that table, indicate the datasets used in the experimentation, the relevant parameters for the main experiments, the algorithms used for comparison and the top performers, respectively.

Algorithm	**Datasets**	**Main Experiments**	**Comparison with**	**Top Algorithms**
KAnalyze (*Audano and Vannberg* [24])	Human chromosome 1 (hg19 Chr1, 249 MB) and NA18580 (1.5 million sequence reads; 453 MB). HG01889 ($\approx$ 72 GB with about a million of reads)	$k = 31$	Jellyfish 1.1.10, DSK 1.5280, custom Perl script	KAnalyze, Jellyfish 1
Turtle (*Roy et al.* [236])	D. Melanogaster (size 122 Mbp), G. Gallus (size 1×10^3 Mbp), Z. Mays (size 2.9×10^3 Mbp), H. Sapiens (size 3.3×10^3 Mbp)	$k = 31$, 48, 64	Jellyfish, DSK, BFCounter	cTurtle
Khmer (*Zhang et al.* [305])	5 soil metagenomic read datasets: 1.90 GB, 2.17 GB, 3.14 GB, 4.05 GB, 5.00 GB (entire dataset)	$k = 22$	Tallymer 1.3.4, Jellyfish 1, BFCounter 1.0, DSK 1.5031, KMC 0.3, Turtle 0.3, KAnalyze	KMC, Turtle, Jellyfish 1
KmerStream (*Melsted and Halldórsson* [193])	Dataset of 2,656 whole genome sequenced individuals using Illumina HiSeq sequencers. Homo Sapiens chr 14 (36 M reads) and B. Impatiens (303 M reads).	$k = 21$, 31	KmerGenie	KmerStream

Table A.2: *Continued.* A synopsis of the experimental setup used to evaluate the algorithms listed in Table A.1. The new columns, with respect to that table, indicate the datasets used in the experimentation, the relevant parameters for the main experiments, the algorithms used for comparison and the top performers, respectively.

Algorithm	**Datasets**	**Main Experiments**	**Comparison with**	**Top Algorithms**
MSPKC (*Li and Yan* [175])	Short-reads datasets (bird 107 GB, snake 182 GB, fish 137 GB, soybean 40 GB)	Experiments on a single thread with $k = 31$ and various levels of coverage. Preliminary tests on multi-threading version called MSPKmer-Counter(MT)	Jellyfish 1 (only memory, also disk), BFCounter	Jellyfish 1 (RAM only), MSPKC, Jellyfish 1
KMC2 (*Deorowicz et al.* [84])	5 datasets from 10 to 313 GB	$k = 28, 55$	Jellyfish 2, KAnalyze, MSPKmerCounter, Turtle, DSK, KMC 1.	KMC2, Jellyfish 2, DSK

A.2.1 Algorithms for Exact Cumulative Statistics

Among the algorithms designed for CS, we have included in this study only the ones that provide exact statistics. We have chosen the most representative according to the literature, for each type of computer architecture we are interested in. Moreover, we include the latest release of each algorithm since writing of this dissertation.

The algorithms so selected are as follows. First, the most representative of the algorithms that are disk-based and that work in a shared-memory environment using multi-threading: *KAnalyze* [24], *Jellyfish2*, an evolution of Jellyfish [184], *KMC2* [84] and *DSK* [231]. Then, we selected *BioPig* (k-mer counting module) [37, 209] as a representative of algorithms supported by distributed architectures. It is to be pointed out that all these algorithms work by skipping all the k-mers

of an input sequence containing at least one N character. We also observe that BioPig and KMC2 have been designed to work on short sequences only. This could imply that may exhibit very bad performances when processing long sequences or fail at all to work.

A careful profiling of some of the most successful methods that have been developed for CS is also presented in Section A.3. From these analyses, it is evident that they do not scale well with computational resources.

In the next subsections we describe some popular tools used to exactly count the k-mer frequencies.

A.2.1.1 Tallymer

The authors in [161] present *Tallymer*, a flexible and memory-efficient collection of programs for k-mer counting, indexing, and searching of large sequence sets. Tallymer is part of the *GenomeTools* [114] software. The authors have employed enhanced suffix arrays to compute the counts and construct the k-mer frequency index from which they can efficiently retrieve the counter of each k-mer. However, Tallymer does not support multi-threading.

A.2.1.2 Meryl

Meryl [195] comes from the k-mer tool of the Celera assembler [200], and it is a multi-threaded and multi-process k-mer counter. This tool is capable of generating the k-mer counting table and of performing simple operations on multiple tables (*e.g.*, adding counts, subtracting counts, logical operations) along with reporting statistics on individual tables (*e.g.*, histograms).

A.2.1.3 Jellyfish

Marçais and Kingsford in [184] have proposed a k-mer counting algorithm called *Jellyfish* (version 1.x, *i.e.*, v1.x), which is fast and memory efficient. It is based on a multi-threaded, lock-free hash table for counting k-mers up to 31 bases in length (only v1.x). Jellyfish stores k-mer counts in memory hash table, and makes

use of disk storage to scale to larger datasets. It uses several lock-free data structures that exploit a widely available hardware operation called *compare-and-swap* (CAS) to implement efficient shared access to the data structures. In particular, Jellyfish uses lock-free queues for communication between worker threads and a lock-free hash table to store the k-mer frequencies. Jellyfish is also very memory efficient, in fact, it uses an reduced memory usage for a hash table entry and an space-efficient encoding of keys. In particular, it implements a key compression scheme that allows it to use a constant amount of memory per key in the hash table, regardless of the length k of the k-mers counted. It also uses a bit-packed data structure to reduce wasted memory due to memory alignment requirements. The tool stores only a part (prefix) of the k-mer in the hash table, since its suffix can be deduced from the hash position.

When the hash table is full, it could be written to disk as a list of key-value records instead of doubling its size in memory. This situation occurs when there is not enough memory to carry out the entire computation and, therefore, intermediary results are saved to disk. Sorting the output has the advantage that the results can be queried quickly using a binary search, and two or more hash tables to be merged into one easily. The user could decide if use disk operation, instead of do size doubling. In particular, Jellyfish will detect when a hash table needs to expand beyond the available memory and will, instead, write the current k-mer counts to disk, clear the hash table and begin counting afresh.

The final phase is the writing, where the results are sorted and written to disk. In this phase, the operations are bounded by I/O bandwidth.

The second version (v2.x) of Jellyfish does not have any limitation on the size of k-mers, unlike version 1.x which was limited to $k \leq 31$. This version also offers two way to count only high-frequency k-mers (meaning only k-mers with count > 1), which reduces significantly the memory usage. Both methods are based on using Bloom filters [39]. The first method is a one pass approach, which provides approximate count for some percentage of the k-mers. The second method is a two pass approach which provides exact count. In both methods, most of the low-frequency k-mers are not reported.

A.2.1.4 KAnalyze

Audano and Vannberg in [24] have presented *KAnalyze*, a Java program that counts k-mers, and that can process large datasets with 2 GB of memory. The counting phase takes place in two steps over two components, *i.e.*, split component and merge component. The split component writes sorted subsets of data to disk, and the merge component accumulates counts from each subset. Split and merge operations can be performed in multiple steps and, therefore, exploiting multi-threading.

In particular, the split component reads k-mers into a memory array until it is full. Then the array is sorted, and k-mers are counted by traversing the sorted array. Each k-mer and its count are written to disk. The memory array is then filled with the next set of k-mers, and a new file of k-mer counts is created. The process repeats until all k-mers have been written. Finally, the merge component reads k-mers and their counts from each file, and sums the counts for each k-mer.

A.2.1.5 MSPKmerCounter

Li and Yan in [175] have described *MSPKmerCounter* (MSPKC), a disk-based approach, to efficiently perform k-mer counting for large genomes using a small amount of memory. The approach is based on *Minimum Substring Partitioning* (MSP) that breaks short reads into multiple disjoint partitions such that each partition can be loaded into memory and processed individually. By leveraging the overlaps among the k-mers derived from the same short read, MSP can achieve big compression ratio so that the I/O cost can be significantly reduced.

A.2.1.6 KMC

Preliminary Version of KMC In [83] *Deorowicz et al.* have proposed *K-mer Counter* (KMC), a simple, efficient, parallel disk-based algorithm for k-mer counting. The basic idea is to obtain a compact on-disk dictionary structure with k-mers as keys and their counts as values. The structure can then be read sequentially, or individual k-mers (with their associated counts) can be found using the standard binary search technique. The proposed technique follows the

disk-based distribution sort paradigm. In the first phase, called distribution, the reads are scanned one by one, all the k-mers are extracted from each and sent each to one of multiple disk files based on the k-mer prefix of length p_1. The first phase starts with storing the data in buffers in the main memory where another prefix part, of length p_2, is removed from each k-mer, and the prefix counts are maintained for further recovery. Once the buffer reaches the predefined capacity, its content is sent to a file. The k-mers scattered over hundreds of files are the outcome of the distribution phase. Each file corresponds to a unique prefix of length p_1. In each file, the k-mers are also grouped by their successive p_2 symbols. Removing the prefixes reduces the disk usage depending on the value of k. The sorting phase collects the data from disk in the order of lexicographically sorted prefixes of length p_1, it recovers the p_2-symbol long prefixes, then it sorts the k-mers, it counts their frequencies (after sorting repeating k-mers are at adjacent positions), and (optionally) it removes unique k-mers. These steps are implemented as parallel algorithm using threads. In KMC the space resources are bounded, *i.e.*, the RAM usage is user-selected and the upper bound on the amount of disk space can be approximately estimated from standard input parameters.

KMC2 *Deorowicz et al.* in [84] have presented a new version for KMC that borrows from the efficient architecture of preliminary version of KMC, but reduces the disk usage several times (sometimes about 10 times) and improves the speed usually about twice. The experiments also show that the memory usage of KMC2 is even smaller than its predecessor.

There are two main ideas behind these improvements. The first is the use of signatures of k-mers that allow significant reduction of temporary disk space. These were used for the first time for the k-mer counting in MSPKmerCounter [175], but the modification in KMC2 significantly reduces the main memory requirements and the disk space. The second main novelty is the use of (k, x)-mers for reduction of the amount of data to sort. In particular, instead of sorting some amount of k-mers, the authors sort a much smaller portion of (k, x)-mers and then obtain the statistics for k-mers in the post-processing phase.

Experiments show that it usually offers the fastest solution to the considered problem, while demanding a relatively small amount of memory.

A.2.1.7 DSK

Rizk et al. in [231] have presented *Disk Streaming of K-mers* (DSK), a streaming algorithm for k-mer counting, which only requires a fixed user-defined amount of memory and disk space. The multi-set of all k-mers present in the reads is partitioned, and the partitions are saved to disk. Then, each partition is separately loaded in memory in a temporary hash map. The k-mer counts are returned by traversing each hash table. This tool uses little memory, but its processing time is increased due to more iterations, thus the I/O is increased. As the algorithm relies heavily on I/O to the disk, the authors also use a solid-state drive (DSK-SSD variant). In this configuration, the algorithm is no longer limited by disk I/O and it could benefit from multi-threading. DSK does not provide random access to k-mer counts, but only an arbitrarily small subset of k-mers is loaded in memory at any time.

A.2.1.8 BioPig

In [37] is introduced the *BioPig* sequence analysis toolkit as one of the solutions that scale to data and computation. It is build on Hadoop and Pig dataflow language [213]. Pig is a flexible data scripting language that uses Hadoop data structure and MapReduce framework to process very large data files in parallel and combine the results (see Section 2.4.5 for details about Pig). In particular, BioPig extends Pig with capability of sequence analysis.

Nordberg et al. in [209] have discussed the design principles of BioPig, they give examples on use of this toolkit for specific sequence analysis tasks, and they compare its performance with alternative solutions on different platforms. Using the BioPig modules, they provide a set of scripts that show the functionality provided by the framework. Given a set of sequences, the module *pigKmer* of BioPig computes the frequencies of each k-mer and outputs a histogram of the k-mer counts. The histogram of the counts is generated in a second MapReduce iteration. A number of variations of k-mer counting are available: count only the number of unique reads that contain the k-mers or group k-mers within one or two hamming distance.

The lasted BioPig version includes a query for only k-mer counting (*kmer-*

Count.pig) [150]. It is a simple Hadoop-based version of k-mer counting that presents some bottleneck, as experimented in Section 5.4.3.4.

Other Naive Hadoop-based Solutions *Pahadia et al.* in [217] have proposed a naive Hadoop-based solution for k-mer counting similar to BioPig. In addition, each input file is only read line by line, and, from each line, the substrings of length k are easily extracted. In [216] the authors have used this solution for classification of multi-genomic data with $k \leq 4$. Another analogous algorithm for cumulative statistics is given by *Parsian* in [220].

A.3 Comparison between KCH and Other Solutions for CS

In this section we present some comparison results between our solution, *i.e.*, `KCH`, and some popular tools for CS for multi-threading environments. These tools are experimented on a single slave node (see Section 5.4.3.2), while `KCH` uses the same setting presented in Section 5.4.3.4.

We again recall that some of the considered algorithms provide, as output, the k-mer counts in a file saved using a human-readable format whereas other algorithms return these statistics as a binary file. In this last case, we include in our performance also the time required to convert these binary files in a human-readable format, an operation that can be usually carried out using a support tool provided with the k-mer counting algorithm. Some conversion tools are non-parallel.

The selected programs are:

- KMC 2.3.0 (*KMC2*).
  In KMC2 experiments we have selected our CS datasets using canonical k-mers for $k = 3, 7, 15, 31$ varying the number of threads (4, 8, 16, 32). The version of KMC 2.3.0 also works on $k < 10$. The KMC2 signature length (used to extract the super k-mers) was 7. Other details are presented in Section 5.4.3.4. Figure A.2 reports the results of the CS experiments for

KMC2. For ease of comparison and for the convenience of the reader, we also report the performance of `KCH`.

- DSK 2.0.5 (64 bits) with parallel dump (*DSK*).
  In DSK experiments we have selected our CS datasets using canonical k-mers for $k = 7, 15, 31$ varying the number of threads (4, 8, 16, 32). We have not used $k = 3$, because $k \geq 4$ in DSK 2.0.5. DSK uses a parallel tool for transforming the binary representation of k-mers in textual one. Figure A.3 reports the results of the CS experiments for DSK. For ease of comparison and for the convenience of the reader, we also report the performance of `KCH`.

- Jellyfish 2.2.0 (*JF2*).
  In JF2 experiments we have selected our CS datasets (excluding *CS_128GB*) using canonical k-mers for $k = 3, 7, 15, 31$ varying the number of threads (4, 8, 16, 32). Figure A.4 reports the results of the CS experiments for JF2. For ease of comparison and for the convenience of the reader, we also report the performance of `KCH`. The dataset *CS_128GB* was excluded because the execution time was estimated be very high.

- KAnalyze 0.9.7 (*KA*).
  In KA experiments we have selected our CS datasets (excluding *CS_128GB*) using canonical k-mers for $k = 3, 7, 15, 31$ varying the number of threads (4, 8, 16, 32). Figure A.5 reports the results of the CS experiments for KA. For ease of comparison and for the convenience of the reader, we also report the performance of `KCH`. The dataset *CS_128GB* was excluded because the execution time was estimated be very high.

Notice that each of the considered algorithm has been run using, as parameters, the ones yielding the best performance.

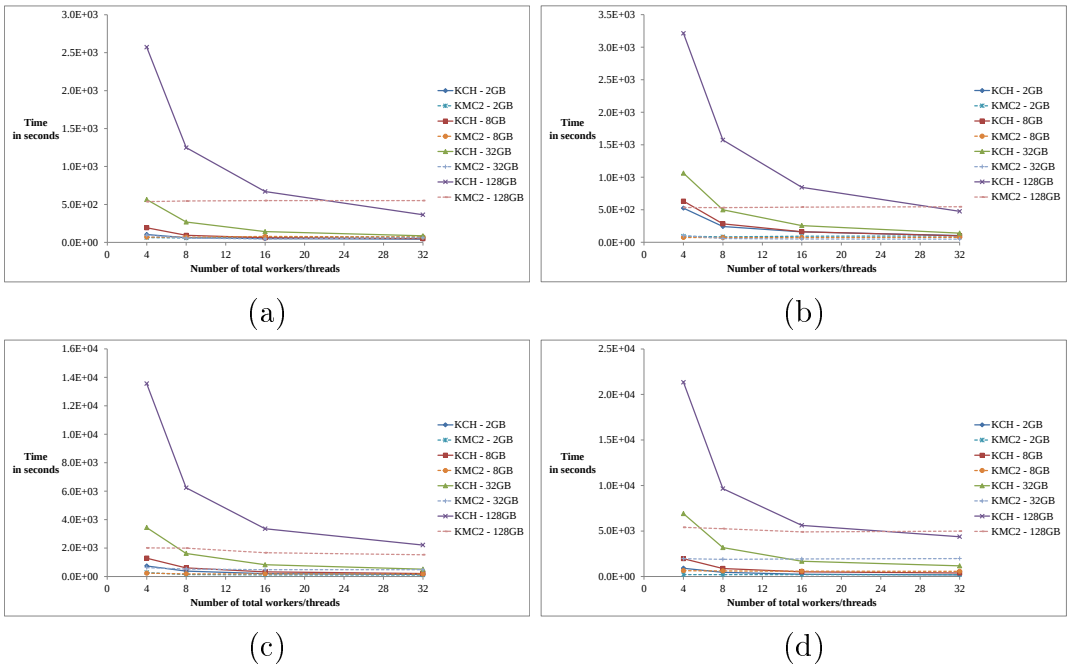

Figure A.2: *Cumulative Statistics.* (a) Scalability of `KCH` with respect to KMC2, for the case $k = 3$, for all datasets, which are indicated in the figure according to the legend to the right. The abscissa gives the number of workers/threads used, while the ordinate gives the corresponding time in seconds. (b)-(d) As in (a), but for $k = 7$, $k = 15$ and $k = 31$, respectively.

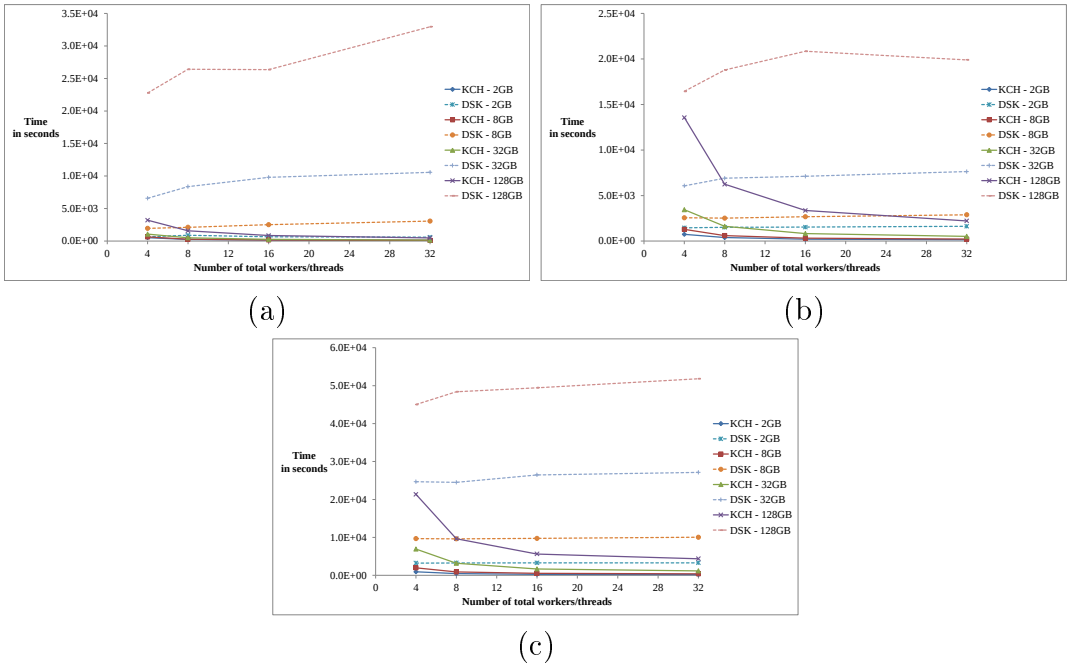

Figure A.3: *Cumulative Statistics.* (a) Scalability of `KCH` with respect to DSK, for the case $k = 7$, for all datasets, which are indicated in the figure according to the legend to the right. The abscissa gives the number of workers/threads used, while the ordinate gives the corresponding time in seconds. (b)-(c) As in (a), but for $k = 15$ and $k = 31$, respectively.

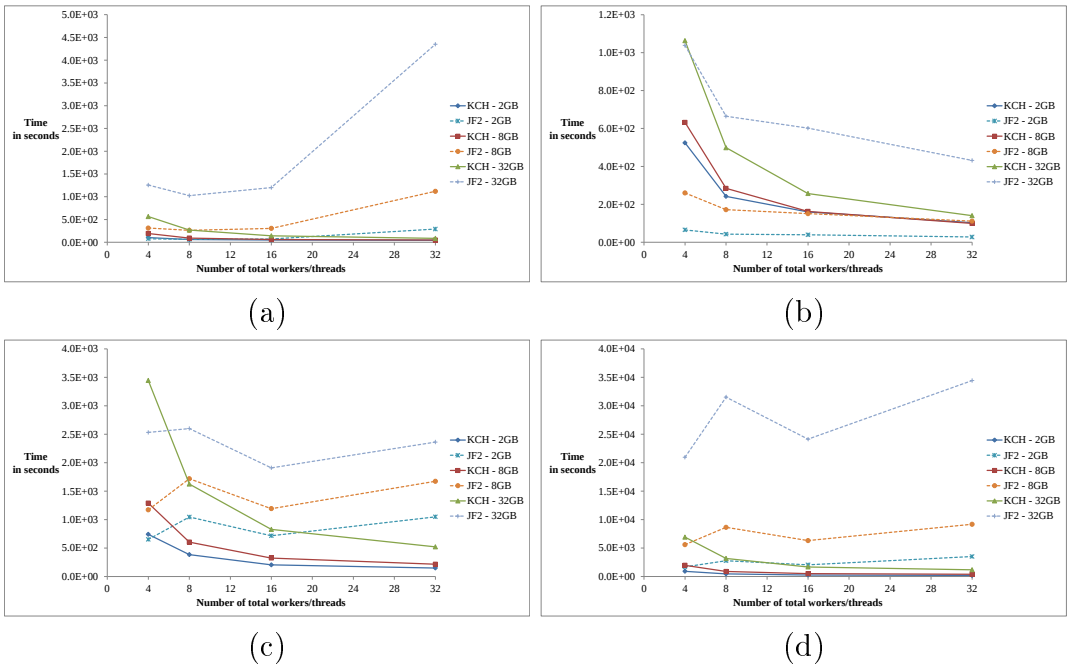

Figure A.4: *Cumulative Statistics.* (a) Scalability of `KCH` with respect to Jellyfish, for the case $k = 3$, for all datasets, which are indicated in the figure according to the legend to the right. The abscissa gives the number of workers/threads used, while the ordinate gives the corresponding time in seconds. (b)-(d) As in (a), but for $k = 7$, $k = 15$ and $k = 31$, respectively.

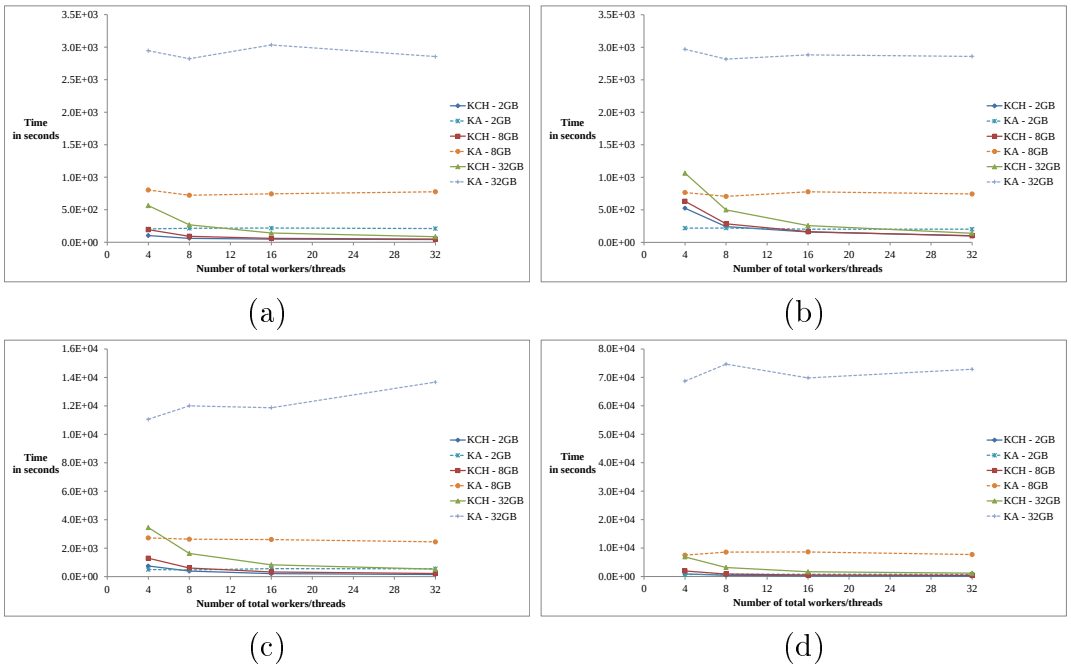

Figure A.5: *Cumulative Statistics.* (a) Scalability of `KCH` with respect to KAnalyze, for the case $k = 3$, for all datasets, which are indicated in the figure according to the legend to the right. The abscissa gives the number of workers/threads used, while the ordinate gives the corresponding time in seconds. (b)-(d) As in (a), but for $k = 7$, $k = 15$ and $k = 31$, respectively.

Appendix B

Publications during the Ph.D.

In Appendix B are listed the publications of Gianluca Roscigno written during the Ph.D. years. Section B.1 presents the list of published papers, while Section B.2 gives the list of submitted or accepted papers (but not yet published). We also include the papers not covered in this thesis.

The updated publications of Gianluca Roscigno are listed at OrcID: `http://orcid.org/0000-0001-6034-150X`.

B.1 Personal Publications

List of papers published during the Ph.D. studies:

- **Nuovi Metodi di Indagine basati su Immagini Digitali e Rumore Caratteristico del Sensore** (Giuseppe Cattaneo, Umberto Ferraro Petrillo, Mario Ianulardo, Gianluca Roscigno), In IISFA Memberbook 2015 DIGITAL FORENSICS: Condivisione della conoscenza tra i membri dell'IISFA ITALIAN CHAPTER, 2015, [50].
- **A PNU-Based Technique to Detect Forged Regions in Digital Images** (Giuseppe Cattaneo, Umberto Ferraro Petrillo, Gianluca Roscigno, Carmine De Fusco), In Advanced Concepts for Intelligent Vision Systems (ACIVS 2015), 2015, DOI: 10.1007/978-3-319-25903-1_42, [51].

- **Alignment-free Sequence Comparison over Hadoop for Computational Biology** (Giuseppe Cattaneo, Umberto Ferraro Petrillo, Raffaele Giancarlo, Gianluca Roscigno), In 44rd International Conference on Parallel Processing Workshops (ICCPW 2015), 2015, DOI: 10.1109/ICPPW.2015.28, [49].

- **Reliable Voice-based Transactions over VoIP Communications** (Giuseppe Cattaneo, Luigi Catuogno, Fabio Petagna, Gianluca Roscigno), In Ninth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS 2015), 2015, DOI: 10.1109/IMIS.2015.20, [47].

- **A Possible Pitfall in the Experimental Analysis of Tampering Detection Algorithms** (Giuseppe Cattaneo, Gianluca Roscigno), In 17th International Conference on Network-Based Information Systems (NBiS), 2014, DOI: 10.1109/NBiS.2014.82, [53].

- **A Scalable Approach to Source Camera Identification over Hadoop** (Giuseppe Cattaneo, Gianluca Roscigno, Umberto Ferraro Petrillo), In IEEE 28th International Conference on Advanced Information Networking and Applications (AINA), 2014, DOI: 10.1109/AINA.2014.47, [55].

- **Experimental Evaluation of an Algorithm for the Detection of Tampered JPEG Images** (Giuseppe Cattaneo, Gianluca Roscigno, Umberto Ferraro Petrillo), In Information and Communication Technology - Proceedings of Second IFIP TC5/8 International Conference (ICT-EurAsia 2014), 2014, DOI: 10.1007/978-3-642-55032-4_66, [54].

B.2 Submitted or Accepted Papers

List of submitted or accepted papers (but not yet published) during the years of Ph.D., in addition papers to be submitted are also reported.

- **Speeding up Alignment-Free Sequence Comparison Algorithms with Hadoop** (Giuseppe Cattaneo, Umberto Ferraro Petrillo, Raffaele Giancarlo, Gianluca Roscigno) - Submitted at *Journal of Supercomputing*.
- **Improving the Experimental Analysis of Tampered Image Detection Algorithms for Biometric Systems** (Giuseppe Cattaneo, Umberto Ferraro Petrillo, Gianluca Roscigno) - *to be submitted*.
- **Achieving Efficient Source Camera Identification on Hadoop** (Giuseppe Cattaneo, Umberto Ferraro Petrillo, Gianluca Roscigno) - Submitted at *Concurrency and Computation: Practice and Experience Journal*.
- **Ensuring Non-repudiability of Human Conversations over VoIP Communications** (Giuseppe Cattaneo, Luigi Catuogno, Fabio Petagna and Gianluca Roscigno) - Accepted at *International Journal of Communication Networks and Distributed Systems* (*in press*).
- **The Design and Engineering of a Fast Hadoop Algorithm for K-mer Statistics** (Umberto Ferraro Petrillo, Gianluca Roscigno, Giuseppe Cattaneo, Raffaele Giancarlo) - *to be submitted*.

References

[1] Thomas Abeel, Yves Van de Peer, and Yvan Saeys. Java-ML: a machine learning library. *Journal of Machine Learning Research*, 10:931–934, 2009.

[2] Alok Aggarwal and Jeffrey S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[3] Frances Allen, G. Almasi, Wanda Andreoni, D. Beece, Bruce J. Berne, A. Bright, Jose Brunheroto, Calin Cascaval, J. Castanos, Paul Coteus, et al. Blue Gene: a vision for protein science using a petaflop supercomputer. *IBM Systems Journal*, 40(2):310–327, 2001.

[4] Jonas S. Almeida, Alexander Grüneberg, Wolfgang Maass, and Susana Vinga. Fractal MapReduce decomposition of sequence alignment. *Algorithms for Molecular Biology*, 7(1):1–12, 2012.

[5] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

[6] Amazon Web Services. Amazon Elastic MapReduce (Amazon EMR). (Available from: `https://aws.amazon.com/elasticmapreduce/`), 2015. [Accessed on 28 December 2015].

[7] Amazon Web Services. Amazon Elastic Compute Cloud (EC2). (Available from: `http://aws.amazon.com/`), 2016. [Accessed on 10 January 2016].

[8] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the Spring Joint Computer Conference, April 18-20, 1967*, pages 483–485. ACM, 1967.

[9] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Wang, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. PACMan: coordinated memory caching for parallel jobs. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 1–14. USENIX Association, 2012.

[10] Greg R. Andrews. *Foundations of parallel and distributed programming.* Addison-Wesley Longman Publishing Co., Inc., 1999.

[11] Ayesha Anwar, K. R. Krish, and Ali R. Butt. On the use of microservers in supporting Hadoop applications. In *IEEE International Conference on Cluster Computing (CLUSTER), 2014*, pages 66–74. IEEE, 2014.

[12] Apache Software Foundation. Hive. (Available from: `https://hive.apache.org/`), 2014. [Accessed on 14 January 2016].

[13] Apache Software Foundation. Apache Hadoop NextGen MapReduce (YARN). (Available from: `http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html`), 2015. [Accessed on 28 December 2015].

[14] Apache Software Foundation. Hadoop. (Available from: `http://hadoop.apache.org/`), July 2015. [Accessed on 15 July 2015].

[15] Apache Software Foundation. HBase. (Available from: `https://hbase.apache.org/`), 2015. [Accessed on 14 January 2016].

[16] Apache Software Foundation. Pig. (Available from: `https://pig.apache.org/`), 2015. [Accessed on 14 January 2016].

[17] Apache Software Foundation. Spark. (Available from: `http://spark.apache.org/`), 2015. [Accessed on 29 December 2015].

[18] Apache Wiki. Hadoop Wiki - PoweredBy. (Available from: `https://wiki.apache.org/hadoop/PoweredBy`), 2016. [Accessed on 29 January 2016].

[19] Alberto Apostolico and Fabio Cunian. The subsequence composition of polypeptides. *Journal of Computational Biology*, 17:1011–1049, 2010.

[20] Alberto Apostolico and Raffaele Giancarlo. Sequence alignment in molecular biology. *Journal of Computational Biology*, 5(2):173–196, 1998.

[21] Raja Appuswamy, Christos Gkantsidis, Dushyanth Narayanan, Orion Hodson, and Antony Rowstron. Scale-up vs scale-out for Hadoop: Time to rethink? In *Proceedings of the 4th annual Symposium on Cloud Computing*, pages 20:1–20:13. ACM, 2013.

[22] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[23] Mikhail J. Atallah. *Algorithms and theory of computation handbook*. CRC press, 1998.

[24] Peter Audano and Fredrik Vannberg. KAnalyze: a fast versatile pipelined k-mer toolkit. *Bioinformatics*, 30(14):2070–2072, 2014.

[25] Yurii S. Aulchenko, Dirk-Jan De Koning, and Chris Haley. Genomewide rapid association using mixed model and regression: a fast and simple method for genomewide pedigree-based quantitative trait loci association analysis. *Genetics*, 177(1):577–585, 2007.

[26] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, pages 1–16, New York, NY, USA, 2002. ACM.

[27] Sebastiano Battiato, Giovanni Maria Farinella, Giovanni Gallo, and Enrico Messina. Naturalness classification of images into DCT domain. In *IS&T/SPIE Electronic Imaging*, volume 7250, pages 1–12. International Society for Optics and Photonics, 2009.

[28] Sevinç Bayram, Hüsrev Taha Sencar, and Nasir Memon. Efficient techniques for sensor fingerprint matching in large image and video databases. In *IS&T/SPIE Electronic Imaging*, volume 7541, pages 1–8. International Society for Optics and Photonics, 2010.

[29] Sevinç Bayram, Hüsrev Taha Sencar, and Nasir Memon. Efficient sensor fingerprint matching through fingerprint binarization. *IEEE Transactions on Information Forensics and Security*, 7(4):1404–1413, 2012.

[30] Sevinç Bayram, Hüsrev Taha Sencar, Nasir Memon, and Ismail Avcibas. Source camera identification based on CFA interpolation. In *IEEE International Conference on Image Processing (ICIP)*, volume 3, pages 69–72. IEEE, 2005.

[31] BBC News Business. Big Data: are you ready for blast-off? (Available from: `http://www.bbc.com/news/business-26383058`), 2014. [Accessed on 4 March 2014].

[32] Gordon Bell, Tony Hey, and Alex Szalay. Beyond the data deluge. *Science*, 323(5919):1297–1298, 2009.

[33] Mordechai Ben-Ari. *Principles of concurrent and distributed programming*. Pearson Education, 2006.

[34] Bonnie Berger, Jian Peng, and Mona Singh. Computational solutions for omics data. *Nature Reviews Genetics*, 14(5):333–346, 2013.

[35] Riza Berkan. Big Data: a blessing and a curse. (Available from: `http://www.searchenginejournal.com/big-data-blessing/53528/`), 2012. [Accessed on 2 December 2015].

[36] Fran Berman, Geoffrey Fox, and Anthony J. G. Hey. *Grid computing: making the global infrastructure a reality*, volume 2. John Wiley & Sons, Inc., 2003.

[37] Karan Bhatia and Zhong Wang. BioPig: developing cloud computing applications for next-generation sequence analysis. Technical report, Ernest Orlando Lawrence Berkeley National Laboratory, Berkeley, CA (US), 2011.

[38] Bioinformatics.org. CS201 High Performance Computing. (Available from: `http://www.bioinformatics.org/wiki/CS201_High_Performance_Computing`), 2013. [Accessed on 17 December 2015].

[39] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[40] Marcus Boden, Martin Schöneich, Sebastian Horwege, Sebastian Lindner, Chris Leimeister, and Burkhard Morgenstern. Alignment-free sequence comparison with spaced k-mers. In *OASIcs-OpenAccess Series in Informatics*, volume 34, pages 24–34. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2013.

[41] Rajkumar Buyya, James Broberg, and Andrzej M. Goscinski. *Cloud computing: principles and paradigms*, volume 87. John Wiley & Sons, Inc., 2010.

[42] Camera and Imaging Products Association. Production, shipment of digital still cameras 2013. (Available from: `http://www.cipa.jp/stats/documents/e/d-2013_e.pdf`), 2013. [Accessed on 16 November 2014].

[43] Camera and Imaging Products Association. Production, shipment of digital still cameras 2014. (Available from: `http://www.cipa.jp/stats/documents/e/d-2014_e.pdf`), 2014. [Accessed on 15 December 2015].

[44] Davide Campagna, Chiara Romualdi, Nicola Vitulo, Micky Del Favero, Matej Lexa, Nicola Cannata, and Giorgio Valle. RAP: a new computer program for de novo identification of repeated sequences in whole genomes. *Bioinformatics*, 21(5):582–588, 2005.

[45] Alberto Castellini, Giuditta Franco, and Vincenzo Manca. A dictionary based informational genome analysis. *BMC Genomics*, 13(1):485, 2012.

[46] Aniello Castiglione, Giuseppe Cattaneo, Maurizio Cembalo, and Umberto Ferraro Petrillo. Experimentations with source camera identification and online social networks. *Journal of Ambient Intelligence and Humanized Computing*, 4(2):265–274, 2013.

[47] Giuseppe Cattaneo, Luigi Catuogno, Fabio Petagna, and Gianluca Roscigno. Reliable voice-based transactions over VoIP communications. In *9th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS 2015)*, pages 101–108, July 2015.

[48] Giuseppe Cattaneo, Pompeo Faruolo, and Umberto Ferraro Petrillo. Experiments on improving sensor pattern noise extraction for source camera identification. In *Sixth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS)*, pages 609–616, July 2012.

[49] Giuseppe Cattaneo, Umberto Ferraro Petrillo, Raffaele Giancarlo, and Gianluca Roscigno. Alignment-free sequence comparison over Hadoop for computational biology. In *44th International Conference on Parallel Processing Workshops (ICCPW 2015)*, pages 184–192. IEEE, September 2015.

[50] Giuseppe Cattaneo, Umberto Ferraro Petrillo, Mario Ianulardo, and Gianluca Roscigno. *IISFA Memberbook 2015 DIGITAL FORENSICS: Condivisione della conoscenza tra i membri dell'IISFA ITALIAN CHAPTER*, chapter XII - Nuovi metodi di indagine basati su immagini digitali e rumore caratteristico del sensore, pages 301–324. IISFA, 2015.

[51] Giuseppe Cattaneo, Umberto Ferraro Petrillo, Gianluca Roscigno, and Carmine De Fusco. A PNU-based technique to detect forged regions in digital images. In *Advanced Concepts for Intelligent Vision Systems (ACIVS 2015)*, pages 486–498. Springer, October 2015.

[52] Giuseppe Cattaneo and Giuseppe Italiano. Algorithm engineering. *ACM Computing Surveys (CSUR)*, 31(3es):582–585, 1999.

[53] Giuseppe Cattaneo and Gianluca Roscigno. A possible pitfall in the experimental analysis of tampering detection algorithms. In *17th International Conference on Network-Based Information Systems (NBiS 2014)*, pages 279–286, September 2014.

[54] Giuseppe Cattaneo, Gianluca Roscigno, and Umberto Ferraro Petrillo. Experimental evaluation of an algorithm for the detection of tampered JPEG images. In *Information and Communication Technology - Proceedings of Second IFIP TC5/8 International Conference, ICT-EurAsia 2014, Bali, Indonesia, April 14-17, 2014*, volume 8407, pages 643–652. Springer, April 2014.

[55] Giuseppe Cattaneo, Gianluca Roscigno, and Umberto Ferraro Petrillo. A scalable approach to source camera identification over Hadoop. In *IEEE 28th International Conference on Advanced Information Networking and Applications (AINA 2014)*, pages 366–373. IEEE, May 2014.

[56] Cheong Xin Chan, Guillaume Bernard, Olivier Poirion, James M. Hogan, and Mark A. Ragan. Inferring phylogenies of evolving sequences without multiple sequence alignment. *Scientific Reports*, 4(6504), 2014.

[57] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: a library for support vector machines. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2(3):1–27, 2011. (Available from: `http://www.csie.ntu.edu.tw/~cjlin/libsvm`).

[58] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[59] Abbas Cheddad, Joan Condell, Kevin Curran, and Paul Mc Kevitt. Digital image steganography: survey and analysis of current methods. *Signal processing*, 90(3):727–752, 2010.

[60] Mo Chen, Jessica Fridrich, Miroslav Goljan, and Jan Lukáš. Determining image origin and integrity using sensor noise. *IEEE Transactions on Information Forensics and Security*, 3(1):74–90, 2008.

[61] Mo Chen, Jessica Fridrich, Jan Lukáš, and Miroslav Goljan. Imaging sensor noise as digital X-ray for revealing forgeries. In *Information Hiding*, pages 342–358. Springer, 2007.

[62] Yanpei Chen, Sara Alspaugh, and Randy Katz. Interactive analytical processing in Big Data systems: A cross-industry study of MapReduce workloads. *Proceedings of the VLDB Endowment*, 5(12):1802–1813, 2012.

[63] Jinkui Cheng, Fuliang Cao, and Zhihua Liu. AGP: a multi-methods web server for alignment-free genome phylogeny. *Molecular Biology and Evolution*, pages 1–6, 2013.

[64] Rayan Chikhi and Paul Medvedev. Informed and automated k-mer size selection for genome assembly. *Bioinformatics*, 30(1):31–37, 2013.

[65] Junho Choi, Chang Choi, Byeongkyu Ko, Dongjin Choi, and Pankoo Kim. Detecting web based DDoS attack using MapReduce operations in cloud computing environment. *Journal of Internet Services and Information Security (JISIS)*, 3(3/4):28–37, November 2013.

[66] Benny Chor, David Horn, Nick Goldman, Yaron Levy, Tim Massingham, et al. Genomic DNA k-mer spectra: models and modalities. *Genome Biology*, 10(10):1–10, 2009.

[67] Rudi Cilibrasi and Paul M. B. Vitányi. Clustering by compression. *IEEE Transactions on Information Theory*, 51(4):1523–1545, 2005.

[68] Cisco. Cisco Visual Networking Index: global mobile data traffic forecast update 2014–2019. (Available from: `http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white_paper_c11-520862.html`), 2015. [Accessed on 2 January 2016].

[69] Cloudera. Data helps solve the world's biggest problems. (Available from: `http://www.cloudera.com`), 2016. [Accessed on 10 January 2016].

[70] Matteo Comin and Davide Verzotto. Beyond fixed-resolution alignment-free measures for mammalian enhancers sequence comparison. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 11:628–637, 2014.

[71] Phillip E. C. Compeau, Pavel A. Pevzner, and Glenn Tesler. How to apply de Bruijn graphs to genome assembly. *Nature Biotechnology*, 29(11):987–991, 2011.

[72] Javier Conejero, Pete Burnap, Omer Rana, and Jeffrey Morgan. Scaling archived social media data analysis using a Hadoop cloud. In *IEEE Sixth International Conference on Cloud Computing (CLOUD), 2013*, pages 685–692. IEEE, 2013.

[73] Contrail Project. Contrail: Assembly of large genomes using cloud computing. (Available from: `http://sourceforge.net/projects/contrail-bio/`), 2013. [Accessed on 30 December 2014].

[74] George F. Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems: concepts and design*. Pearson education, 2005.

[75] Fábio Coutinho, Eduardo Ogasawara, Daniel De Oliveira, Vanessa Braganholo, Alexandre A. B. Lima, Alberto M. R. Dávila, and Marta Mattoso. Data parallelism in bioinformatics workflows using Hydra. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 507–515. ACM, 2010.

[76] CRM. Salesforce Customer Success Platform. (Available from: `http://www.salesforce.com/`), 2016. [Accessed on 10 January 2016].

[77] Crossbow Project. Crossbow: Genotyping from short reads using cloud computing. (Available from: `http://bowtie-bio.sourceforge.net/crossbow/index.shtml`), 2013. [Accessed on 30 December 2014].

[78] Frederica Darema. The SPMD model: past, present and future. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 1–1. Springer, 2001.

[79] Toon De Pessemier, Kris Vanhecke, Simon Dooms, and Luc Martens. Content-based recommendation algorithms on the Hadoop MapReduce framework. In *7th International Conference on Web Information Systems and Technologies (WEBIST-2011)*, pages 237–240. Ghent University, Department of Information Technology, 2011.

[80] Dieter De Witte, Michiel Van Bel, Pieter Audenaert, Piet Demeester, Bart Dhoedt, Klaas Vandepoele, and Jan Fostier. A parallel, distributed-memory framework for comparative motif discovery. In *Parallel Processing and Applied Mathematics*, pages 268–277. Springer, 2014.

[81] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Operating Systems Design and Implementation (OSDI)*, pages 137–150, 2004.

[82] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[83] Sebastian Deorowicz, Agnieszka Debudaj-Grabysz, and Szymon Grabowski. Disk-based k-mer counting on a PC. *BMC Bioinformatics*, 14(1):160, 2013.

[84] Sebastian Deorowicz, Marek Kokot, Szymon Grabowski, and Agnieszka Debudaj-Grabysz. KMC 2: fast and resource-frugal k-mer counting. *Bioinformatics*, 31(10):1569–1576, 2015.

[85] DFRWS. DFRWS (Digital Forensics Research Conference). (Available from: `http://dfrws.org`), 2001. [Accessed on 16 December 2015].

[86] Disco Project. Disco MapReduce. (Available from: `http://discoproject.org/`), 2014. [Accessed on 14 June 2014].

[87] Distributed.net. Distributed.net main page. (Available from: `http://www.distributed.net/Main_Page`), 2015. [Accessed on 9 December 2015].

[88] Ciprian Dobre and Fatos Xhafa. Parallel programming paradigms and frameworks in Big Data era. *International Journal of Parallel Programming*, pages 1–29, 2013.

[89] Matthieu Dorier, Gabriel Antoniu, Franck Cappello, Marc Snir, and Leigh Orf. Damaris: How to efficiently leverage multicore parallelism to achieve scalable, jitter-free I/O. In *IEEE International Conference on Cluster Computing (CLUSTER), 2012*, pages 155–163. IEEE, 2012.

[90] Jake Drew and Michael Hahsler. Strand: fast sequence comparison using MapReduce and locality sensitive hashing. In *Proceedings of the 5th ACM Conference on Bioinformatics, Computational Biology, and Health Informatics*, pages 506–513. ACM, 2014.

[91] Richard Durbin, Sean Eddy, Anders Krogh, and Graeme Mitchison. *Biological sequence analysis: probabilistic models of proteins and nucleic acids*. Cambridge University Press, New York, NY, USA, 1998.

[92] Robert C. Edgar. MUSCLE: multiple sequence alignment with high accuracy and high throughput. *Nucleic Acids Research*, 32(5):1792–1797, 2004.

[93] Victor Eijkhout. *Introduction to High Performance Scientific Computing*. Lulu.com, 2015.

[94] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative MapReduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 810–818. ACM, 2010.

[95] Khaled Elmeleegy. Piranha: optimizing short jobs in Hadoop. *Proceedings of the VLDB Endowment*, 6(11):985–996, 2013.

[96] Tamer Elsayed, Jimmy Lin, and Douglas W. Oard. Pairwise document similarity in large collections with MapReduce. In *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics on Human Language Technologies*, pages 265–268, 2008.

[97] Constantinos Evangelinos and Chris N. Hill. Cloud computing for parallel scientific HPC applications: Feasibility of running coupled atmosphere-ocean climate models on Amazon's EC2. In *1st Workshop on Cloud Computing and its Applications (CCA)*, pages 1–6, 2008.

[98] Hany Farid. Exposing digital forgeries from JPEG ghosts. *IEEE Transactions on Information Forensics and Security*, 4(1):154–160, 2009.

[99] FBI - Computer Forensics Labs. RCFL program annual report for fiscal year 2008. (Available from: `http://www.fbi.gov/news/stories/2009/august/rcfls_081809`), 2008. [Accessed on 16 December 2015].

[100] Corrado Federici. AlmaNebula: a computer forensics framework for the cloud. *Procedia Computer Science*, 19:139–146, 2013.

[101] Xin Feng, Robert Grossman, and Lincoln Stein. PeakRanger: a cloud-enabled peak caller for ChIP-seq data. *BMC Bioinformatics*, 12(1):1–11, 2011.

[102] Paolo Ferragina, Raffaele Giancarlo, Valentina Greco, Giovanni Manzini, and Gabriel Valiente. Compression-based classification of biological sequences and structures via the universal similarity metric: experimental assessment. *BMC Bioinformatics*, 8:252, 2007.

[103] Benjamin Fish, Jeremy Kun, Adám D Lelkes, Lev Reyzin, and György Turán. On the computational complexity of MapReduce. In *Distributed Computing*, volume 9363, pages 1–15. Springer, 2015.

[104] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 100(9):948–960, 1972.

[105] Lukas Forer, Tomislav Lipic, Sebastian Schonherr, Hansi Weisensteiner, Davor Davidovic, Florian Kronenberg, and Enis Afgan. Delivering bioinformatics MapReduce applications in the cloud. In *37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2014*, pages 373–377. IEEE, 2014.

[106] Ian Foster and Carl Kesselman. What is the grid. *Daily News and Information for the Global Grid Community*, 1(6), 2002.

[107] Ian Foster, Yong Zhao, Ioan Raicu, and Shiyong Lu. Cloud computing and grid computing 360-degree compared. In *Grid Computing Environments Workshop, 2008, (GCE'08)*, pages 1–10. IEEE, 2008.

[108] Geoffrey Fox, Xiaohong Qiu, Scott Beason, Jong Choi, Jaliya Ekanayake, Thilina Gunarathne, Mina Rho, Haixu Tang, Neil Devadasan, and Gilbert Liu. Biomedical case studies in data intensive computing. In *Cloud Computing*, volume 5931, pages 2–18. Springer, 2009.

[109] Christopher Frank. Forbes: improving decision making in the world of Big Data. (Available from: `http://www.forbes.com/sites/christopherfrank/2012/03/25/improving-decision-making-in-the-world-of-big-data/`), 2012. [Accessed on 2 December 2015].

[110] Jessica Fridrich, Miroslav Goljan, and Rui Du. Steganalysis based on JPEG compatibility. In *International Symposium on the Convergence of IT and Communications (ITCom)*, volume 4518, pages 275–280. International Society for Optics and Photonics, 2001.

[111] Jessica Fridrich, Jan Lukáš, and Miroslav Goljan. Detecting digital image forgeries using sensor pattern noise. In *SPIE, Electronic Imaging, Security, Steganography, and Watermarking of Multimedia Contents VIII*, volume 6072, pages 1–11, 2006.

[112] Massimo Gaggero, Simone Leo, Simone Manca, Federico Santoni, Omar Schiaratura, and Gianluigi Zanetti. Parallelizing bioinformatics applications with MapReduce. *Cloud Computing and its Applications*, pages 22–23, 2008.

[113] Peter B. Galvin, Greg Gagne, and Abraham Silberschatz. *Operating system concepts - Ninth edition.* John Wiley & Sons, Inc., 2013.

[114] Genome Informatics Research Group and Center for Bioinformatics, University of Hamburg. GenomeTools - the versatile open source genome anal-

ysis software. (Available from: `http://genometools.org`), 2016. [Accessed on 25 January 2016].

[115] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. *ACM SIGOPS Operating Systems Review*, 37(5):29–43, 2003.

[116] Raffaele Giancarlo, Simona E. Rombo, and Filippo Utro. Compressive biological sequence analysis and archival in the era of high-throughput sequencing technologies. *Briefings in Bioinformatics*, 15(3):390–406, 2014.

[117] Raffaele Giancarlo, Simona E. Rombo, and Filippo Utro. Epigenomic k-mer dictionaries: shedding light on how sequence composition influences in vivo nucleosome positioning. *Bioinformatics*, 2015.

[118] Raffaele Giancarlo, Davide Scaturro, and Filippo Utro. Textual data compression in computational biology: a synopsis. *Bioinformatics*, 25:1575–1586, 2009.

[119] Raffaele Giancarlo, Davide Scaturro, and Filippo Utro. Textual data compression in computational biology: algorithmic techniques. *Computer Science Review*, 6(1):1–25, 2012.

[120] Thomas Gloe. Feature-based forensic camera model identification. In *Transactions on Data Hiding and Multimedia Security VIII*, pages 42–62. Springer, 2012.

[121] Thomas Gloe, Karsten Borowka, and Antje Winkler. Feature-based camera model identification works in practice. In *Information Hiding*, pages 262–276. Springer, 2009.

[122] Miroslav Goljan, Jessica Fridrich, and Tomáš Filler. Large scale test of sensor fingerprint camera identification. In *IS&T/SPIE, Electronic Imaging, Security and Forensics of Multimedia Contents XI*, volume 7254, pages 1–12. International Society for Optics and Photonics, 2009.

[123] Miroslav Goljan, Jessica Fridrich, and Tomáš Filler. Managing a large database of camera fingerprints. In *SPIE Conference on Media Forensics*

and Security, volume 7541, pages 1–12. International Society for Optics and Photonics, 2010.

[124] Navid Golpayegani and Milton Halem. Cloud computing for satellite data processing on high end compute clusters. In *IEEE International Conference on Cloud Computing*, pages 88–92. IEEE, 2009.

[125] Google. Cloud Dataflow. (Available from: `https://cloud.google.com/dataflow/`), 2015. [Accessed on 16 January 2016].

[126] Google. Google Cloud Platform. (Available from: `https://cloud.google.com/appengine/`), 2015. [Accessed on 10 January 2016].

[127] Jim Gray and Alex Szalay. eScience - a transformed scientific method. In presentation to the Computer Science and Technology Board of the National Research Council (NRC-CSTB). (Available from: `http://research.microsoft.com/en-us/um/people/gray/talks/nrc-cstb_escience.ppt`), 2007.

[128] Valentina Greco and Raffaele Giancarlo. Grid-K: a cometa VO service for compression-based classification of biological sequences and structures. *Symposium GRID Open Days at the University of Palermo, Italy*, pages 87–93, 2007.

[129] Alessandro Guarino. Digital forensics as a Big Data challenge. In *ISSE 2013 Securing Electronic Business Processes*, pages 197–203. Springer, 2013.

[130] Martyn Guest, Giovanni Aloisio, Richard Kenway, et al. The scientific case for HPC in Europe 2012-2020. Technical report, PRACE, October 2012. `http://www.prace-ri.eu/prace-the-scientific-case-for-hpc/`, 2012.

[131] Dan Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology.* Cambridge University Press, New York, NY, USA, 1997.

[132] John L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[133] Per Brinch Hansen. *The architecture of concurrent programs*. Prentice-Hall, Inc., 1977.

[134] Harvard-Lipper Center for Computational Genetics. Reverse and/or complement DNA sequences. (Available from: `http://arep.med.harvard.edu/labgc/adnan/projects/Utilities/revcomp.html`), 2016. [Accessed on 6 March 2016].

[135] Seyyed Mohsen Hashemi and Amid Khatibi Bardsiri. Cloud vs. Grid computing. *ARPN Journal of Systems and Software*, 2(5), 2012.

[136] Bernhard Haubold. Alignment-free phylogenetics and population genetics. *Briefings in Bioinformatics*, 15(3):407–418, 2014.

[137] John Healy, Elizabeth E. Thomas, Jacob T. Schwartz, and Michael Wigler. Annotating large genomes with exact word matches. *Genome Research*, 13(10):2306–2315, 2003.

[138] Dominique Heger. Hadoop performance tuning - a pragmatic & iterative approach. *CMG Journal*, 4:97–113, 2013.

[139] Christopher M. Hill, Carl H. Albach, Sebastian G. Angel, and Mihai Pop. K-mulus: strategies for BLAST in the cloud. In *Parallel Processing and Applied Mathematics*, pages 237–246. Springer, 2014.

[140] Michael Höhl and Mark A. Ragan. Is multiple-sequence alignment required for accurate inference of phylogeny? *Systematic Biology*, 56(2):206–221, 2007.

[141] Sebastian Horwege, Sebastian Lindner, Marcus Boden, Klaus Hatje, Martin Kollmar, Chris-André Leimeister, and Burkhard Morgenstern. Spaced words and kmacs: fast alignment-free sequence comparison based on inexact word matches. *Nucleic Acids Research*, 42(W1):7–11, 2014.

[142] Yongjian Hu, Chang-Tsun Li, and Zhimao Lai. Fast source camera identification using matching signs between query and reference fingerprints. *Multimedia Tools and Applications*, 74(18):7405–7428, 2014.

[143] Shadi Ibrahim, Hai Jin, Lu Lu, Li Qi, Song Wu, and Xuanhua Shi. Evaluating MapReduce on virtual machines: the Hadoop case. In *Cloud Computing*, volume 5931, pages 519–528. Springer, 2009.

[144] Illumina. Human genome Illumina dataset. (Available from: `ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA010/SRA010896/SRX016231/`), May 2015. [Accessed on 4 May 2015].

[145] Illumina. An introduction to Next-Generation Sequencing Technology. (Available from: `http://www.illumina.com/content/dam/illumina-marketing/documents/products/illumina_sequencing_introduction.pdf`), 2016. [Accessed on 19 January 2016].

[146] Intel - Universal Parallel Computing Research Centers. Parallel computing: background. (Available from: `http://www.intel.com/pressroom/kits/upcrc/ParallelComputing_backgrounder.pdf`), 2008. [Accessed on 9 January 2016].

[147] Joab Jackson. Google service analyzes live streaming data. (Available from: `http://www.infoworld.com/d/business-intelligence/google-service-analyzes-live-streaming-data-245079`), June 2014. [Accessed on 3 December 2015].

[148] David B. Jaffe, Jonathan Butler, Sante Gnerre, Evan Mauceli, Kerstin Lindblad-Toh, Jill P. Mesirov, Michael C. Zody, and Eric S. Lander. Whole-genome sequence assembly for mammalian genomes: Arachne 2. *Genome Research*, 13(1):91–96, 2003.

[149] Law Jia Jge. The history, evolution & trends in distributed computing. (Available from: `https://prezi.com/9gobleqbzgp-/the-history-evolution-trends-in-distributed-computing/`), 2013. [Accessed on 9 December 2015].

[150] JGI-Bioinformatics. BioPig - GitHub. (Available from: `https://github.com/JGI-Bioinformatics/biopig`), November 2015. [Accessed on 11 November 2015].

[151] Shrinivas B. Joshi. Apache Hadoop performance-tuning methodologies and best practices. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, pages 241–242. ACM, 2012.

[152] Gideon Juve, Ewa Deelman, Karan Vahi, Gaurang Mehta, Bruce Berriman, Benjamin P. Berman, and Phil Maechling. Scientific workflow applications on Amazon EC2. In *5th IEEE International Conference on E-Science Workshops, 2009*, pages 59–66. IEEE, 2009.

[153] Scott D. Kahn. On the future of genomic data. *Science*, 331(6018):728–729, 2011.

[154] Alan Kaminsky. BIG CPU, BIG DATA: solving the world's toughest computational problems with parallel computing. *Mountain View, CA: Creative Commons*, 2013.

[155] Ramin Karimi, Ladjel Bellatreche, Patrick Girard, Ahcene Boukorca, and Andras Hajdu. BINOS4DNA: bitmap indexes and NoSQL for identifying species with DNA signatures through metagenomics samples. In *Information Technology in Bio- and Medical Informatics*, pages 1–14. Springer, 2014.

[156] David R. Kelley, Michael C. Schatz, Steven L. Salzberg, et al. Quake: quality-aware detection and correction of sequencing errors. *Genome Biology*, 11(11):1–13, 2010.

[157] Ankit Khandelwal. Business insights from Big Data. (Available from: `http://bigdataanalytics.blogspot.it/2010/10/business-insights-from-big-data.html`), October 2013. [Accessed on 7 October 2013].

[158] Vinh Ngoc Khuc, Chaitanya Shivade, Rajiv Ramnath, and Jay Ramanathan. Towards building large-scale distributed systems for Twitter sentiment analysis. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 459–464. ACM, 2012.

[159] Yong-Il Kim, Yoo-Kang Ji, Sun Park, Ok-kyoon Ha, Izzat Alsmadi, Ikdam AlHami, Saif Kazakzeh, Soyoung Hwang, Donghui Yu, Byung Do Chung, et al. Big text data clustering using class labels and semantic feature based on Hadoop of cloud computing. *International Journal of Software Engineering and its Applications*, 8(4):1–10, 2014.

[160] Kenji Kurosawa, Kenro Kuroki, and Naoki Saitoh. CCD fingerprint method-identification of a video camera from videotaped images. In *International Conference on Image Processing (ICIP)*, volume 3, pages 537–540, 1999.

[161] Stefan Kurtz, Apurva Narechania, Joshua C. Stein, and Doreen Ware. A new method to compute k-mer frequencies and its application to annotate large repetitive plant genomes. *BMC Genomics*, 9(1):517, 2008.

[162] Yang Lai and Shi ZhongZhi. An efficient data mining framework on Hadoop using Java persistence API. In *IEEE 10th International Conference on Computer and Information Technology (CIT), 2010*, pages 203–209. IEEE, 2010.

[163] Ben Langmead, Kasper D. Hansen, Jeffrey T. Leek, et al. Cloud-scale RNA-sequencing differential expression analysis with Myrna. *Genome Biology*, 11(8):1–11, 2010.

[164] Ben Langmead, Michael C. Schatz, Jimmy Lin, Mihai Pop, and Steven L. Salzberg. Searching for SNPs with cloud computing. *Genome Biology*, 10(11):1–10, 2009.

[165] Ben Langmead, Cole Trapnell, Mihai Pop, Steven L. Salzberg, et al. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10(3):R25, 2009.

[166] Jooyoung Lee and Sungyong Un. Digital forensics as a service: a case study of forensic indexed search. In *International Conference on ICT Convergence (ICTC)*, pages 499–503. IEEE, 2012.

[167] Yeonhee Lee and Youngseok Lee. Toward scalable Internet traffic measurement and analysis with Hadoop. *ACM SIGCOMM Computer Communication Review*, 43(1):5–13, 2013.

[168] Youngseok Lee, Wonchul Kang, and Hyeongu Son. An Internet traffic analysis method with MapReduce. In *IEEE/IFIP Network Operations and Management Symposium Workshops (NOMS Wksps), 2010*, pages 357–361. IEEE, 2010.

[169] Arnaud Lefebvre, Thierry Lecroq, Hélene Dauchel, and Joël Alexandre. FORRepeats: detects repeats on entire chromosomes and between genomes. *Bioinformatics*, 19(3):319–326, 2003.

[170] Chris-André Leimeister, Marcus Boden, Sebastian Horwege, Sebastian Lindner, and Burkhard Morgenstern. Fast alignment-free sequence comparison using spaced-word frequencies. *Bioinformatics*, 30(14):1991–1999, 2014.

[171] Simone Leo, Federico Santoni, and Gianluigi Zanetti. Biodoop: bioinformatics on Hadoop. In *International Conference on Parallel Processing Workshops, 2009 (ICPPW'09)*, pages 415–422. IEEE, 2009.

[172] Kuo-Bin Li. ClustalW-MPI: ClustalW analysis using distributed and parallel computing. *Bioinformatics*, 19(12):1585–1586, 2003.

[173] Min Li, Liangzhao Zeng, Shicong Meng, Jian Tan, Li Zhang, Ali R. Butt, and Nicholas Fuller. MRONLINE: MapReduce online performance tuning. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, pages 165–176. ACM, 2014.

[174] Ruiqiang Li, Yingrui Li, Karsten Kristiansen, and Jun Wang. SOAP: short oligonucleotide alignment program. *Bioinformatics*, 24(5):713–714, 2008.

[175] Yang Li and Xifeng Yan. MSPKmerCounter: a fast and memory efficient approach for k-mer counting. *draft, (Available from: http://cs.ucsb.edu/~yangli/paper/bio14_li.pdf)*, 2014.

[176] Ross A. Lippert, Haiyan Huang, and Michael S. Waterman. Distributional regimes for the number of k-word matches between two random sequences. *Proceedings of the National Academy of Sciences*, 99(22):13980–13989, 2002.

[177] Yang Liu, Xiaohong Jiang, Huajun Chen, Jun Ma, and Xiangyu Zhang. MapReduce-based pattern finding algorithm applied in motif detection for prescription compatibility network. In *Advanced Parallel Processing Technologies*, pages 341–355. Springer, 2009.

[178] Scott Lloyd and Quinn Snell. Accelerated large-scale multiple sequence alignment. *BMC Bioinformatics*, 12:466, 2011.

[179] Glenn Lockwood. DNA sequencing: Not quite HPC yet. (Available from: `http://www.nextplatform.com/2015/03/03/dna-sequencing-not-quite-hpc-yet/`), 2015. [Accessed on 11 February 2016].

[180] Wei Lu, Jun Huang, and Lin Hong. Massive data MapReduce fingerprint discriminant algorithm based on Hadoop. In *Applied Mechanics and Materials*, volume 263, pages 2655–2660. Trans Tech Publications, 2012.

[181] Jan Lukáš, Jessica Fridrich, and Miroslav Goljan. Digital camera identification from sensor pattern noise. *IEEE Transactions on Information Forensics and Security*, 1:205–214, November 2006.

[182] Nancy A. Lynch. *Distributed algorithms*. Morgan Kaufmann, 1996.

[183] David Makovoz and Iffat Khan. Mosaicking with MOPEX. In *Astronomical Data Analysis Software and Systems XIV*, volume 347, pages 81–85, 2005.

[184] Guillaume Marçais and Carl Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770, 2011.

[185] Ami Marowka. The GRID: Blueprint for a new computing infrastructure. *Scalable Computing: Practice and Experience*, 3(3), 2001.

[186] Marco Masseroli, Barend Mons, Erik Bongcam-Rudloff, Stefano Ceri, Alexander Kel, François Rechenmann, Frederique Lisacek, and Paolo Romano. Integrated Bio-Search: challenges and trends for the integration, search and comprehensive processing of biological information. *BMC Bioinformatics*, 15(1):1–15, 2014.

[187] Marco Masseroli, Matteo Picozzi, Giorgio Ghisalberti, and Stefano Ceri. Explorative search of distributed bio-data to answer complex biomedical questions. *BMC Bioinformatics*, 15(1):1–14, 2014.

[188] MATLAB. MATLAB Distributed Computing Server. (Available from: `http://www.mathworks.com/products/distriben/index.html`), August 2013. [Accessed on 17 August 2013].

[189] MATLAB. MATLAB Parallel Computing Toolbox. (Available from: `http://www.mathworks.com/products/parallel-computing/`), August 2013. [Accessed on 17 August 2013].

[190] Andréa Matsunaga, Maurício Tsugawa, and José Fortes. CloudBLAST: combining MapReduce and virtualization on distributed resources for bioinformatics applications. In *IEEE Fourth International Conference on eScience, 2008. eScience'08*, pages 222–229. IEEE, 2008.

[191] Suzanne J. Matthews and Tiffani L. Williams. MrsRF: an efficient MapReduce algorithm for analyzing large collections of evolutionary trees. *BMC Bioinformatics*, 11(Suppl 1):1–9, 2010.

[192] Aaron McKenna, Matthew Hanna, Eric Banks, Andrey Sivachenko, Kristian Cibulskis, Andrew Kernytsky, Kiran Garimella, David Altshuler, Stacey Gabriel, Mark Daly, et al. The genome analysis toolkit: a MapReduce framework for analyzing next-generation DNA sequencing data. *Genome Research*, 20(9):1297–1303, 2010.

[193] Páll Melsted and Bjarni V. Halldórsson. KmerStream: streaming algorithms for k-mer abundance estimation. *Bioinformatics*, 30(24):3541–3547, 2014.

[194] Pall Melsted and Jonathan K. Pritchard. Efficient counting of k-mers in DNA sequences using a Bloom filter. *BMC Bioinformatics*, 12(1):1–7, 2011.

[195] Meryl. Getting started with Meryl. (Available from: `http://kmer.sourceforge.net/wiki/index.php/Getting_Started_with_Meryl`), 2012. [Accessed on 30 June 2015].

[196] Cade Metz. Meet the data brains behind the rise of Facebook. (Available from: `http://www.wired.com/2013/02/facebook-data-team/`), 2013. [Accessed on 2 December 2015].

[197] Microsoft Corporation. Dryad. (Available from: `http://research.microsoft.com/en-us/projects/dryad/`), 2011. [Accessed on 19 June 2014].

[198] Microsoft Corporation. Azure. (Available from: `https://azure.microsoft.com`), 2015. [Accessed on 28 December 2015].

[199] M. Kivanc Mihcak, Igor Kozintsev, and Kannan Ramchandran. Spatially adaptive statistical modeling of wavelet image coefficients and application to denoising. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 6, pages 3253–3256. IEEE, Mar 1999.

[200] Jason R. Miller, Arthur L. Delcher, Sergey Koren, Eli Venter, Brian P. Walenz, Anushka Brownley, Justin Johnson, Kelvin Li, Clark Mobarry, and Granger Sutton. Aggressive assembly of pyrosequencing reads with mates. *Bioinformatics*, 24(24):2818–2824, 2008.

[201] Bernard M. E. Moret. Towards a discipline of experimental algorithmics. *Data Structures, Near Neighbor Searches, and Methodology: Fifth and Sixth DIMACS Implementation Challenges*, 59:197–213, 2002.

[202] David W. Mount. Sequence and genome analysis. *Bioinformatics: Cold Spring Harbour Laboratory Press: Cold Spring Harbour*, 2, 2004.

[203] Eugene W. Myers, Granger G. Sutton, Art L. Delcher, Ian M. Dew, Dan P. Fasulo, Michael J. Flanigan, Saul A. Kravitz, Clark M. Mobarry, Knut

H. J. Reinert, Karin A. Remington, et al. A whole-genome assembly of drosophila. *Science*, 287(5461):2196–2204, 2000.

[204] National Center for Biotechnology Information European (NCBI) and Bioinformatics Institute (EBI). NCBI/EBI sequence read archive. (Available from: `http://www.ncbi.nlm.nih.gov/sra/`), 2015. [Accessed on 4 May 2015].

[205] National Center for Biotechnology Information (NCBI). BLAST: Basic Local Alignment Search Tool. (Available from: `http://blast.ncbi.nlm.nih.gov/Blast.cgi`), 2014. [Accessed on 30 December 2014].

[206] Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970.

[207] Matti Niemenmaa, Aleksi Kallio, André Schumacher, Petri Klemelä, Eija Korpelainen, and Keijo Heljanko. Hadoop-BAM: directly manipulating next generation sequencing data in the cloud. *Bioinformatics*, 28(6):876–877, 2012.

[208] NIST. NIST Cloud Computing Program. (Available from: `http://www.nist.gov/itl/cloud/`), 2015. [Accessed on 2 January 2016].

[209] Henrik Nordberg, Karan Bhatia, Kai Wang, and Zhong Wang. BioPig: a Hadoop-based analytic toolkit for large-scale sequence data. *Bioinformatics*, 29(23):3014–3019, 2013.

[210] Graeme Noseworthy. Infographic: managing the big flood of Big Data in digital marketing. (Available from: `http://analyzingmedia.com/2012/infographic-big-flood-of-big-data-in-digital-marketing/`), 2012. [Accessed on 2 December 2015].

[211] Kary Ocaña and Daniel de Oliveira. Parallel computing in genomic research: advances and applications. *Advances and Applications in Bioinformatics and Chemistry: AABC*, 8:23, 2015.

[212] Aisling O'Driscoll, Jurate Daugelaite, and Roy D. Sleator. 'Big Data', Hadoop and cloud computing in genomics. *Journal of Biomedical Informatics*, 46(5):774–781, 2013.

[213] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 1099–1110. ACM, 2008.

[214] Owen O'Malley. Terabyte sort on Apache Hadoop. *Yahoo, (Available from: http://sortbenchmark.org/YahooHadoop.pdf)*, pages 1–3, 2008.

[215] Openstack.org. OpenStack. (Available from: `https://www.openstack.org/`), 2015. [Accessed on 2 January 2016].

[216] Mayank Pahadia, Akash Srivastava, Divyang Srivastava, and Nagamma Patil. Classification of multi-genomic data using MapReduce paradigm. In *International Conference on Computing, Communication & Automation (ICCCA), 2015*, pages 678–682. IEEE, 2015.

[217] Mayank Pahadia, Akash Srivastava, Divyang Srivastava, and Nagamma Patil. Genome data analysis using MapReduce paradigm. In *Second International Conference on Advances in Computing and Communication Engineering (ICACCE), 2015*, pages 556–559. IEEE, 2015.

[218] Gary Palmer et al. A road map for digital forensic research. In *First Digital Forensic Research Workshop, Utica, New York*, pages 27–30, 2001.

[219] Christos H. Papadimitriou. *Computational complexity*. John Wiley & Sons, Inc., 2003.

[220] Mahmoud Parsian. *Data algorithms - recipes for scaling up with Hadoop and Spark*. O'Reilly Media, July 2015.

[221] Luca Pinello, Giosuè Lo Bosco, Bret Hanlon, and Guo-Cheng Yuan. A motif-independent metric for DNA sequence specificity. *BMC Bioinformatics*, 12(1), 2011.

[222] Luca Pinello, Giosuè Lo Bosco, and Guo-Cheng Yuan. Applications of alignment-free methods in epigenomics. *Briefings in Bioinformatics*, 15:419–430, 2013.

[223] Precision Optical Imaging. ISO Noise Chart 15739. (Available from: `http://www.precisionopticalimaging.com/products/products.asp?type=15739`), 2011. [Accessed on 16 June 2014].

[224] Judy Qiu, Jaliya Ekanayake, Thilina Gunarathne, Jong Youl Choi, Seung-Hee Bae, Yang Ruan, Saliya Ekanayake, Stephen Wu, Scott Beason, Geoffrey Fox, et al. Data intensive computing for bioinformatics. *Bioinformatics: Concepts, Methodologies, Tools, and Applications*, 287, 2013.

[225] Xiaohong Qiu, Jaliya Ekanayake, Scott Beason, Thilina Gunarathne, Geoffrey Fox, Roger Barga, and Dennis Gannon. Cloud technologies for bioinformatics applications. In *Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers*, pages 1–10. ACM, 2009.

[226] Atanas Radenski and Louis Ehwerhemuepha. Speeding-up codon analysis on the cloud with local MapReduce aggregation. *Information Sciences*, 263:175–185, 2014.

[227] Nallanthighal S. Raghava and Shelly. D. Iris recognition on Hadoop: a biometrics system implementation on cloud computing. In *IEEE International Conference on Cloud Computing and Intelligence Systems (CCIS)*, pages 482–485. IEEE, 2011.

[228] Zeehasham Rasheed and Huzefa Rangwala. A Map-Reduce framework for clustering metagenomes. In *IEEE 27th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013*, pages 549–558. IEEE, 2013.

[229] Daniele Ravì, M. Bober, Giovanni Maria Farinella, Mirko Guarnera, and Sebastiano Battiato. Semantic segmentation of images exploiting DCT based features and random forest. *Pattern Recognition*, 52:260–273, 2016.

[230] Gesine Reinert, David Chew, Fengzhu Sun, and Michael S. Waterman. Alignment-free sequence comparison (I): statistics and power. *Journal of Computational Biology*, 16(12):1615–1634, 2009.

[231] Guillaume Rizk, Dominique Lavenier, and Rayan Chikhi. DSK: k-mer counting with very low memory usage. *Bioinformatics*, 29(5):652–653, 2013.

[232] Guillaume Rizk, Dominique Lavenier, and Rayan Chikhi. Software: DSK. (Available from: `http://minia.genouest.org/dsk/`), 2015. [Accessed on 3 May 2015].

[233] Eric Roberts. A brief history of the Internet. (Available from: `http://cs.stanford.edu/people/eroberts/courses/soco/projects/2001-02/distributed-computing/html/history.html`), 2006. [Accessed on 3 January 2016].

[234] Vassil Roussev, Liqiang Wang, Golden Richard, and Lodovico Marziale. A cloud computing platform for large-scale forensic computing. In *Advances in Digital Forensics V*, pages 201–214. Springer, 2009.

[235] Antony Rowstron, Dushyanth Narayanan, Austin Donnelly, Greg O'Shea, and Andrew Douglas. Nobody ever got fired for using Hadoop on a cluster. In *Proceedings of the 1st International Workshop on Hot Topics in Cloud Data Processing*, pages 1–5. ACM, 2012.

[236] Rajat Shuvro Roy, Debashish Bhattacharya, and Alexander Schliep. Turtle: identifying frequent k-mers with cache-efficient algorithms. *Bioinformatics*, 30(14):1950–1957, 2014.

[237] G. Sudha Sadasivam and G. Baktavatchalam. A novel approach to multiple sequence alignment using Hadoop data grids. In *Proceedings of the 2010 Workshop on Massive Data Analytics on the Cloud*, pages 1–7. ACM, 2010.

[238] Esen Sagynov. Commercial and open source Big Data platforms comparison. (Available from: `http://architects.dzone.com/articles/commercial-and-open-source-big`), October 2013. [Accessed on 6 October 2013].

[239] Naruya Saitou and Masatoshi Nei. The neighbor-joining method: a new method for reconstructing phylogenetic trees. *Molecular Biology and Evolution*, 4(4):406–425, 1987.

[240] Michael C. Schatz. BlastReduce: high performance short read mapping with MapReduce. *University of Maryland, (Available from: http://cgis.cs.umd.edu/Grad/scholarlypapers/papers/MichaelSchatz.pdf)*, 2008.

[241] Michael C. Schatz. CloudBurst: highly sensitive read mapping with MapReduce. *Bioinformatics*, 25(11):1363–1369, 2009.

[242] Michael C. Schatz, Daniel Sommer, David Kelley, and Mihai Pop. De novo assembly of large genomes using cloud computing. In *Proceedings of the Cold Spring Harbor Biology of Genomes Conference*, 2010.

[243] Matthias Scholz, Adrian Tett, and Nicola Segata. Chapter 5 - Computational tools for taxonomic microbiome profiling of shotgun metagenomes. In Jacques Izard and Maria C. Rivera, editors, *Metagenomics for Microbiology*, pages 67 – 80. Academic Press, Oxford, 2015.

[244] André Schumacher, Luca Pireddu, Matti Niemenmaa, Aleksi Kallio, Eija Korpelainen, Gianluigi Zanetti, and Keijo Heljanko. SeqPig: simple and scalable scripting for large sequencing data sets in Hadoop. *Bioinformatics*, 30(1):119–120, 2014.

[245] Science Daily. Big Data, for better or worse: 90% of world's data generated over last two years. (Available from: `http://www.sciencedaily.com/releases/2013/05/130522085217.htm`), 2013. [Accessed on 16 January 2016].

[246] Nicola Segata, Levi Waldron, Annalisa Ballarini, Vagheesh Narasimhan, Olivier Jousson, and Curtis Huttenhower. Metagenomic microbial community profiling using unique clade-specific marker genes. *Nature Methods*, 9(8):811–814, 2012.

[247] Kulesh Shanmugasundaram, Nasir Memon, Anubhav Savant, and Herve Bronnimann. ForNet: a distributed forensics network. In *Computer Network Security*, pages 1–16. Springer, 2003.

[248] James E. Short, Roger E. Bohn, and Chaitanya Baru. How much information? 2010 report on enterprise server information. uc san diego (ucsd) global information industry center. (Available from: `http://clds.sdsc.edu/sites/clds.sdsc.edu/files/pubs/ESI-Report-Jan2011.pdf`), 2011. [Accessed on 5 January 2016].

[249] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop distributed file system. In *IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), 2010*, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[250] Gregory E. Sims, Se-Ran Jun, Guohong A. Wu, and Sung-Hou Kim. Alignment-free genome comparison with feature frequency profiles (FFP) and optimal resolutions. *Proceedings of the National Academy of Sciences*, 106(8):2677–2682, 2009.

[251] Gregory E. Sims and Sung-Hou Kim. Whole-genome phylogeny of Escherichia coli/Shigella group by feature frequency profiles (FFPs). *Proceedings of the National Academy of Sciences*, 108(20):8329–8334, 2011.

[252] Suzanne S. Sindi, Brian R. Hunt, and James A. Yorke. Duplication count distributions in DNA sequences. *Physical Review E*, 78(6):061912, 2008.

[253] Craig Smith. By the numbers: 200+ amazing Facebook statistics. (Available from: `http://expandedramblings.com/index.php/by-the-numbers-17-amazing-facebook-stats/15/`), 2015. [Accessed on 21 December 2015].

[254] Temple F. Smith and Michael S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

[255] Peter H. A. Sneath and Robert R. Sokal. Unweighted pair group method with arithmetic mean. *Numerical Taxonomy*, pages 230–234, 1973.

[256] Robert R. Sokal and Michener Charles D. A statistical method for evaluating systematic relationships. *University of Kansas Science Bulletin*, 38:1409–1438, 1958.

[257] Kai Song, Jie Ren, Gesine Reinert, Minghua Deng, Michael S. Waterman, and Fengzhu Sun. New developments of alignment-free sequence comparison: measures, statistics and next-generation sequencing. *Briefings in Bioinformatics*, 15(3):343–353, 2013.

[258] Stanford University. Folding@home. (Available from: `http://folding.stanford.edu`), 2013. [Accessed on 16 January 2015].

[259] Erich Strohmaier, Jack Dongarra, Horst Simon, and Martin Meuer. TOP500 lists. (Available from: `http://www.top500.org/lists/`), 2016. [Accessed on 11 January 2016].

[260] Aravind Subramanian, Pablo Tamayo, Vamsi K. Mootha, Sayan Mukherjee, Benjamin L. Ebert, Michael A. Gillette, Amanda Paulovich, Scott L. Pomeroy, Todd R. Golub, Eric S. Lander, et al. Gene set enrichment analysis: a knowledge-based approach for interpreting genome-wide expression profiles. *Proceedings of the National Academy of Sciences*, 102(43):15545–15550, 2005.

[261] Chris Sweeney, Liu Liu, Sean Arietta, and Jason Lawrence. HIPI: a Hadoop Image Processing Interface for image-based MapReduce tasks. (Available from: `http://hipi.cs.virginia.edu/`), 2011. [Accessed on 17 May 2014].

[262] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed systems*. Prentice-Hall, 2007.

[263] Ronald C. Taylor. An overview of the Hadoop/MapReduce/HBase framework and its current applications in bioinformatics. *BMC Bioinformatics*, 11(Suppl 12):1–6, 2010.

[264] Thanh Hai Thai, Remi Cogranne, and Florent Retraint. Camera model identification based on the heteroscedastic noise model. *IEEE Transactions on Image Processing*, 23(1):250–263, 2014.

[265] Julie D. Thompson, Desmond G. Higgins, and Toby J. Gibson. CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Research*, 22(22):4673–4680, 1994.

[266] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu, and Raghotham Murthy. Hive - a petabyte scale data warehouse using Hadoop. In *IEEE 26th International Conference on Data Engineering (ICDE), 2010*, pages 996–1005. IEEE, 2010.

[267] David C. Torney, Christina Burks, Daniel Davison, and Kart M. Sirotkin. Computation of d2: a measure of sequence dissimilarity. In *Computers and DNA: the proceedings of the Interface between Computation Science and Nucleic Acid Sequencing Workshop*. Redwood City, Calif.: Addison-Wesley Pub. Co., 1990., 1990.

[268] Duy Tin Truong, Eric A. Franzosa, Timothy L. Tickle, Matthias Scholz, George Weingart, Edoardo Pasolli, Adrian Tett, Curtis Huttenhower, and Nicola Segata. Metaphlan2 for enhanced metagenomic taxonomic profiling. *Nature Methods*, 12(10):902–903, 2015.

[269] DB Tsai and Jenny Thompson. Implementing the in-mapper combiner for performance gains in Hadoop. (Available from: `https://www.dbtsai.com/blog/2013/implementing-the-in-mapper-combiner-for-performance-gains-in-hadoop/`), 2013. [Accessed on 5 February 2016].

[270] Radu Tudoran, Alexandru Costan, and Gabriel Antoniu. Mapiterativereduce: a framework for reduction-intensive data processing on Azure clouds. In *Proceedings of third International Workshop on MapReduce and its Applications Date*, pages 9–16. ACM, 2012.

[271] University Alliance - Villanova University. What is Big Data? (Available from: http://www.villanovau.com/resources/bi/what-is-big-data/#.VpuZUFKo2HR), 2015. [Accessed on 17 January 2016].

[272] University of Göttingen - Dep. of Bioinformatics. The Spaced Words approach to alignment-free sequence comparison. (Available from: http://spaced.gobics.de/), 2015. [Accessed on 7 January 2015].

[273] University of Virginia. FASTA Sequence Comparison. (Available from: http://fasta.bioch.virginia.edu/fasta_www2/fasta_list2.shtml), 2014. [Accessed on 30 December 2014].

[274] Jee Vang. The "in-mapper combining" design pattern for Map/Reduce programming in Java. (Available from: https://vangjee.wordpress.com/2012/03/07/the-in-mapper-combining-design-pattern-for-mapreduce-programming/), 2012. [Accessed on 5 February 2016].

[275] Vladimir N. Vapnik. *The nature of statistical learning theory*. Springer Science & Business Media, 2013.

[276] Hal Varian. Hal Varian on how the web challenges managers. (Available from: http://www.mckinsey.com/insights/innovation/hal_varian_on_how_the_web_challenges_managers), 2009. [Accessed on 2 January 2016].

[277] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache Hadoop YARN: yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, pages 1–16. ACM, 2013.

[278] Sebastiano Vigna. fastutil: Fast & compact type-specific collections for Java. (Available from: http://fastutil.di.unimi.it/), 2015. [Accessed on 22 May 2015].

[279] Susana Vinga. Editorial: Alignment-free methods in computational biology. *Briefings in Bioinformatics*, 15:341–342, 2014.

[280] Susana Vinga and Jonas Almeida. Alignment-free sequence comparison - a review. *Bioinformatics*, 19:513–523, 2003.

[281] Virus Encyclopedia. Creeper. (Available from: `http://virus.wikidot.com/creeper`), 2015. [Accessed on 9 December 2015].

[282] Jeffrey Scott Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys (CsUR)*, 33:209–271, 2001.

[283] Mikhail Vorontsov. Large HashMap overview: JDK, FastUtil, Goldman Sachs, HPPC, Koloboke, Trove – January 2015 version. (Available from: `http://java-performance.info/hashmap-overview-jdk-fastutil-goldman-sachs-hppc-koloboke-trove-january-2015/`), 2015. [Accessed on 3 May 2015].

[284] Panagiotis D. Vouzis and Nikolaos V. Sahinidis. GPU-BLAST: using graphics processors to accelerate protein sequence alignment. *Bioinformatics*, 2010.

[285] Dennis P. Wall, Parul Kudtarkar, Vincent A. Fusaro, Rimma Pivovarov, Prasad Patil, and Peter J. Tonellato. Cloud computing for comparative genomics. *BMC Bioinformatics*, 11(1):1–12, 2010.

[286] Gregory K. Wallace. The JPEG still picture compression standard. *IEEE Transactions on Consumer Electronics*, 38(1):18–34, 1992.

[287] Lin Wan, Gesine Reinert, Fengzhu Sun, and Michael S. Waterman. Alignment-free sequence comparison (II): theoretical power of comparison statistics. *Journal of Computational Biology*, 17(11):1467–1490, 2010.

[288] Chunyu Wang, Maozu Guo, and Yang Liu. EST clustering in large dataset with MapReduce. In *First International Conference on Pervasive Computing Signal Processing and Applications (PCSPA), 2010*, pages 968–971. IEEE, 2010.

[289] Jianwu Wang, Daniel Crawl, and Ilkay Altintas. Kepler + Hadoop: a general architecture facilitating data-intensive applications in scientific workflow systems. In *Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science*, pages 1–8. ACM, 2009.

[290] Qiong Wang, George M. Garrity, James M. Tiedje, and James R. Cole. Naive bayesian classifier for rapid assignment of rRNA sequences into the new bacterial taxonomy. *Applied and Environmental Microbiology*, 73(16):5261–5267, 2007.

[291] Tom White. The small files problem. Cloudera. (Available from: `http://blog.cloudera.com/blog/2009/02/the-small-files-problem/`), 2009. [Accessed on 5 April 2014].

[292] Tom White. *Hadoop: the definitive guide, 3rd edition.* O'Reilly Media / Yahoo Press, May 2012. Storage and Analysis at Internet Scale.

[293] Dag Wieers. Dstat: Versatile resource statistics tool. (Available from: `http://dag.wiee.rs/home-made/dstat/`), 2012. [Accessed on 6 July 2014].

[294] Marek S. Wiewiórka, Antonio Messina, Alicja Pacholewska, Sergio Maffioletti, Piotr Gawrysiak, and Michał J. Okoniewski. SparkSeq: fast, scalable, cloud-ready tool for the interactive genomic data analysis with nucleotide precision. *Bioinformatics*, 30(18):2652–2653, 2014.

[295] Keith Wiley, Andrew Connolly, Jeff Gardner, Simon Krughoff, Magdalena Balazinska, Bill Howe, YongChul Kwon, and Yingyi Bu. Astronomy in the cloud: using MapReduce for image co-addition. *Astronomy*, 123(901):366–380, 2011.

[296] Keith Wiley, Andrew Connolly, Simon Krughoff, Jeff Gardner, Magdalena Balazinska, Bill Howe, YongChul Kwon, and Yingyi Bu. Astronomical image processing with Hadoop. In *Astronomical Data Analysis Software and Systems XX*, volume 442, page 93, 2011.

[297] Derrick Wood and Steven Salzberg. Kraken: ultrafast metagenomic sequence classification using exact alignments. *Genome Biology*, 15:R46, 2014.

[298] Muneto Yamamoto and Kunihiko Kaneko. Parallel image database processing with MapReduce and performance evaluation in pseudo distributed mode. *International Journal of Electronic Commerce Studies*, 3(2):211–228, 2013.

[299] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. Mapreduce-merge: simplified relational data processing on large clusters. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, pages 1029–1040. ACM, 2007.

[300] Kuan Yang and Liqing Zhang. Performance comparison between k-tuple distance and four model-based distances in phylogenetic tree reconstruction. *Nucleic Acids Research*, 36(5):1–9, 2008.

[301] Shuiming Ye, Qibin Sun, and Ee-Chien Chang. Detecting digital image forgeries by measuring inconsistencies of blocking artifact. In *IEEE International Conference on Multimedia and Expo 2007*, pages 12–15. IEEE, July 2007.

[302] Huiguang Yi and Li Jin. Co-phylog: an assembly-free phylogenomic approach for closely related organisms. *Nucleic Acids Research*, 41(7):e75, 2013.

[303] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, volume 8, pages 1–14, 2008.

[304] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, volume 10, page 10, 2010.

[305] Qingpeng Zhang, Jason Pell, Rosangela Canino-Koning, Adina Chuang Howe, and C. Titus Brown. These are not the k-mers you are looking for: efficient online k-mer counting using a probabilistic data structure. *PloS one*, 9(7):1–13, 2014.

[306] Guoguang Zhao, Cheng Ling, and Donghong Sun. SparkSW: scalable distributed computing system for large-scale biological sequence alignment. In *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), 2015*, pages 845–852. IEEE, 2015.

[307] Weizhong Zhao, Huifang Ma, and Qing He. Parallel k-means clustering based on MapReduce. In *Cloud Computing*, pages 674–679. Springer, 2009.

[308] Ping Zhou, Jingsheng Lei, and Wenjun Ye. Large-scale data sets clustering based on MapReduce and Hadoop. *Journal of Computational Information Systems*, 7(16):5956–5963, 2011.

[309] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[310] Quan Zou, Xu-Bin Li, Wen-Rui Jiang, Zi-Yu Lin, Gui-Lin Li, and Ke Chen. Survey of MapReduce frame operation in bioinformatics. *Briefings in Bioinformatics*, 15(4):637–647, 2013.

Nomenclature

Acronyms

ASIC Application-Specific Integrated Circuits

BAM Binary Alignment/Map

bp Base Pair

BS Block Size

CAS Compare-and-Swap

CC Cloud Computing

CGR Chaos Game Representation

CHI Camera Hardware Identification

CLI Chromosome-Level Independence

CNCPO Centro Nazionale per il Contrasto alla Pedopornografia Online

CORBA Common Object Request Broker Architecture

COTS Commercial Off-The-Shelf

CPU Central Processing Unit

CS Cumulative Statistics

DAG Directed Acyclic Graph

DBMS DataBase Management System

D&C Divide and Conquer

DCOM Distributed Computing Object Model

DF Digital Forensics

DIF Digital Image Forensics

DS Distributed System

DSK Disk Streaming of K-mers

EB Exabyte

ERP Enterprise Resource Planning

ETL Extract, Transform, Load

FFP Feature Frequency Profile

FPGA Field-Programmable Gate Arrays

GB Gigabyte

GC Grid Computing

GFS Google File System

GPGPU General-Purpose computing on Graphics Processing Units

GPU Graphics Processing Unit

HAFS Alignment-free Sequence Comparison on Hadoop

HDFS Hadoop Distributed File System

HPC High Performance Computing

HSCI Hadoop for Source Camera Identification

IaaS Infrastructure-as-a-Service

i.i.d. independent and identically distributed

IDL Interface Description Language

I/O Input/Output

JF2 Jellyfish v2

JSD Jensen-Shannon Divergence

KA KAnalyze

KB Kilobyte

KCH *K*-mer Counting on Hadoop

KDD Knowledge Discovery in Databases

KLD Kullback-Leibler Divergence

KMC2 *K*-mer Counter v2

KMC *K*-mer Counter

KPI Key Performance Indicator

LAN Local Area Network

LS Local Statistics

MB Megabyte

Mbp Megabase Pair

Mbps Megabit per second

MC Mobile Computing

MIMD Multiple Instruction streams Multiple Data streams

MISD Multiple Instruction streams Single Data stream

MPI Message Passing Interface

MPMD Multiple Programs Multiple Data streams

MPP Massively Parallel Processing

MR MapReduce

MSA Multiple Sequence Alignment

MSP Minimum Substring Partitioning

MSPKC MSPKmerCounter

NAS Network Attached Storage

NCBI National Center for Biotechnology Information

NJ Neighbor Joining

NN HA NameNode High Availability

OLAP On-Line Analytic Processing

OS Operating System

OSN Online Social Network

P2P Peer-to-Peer

PaaS Platform as-a-Service

PB Petabyte

PC Personal Computer

PSA Pairwise Sequence Alignment

QoS Quality of Service

RAM Random Access Memory

RF Replication Factor

RMI Remote Method Invocation

RN Residual Noise

RP Reference Pattern

RR Recognition Rate

RT Running Time

SaaS Software-as-a-Service

SAN Storage Area Network

SIMD Single Instruction stream Multiple Data streams

SISD Single Instruction stream Single Data stream

SMP Symmetric Multi Processor

SNP Single Nucleotide Polymorphisms

SPMD Single Program Multiple Data streams

SQL Structured Query Language

SSE Streaming SIMD Extension

TB Terabyte

UC Ubiquitous Computing

UPGMA Unweighted Pair Group Method with Arithmetic Mean

URL Uniform Resource Locator

VM Virtual Machine

WAN Wide Area Network

WWW World Wide Web

YARN Yet Another Resource Negotiator

YB Yottabyte

ZB Zettabyte