Benchmarking Spark Distributed Data Structures: A Sequence Analysis Case Study

Umberto Ferraro Petrillo(✉) and Roberto Vitali

Università di Roma "La Sapienza", 00185 Roma, Italy
{umberto.ferraro,roberto.vitali}@uniroma1.it

Abstract. Big Data technologies are recognized by many as a promising solution for the efficient management and analysis of the enormous amount of genomic data available thanks to Next-Generation Sequencing technologies. Despite this, they are still used in a limited number of cases, mostly because of their complexity and of their relevant hidden computational constants. The introduction of Spark is changing this scenario, by delivering a framework that can be used to write very complex and efficient distributed applications using only few lines of codes. Spark offers three types of distributed data structures that are almost functionally equivalent but are very different in their implementations. In this paper, we briefly review these data structures and analyze their advantages and disadvantages, when used to solve a paradigmatic bioinformatics problem on a Hadoop cluster: the k-mer counting.

Keywords: Spark · k-mers Counting · Distributed computing · Performance analysis

1 Introduction

The introduction of next-generation sequencing technologies (in short, NGS) has changed the landscape of biology [1,2], thanks to the possibility of sequencing DNA at a much faster speed than the one achievable with traditional Sanger sequencing approach [3]. This advancement has raised also new methodological and technological challenges. One of these is about the proper approach to adopt for managing and processing timely the vast amount of data that is produced thanks to NGS technologies.

A solution that is gaining popularity is to resort to the technologies that have been developed for dealing with Big Data. By this term, we refer to the problem of storing, managing and processing data that may be significantly big with respect to several dimensions like size, diversity or generation rate. A very popular approach to Big Data processing, allowing for the analysis of enormous datasets, is the one based on MapReduce [4]. It is a computational paradigm that works by organizing an elaboration in two consecutive steps. In the first step, a *map* function is used to process, filter and/or transform input data. In the second step, a *reduce* function is used to aggregate the output of the map functions. Map

DOI: 10.1007/978-3-319-57711-1_7

and reduce functions are executed as tasks on the nodes of a distributed system, namely, a network of computational nodes that cooperate, sending messages each other, to achieve a common goal. The most used implementation of this paradigm is Apache Hadoop [5]. Despite being the first framework to provide a full implementation of the MapReduce paradigm, Hadoop is often criticized for a number of issues, first being its disappointing performance when used for running iterative tasks (see, e.g., [6]). A competing framework is gaining a lot of attention in the very recent years: Apache Spark [7]. It is a sort of evolution of Hadoop, but with some important differences allowing it to outperform its predecessor in many application scenarios. First of all, wherever there is enough RAM, Spark is able to perform iterative computations in-memory, without having to write intermediate data on disk, as required by Hadoop. In addition, it is more flexible than Hadoop, because it provides a rich set of distributed operations other than the ones required for implementing the MapReduce paradigm.

Indeed, one of the aspects that has the deepest impact on the performance of a distributed application, is the pattern used to distribute and process data among the different nodes of a network. This is especially the case of bioinformatics application, where even a single genomic sequence may be several gigabytes long. In this context, a poor data layout may prevent even an efficient algorithm to exploit the parallelism of a distributed system. From this viewpoint, the Spark feature that most marks the difference with respect to Hadoop is the availability of ready-to-use distributed data structures. These allow to manage and process in a standard and consistent way the data of an application while leaving to Spark the responsibility of partitioning this data and their elaboration. It is interesting to note that Spark offers three different types of distributed data structures. These are almost functionally identical and choosing which of them to use when developing a bioinformatics application may not be simple.

The goal of this paper is to investigate the complexity and the performance of the different distributed data structures offered by Spark, with the aim of providing useful hints to the bioinformatics community about which is the best option to choose, and when. This has been done by analyzing the three different solutions when used for the implementation of a typical sequence analysis algorithmic pattern: the counting of the distinct k-mers existing in an input sequence of characters. The three implementations we developed have been tested on a reference dataset to determine their relative performance and provide insightful hints about which of them to prefer when dealing with such a scenario.

Organization of the paper. The paper is structured as follows. In Sect. 2, we briefly discuss the current state of adoption of Big Data technologies for genomic computation. The Spark framework and the distributed data structures it offers are presented in Sect. 3. In Sect. 4, we state the objective of this paper and present the k-mer counting problem that has adopted as reference scenario for evaluating the different types of Spark distributed data structures. In Sect. 5 we outline the setting of our experiments and briefly discuss their results. Finally, in Sect. 6 we provide some concluding remarks for our work.

2 Related Work

The adoption of Big Data related technologies for accelerating the solution of bioinformatics problems has proceeded at a slow pace in the past years for several reasons. One of these is that the complexity of a framework like Hadoop adds to a distributed computation a significant amount of overhead, thus making it convenient only when processing enormous amount of data and/or when using distributed facilities counting hundreds or thousands of nodes. Instead, the same computation carried on a stand-alone workstation is able to exploit almost all the processing power of the underlying machine as the logic required to coordinate several concurrent processes running on the same machine is much simpler. Despite this, there are several relevant contributions worth to be mentioned.

One of the first and most noteworthy is GATK (see [8]). It introduces a structured programming framework designed to ease the development of efficient and robust analysis tools for next-generation DNA sequencers using MapReduce. A problem that often arises when writing an Hadoop sequence analysis application is the adaptation of the formats used for maintaining genomic sequences to the standard file format used by Hadoop. This problem has been addressed by Niemenmaa *et al.* in [9]. They proposed a software library for the generic and scalable manipulation of aligned next-generation sequencing data stored using the BAM format. The same problem has been further addressed by Massie *et al.* in [10]. In this case, the authors did not resort to an existing file format, like in [9], but proposed a new file format (i.e., ADAM) explicitly designed for indexing and managing genomic sequences on a distributed MapReduce system like Hadoop or Spark. There have also been several contributions about the usage of MapReduce and Hadoop for the solution of specific application problems. To name some, the work by Cattaneo *et al.* in [11,12] describes a MapReduce distributed framework based on Hadoop able to evaluate the alignment-free distance among genomic sequences using a variety of dissimilarity functions and in a scalable way.

The advent of Spark is slowly changing this scenario, as there is an increasing number of contributions developed using this technology and aiming at introducing solutions that are not only scalable but also efficient. This is the case of SparkSeq, a general-purpose, flexible and easily extendable library for genomic cloud computing presented in [13]. It can be used to build genomic analysis pipelines and run them in an interactive way. Another work worth to be mentioned is the one described in [14]. It provides a comprehensive study on a set of distributed algorithms implemented in Spark for genomic computation adopting efficient statistical approaches. The main objective is the study of the performance of the proposed algorithms with respect to more traditional ones.

3 Spark

Spark is a framework for general-purpose distributed processing of Big Data. It is able to exploit the computational capabilities of several calculators at once, by providing an uniform and abstract view of these as a computing cluster. Spark

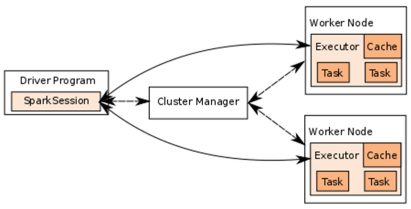


Fig. 1. Spark architecture

can be seen as a sort of evolution of Hadoop, as inherits the same MapReduce based distributed programming paradigm. In addition, it offers a wide range of ready-to-use operators and transformations that are often needed when developing a distributed application and that, although being possible with Hadoop, would require some work to be developed from scratch.

The Spark architecture (see Fig. 1) is composed by three main components: (a) the *driver program*, that is in charge to setup the Spark environment and launch the computation; (b) a *cluster manager* service, that is in charge of managing the distributed computation, assigning resources and scheduling the execution of one or more *tasks* on each node of the cluster; (c) several different *worker nodes*, in charge of carrying out the real computation, where each node is able to execute one or more tasks in parallel by spanning a corresponding number of *executors*. Notice that, apart from the cluster manager available with Spark, it is also possible to use third-party managers, such as Hadoop *YARN* [5].

3.1 The Programming Model

In a typical Spark application, the *driver program* begins the execution by loading the input data in a distributed data structure. This is essentially a collection of objects that is partitioned over the nodes of a cluster. Once data has been loaded, the execution proceeds by means of a sequence of distributed operations. Following the same *move computation close to data* philosophy that inspired Hadoop, Spark tries to run these operations directly on the nodes hosting the data that they are required to process. This is done to reduce the overhead that will be otherwise required to transfer big amount of data over the network for processing them elsewhere.

Distributed data structures available with Spark support two types of distributed operations: *Actions* and *Transformations*. The former may essentially be divided in three categories:

- **reduce:** apply a cumulative operation to the elements of a distributed collection of objects, so that multiple input objects are aggregated and combined in a single object belonging to an output distributed collection of objects;
- **collect:** gather all the objects of a distributed collection, or a subset of them, and send them to the driver program, where these are made available as a collection of local objects;

– **save:** writes the elements of distributed collection of objects on an external storage.

The latter may essentially divided in the three categories:

– **map:** map a distributed collection of objects in another distributed collection of objects. The new objects result from the application of a given function on each of the input objects;
– **filter:** filter the elements of a distributed collection of objects, returning a new distributed collection containing only elements satisfying an input-provided condition;
– **set operations:** combine two distributed collection of objects in a single one by means of a set operator.

The distributed part of an application run with Spark is logically divided in *stages*, where each stage corresponds to a transformation or an action. Stages related to transformations are run by Spark in a *lazy* way. This means that they are not run as soon as they are encountered during the execution of a program but only when and if their result is needed to accomplish a subsequent step of the application.

3.2 Distributed Data Structures

Spark provides three types of distributed data structures: *Resilient Distributed Dataset*, *DataFrame* and *DataSet*. These data structures share some relevant properties. First, they do all support in-memory computations. This means that, provided that there is enough memory space, their content may be partially or entirely cached in memory. This is especially useful when executing subsequent or iterative tasks targeting the same data. If the available memory is not enough, as when processing very large amount of data, their content may be selectively spilled to disk and retrieved in memory when required. The developer can choose also if and how to replicate this data, so to make the computation resilient with respect to hardware or network faults (see [7] for examples).

Resilient Distributed Dataset. The Resilient Distributed Dataset (in short, `RDD`) has been the first type of distributed data structure available with Spark. It is a virtual data structure encapsulating a collection of object-oriented datasets spread over the nodes of a computing cluster. The object-oriented nature of these datasets implies all the advantages and the disadvantages of this paradigm. For instance, it is the developer that chooses how the data stored in a `RDD` can be processed, by defining some proper methods on the objects storing that data. `RDD`s can be created by importing a dataset from an external storage or from the network, by issuing some special-purpose functions provided by Spark for making distributed a local dataset or as the result of the execution of a transformation over another `RDD`.

`RDD`s have also some drawbacks. For example, every time there is need to transfer elsewhere the content of a `RDD` (e.g. when performing a reduce operation), Spark has to marshall and encode, one-by-one, all the elements of that

RDD as well as their associated metadata. The reverse of this operation, then, has to be performed on each node receiving those elements. Similarly, whenever the content of a RDD is destroyed, the underlying java virtual machine has to claim back the memory used by each of the objects contained in the RDD. Since RDDs are often used to maintain collections counting millions or billions of elements, this overhead may severely burden the performance of a Spark application.

DataFrame. The DataFrame is a distributed data structure introduced in Spark to overcome some of the performance issues of RDDs. Instead of using a collection of objects, DataFrames maintain data in a relational-database fashion, providing a flat table-like representation supported by the definition of a schema. This has several important advantages. First, manipulation of large amount of data can be carried out using an SQL-like engine rather than requiring the execution of methods on each of the element to be processed. Second, by avoiding the usage of objects for storing the individual elements of a collection, the transmission of a chunk of a DataFrame to a node tends to be very fast. Third, since the meta-data describing the elements of a collection are the same for all these elements and are known in advance, there is no need of transmitting them when moving parts of a DataFrame, thus achieving a substantial saving in communication time. Finally, the adoption of an SQL based approach to the processing of data allows for several optimizations (see [15]). Even DataFrames suffer of some serious drawbacks. To name one, the dismissal of the object-oriented approach in favor of the SQL-like engine makes the resulting applications less robust as it is virtually impossible for the compiler to verify the type-safety of an application.

DataSet. The DataSet is a distributed data structure introduced to mix the best of the two previous technologies by guaranteeing the same performance of DataFrames while allowing to model data after the object oriented paradigm, as when using RDDs. This is mainly achieved thanks to two solutions. The first is the introduction of a new *encoding* technology able to marshall quickly and in a step a collection of objects. We recall that RDD need to marshall individually each object of a collection by means of the Java standard serialization framework. The second is the possibility of operating on the elements of a DataSet using an object-oriented interface while retaining their internal relational representation. On a side, this allows to perform the safety checks at compile time, thus making the applications more robust. On the other side, this allow to maintain all the performance advantages introduced with DataFrames.

4 Objective of the Paper

The three types of distributed data structures available with Spark, as well as the wide range of transformations and actions they provide, allow to write complex distributed applications in a few lines of code and without requiring advanced programming skills. This is a relevant feature as, typically, one of major issues preventing from using a distributed approach to solve a problem is the time and the cost required to develop such a solution. However, this simplicity comes at a cost. By delegating to Spark most of the work about how to organize and process distributed data structures, the developer takes the risk of sacrificing the efficiency of his code.

In this paper, we deal with this problem by focusing on assessing the performance trade-offs related to the choice of the distributed data structure type among the three offered by Spark, when developing a bioinformatics application. We use as a case study a simple problem that is fundamental when performing genomic sequence analysis: the k-mer counting problem.

4.1 The k-mer Counting Problem

Given a string S, we denote with term *k-mer* all the possible substrings of S having size k. The k-mer counting problem refers to the problem of counting the number of occurrences of each *k-mer* k in S. It is a very common and (apparently) simple task that is often used as a building block in the development of more complex sequence analysis applications such as genome assembly or sequence alignment (see, e.g., [16]).

The problem of counting the k-mers of a sequence is paradigmatic with respect to the class of problems that would benefit from the adoption of a distributed solution. On a side, it is apparently easy to solve as its algorithmic formulation is very simple and straightforward. This simplifies as well its distributed reformulation, as it represents a typical case of an embarrassingly parallel problem. On the other side, real-world scenarios often require to process either a huge number of sequences or sequences having a huge size (i.e., gigabytes of characters). Consequently, there is both a time-related problem (i.e., processing huge amount of data using a single machine could require days or weeks) and a memory-related problem (i.e., the memory required to keep the k-mers counts may span also tens or hundreds of gigabytes when using large values of k and huge sequences). The convenience of this approach is also witnessed by the several scientific contributions proposed so far (see, e.g., [11,12,17,18]), introducing clever solutions for counting k-mers in a parallel or distributed setting.

5 Experimental Study

In our experimental study we first developed three different solutions to the k-mer counting problem using Spark. These solutions are identical in their output, provided the same input, but differ in the distributed data structures they use. Then, we performed a comparative experimental analysis of these codes by measuring their performance when run on a reference testing dataset.

5.1 The Proposed Implementations

We report in Listings 1.1, 1.2 and 1.3 the pseudo-code of our three implementations (full source code not reported and available upon request): `RDD`, `DataSet`

Listing 1.1. Pseudo-code of k-mer counting implemented using Resilient Distributed Datasets

```
1     input = readTextFile(filename);
2     kmers = input.flatMapToPair(new KMerExtractor());
3     kmers_count = kmers.reduceByKey(new KMerAggregator()
          );
4     writeFile(kmers_count);
```

and `DataFrame`. As already said, the three solutions are equivalent, except for the particular type of distributed data structure used by each of them.

The first solution (Listing 1.1) uses a `RDD` to collect all the string lines of an input file, where each line corresponds a different genomic sequence. Then, it applies to each line a map function, `KMerExtractor`, that scans it returning all the k-mers it contains as a `RDD` of pairs $(k\text{-}mer, 1)$ (line 2). All these pairs are aggregated by the `KMerAggregator` reduce function (line 3), thus returning a `RDD` containing the final counts. The result is saved to file (line 4).

The second solution (Listing 1.2) extracts the k-mers from an input file as the first solution (line 1–2). Then, it builds a new schema definition, needed to establish the structure of the `DataFrame` used for storing the k-mers (line 3). Then, a new `DataFrame` is created using this definition and the collection of extracted k-mers (line 4). Once the `DataFrame` is ready, it is queried through an SQL query (line 5–6) for the k-mer counts. The result is saved to file (line 7).

The third solution (Listing 1.3) mimics the second one, but without the need of defining an explicit schema. In details, it first extracts k-mers from an input file as in the previous cases (line 1–2). The results of the extraction is saved in a Dataset. Its schema is automatically determined according to the data type of the k-mers. Then, it is queried (line 3) by running some of the standard methods available with this data structure (i.e., `groupBy` and `count`), instead of using an SQL query. The k-mer counts resulting from the query is saved to file (line 4).

Listing 1.2. Pseudo-code of k-mer counting implemented using DataFrames

```
1     input = readTextFile(filename);
2     kmers = input.flatMap(new KMerExtractor());
3     schema = CreateNewSchema(schema definition);
4     createDataFrame("kmers", schema, kmers);
5     String q = "select kmer, count(kmer) as count from
          kmers group by kmer";
6     kmers_count = spark.sql(q);
7     writeFile(kmers_count);
```

Listing 1.3. Pseudo-code of k-mer counting implemented using DataSets

```
    input = readTextFileasDataset(filename);
    kmers = input.flatMap(new KMerExtractor());
    kmers_count = kmers.groupBy("kmer").count();
    writeFile(kmers_count);
```

5.2 Dataset

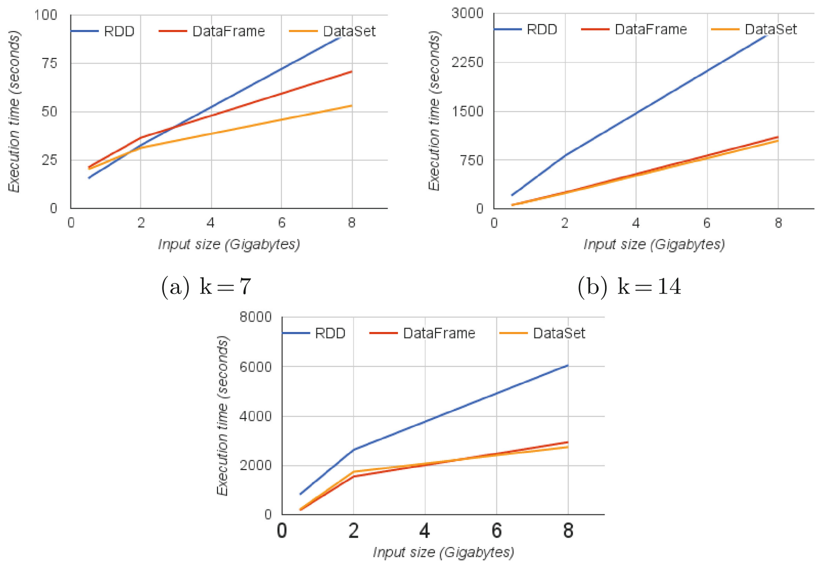
The experiments have been conducted on a dataset of four randomly-generated FASTA [19] files of increasing size. Each file has been generated as a collection of short-sequences, with each sequence being introduced by a text comment line and containing at most 100 characters drawn from the alphabet $\{A, C, G, T\}$. The overall size of the used files is, respectively, of about: 512 MB, 2 GB, 8 GB. These sizes have been chosen to represent the class of problems that are difficult to manage with a sequential approach and would benefit of a distributed solution.

5.3 Configuration

Our experiments have been conducted on a five-nodes Hadoop cluster, with one node acting as *resource manager* for the cluster and the remaining nodes being used as worker nodes. Each node of this cluster is equipped with a 16-core Intel Xeon E5-2630@2.40 GHz processor, with 64 GB of RAM. During the experiments, we varied the number of executors on each node from 1 to 4, to assess the scalability of the proposed solutions. Moreover, we organized input files in blocks having size at most 64MB, with each block available on two different nodes of the cluster. Such configuration allows for a better distribution of the workload but without affecting the performance of the whole system.

5.4 Results

In our first experiment, we have measured the performance of `RDD`, `DataFrame` and `DataSet` when run on sequences of increasing size and using increasing values of k. Its purpose has been to analyze the behavior of the three types of distributed data structures in a context where the size of these data structures could exceed the RAM memory available to a node. The experimental result, reported in Fig. 2, shows that when dealing with very small sized problems (i.e., size = 512 MB, k = 7) `RDD` is the implementation achieving the best performance. We recall that in this setting the number of possible distinct k-mers is very small. Consequently, `RDD` has to manage a very small number of objects. As soon as the size of the problem increases, the performance of this implementation quickly deteriorates because of the too many k-mers to be handled. Instead, the other two implementations exhibit an increase in their execution time that is linear with respect to the size of the problem. This is clearly due to their different strategy used to maintain k-mers in memory, that reveals to be much more



(a) k = 7 (b) k = 14

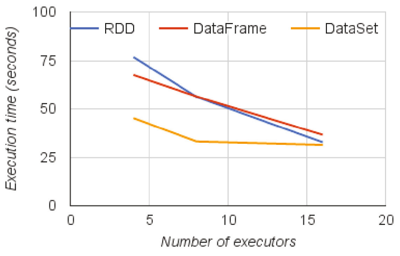
(c) k = 28

Fig. 2. Execution time, in seconds, of `RDD`, `DataFrame` and `DataSet` when processing random sequences of increasing size under different assignments of k

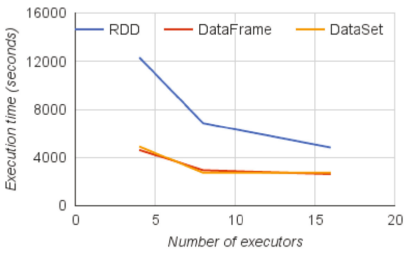
efficient when the number of k-mers to manage increases. We notice also that `DataSet` performs slightly better than `DataFrame`, mostly because it is able to encode k-mers faster (see Sect. 3.2).

In our second experiment, we have measured the scalability of the three considered implementations when run on a small problem instance and on a large problem instance using a cluster of increasing size. The two cases are representative of a scenario where the distributed data structures are either small enough to fit in the main memory or large enough to require their partial backup on external memory. The increasing size of the cluster has been simulated by increasing the number of executors per node (see Sect. 3), for an overall number of 4, 8 and 16 executors.

The experimental results on the small problem instance dataset, reported in Fig. 3, confirm that `DataSet` is the fastest of the three implementations. However, we notice that the scalability of `RDD` is much better. As expected, this phenomenon is due to the fact that, for such a small dataset, the usage of a high number of executors allows `RDD` to keep all the k-mers counts in memory, thus becoming competitive with the other two implementations. For the same reason, `RDD` enjoys a linear speed-up proportional to the number of executors. Instead, the performance of `DataSet` offers small room for improvement, as there is no noticeable gain when switching from 8 executors to 16 executors. Speaking of the large problem instance, we observe that none of the three codes is able to scale linearly with the number of executors. This may be explained by considering the



(a) 2 GB 7-sized k-mers



(b) 8 GB 28-sized k-mers

Fig. 3. Scalability of `RDD`, `DataFrame` and `DataSet` on a cluster with an increasing number of executors, when extracting k-mers from: (a) a 2 GB random sequences with k = 7; (b) a 8 GB random sequences with k = 28;

I/O bound nature of the k-mer counting activity, that becomes more evident when processing very large files. In such a scenario, most of the time is spent reading data from the external memory. Running several executors on the same node implies that they will contend the access to the disk when trying to read at the same time their respective input blocks, thus preventing the possibility of fully exploiting their computational resources.

6 Conclusion

The objective of this work has been to assess how the choice of the particular type of distributed data structure to be used for implementing a sequence analysis application with Spark affects its performance. We observed that three variants of the same code (a k-mer counting algorithm), having an identical behavior and undistinguishable in their output, but using different types of distributed data structures, exhibit very different performance. A direction worth to be investigated would be the analysis of more complex sequence analysis application patterns. This would allow to better assess the architectural peculiarities of the different types of the Spark distributed data structures. Moreover, given the internal complexity of Spark and the availability of a large number of settings influencing its performance, another promising direction would be to repeat these experiments on a larger scale and under a much broader range of configurations.

References

1. Pop, M., Salzberg, S.L.: Bioinformatics challenges of new sequencing technology. Trends Genet. **24**(3), 142–149 (2008)
2. Schuster, S.C.: Next-generation sequencing transforms today's biology. Nature **200**(8), 16–18 (2007)
3. Sanger, F., Nicklen, S., Coulson, A.R.: DNA sequencing with chain-terminating inhibitors. Proc. Natl. Acad. Sci. **74**(12), 5463–5467 (1977)

4. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
5. Apache: Hadoop. http://hadoop.apache.org/
6. Zhang, Y., Gao, Q., Gao, L., Wang, C.: iMapReduce: a distributed computing framework for iterative computation. J. Grid Comput. **10**(1), 47–68 (2012). http://dx.doi.org/10.1007/s10723-012-9204-9
7. Apache: Spark. http://spark.apache.org/
8. McKenna, A., Hanna, M., Banks, E., Sivachenko, A., Cibulskis, K., Kernytsky, A., Kiran, G., Altshuler, D., Gabriel, S., Daly, M., DePristo, M.A.: The genome analysis toolkit: a MapReduce framework for analyzing next-generation DNA sequencing data. Genome Res. **20**(9), 1297–1303 (2010). http://genome.cshlp.org/content/20/9/1297.abstract
9. Niemenmaa, M., Kallio, A., Schumacher, A., Klemelä, P., Korpelainen, E., Heljanko, K.: Hadoop-BAM: directly manipulating next generation sequencing data in the cloud. Bioinformatics **28**(6), 876–877 (2012)
10. Massie, M., Nothaft, F., Hartl, C., Kozanitis, C., Schumacher, A., Joseph, A.D., Patterson, D.A.: ADAM: Genomics formats and processing patterns for cloud scale computing. University of California, Berkeley Technical report, No. UCB/EECS-2013 207 (2013)
11. Cattaneo, G., Ferraro-Petrillo, U., Giancarlo, R., Roscigno, G.: Alignment-free sequence comparison over Hadoop for computational biology. In: Proceedings of 44th International Conference on Parallel Processing Workshops, ICPPW, pp. 184–192 (2015)
12. Cattaneo, G., Ferraro-Petrillo, U., Giancarlo, R., Roscigno, G.: An effective extension of the applicability of alignment-free biological sequence comparison algorithms with Hadoop. J. Supercomputing. **73**(4), 1467–1483 (2017)
13. Wiewiórka, M.S., Messina, A., Pacholewska, A., Maffioletti, S., Gawrysiak, P., Okoniewski, M.J.: SparkSeq: fast, scalable, cloud-ready tool for the interactive genomic data analysis with nucleotide precision. Bioinformatics (2014)
14. Bahmani, A., Sibley, A.B., Parsian, M., Owzar, K., Mueller, F.: SparkScore: leveraging apache spark for distributed genomic inference. In: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops, pp. 435–442, May 2016
15. Xin R., R.J.: Project tungsten: Bringing Spark closer to bare metal. https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html
16. Giancarlo, R., Rombo, S.E., Utro, F.: Epigenomic k-mer dictionaries: shedding light on how sequence composition influences in vivo nucleosome positioning. Bioinformatics (2015)
17. Deorowicz, S., Kokot, M., Grabowski, S., Debudaj-Grabysz, A.: KMC2: fast and resource-frugal k-mer counting. Bioinformatics **31**, 1569–1576 (2015)
18. Ferraro Petrillo, U., Roscigno, G., Cattaneo, G., Giancarlo, R.: FASTdoop: a versatile and efficient library for the input of FASTA and FASTQ files for MapReduce Hadoop bioinformatics applications. Bioinformatics (2017). https://dx.doi.org/10.1093/bioinformatics/btx010
19. Wikipedia: FASTA format – Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/FASTA_format