DNA short read alignment on apache spark

Maryam AlJame and Imtiaz Ahmad
Department of Computer Engineering, Kuwait University, Kuwait City, Kuwait

Received 6 February 2019
Revised 19 April 2019
Accepted 24 April 2019

Abstract

The evolution of technologies has unleashed a wealth of challenges by generating massive amount of data. Recently, biological data has increased exponentially, which has introduced several computational challenges. DNA short read alignment is an important problem in bioinformatics. The exponential growth in the number of short reads has increased the need for an ideal platform to accelerate the alignment process. Apache Spark is a cluster-computing framework that involves data parallelism and fault tolerance. In this article, we proposed a Spark-based algorithm to accelerate DNA short reads alignment problem, and it is called Spark-DNAligning. Spark-DNAligning exploits Apache Spark 's performance optimizations such as broadcast variable, join after partitioning, caching, and in-memory computations. Spark-DNAligning is evaluated in term of performance by comparing it with SparkBWA tool and a MapReduce based algorithm called CloudBurst. All the experiments are conducted on Amazon Web Services (AWS). Results demonstrate that Spark-DNAligning outperforms both tools by providing a speedup in the range of 101–702 in aligning gigabytes of short reads to the human genome. Empirical evaluation reveals that Apache Spark offers promising solutions to DNA short reads alignment problem.

Keywords Cloud, Cluster, DNA, MapReduce, Spark

Paper type Original Article

1. Introduction

Bioinformatics is an emerging field with applications in protein structure prediction, sequence alignment, drug design, gene finding and more. Within the recent years, biological data volume has increased enormously. Bioinformatics uses computational tools and applications to organize and analyze the massive amount of data sets that are produced from high-throughput biological studies. Thus, bioinformatics combined with math, computer science, and statistics helps to develop algorithms that solve practical problems for managing and analyzing vast amounts of biological data.

Discovered by two biologists, James Watson and Francis Crick in 1953, deoxyribonucleic acid (DNA) is the blueprint of life as it encodes genetic instructions [1]. Two twisted paired strands construct the DNA, each strand consisting of a sequence of four nucleotide base: adenine (A), cytosine (C), guanine (G), and thymine (T) [2]. Within the nucleus of living cells,

The authors would like to thank unanimous reviewers for their valuable comments and suggestions in improving the quality of the article. Thanks are also extended to Dr. Paul Karlsrud for his immense help in running the SparkBWA tool.

Publishers note: The publisher wishes to inform readers that the article "DNA short read alignment on apache spark" was originally published by the previous publisher of *Applied Computing and Informatics* and the pagination of this article has been subsequently changed. There has been no change to the content of the article. This change was necessary for the journal to transition from the previous publisher to the new one. The publisher sincerely apologises for any inconvenience caused. To access and cite this article, please use Aljame, M., Ahmad, I. (2019) "DNA short read alignment on apache spark", *Applied Computing and Informatics*. Vol. ahead-of-print No. ahead-of-print. https://10.1016/j.aci.2019.04.002. The original publication date for this paper was 26/04/2019.

Applied Computing and Informatics
Vol. 19 No. 1/2, 2023
pp. 64-81
Emerald Publishing Limited
e-ISSN: 2210-8327
p-ISSN: 2634-1964
DOI 10.1016/j.aci.2019.04.002

there are chromosomes that packaged the DNA. In a nucleus, the collection of DNA of all chromosomes is defined as genome.

Sequencing techniques are used by biologists to study the genomic sequence. One of the sequencing techniques is Next Generation Sequencing (NGS). In a single run, NGS can generate terabytes of data, this increased in throughput has exceeded Moore's Law [3]. Recent experiments have shown that the NGS's throughput is increasing 3-5x every year [4]. Short reads are obtained from NGS experiments. Mapping short reads to a long reference genome with a minimum number of mismatches is a challenge which brings forth a problem called short reads alignment. Such type of mapping is a computationally expensive operation since it requires matching a huge amount of short reads across an immense reference genome [5]. Next Generation Sequencing technique introduces advanced challenges in short reads alignment [6]. Numerous tools and algorithms have been developed to overcome NGS computational challenges and hurdles. Such tools and algorithms battle to improve accuracy, increase mapped read rate and better memory usage by decreasing memory consumption whilst increasing throughput [2]. Mainly, hash tables method and Burrows-Wheeler Transform (BWT) alignment are the most common algorithms used to solve the short reads alignment problem. BWT alignment is based on FM-index. Both hash tables and BWT are implemented on different parallel platforms to achieve hardware acceleration. Some of the parallel platforms are Graphics Processing Units (GPUs), multi-processors, and specialized acceleration devices such as Field Programmable Gate Arrays (FPGAs).

BWT has been implemented on a few hardware platforms. On the flipside, several software tools based on BWT have been developed to solve NGS reads alignment problem, such as Burrows-Wheeler Alignment (BWA) [7]. The development of BWA-mem [8] is meant to overcome the difficulty faced by the BWA tool in regards to gaps and indels challenges. Moreover, Bowtie [9] uses FM-index search while its subsequent Bowtie2 [10] uses dynamic programming to add the ability to count gaps. Further, a key implementation in parallel platform is SparkBWA [11] which is an example of BWA on Apache Spark.

Hash tables method has been used widely in read alignment tools and has been implemented in both software tools and on different hardware platforms. Software tools include: The Basic Local Alignment Search Tool (BLAST) [12], RMAP [13], SeqMap [14], CloudBurst [15] which is a distributed RMAP version, etc. It is worth mentioning that CloudBurst is based on the parallel platform Apache Hadoop.

Though NGS throughput is growing rapidly in addition to all the above mentioned implementations, biologists still need solutions with performance optimizations including accelerating short reads alignment process and an ability to run it efficiently with less power consumption and less memory utilization. To accomplish such performance optimizations, in this article, Apache Spark has been used to solve DNA short read alignment problem. Apache Spark is a promising cluster-computing framework that introduces Resilient Distributed Datasets (RDDs) which have data parallelism and fault tolerance implicitly. In this article, an algorithm based on Apache Spark, called Spark-DNAligning, has been developed and tested to effectively solve the DNA short read alignment problem. Spark-DNAligning results demonstrate the benefits of utilizing Apache Spark as it improves performance, in comparison to SparkBWA and one of Apache Hadoop implementations, CloudBurst.

The remainder of the article is organized as follows: Section 2 discusses problem formulation and presents the related work. The architecture of Apache Spark platform is described in Section 3. Section 4 gives an overview of Spark-DNAligning, a novel Spark algorithm to deal with DNA short read alignment. The experiments conducted and the performance evaluation results of the proposed algorithm are reported in Section 5. Finally, conclusions are described in Section 6.

2. Problem formulation and related work

The current section consists of two subsections. The first subsection describes DNA short read alignment problem and gives a brief history of it. The second subsection provides an overview of the related work on different parallel platforms.

2.1 DNA short read problem formulation

Next-generation sequencing (NGS), also known as high-throughput sequencing, is a term used to describe several sequencing technologies namely Illumina (Solexa), Roche 454, Ion torrent Proton/PGM, and SOLiD. Such recent sequencing technologies accelerate DNA and RNA sequencing. 1000 Genomes Project is an example of Next-generation DNA sequencing (NGS) project [16]. NGS projects have brought forth a new revolution as they have among other aspects, increased the capability to understand genetic variation among individuals, and have helped to characterize cancers at the genomic level. For instance, biologists and health care providers need DNA sequencing technologies in several applications including finding pathogenic genes, molecular cloning, and breeding [17]. To serve the intended purpose, DNA sequencing technologies should have various critical characteristics such as accurate, easy of use and fast execution time. As an example, Figure 1 illustrates Whole-genome sequencing (WGS) workflow. The first step is the sequencing step where a sequencer sequences the human genome by fragmenting it into numerous short chunks named short reads [18]. The sequencing step generates plenty of short reads, almost billions in number. In particular, sequencers generate lengthy short reads ranging between 100 and 500 bp [19]. As shown in Figure 1, the next step is aligning all the short reads to a reference genome [18]. Some short reads will be matched exactly to the reference genome while other short reads will have some mismatches. For example, the short read "GGTCTGGATGC" will be aligned to index three with one mismatch. It is clearly illustrated in Figure 1 where the mismatch has a bold character, seen in the character "G" as it should be "A".

The main concerns in DNA short read alignment problem are execution time and memory usage. Fortunately, the two concerns can be handled effectively on parallel platforms such as Apache Hadoop, Apache Spark, GPU based implementations, as well as using Field Programmable Gate Arrays (FPGAs) device. Recent studies assert that parallel platforms are promising solutions for DNA short read alignment problem.

2.2 Related work

Recently, in bioinformatics, cloud computing and parallel platforms are considered new emerging promising solutions to handle critical issues such as computation and memory footprint [20]. MapReduce is a computing framework developed by Google whose execution is parallel on a large cluster of commodity machines in a significantly scalable manner [21]. MapReduce is available on different implementations such as Phoenix, Hadoop, Mars and

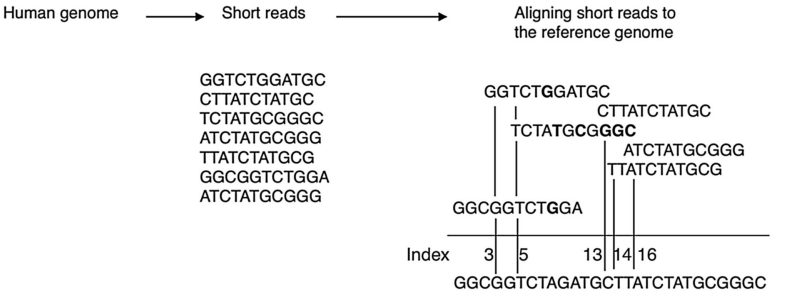


Figure 1.
Whole-genome sequencing workflow.

many others [22]. MapReduce has been the key framework for many emerging tools in bioinformatics such as CloudAligner [22], CloudBurst [15], Crossbow [23], and SeqMapreduce [24]. CloudBurst is a parallel seed-and-extend read-mapping algorithm developed based on Hadoop MapReduce implementation. CloudBurst boasts of several added enhancements. The essential one is near linear parallel speedup which implies that CloudBurst running time scales linearly according to reads number [15]. Though it possesses various strong features as seen, CloudBurst is not without limitations. For example, because input files are in FASTA format, CloudBurst does not support many common formats in bioinformatics as FASTQ[25] format. In addition, in a study undertaken by Schatz (2009), another limitation reported is the failure to align 7 M reads with four mismatches to the human genome, reporting near 25 billion aligns before stopping due to lack of memory. SeqMapReduce is another Hadoop MapReduce mapping sequences tool. The main advantage SeqMapReduce has over CloudBurst is in-memory seed-and-extension and a late emission strategy [24]. Hence, the overall performance is improved by reducing the need to store the intermediate key/value pairs on the cluster as well as the reduction of communications overhead [24]. Furthermore, Crossbow uses Hadoop MapReduce to build a parallel cloud-computing tool that explores the benefits of Bowtie [23] and SNP caller SOAPsnp [7] in terms of speed and accuracy, respectively. CloudAligner is a Hadoop-based sequence alignment tool developed by Tung Nguyen et al. [22]. It has a parallel processing and it partitions the reference genome and reads in a way that improves performance. CloudAligner is highly scalable and has the ability to run longer reads in comparison to other existing tools, in addition to its support for many different input/output file formats. Furthermore, there are various Burrows-Wheeler aligner based tools that utilize Hadoop MapReduce implementations to enhance BWA performance. For example, Halvade [26], SEAL [27], and BigBWA [28]. BigBWA is a sequence alignment tool based on the original BWA source code, which means that BigBWA will be compatible with BWA updated versions. All BigBWA characteristics can also be found in SparkBWA [11]. The major difference is that instead of Hadoop, SparkBWA is based on Apache Spark. SparkBWA uses Java Native Interface (JNI) to call all BWA methods. Also, SparkBWA supports all BWA algorithms; BWA-backtrack [7], BWA-SW [29] and BWA-MEM [8]. StreamBWA is a Spark-based sequence aligning tool that explores Spark streaming to process the input data in real time on a cluster [30]. Unlike SparkBWA, StreamBWA just supports BWA-MEM algorithm. To enhance and boost other problems in Bioinformatics, there has been the development of many Spark-based tools as reported by Li et al. [31] including MetaSpark [32] and Sparkga [33]. Further, the survey [34] lists several Spark based applications and algorithms used in Bioinformatics field such as alignment, mapping, assembly, sequence analysis, drug discovery, and others.

3. Apache spark fundamentals

This section provides a background knowledge of Apache Spark and its fundamentals. It is organized as follows: the first subsection provides an overview of Apache Spark fundamentals. The second subsection discusses Spark programming model.

3.1 An overview of apache spark fundamentals

In the last decade, cluster computing frameworks have gained popularity as they provide impressive solutions to process big data. MapReduce [21] and Dryad [35] in particular are cluster computing frameworks that have been widely used to speedup parallel computations. One limitation though is that they are inefficient for reusing intermediate results among several computations. Reusing intermediate results is an important step in many algorithms such as machine learning algorithms and graph algorithms [36]. The

mentioned algorithms have emerging applications like K-means clustering, PageRank, and logistic regression. In fact, iterative algorithms and interactive data mining tools require in-memory computations to run efficiently. The two types of applications have motivated Spark's RDDs idea [37]. Apache Spark is an emerging cluster computing framework that proposes a novel distributed memory abstraction called Resilient Distributed Datasets (RDDs). Spark pipelines processing of tasks depicted by Directed Acyclic Graph (DAG) is extended from a bipartite MapReduce paradigm. RDDs empower Spark to outperform the prior cluster computing frameworks as RDDs support in-memory computations without the need to replicate data. Consequently, querying data in Spark is faster than Hadoop which is a disk-based engine. Certainly, by comparing Spark with the traditional Hadoop, the main speed gain is the way Spark handles intermediate computations [37]. First, Spark reads data from a distributed storage system. Then, Spark caches the data on local RAMs as persisted RDDs. To that effect, Spark uses RAMs effectively to hold intermediate computations results thereby efficiently improving performance by reducing overheads significantly. As shown in Figure 2, an additional feature is that Spark core provides task scheduling, memory management, basic I/O functionalities and fault recovery, services which can be exposed through all the supported APIs. Spark demands two requirements; a cluster manager and a distributed storage system. Currently, Spark supports three different cluster managers namely Standalone, Apache Mesos [38] and Hadoop YARN [39].

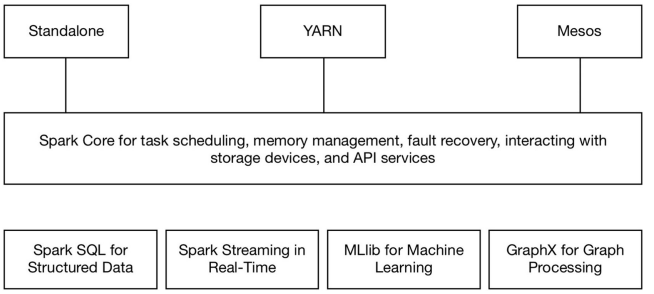


Figure 2.
Apache Spark software system core architecture.

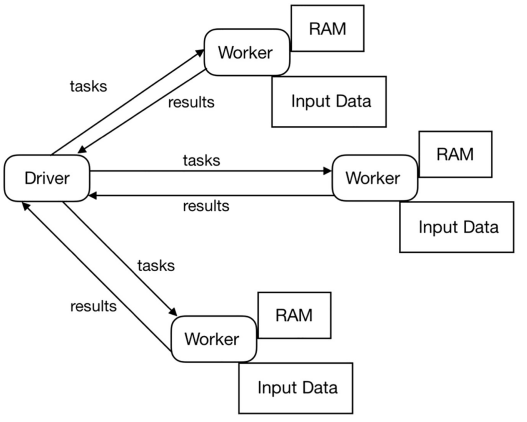


Figure 3.
Spark architecture.

Spark is built based on a master/slave architecture where it has one master referred to as a driver program and various executors (workers) as depicted in Figure 3. The driver program plays important roles and aside from its main functions, it also constructs RDDs with all their parallel operations (transformations and actions). In addition, the driver program has the responsibility to run all the said parallel operations on a cluster [37].

3.2 Spark programming model

To develop a Spark application, developers implement Spark application flow control on a high-level by writing a driver program [40]. In Spark, a driver program is written in Scala. Scala programming language is a functional programming language that is statically typed for the Java VM. In addition, other functional programming languages including Java, Python and R can also be used to write a Spark driver program. Operations in Spark applications are executed in parallel. In Spark, parallel programming is available in two fundamental abstractions: Resilient Distributed Datasets (RDDs), and parallel operations which are applied on RDDs [40]. A resilient distributed dataset (RDD) is one of the fundamental abstraction in Spark parallel programming. An RDD is an immutable collection of objects that is partitioned across all machines in the cluster and performs in-memory computations [36]. RDDs in Spark possess distributed shared memory advantages without latency issues [37]. Operations on RDDs are based on coarse-grained transformations and as a consequence, the same operation is applied to several data items at the same time. There are numerous parallel operations such as map, filer, join, reduce, and collect among others. Mainly operations on RDDs are of two types; transformations and actions. Transformations are lazily evaluated. Afterward, the transformation operations execute only when an action operation is triggered.

Spark operations are invoked by passing closures to them. As a functional programming concept, closures can access variables within closures' scope. As a rule, closure's needed variables are copied to the worker node. In some cases, that copy consumes a huge amount of resources in regards to memory and network traffic. To overcome such limitations, Spark offers two kinds of shared variables; Broadcast variables and Accumulators variables. Broadcast variables are read-only variables which are distributed and copied to the workers' nodes only once. Consequently, broadcast variables are very useful and helpful in a situation where multiple parallel operations need to use the same data without modifying it. In particular, a lookup table is commonly used as a broadcast variable.

In Apache Spark RDDs dependency occurs between parent RDD partitions and its child RDD partitions. There are two types of RDDs dependency; narrow dependency and wide dependency. Narrow dependency means each partition in the child RDD depends on at most one partition in the parent RDD. Wide dependency, on the other hand, means that each partition in the child RDD depends on several partitions in the parent RDD. In light of such facts, narrow dependency then becomes more effective than wide dependency.

4. Proposed algorithm

Spark-DNAligning algorithm is a new DNA short read alignment algorithm developed based on Apache Spark. This section provides a detailed explanation of the proposed algorithm referred to as the Spark-DNAligning algorithm. The first subsection gives an overview of the Spark-DNAligning workflow. The next subsection demonstrates Spark-DNAligning Algorithm Design.

4.1 Workflow of the proposed algorithm

This subsection explains the workflow of the proposed algorithm. Figure 4 depicts Spark-DNAligning Algorithm workflow. As shown, there are three main steps: (1) Indexing the reference genome, (2) DNA short reads preparation, and (3) Aligning. Almost all short reads

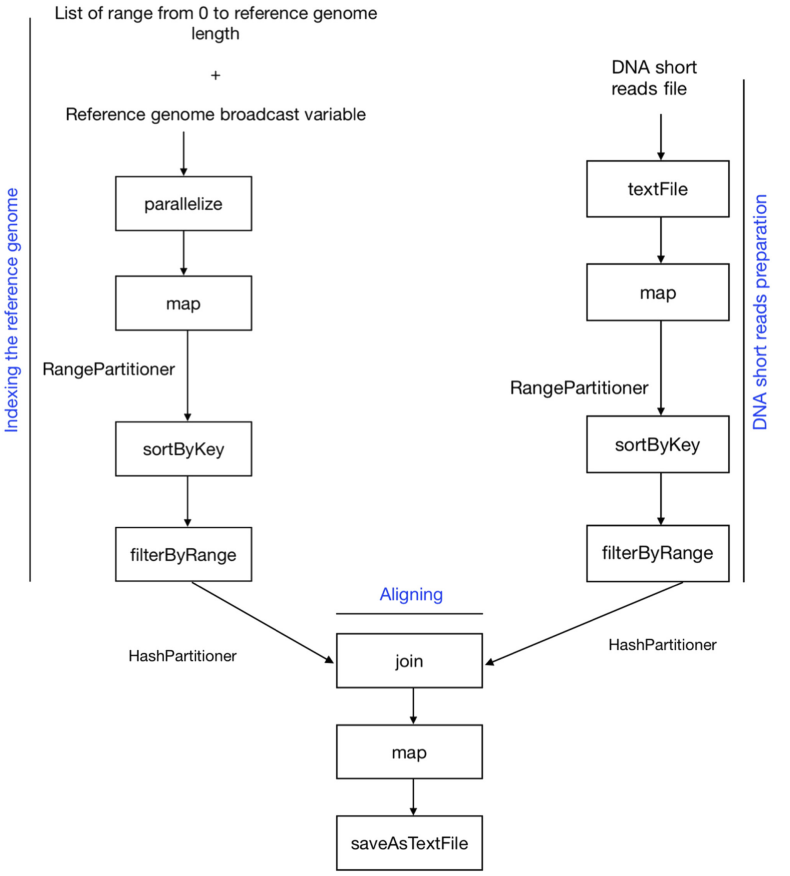
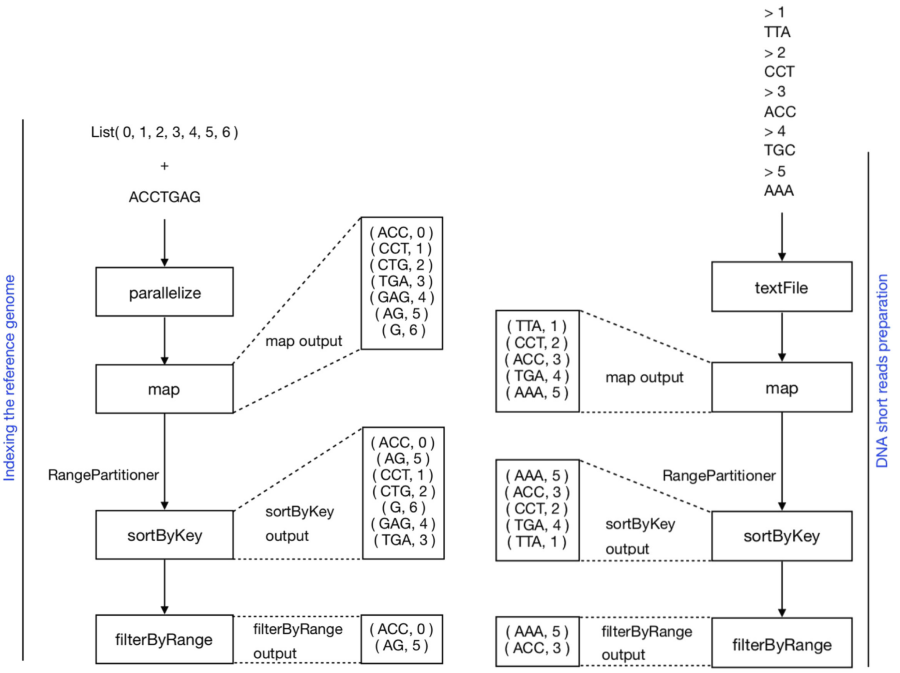


Figure 4.
Spark-DNAligning algorithm workflow.

alignment algorithms index the reference genome to speedup aligning steps. Each algorithm has its own way of indexing the reference genome or it simply loads the available index files that have been previously generated. Spark-DNAligning explores many Spark features to implement its own reference genome indexing steps. In Figure 4, the left side illustrates the indexing steps. First, Spark's parallelize method needs two inputs: (1) List of range starting from zero to DNA reference length, and (2) Reference genome broadcast variable. Figure 5a displays the indexing steps using a simple example. Assuming that the reference genome is "ACCTGAG", the List starts from zero up to n-1 where n is length of the reference genome. Thus, in the example, the List starts from zero up to 6. The map output represents the reference genome indexes. In Figure 5a for instance, index zero is represented by the pair (ACC, 0) which means the DNA segment "ACC" is located at index zero in the reference genome. More details regarding that are provided later. SortByKey sorts the pairs alphabetically in ascending order based on keys. Afterward, filterByRange in the given example in Figure 5a filters all pairs that start will (A) nitrogen base. The same filter is applied to the rest of DNA nitrogen bases C, G, and T.

DNA short reads preparation steps are shown on the right side of Figure 4. The first step is loading DNA short reads from a file by calling textFile Spark method. Figure 5b demonstrates a simple example with five DNA short read records. Each read has a length of



(a) Indexing the reference genome (b) DNA short reads preparation steps

Figure 5.
Spark-DNAligning algorithm.

three characters. As shown in Figure 5b, map output is a pair RDD of (DNA_read, DNA_read_ID). Afterwards, sortByKey and filterByRange work similar to indexing steps. Thus, as depicted in Figure 5b, the filterByRange output is two pairs (AAA, 5) and (ACC, 3) as in the example, filter is for (A) nitrogen base. The next step is aligning which is done by applying Spark's join method to the previously generated two RDDs is illustrated in Figure 6. In the given example, join output is (ACC, (3, 0)). It implies the short read "ACC" with ID 3 is located at index zero in the reference genome. Indeed, there is no need for the segment ACC because Spark-DNAligning users are able to identify it from DNA_read_ID which is 3 in the given example. Spark-DNAligning applies map transformation to omit DNA segment from

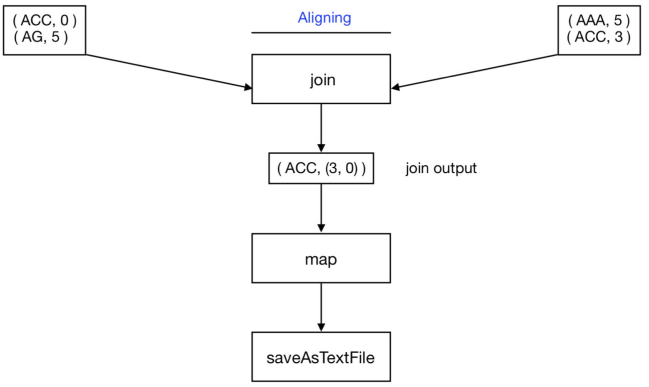


Figure 6.
Spark-DNAligning algorithm aligning steps.

all join output pairs for several reasons. Key among them is memory usage. In the final step, results are saved.

4.2 Spark-DNAligning algorithm design

This subsection provides a detailed description of the design and implementation of Spark-DNAligning Algorithm. The proposed algorithm accepts input files in FASTA format and output the results as a SAM file.

Algorithm 1 Spark-DNAligning Algorithm

INPUT: DNA reference file *ref*, DNA short reads *reads*, DNA short reads length *readsLen*, Number of partitions *partitionsNum*
OUTPUT: SAM file.
1: *ref* = sc.broadcast(load("Ref.fa")) ▷Load DNA reference from *ref.fa* file
2: *refLength* = get *ref* length
3: indexes = sc.parallelize(List.range(0, refLength), *partitionsNum*)
4: (DNA segment, location) ← map(indexes) ▷ Indexing the reference genome
5: reads = sc.textFile("s3n://path/to/shortReads.fa.bz2", *partitionsNum*)
6: (read, ID) ← map(reads) ▷*reads* preparation steps
7: sortedIndexes ← *indexesRDD*.partitionBy(RangePartitioner).sortByKey()
8: sortedReads ← *readsRDD*.partitionBy(RangePartitioner).sortByKey()
9: *indexesRDD*.unpersist() ▷cleaning step
10: *readsRDD*.unpersist() ▷cleaning step
11: **repeat**
12: baseIndexes ← sortedIndexes.filterByRange(from first record starting with *base* to last record starting with *base*)
13: baseReads ← sortedReads.filterByRange(from first record starting with *base* to last record starting with *base*)
14: ▷Aligning step
15: mappedReads ← baseReads.join(baseIndexes).map(to SAM format)
16: unmappedReads ← baseReads.map.subtractByKey(mappedReads).map(to SAM format)
17: ▷ Save the final results on Amazon S3
18: saveAsTextFile("s3://path/to/baseoutput")
19: baseIndexes.unpersist() ▷cleaning step
20: baseReads.unpersist() ▷cleaning step
21: baseresult.unpersist() ▷cleaning step
22: **until** for all nitrogen bases a, c, g, and t

Spark-DNAligning takes advantage of five key concepts in Apache Spark. The concepts are broadcast variable, pair RDD, sortByKey, filterByRange, and join. Broadcast variables are a valuable feature in Apache Spark as they provide a way to let all nodes (workers) in the

cluster access the same read-only variable in a handy distributable manner that decreases communications overhead. Apache Spark broadcast variable gains its value by means of caching a broadcast variable on each node in the cluster instead of carrying a copy of it with each task. Absolutely with a simple thought, in DNA short read alignment problem, all DNA reads need to be compared with the DNA reference. Thus, DNA reference is a perfect candidate to be a Spark broadcast variable. As shown in Algorithm 1 line 1, Spark-DNAligning Algorithm makes a broadcast variable of the DNA reference after loading it from a FASTA file. Consequently, DNA reference is available to all nodes (workers) in the cluster with less communications overhead.

After that, in Algorithm 1 line 2, the DNA reference length is calculated. The length is used to create a Scala collection of a range of numbers starting from zero to the DNA reference length as it is written in line 3, in Algorithm 1. As shown in Figure 7a, the collection is parallelized by calling SparkContext's parallelize method to compose a distributed dataset (indexes RDD) which executes in a parallel manner. The DNA reference indexes by applying map transformation on the previous generated RDD (indexes RDD). As mentioned in Spark documentation, map transformation applies a function to each dataset element then returns the results as a new RDD. So, in the proposed algorithm, map transformation is used to create a pair RDD of DNA segment as a key, and its location as a value (DNA_segment, location). DNA segment is generated by calling slice Scala function on the DNA reference broadcast variable. Slice Scala function takes two parameters: start index, and last index. It then returns a substring which contains only the characters at the start index through the last index. The last index of the slice function is equal to location plus short reads length in terms of the number of base pairs (bp). For example, assuming that DNA reference is "gctattgaatc", short reads bp is equal to 3 and location is equal to 7. Then, DNA_segment starts from the seventh character of the DNA reference and ends at character 10 since location plus short reads bp is equal to 7+3 which is equal to 10. Thus, the map iteration generates the pair element (aat, 7) where "aat" is the DNA_segment and is located at character 7 in the DNA reference. Figure 7a illustrates map transformation results in refIndexes RDD. The next step is to sort the key/value pairs based on key values by calling sortByKey( ) on refIndexes RDD. The result is the construction of a new RDD which is sortedIndexes RDD in Figure 7a.

DNA short reads are stored in Amazon S3, as illustrated in Figure 7b. RDD of the DNA short reads is constructed by calling SparkContext's textFile method followed by applying map transformation to construct a pair RDD with DNA short read as a key and DNA short read ID as a value (DNA_read, DNA_read_ID). Without a doubt, communication in a distributed system is really expensive. Accordingly, minimizing network traffic improves performance significantly. In Spark, to reduce communications, a developer has the option of controlling RDD partitioning. For that purpose, Spark-DNAligning takes partitions number as an input to enable users to tune partitions number smoothly. Apache Spark has two type of partitioning: Hash Partitioning, and Range Partitioning. In addition, developers can create custom partitioning to satisfy their application needs.

It is more efficient to use Range Partitioning with pair RDDs that have keys in a particular ordering. Consequently, elements with the same range of keys appear on the same machine in the cluster. Spark-DNAligning utilizes Range Partitioning by applying it on both refIndexes RDD and reads RDD before calling sortByKey( ) as depicted in Algorithm 1 in line 7 and line 8. Following that action, the resulting RDDs are persisted. On the other hand, refIndexes RDD and reads RDD are deleted from memory by applying unpersist( ) command to them. Actually, Spark uses least-recently-used (LRU) algorithm to delete old unused data. In Spark-DNAligning though, instead of waiting for Spark automatic memory cleaning, Spark-DNAligning calls unpersist( ) method manually. Such RDDs's persistence control enhances memory usage.

DNA consists of four nitrogen bases: adenine (A), guanine (G), cytosine (C) and thymine (T). For several benefits like memory usage and performance improvement, Spark-DNAligning

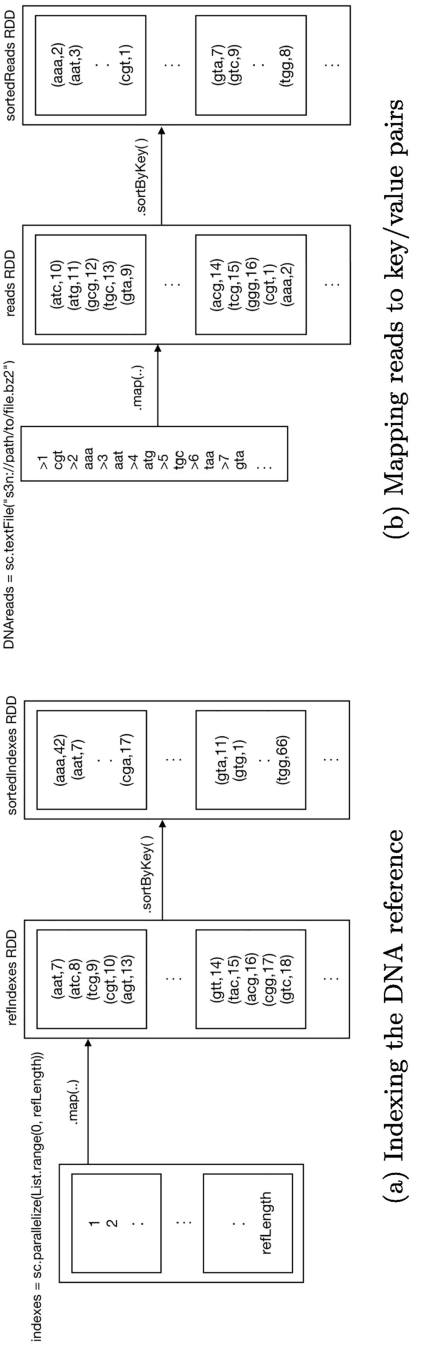


Figure 7.
Spark-DNAligning algorithm example.

algorithm aligns all DNA short reads that begin with adenine (A) first followed by short reads that begin with cytosine (C), guanine (G), and thymine (T), respectively. To achieve such aligning, Spark-DNAligning uses Spark filterByRange transformation with Scala regular expression. Algorithm 1 shows that filterByRange is applied on both of sortedIndexes RDD and sortedReads RDD. Spark filterByRange transformation takes two parameters: the lower key, and the upper key, constructing an RDD from only the elements within the given range. Spark-DNAligning exploits Scala regular expression to specify the range to filterByRange transformation. Regular expression helps in setting filterByRange lower key to the first read starting with one of the nitrogen bases, and sets filterByRange upper key to the last read starting with the same nitrogen base. Figure 8 shows adenine (A) nitrogen base case, in sortedIndexes and sortedReads. Adenine (A) nitrogen base range starts from the first record beginning with character A to the last record beginning with character A. In details, all records are sorted so the range starts from records which have "AA" prefix and ends with records which have "ATT" prefix. The same is witnessed for cytosine (C) base, guanine (G) base, and thymine (T) base. The final phase is joining. Spark join transformation takes two pair RDDs and merges pairs with identical keys. Figure 8 illustrates join in the Spark-DNAligning algorithm for adenine (A) nitrogen base. As mentioned earlier, the aIndexes RDD has DNA segment as a key while aReads RDD has DNA read as a key. As a result, joining identical keys performs the

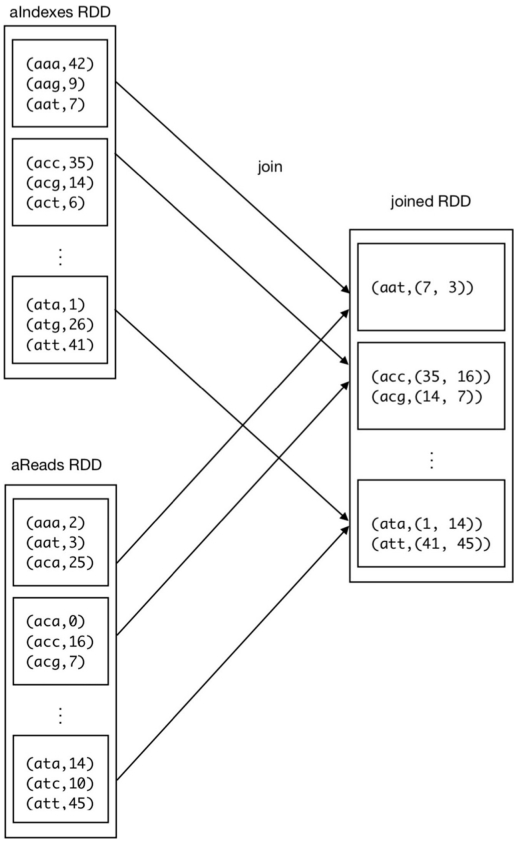


Figure 8.
Spark-DNAligning filterByRange and join for adenine (A) nitrogen base.

alignment and produces pair RDD with DNA_read as a key and the tuple (DNA_ID, location) as a value.

Before saving the aligning results, Spark-DNAligning applied map transformation to format the aligning results to SAM format. Further, the proposed algorithm used subtractByKey( ) to get the unmapped reads as written in Algorithm 1 line 15. The final results are saved on Amazon S3 by calling saveAsTextFile( ) Spark's action. Moreover, a performance improvement point to consider is join shuffling. In fact, join without co-partitioned inputs is a wide dependency which requires shuffling while join with co-partitioned inputs is a narrow dependency without shuffling. For that reason, Spark-DNAligning applies HashPartitioner for both RDDs before calling join as demonstrated in Figure 4. The partitioning dramatically improves Spark-DNAligning performance.

5. Performance evaluation

The section evaluates the proposed algorithm by comparing it with two different tools: SparkBWA [11] and CloudBurst [15]. SparkBWA is a Burrows-Wheeler Alignment Tool based on Apache Spark. SparkBWA includes all the three different types of BWA alignment algorithms: BWA-backtrack, BWA-SW and BWA-MEM. CloudBurst is a Hadoop based, read-mapping algorithm built according to the seed-and-extend algorithm. The first subsection explains the required setups. The second subsection demonstrates all the details of both datasets and machines that are used in the evaluation experiments. The last subsection compares Spark-DNAligning with SparkBWA and CloudBurst in term of execution time.

5.1 Spark-DNAligning setup

Spark-DNAligning has been evaluated on Amazon Elastic MapReduce (EMR), which utilizes the elastic infrastructure of Amazon EC2 and Amazon S3. EMR uses Amazon S3 for data storage, also it can use other types like Amazon DynamoDB. Amazon Elastic MapReduce has several advantages including reliability, security, flexibility, and low cost just to name a few. Moreover, EMR can execute many distributed frameworks such as Apache Spark, HBase, Flink, and Presto. By default, Elastic MapReduce uses Yarn cluster manager.

5.2 Datasets description and machines details

For the experiment, Spark-DNAligning is tested in terms of performance by comparing its execution time with the execution time for both of SparkBWA and CloudBurst. The data used in the testing performance experiment is retrieved from the 1000 Genomes Project [41]. Mainly, five different datasets are used in the testing experiments. The size of the first two datasets is within the range of megabytes while that of the remaining datasets is within the range of gigabytes. Table 1 lists the main characteristics of those input datasets.

Table 1.
Datasets characteristic.

Dataset name	Number of short reads	Short reads length (bp)	Size	Reference genome	Length (bp)
100k	100,000	36	4.4MB	s_suis	2,007,491
SRR001113	103,500	47	10MB	Chr22	50,818,468
ERR000589	23,928,016	51	3.61GB	Chr22	50,818,468
SRR062634	48,297,986	100	12.69GB	Chr19	58,617,616
SRR642648	197,502,074	100	51.88GB	Chr19	58,617,616

For the first two datasets which are in megabytes, Spark-DNAligning testing performance experiments are ran on EMR Yarn cluster with four Amazon EC2 instances, one master node, and three worker nodes. Amazon EC2 instances are from general purpose family of type m4.xlarge. The characteristics of m4.xlarge is 4 vCPU and 16 Memory (GiB). Thus, the total number of cores in the cluster is 16 cores. On the other hand, all the gigabytes datasets the last three rows in Table 1, are ran on EMR Yarn cluster with four nodes (workers) and one master node which places the total nodes in the cluster at five. All nodes are of type m4.16xlarge which have the following characteristics 64 vCPU and 256 Memory (GiB). Thus, the cluster has 320 cores in total.

In Apache Spark, it is commonly known that number of partitions affects performances significantly. Therefore, for each dataset, Spark-DNAligning was ran using a different number of partitions. After analyzing the effect arising from varying the number of partitions, the best partitions number is picked for each dataset. In Apache Spark, there is no straight rule that sets or governs the number of partitions. However, there are some conventions which are commonly used in Apache Spark community. Some of the conventions are; the minimum number of partitions is equal to the number of cores in the cluster, then multiply it by 1.5 until the performance stops improving. To that end, Spark-DNAligning has been run on several numbers of partitions. Table 2 lists the execution time with different number of partitions for ERR000589, SRR062634, and SRR642648. Spark-DNAligning has been run on a cluster of 320 cores. Thus, setting number of partitions to 320 activates all cores in the cluster and prevents them to be idle. Table 2 shows that the execution time for ERR000589 short reads with 480 partitions is faster than 720 partitions by 13.611 s and 320 partitions by 12.106 s. For SRR642648 short reads case, the fastest execution time is reached with 480 partitions also. On the other hand, the case of SRR062634 short reads has the fastest execution time with 320 partitions which is 392.408 s. As illustrated in Table 2 ERR000589 with 480 partitions needs 404.636 s to complete the job. Actually, the difference between the execution time with 320 and 480 partitions is just 12.228 s, which is a small difference. Accordingly, Table 2 indicates that 480 partitions is the optimum number of partitions for the cluster in the testing experiments.

5.3 Performance evaluation

In order to evaluate the performance of the proposed algorithm, the aforementioned five datasets have been used to run the three tools: Spark-DNAligning, SparkBWA and CloudBurst. After which their execution times have been analyzed and compared. Regrettably, for the largest two datasets SparkBWA failed to complete the execution on the current testing environment. Although, it is successfully completed the job for the two datasets as reported [11]. However, the testing environment in [11] is bigger than the available environment in this article. For CloudBurst cases, since it is based on Apache Hadoop, it becomes necessary to set the proper number of mappers and reducers. In the testing experiments, there are 3200 mappers and 640 reducers. Worth noting is that both CloudBurst and Spark-DNAligning have a preprocessing stage. However, SparkBWA doesn't demand a preprocessing stage. In fact, input files are prepared in the preprocessing

Table 2. Execution time (seconds) with different numbers of partitions.

Dataset name	Number of partitions		
	320	480	720
ERR000589	366.951	354.845	368.456
SRR062634	392.408	404.636	421.257
SRR642648	691.028	678.668	685.246

stage which means that the stage requires time. For comparison fairness, the preprocessing stage time has been included. All the experiment results, details and numbers are listed in Table 3 and Table 4. The performance evaluation reveals several conclusions. Spark-DNAligning shows a perfect scalability with the number of short reads. For instance, the biggest dataset (SRR642648) which has 197,502,074 short reads, Spark-DNAligning needs 678.668s to complete the alignment. For the smallest dataset (100k) with 100,000 short reads, Spark-DNAligning requires 64.499s. Hence, increasing the short reads by 197,402,074 just increased the execution time by 614.169s. On the other hand, for the smallest and the biggest datasets, CloudBurst requires 453.373, and 2,305.023 s respectively. Thus, the execution time increased by 1851.65s. Results confirm the great improvement of in-memory computations which is a core fundamental in Spark. Subsequently, Spark with its in-memory computations is a well competitive for Hadoop-based tools as CloudBurst in this experiment. Furthermore, Table 3 verifies that Spark-DNAligning outperforms SparkBWA in all cases with a minimum speedup of 101.417 in SRR001113 case. According to the results, Spark-DNAligning has a better overall performance in comparison to SparkBWA and CloudBurst. To cement the benefits of the proposed algorithm, it is worth noting that Spark-DNAligning recorded the speedup ranges from 101 to 702.

6. Conclusions

Recently, DNA sequencing speed has increased dramatically in most Next Generation Sequencing (NGS) applications and because of it, a massive amount of data is generated from DNA sequencing. That massive amount of data requires a scalable tool that has the ability to perform computations on a high-performance level. Additionally, sequencing tools need to be fast without using a large memory size. In fact, parallel platforms are promising solutions to address all the aforementioned needs. In this article, Apache Spark has been used to solve DNA short read alignment problem. Apache Spark is a scalable cluster-computing framework which efficiently executes computations in-memory with a robust fault-tolerant mechanism achieved by using resilient distributed dataset (RDD). Undoubtedly, in-memory

Table 3.
The execution time (seconds) for Spark-DNAligning, SparkBWA and CloudBurst.

Dataset name	Spark-DNAligning	SparkBWA	CloudBurst
100k s_suis	64.499	108.633	453.373
Speedup	(108.633/64.499)*100 = 168.426		(453.373/ 64.499)*100 = 702.915
SRR001113 chr22	229.446	232.698	556.522
Speedup	(232.698/229.446)*100 = 101.417		(556.522/ 229.446)*100 = 242.550
ERR000589 chr22	354.845	366.038	577.554
Speedup	(366.038/ 354.845)*100 = 103.154		(577.554/ 354.845)*100 = 162.762

Table 4.
The execution time (seconds) for both of Spark-DNAligning and CloudBurst.

Dataset name	Spark-DNAligning	CloudBurst
SRR062634 chr19	404.636	914.004
Speedup	(914.004/ 404.636)*100 = 225.883	
SRR642648 chr19	678.668	2,305.023
Speedup	(2,305.023/ 678.668)*100 = 339.639	

computations have a low latency which improves performance dramatically. Spark-DNAligning parallelizes the aligning processes by distributing short reads on a cluster. Also, Spark-DNAligning enhances performance by utilizing the Spark broadcast variable to make the reference genome available to all nodes in the cluster. Spark-DNAligning is designed in such a way that gives its users the ability to control and tune partitions number which is crucial since results have shown that the partitions number impacts performance significantly. The source code of Spark-DNAligning is publicly available under the MIT license on the GitHub repository:https://github.com/Maryom/Spark-DNAligning. Spark-DNAligning has been evaluated by comparing it with SparkBWA and CloudBurst tools in terms of performance. CloudBurst is a Hadoop based DNA short reads alignment tool. While SparkBWA is BWA based on Apache Spark. Comparisons have been executed on two different YARN clusters with five datasets. Spark-DNAligning speedup has been reached up to 702. Spark-DNAligning boosts performance due to Spark in-memory computations and many other performance optimizations. There is a strong belief that Spark-DNAligning will help biologists in accelerating short read alignment process.

References

[1] J.D. Watson, F.H. Crick, Molecular structure of nucleic acids, Resonance 9 (11) (2004) 96–98.

[2] A. Al Kawam, S. Khatri, A. Datta, A survey of software and hardware approaches to performing read alignment in next generation sequencing, IEEE/ACM Trans. Comput. Biol. Bioinf. 14 (6) (2017) 1202–1213.

[3] J.S.Kim, D.Senol, H.Xin, D.Lee, S.Ghose, M.Alser, H.Hassan, O.Ergin, C.Alkan, O.Mutlu, Grim-filter: fast seed filtering in read mapping using emerging memory technologies, arXiv preprint arXiv:1708.04329.

[4] J. Arram, T. Kaplan, W. Luk, P. Jiang, Leveraging FPGAs for accelerating short read alignment, IEEE/ACM Trans. Comput. Biol. Bioinf. (TCBB) 14 (3) (2017) 668–677.

[5] E.B. Fernandez, J. Villarreal, S. Lonardi, W.A. Najjar, Fhast: FPGA-based acceleration of bowtie in hardware, IEEE/ACM Trans. Comput. Biol. Bioinf. (TCBB) 12 (5) (2015) 973–981.

[6] S. Canzar, S.L. Salzberg, Short read mapping: an algorithmic tour, Proc. IEEE 105 (3) (2017) 436–458.

[7] H. Li, R. Durbin, Fast and accurate short read alignment with burrows–wheeler transform, Bioinformatics 25 (14) (2009) 1754–1760.

[8] H. Li, Aligning sequence reads, clone sequences and assembly contigs with bwa-mem, arXiv preprint arXiv:1303.3997.

[9] B. Langmead, C. Trapnell, M. Pop, S.L. Salzberg, Ultrafast and memory-efficient alignment of short dna sequences to the human genome, Genome Biol. 10 (3) (2009) R25.

[10] B. Langmead, S.L. Salzberg, Fast gapped-read alignment with bowtie 2, Nat. Methods 9 (4) (2012) 357.

[11] J.M. Abuín, J.C. Pichel, T.F. Pena, J. Amigo, Sparkbwa: speeding up the alignment of high-throughput dna sequencing data, PloS One 11 (5) (2016) e0155461.

[12] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, D.J. Lipman, Basic local alignment search tool, J. Mol. Biol. 215 (3) (1990) 403–410.

[13] A.D. Smith, Z. Xuan, M.Q. Zhang, Using quality scores and longer reads improves accuracy of Solexa read mapping, BMC Bioinf. 9 (1) (2008) 128.

[14] H. Jiang, W.H. Wong, Seqmap: mapping massive amount of oligonucleotides to the genome, Bioinformatics 24 (20) (2008) 2395–2396.

[15] M.C. Schatz, Cloudburst: highly sensitive read mapping with mapreduce, Bioinformatics 25 (11) (2009) 1363–1369.

[16] A. McKenna, M. Hanna, E. Banks, A. Sivachenko, K. Cibulskis, A. Kernytsky, K. Garimella, D. Altshuler, S. Gabriel, M. Daly, et al., The genome analysis toolkit: a mapreduce framework for analyzing next-generation dna sequencing data, Genome Res. 20 (9) (2010) 1297–1303.

[17] L. Liu, Y. Li, S. Li, N. Hu, Y. He, R. Pong, D. Lin, L. Lu, M. Law, Comparison of next-generation sequencing systems, BioMed Res. Int. (2012).

[18] S. Zhao, K. Watrous, C. Zhang, B. Zhang, Cloud computing for next-generation sequencing data analysis, in: Cloud Computing-Architecture and Applications, InTech, Rijeka, 2017, pp. 29–51.

[19] S. Jünemann, F.J. Sedlazeck, K. Prior, A. Albersmeier, U. John, J. Kalinowski, A. Mellmann, A. Goesmann, A. Von Haeseler, J. Stoye, et al., Updating benchtop sequencing performance comparison, Nat. Biotechnol. 31 (4) (2013) 294.

[20] J.T. Dudley, Y. Pouliot, R. Chen, A.A. Morgan, A.J. Butte, Translational bioinformatics in the cloud: an affordable alternative, Genome Med. 2 (8) (2010) 51.

[21] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113.

[22] T. Nguyen, W. Shi, D. Ruden, Cloudaligner: a fast and full-featured mapreduce based tool for sequence mapping, BMC Res. Notes 4 (1) (2011) 171.

[23] B. Langmead, M.C. Schatz, J. Lin, M. Pop, S.L. Salzberg, Searching for SNPs with cloud computing, Genome Biol. 10 (11) (2009) R134.

[24] Y. Li, S. Zhong, Seqmapreduce: software and web service for accelerating sequence mapping, Critical Assessment of Massive Data Anaysis (CAMDA) (2009).

[25] P.J. Cock, C.J. Fields, N. Goto, M.L. Heuer, P.M. Rice, The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants, Nucl. Acids Res. 38 (6) (2009) 1767–1771.

[26] D. Decap, J. Reumers, C. Herzeel, P. Costanza, J. Fostier, Halvade: scalable sequence analysis with mapreduce, Bioinformatics 31 (15) (2015) 2482–2488.

[27] L. Pireddu, S. Leo, G. Zanetti, Seal: a distributed short read mapping and duplicate removal tool, Bioinformatics 27 (15) (2011) 2159–2160.

[28] J.M. Abuín, J.C. Pichel, T.F. Pena, J. Amigo, Bigbwa: approaching the burrows–wheeler aligner to big data technologies, Bioinformatics 31 (24) (2015) 4003–4005.

[29] H. Li, R. Durbin, Fast and accurate long-read alignment with burrows–wheeler transform, Bioinformatics 26 (5) (2010) 589–595.

[30] H. Mushtaq, N. Ahmed, Z. Al-Ars, Streaming distributed dna sequence alignment using apache spark, in: 2017 IEEE 17th International Conference on Bioinformatics and Bioengineering (BIBE), IEEE, 2017, pp. 188–193.

[31] X. Li, G. Tan, C. Zhang, X. Li, Z. Zhang, N. Sun, Accelerating large-scale genomic analysis with spark, in: 2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM), IEEE, 2016, pp. 747–751.

[32] W. Zhou, R. Li, S. Yuan, C. Liu, S. Yao, J. Luo, B. Niu, Metaspark: a spark-based distributed processing tool to recruit metagenomic reads to reference genomes, Bioinformatics 33 (7) (2017) 1090–1092.

[33] H. Mushtaq, F. Liu, C. Costa, G. Liu, P. Hofstee, Z. Al-Ars, Sparkga: a spark framework for cost effective, fast and accurate dna analysis at scale, in: Proceedings of the 8th ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics, ACM, 2017, pp. 148–157.

[34] R. Guo, Y. Zhao, Q. Zou, X. Fang, S. Peng, Bioinformatics applications on apache spark, GigaScience 7 (8) (2018) giy098.

[35] M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly, Dryad: distributed data-parallel programs from sequential building blocks, in: ACM SIGOPS operating systems review, vol. 41, ACM, 2007, pp. 59–72.

[36] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M.J. Franklin, S. Shenker, I. Stoica, Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing, in: Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, USENIX Association, 2012, pp. 2–2.

[37] K. Hwang, Cloud Computing for Machine Learning and Cognitive Applications: A Machine Learning Approach, MIT Press, 2017.

[38] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A.D. Joseph, R.H. Katz, S. Shenker, I. Stoica, Mesos: a platform for fine-grained resource sharing in the data center, in: NSDI, vol. 11, 2011, 22–22.

[39] V.K. Vavilapalli, A.C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al., Apache Hadoop YARN: yet another resource negotiator, in: Proceedings of the 4th annual Symposium on Cloud Computing, ACM, 2013, p. 5.

[40] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, I. Stoica, Spark: cluster computing with working sets, HotCloud 10 (10-10) (2010) 95.

[41] G.P. Consortium, et al., A map of human genome variation from population-scale sequencing, Nature 467 (7319) (2010) 1061.

Corresponding author
Maryam AlJame can be contacted at: maryam.aljame@eng.ku.edu.kw