# SMusket: Spark-based DNA error correction on distributed-memory systems

Roberto R. Expósito*, Jorge González-Domínguez, Juan Touriño

*Universidade da Coruña, CITIC, Computer Architecture Group, Campus de Elviña, 15071 A Coruña, Spain*

**Abstract**

Next-Generation Sequencing (NGS) technologies have revolutionized genomics research over the last decade, bringing new opportunities for scientists to perform groundbreaking biological studies. Error correction in NGS datasets is considered an important preprocessing step in many workflows as sequencing errors can severely affect the quality of downstream analysis. Although current error correction approaches provide reasonably high accuracies, their computational cost can be still unacceptable when processing large datasets. In this paper we propose SparkMusket (SMusket), a Big Data tool built upon the open-source Apache Spark cluster computing framework to boost the performance of Musket, one of the most widely adopted and top-performing multithreaded correctors. Our tool efficiently exploits Spark features to implement a scalable error correction algorithm intended for distributed-memory systems built using commodity hardware. The experimental evaluation on a 16-node cluster using four publicly available datasets has shown that SMusket is up to 15.3 times faster than previous state-of-the-art MPI-based tools, also providing a maximum speedup of 29.8 over its multithreaded counterpart. SMusket is publicly available under an open-source license at https://github.com/rreye/smusket.

*Keywords:* Next-Generation Sequencing (NGS), Sequence analysis, Big Data, Apache Spark, Error correction

## 1. Introduction

In recent years, the volume of biological data has increased exponentially due to significant advances in throughput and cost of Next-Generation Sequencing (NGS) platforms [1]. These advances are providing new opportunities to researchers to better understand genetic variation among individuals, helping to characterize complicated diseases like cancer at the genomic level. Nowadays, NGS technologies are able to generate up to terabytes of raw data in a single sequencing run, and this trend is expected to continue to increase in the coming years [2]. Apart from lower cost and increased throughput, NGS technologies also introduce, as a downside, higher error rates in the DNA sequence fragments (so-called reads) compared to traditional Sanger sequencing methods [3], which degrades the quality of downstream analysis and complicates the data processing for many biological studies such as *de novo* genome assembly [4] or short-read mapping [5]. Therefore, an important but computationally intensive and time-consuming preprocessing step in many NGS pipelines is read error correction, which improves not only the quality of downstream analysis but also the accuracy and speed of all the tools used in the pipeline.

The explosion in the amount of available biological data is introducing heavy computational and storage challenges on current systems. Many data analysis pipelines require significant runtimes to transform raw data into valuable information for clinical diagnosis and discovery. Correcting sequencing errors in massive

*Corresponding author

*Email addresses:* roberto.rey.exposito@udc.es (Roberto R. Expósito), jgonzalezd@udc.es (Jorge González-Domínguez), juan@udc.es (Juan Touriño)

NGS datasets in reasonable time can be tackled by relying on parallel computing techniques. However, most of the existing state-of-the-art error correction tools are limited to shared-memory systems or they require specific hardware devices or features. The emergence of Big Data technologies such as the MapReduce paradigm introduced by Google [6] has enabled the deployment of large applications on distributed-memory systems built using commodity off-the-shelf hardware, which can be executed in a highly scalable manner. As a consequence, Big Data and cloud computing have gained much attention in bioinformatics and biomedical fields when dealing with challenges posed by abundant biological data [7, 8, 9, 10, 11].

In this paper we present SparkMusket (SMusket), an error correction tool built upon the open-source Apache Spark framework [12] to exploit the parallel capabilities of Big Data technologies on distributed-memory systems. Spark is a popular cluster computing framework that supports efficient in-memory computations by relying on a distributed-memory abstraction known as Resilient Distributed Datasets (RDD) [13], which provide data parallelism and fault tolerance implicitly. Our tool reimplements on top of the Spark programming model an accurate error correction algorithm from Musket [14], a top-performing and widely used multithreaded corrector based on the $k$-mer spectrum-based method [15] that provides three correction techniques in a multistage workflow. SMusket currently supports the processing of both single- and paired-end DNA reads stored in standard unaligned sequence formats (FASTQ/FASTA). The main contributions of this paper are:

- A thorough literature review and taxonomy on error correction methods and tools for DNA reads.
- A detailed description of SMusket, a distributed error correction tool that efficiently takes advantage of several Spark features (e.g, RDDs, broadcast variables) to fully exploit the performance of Big Data clusters.
- An extensive experimental evaluation of SMusket on a 16-node cluster using four publicly available real datasets that demonstrates the performance benefits of the proposed Spark-based algorithm when compared to previous state-of-the-art MPI- and multithreaded-based tools.

The remainder of the paper is structured as follows: Section 2 introduces the background of the paper. Section 3 discusses the related work. The design and implementation of our tool is described in Section 4. Section 5 presents the experimental results carried out on a Spark cluster to assess the performance of SMusket together with a comparison with representative related tools. Finally, Section 6 concludes the paper and proposes future work.

## 2. Background

This section introduces the main concepts and technologies involved that are necessary to understand our proposal: the $k$-mer spectrum-based (or $k$-spectrum-based) correction method (Section 2.1), the MapReduce model (Section 2.2) and the Apache Spark framework (Section 2.3).

### *2.1. k-spectrum-based error correction*

Due to the importance of DNA error correction, multiple methods have been proposed to cope with the variety of sequencing errors introduced by NGS technologies. The general idea to correct an error in a specific genomic position is based on obtaining all the reads that cover such position and examining the base in that position from all these reads. So, reads that contain an erroneous base can be corrected relying on the majority of reads that have this base correct, as sequencing errors occur infrequently and randomly. As the source genome is unknown, reads from the same genomic location are inferred by assuming that they typically share subreads of a fixed length $k$, the so-called $k$-mers.

Based on this general idea, existing error correction methods can be categorized into three major classes according to the literature [16, 17]: (1) the $k$-spectrum-based approach (Reptile [18], Quake [19], CUDA-EC [20], DecGPU [21], SGA [22], RACER [23], Musket [14], Lighter [24], BLESS [25], BLESS2 [26], BFC [27], FADE [28], RECKONER [29], SPECTR [30], ZEC [31]); (2) the suffix tree/array-based approach

(SHREC [32], Hybrid-SHREC [33], HiTEC [34], Fiona [35]); and (3) the multiple sequence alignment-based approach (Coral [36], ECHO [37], CloudRS [38], CloudEC [39], MEC [40]). Among them, the most advanced and extended one is the $k$-spectrum-based approach due to its efficiency, high accuracy, good scalability and competitive performance. So, we will focus only on this approach in the remainder of the paper.

The main idea of the $k$-spectrum-based method [15] consists of decomposing the input reads into the set of all $k$-mers present in them. Next, the number of times each $k$-mer occurs in the input reads (i.e., its multiplicity) is counted (the $k$-mer counting step). Because multiple reads are generated from a similar genomic location, it is highly probable that these reads contain the same $k$-mer. So, if the multiplicity of a certain $k$-mer is very low we can assume that it contains erroneous bases. Following this assumption, $k$-mers with multiplicity equal to or greater than a certain threshold $M$ (the $k$-mer multiplicity threshold) are called solid or trusted $k$-mers, whereas the remaining ones are called weak or untrusted $k$-mers. On the one hand, solid $k$-mers are highly likely to occur in the genome, being unlikely to have been altered by sequencing errors. Hence, reads that only contain solid $k$-mers are deemed to be error free. On the other hand, reads that contain weak $k$-mers are corrected by repeatedly converting them into solid $k$-mers until there are no more weak $k$-mers in the read (error correction step). After correction, only solid $k$-mers are kept (if possible).

Although the $k$-mer multiplicity threshold ($M$) that separates solid $k$-mers from weak ones can be generally specified by the user, most tools can also automatically determine an appropriate value for $M$ from the $k$-mer multiplicity histogram, but the specific algorithm used to determine $M$ from such histogram depends on each particular tool. Overall, $k$-spectrum-based tools mainly differ in the specific strategy used for implementing the error correction routine for weak $k$-mers, which ultimately determines its accuracy and computational efficiency.

### *2.2. MapReduce*

MapReduce is a parallel programming model and an associated implementation proposed by Google engineers [6] for the storage and processing of large datasets over a cluster of commodity machines. This model allows transparent parallelization by relying on two user-defined functions that have existed for decades in functional programming: *Map* and *Reduce*. MapReduce adopts a data-parallel approach that first partitions the input dataset into multiple splits or chunks, each one containing many records in a <key,value> pair format, and then processes those splits in parallel running multiple instances of the *Map* and *Reduce* functions (the map and reduce tasks). The user-defined *Map* function is first applied to transform the input <key,value> pairs into other intermediate ones. After all map tasks have been completed, the intermediate pairs are sorted and grouped together according to their keys. Next, a shuffle phase is performed to transfer the intermediate pairs across the network so that all the values with the same key are sent to the same reduce task, which merges them into a single list to form the input of the *Reduce* function. Finally, the reduce tasks produce the final output also in the form of <key,value> pairs by applying the user-defined *Reduce* function. Overall, MapReduce automates and takes care of most of the heavy tasks from the programmer's point of view: data partitioning, execution scheduling, fault tolerance and management of inter-process communications between map and reduce tasks.

In order to efficiently support the MapReduce model, Google developed the Google File System (GFS) [41]. GFS is a distributed, block-oriented file system specifically designed to provide high bandwidth by partitioning and replicating data across multiple commodity machines. This file system has built-in fault tolerance by using a data block replication scheme, and the number of times that GFS replicates each block over the cluster is defined as the replication factor. Relying on GFS, MapReduce attempts to schedule map tasks on the cluster machines where the input data blocks reside, improving data locality and minimizing data movements across the network.

Regarding open-source MapReduce projects, Apache Hadoop [42] is the most popular implementation derived from Google's proprietary one. Basically, Hadoop consists of three components or layers: (1) the Hadoop MapReduce engine as data processing layer; (2) the Hadoop Distributed File System (HDFS) [43] as storage layer that mimics GFS; and (3) Yet Another Resource Negotiator (YARN) [44] as resource management layer. During the last decade, Hadoop and its vast ecosystem have become the most significant platform for Big Data processing. This framework generally shows good performance and scalability when

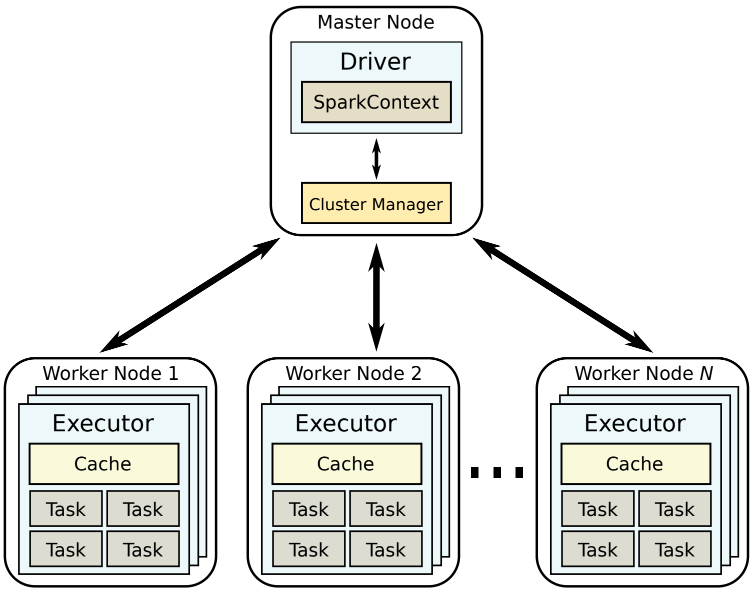


Figure 1: Overview of the Spark architecture and cluster deployment

executing embarrassingly parallel applications that require a single MapReduce job, provided that the volume of intermediate data between the map and reduce phases is not huge. As Hadoop is a disk-based data processing engine, one important limitation is its inefficiency for reusing intermediate results among several computations or MapReduce jobs. Reusing intermediate results is important in many iterative workloads such as machine learning and graph algorithms, which obtain poor performance as HDFS has to be used to store such data in disk (i.e., different MapReduce jobs cannot share data directly).

### *2.3. Apache Spark*

Apache Spark [12, 45] is a cluster computing framework that has been specifically designed to overcome the Hadoop limitations. Spark supports efficient in-memory computations in a fault-tolerant manner by introducing the core concept of Resilient Distributed Dataset (RDD) [13]. An RDD is an immutable, partitioned collection of data distributed across the cluster that can be operated in parallel and cached in memory to be reused in multiple MapReduce-like operations. RDDs can be created by parallelizing an existing collection of objects (e.g., a list) or by loading an external dataset from a distributed file system (e.g., HDFS). By reusing RDDs, programmers can perform iterative computations without writing intermediate results to disk, clearly outperforming Hadoop. Furthermore, another interesting feature is that RDDs recover automatically from failure, providing fault tolerance without replication.

At a high level, Spark uses a master/worker architecture as depicted in Figure 1. On the one hand, the *Driver* program, which usually runs on the master node, executes the main function of the Spark application. For each application, the *Driver* first creates a *SparkContext* that acts as the central coordinator and then defines the RDDs and parallel operations to be carried out over them. This *Driver* can be written in any programming language currently supported by Spark (e.g., Scala, Java). On the other hand, each worker node runs one or more *Executor* processes that are in charge of storing RDDs and effectively performing the computations over them. Tasks are the smallest computational units that can be run in parallel over an RDD by an *Executor*, and are scheduled on a per-core basis (i.e., one task per core). Hence, a set of tasks form a Spark job and a set of jobs form a Spark application. The *SparkContext* entity is responsible for creating, scheduling and sending individual tasks to be executed on *Executors*, allocating the required computational resources through a cluster manager. Currently, Spark supports Hadoop YARN [44], Mesos [46] and Kubernetes [47] as cluster managers, also providing its own implementation known as Spark Standalone.

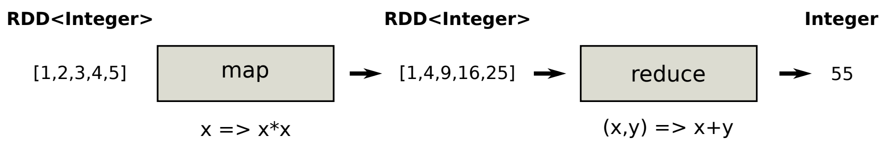


Figure 2: Spark example of *map* transformation and *reduce* action

Spark also provides a richer programming API than Hadoop by extending the MapReduce model to allow for more flexible data-parallel operations over RDDs. These operations can be classified into two types: transformations and actions. On the one hand, transformations (e.g., *map*, *filter*, *join*) are operations that create a new RDD from an existing one. For instance, the *map* transformation processes each RDD element through a user-defined function and returns a new RDD representing the results. It is important to note that transformations are lazily evaluated, so they do not compute anything until an action that requires the result from them is triggered. Therefore, there is no need to worry about memory usage when chaining a number of transformations. On the other hand, actions are operations that return non-RDD values, converting the laziness of transformations into actual computation. They can be used to either return a result to the *Driver* (e.g., *reduce*, *collect*), or to store the content of an RDD in external storage after running a certain computation (e.g., *saveAsTextFile*). For instance, the *reduce* action aggregates all the RDD elements according to a user-defined function and returns the final result to the *Driver* program. Figure 2 shows a simple example of applying a *map* transformation chained with a *reduce* action over an RDD of type *Integer*. The user-defined functions executed over the RDD are shown below the corresponding boxes for the *map* and *reduce* operations.

Finally, each transformed RDD may be recomputed each time an action is performed on it. However, Spark allows to explicitly persist (i.e., cache) an RDD in memory to keep the RDD elements on the cluster for much faster access the next time they are queried.

## 3. Related work

The exploitation of Big Data technologies such as MapReduce and Spark to accelerate the storage, processing and visualization of large datasets has transformed multiple disciplines through the new knowledge these technologies help to generate. Representative examples in the literature include several fields such as weather forecasting [48], healthcare [49], medical imaging simulation [50], social networks [51], industrial IoT [52] and deep learning [53]. In the particular case of the bioinformatics field, the Big Data paradigm is gaining increasing attention [7, 9, 54] as a key mechanism to deal with the severe challenges posed by modern NGS technologies. As a consequence, many bioinformatics tools implemented on top of open-source Big Data technologies have emerged in recent years, from DNA/RNA sequence alignment [55, 56, 57] to *de novo* genome assembly [58, 59]. Next, we will focus on previous bioinformatics tools intended for NGS error correction using the $k$-spectrum-based approach, both for shared- and distributed-memory systems.

### *3.1. $k$-spectrum-based parallel correctors*

Although the accuracy of error correction provided by state-of-the-art $k$-spectrum-based tools has proved to be relatively high [60], their throughput needs improvement when dealing with increasingly large NGS datasets. In the $k$-spectrum-based method, correcting one read is independent of correcting the others, which means that there is a lot of potential parallelism to be extracted from the algorithms if they are efficiently implemented.

We can find in the literature some previous works that exploit parallel and distributed architectures to increase the performance of error correction tools. In fact, most of the $k$-spectrum-based tools cited in Section 2.1 provide a parallel implementation through multithreading to support shared-memory systems, except Reptile [18] and BLESS [25] which are limited to a single core. The following tools fall into the multithreading-based parallel approach: Quake [19], SGA [22], RACER [23], Musket [14], Lighter [24], BLESS2 [26] (a parallel version of the original BLESS tool), BFC [27], RECKONER [29], SPECTR [30] and

ZEC [31]. However, their main drawback is that their scalability is limited to a single node, except for three of them (BLESS2, SPECTR and ZEC) that also provide support for distributed-memory systems using the Message Passing Interface (MPI) [61]. On the one hand, SPECTR features a highly efficient vectorized SIMD algorithm that has been designed to be executed on specific hardware. Concretely, SPECTR requires a CPU microarchitecture that supports the AVX-512 instruction set or the availability of an Intel Xeon Phi many-core accelerator (discontinued in mid-2018). On the other hand, both BLESS2 and ZEC combine MPI with multithreading using the OpenMP standard [62], so they can be executed on any commodity machine without specific hardware requirements. Finally, CUDA-EC [20], DecGPU [21] and FADE [28] are other parallel tools that, similarly to SPECTR, also require specific hardware. CUDA-EC and DecGPU take advantage of the computing power of NVIDIA GPUs by relying on the CUDA programming model [63], whereas FADE exploits fine-grained parallelism in FPGA hardware.

Our main goal is to provide a scalable error correction tool intended for distributed-memory systems based on commodity hardware without any specific requirement, similarly to BLESS2 and ZEC. Instead of relying on the classical MPI+OpenMP hybrid parallel approach, we intend to fully exploit the features of Spark to boost performance when correcting large datasets. To the best of our knowledge, SMusket is the first publicly available tool that relies on Big Data technologies to implement the $k$-spectrum-based method. More specifically, our tool is based on the error correction routine implemented by Musket, which has proved to be one of the top-performing correctors in terms of accuracy among the multithreading-based tools according to [60]. Furthermore, Musket is highly cited and widely used by researchers and scientists. SMusket enables extending the good features of the correction algorithm implemented by Musket to distributed-memory systems, significantly speeding up the execution times while providing the same accuracy.

## 4. SMusket implementation

Musket is a command-line tool implemented in C++ and parallelized using threads [14]. Unfortunately, Spark does not support C++ to write the *Driver* program, which would have greatly facilitated our implementation. Among the currently supported languages (Scala, Java, Python and R), we have selected Java to implement SMusket in order to ease code conversion from C++ due to their comparable objected-oriented models and some syntax similarities. However, it is important to remark that although the performance of the Java Virtual Machine (JVM) has increased significantly over the last decade, the computational efficiency of Java is still lower than C++. Moreover, the memory management of both languages is rather different. C++ relies on the programmer to manage memory explicitly, whereas Java provides automatic memory management through the built-in Garbage Collector (GC) included in the JVM, which can consume a considerable amount of computational resources and hinder application performance [64]. Anyway, these drawbacks would be much the same for any of the other languages supported by Spark [65].

Basically, SMusket has been designed as a Java-based, command-line tool that receives as input some arguments that are equivalent to those of Musket. For instance, the path to the input dataset to be corrected or the $k$ value for creating $k$-mers (the $k$-mer length). SMusket currently supports both single- and paired-end reads stored in standard formats (FASTQ/FASTA). The submission of the Spark jobs to the cluster to correct the input dataset is facilitated by the *smusketrun* command included in the SMusket bundle distribution. Our tool includes a detailed README file that explains all the available input arguments for *smusketrun*, provides execution and compilation instructions, and describes advanced configuration options.

### *4.1. Overall workflow*

At the highest level of abstraction, SMusket divides the computation into two main phases as depicted in Figure 3: $k$-spectrum construction and error correction. During the $k$-spectrum construction phase, SMusket first creates all the $k$-mers available in the input dataset and counts their number of occurrences (i.e., their multiplicity). Second, unique $k$-mers (i.e., $k$-mers with multiplicity = 1) are filtered out since they are considered to be largely untrusted. The $k$-mer multiplicity histogram is then generated from the

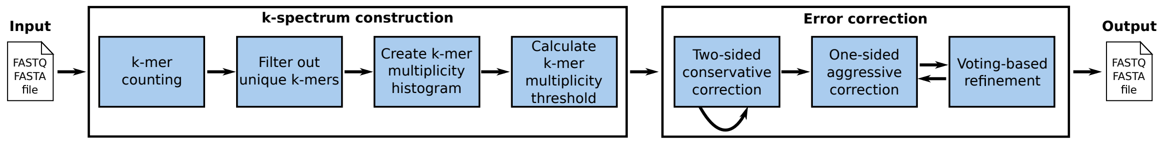


Figure 3: High-level workflow performed by SMusket

multiplicities of non-unique $k$-mers to determine the $k$-mer multiplicity threshold ($M$) that separates solid $k$-mers from weak ones. To determine such threshold $M$ from the histogram, SMusket uses the same algorithm as Musket, which is described in detail in [14].

For the error correction phase, SMusket adopts a multistage correction workflow to correct each input read based on three techniques originally introduced by Musket: (1) a two-sided conservative correction that corrects at most one sequencing error in any $k$-mer of a read. For an error occurring at position $i$, the two-sided correction aims to find a unique alternative base that makes all $k$-mers that cover position $i$ trusted by evaluating both the leftmost and the righmost $k$-mers that cover such position. This technique is performed for a fixed number of iterations (two iterations by default) that can be specified through a command-line option, or until no base is changed. (2) A one-sided aggressive correction in the case that more than one error occurs in a single $k$-mer. To confine false-positive errors, the number of corrections allowed in any $k$-mer of a read are limited to four by default, and this value can also be specified through a command-line option. And (3) a voting-based refinement, which is performed after completing the one-sided correction, attempts to find the correct base by replacing all possible bases at each position of the $k$-mer and checking the solidity of the resulting $k$-mers. This technique aims to reduce the number of new errors that can be introduced during the one-sided correction. The combination of techniques 2 and 3 is also conducted iteratively (2 iterations by default). More specific details about these correction techniques can be obtained from [14]. Basically, SMusket reimplements in Java the whole C++-based error correction routine of Musket, but taking into account the specific features of the JVM as mentioned at the beginning of Section 4. As general guidelines, we have paid special attention to minimize the impact of the GC on performance by reusing Java objects inside methods as much as possible to avoid excessive GC activity at runtime.

It is important to remark that by estimating the $k$-mer multiplicity threshold in the same way as Musket and by reimplementing its correction routine, the sequencing errors corrected by SMusket for a particular input dataset are identical to those corrected by its multithreaded counterpart. By doing so, the accuracy and quality of error correction provided by SMusket is ensured. Specifically, we have checked that the output files generated by SMusket are exact, byte-by-byte replicas of those generated by Musket.

### *4.2. RDD creation*

As explained in Section 2.3, the fundamental Spark feature that enables parallel processing is the concept of RDD [13], a partitioned and fault-tolerant collection of elements distributed across the *Executors* running on the cluster. In order to construct the $k$-mer spectrum, the input dataset must be first translated into an RDD to be operated on by using RDD operations (transformations and actions). As mentioned previously, RDDs can be created either by parallelizing an existing collection of objects or by loading an external dataset from a distributed file system, the second option being the appropriate one in our context as the input DNA reads are stored in a text file. As HDFS is the cornerstone of the Hadoop storage subsystem, which allows data to be processed in a distributed manner, we assume it as the distributed file system in this work. Furthermore, HDFS is not only fully supported by Spark since its inception but also by most of the existing Big Data frameworks and tools.

The most straightforward way to create an RDD from an input text file stored in HDFS is calling the *textFile* method provided by Spark. This method relies on the default implementation provided by Hadoop for processing text-based files stored in HDFS: *TextInputFormat*. This class is a concrete implementation of the abstract *FileInputFormat* class. Unfortunately, *TextInputFormat* is not able to handle properly the FASTQ/FASTA formats due to their specific structure. These sequence formats are text-based files that

```
@SRR317060.19 PAN_0038_FC625T2AAXX:5:3:7882:1292/1
TACAGTATTTTCATTANAGAATTCTTTTATCTTTTCTACATTTTTTAATATGTTCGTAATAATGTATTTTGATTTT
+
GDBBGGDGGGDEGGB@%@3@45773B9<<<+CA4<?<<BBDG3<G@@EFD8BEDB*/27;GDGGGGG>GDDGB@DG
@SRR317060.20 PAN_0038_FC625T2AAXX:5:3:3604:1297/1
AGTACAAGACTTCATCNCAAAAAAAAAAAAAAAACCAAATTTCCCTTTTGCCAAAAAAAATAAGTTCGCAAAAAAA
+
GDBBGGDGGGDEGGB@%@3@45773B9<<<+CA4<?<<BBDG3<G@@EFD8BEDB*/27;GDGGGGG>GDDGB@DG
@SRR317060.21 PAN_0038_FC625T2AAXX:5:3:5688:1308/1
AAAAAGAGCCCGCATTNCCTATTTAATCTTTAGCCAAAATAACAAATCGTGATTCATCACGCTACCTGAGTTCAAA
+
DD>9DB=BA?BDD?D4%1>;==@+?=B@B47<74(-0424==;=6./*2.>0+24??;<1<==?*BD>DDDD<DD?
```

Figure 4: Example of a FASTQ file containing three reads

involve multiple lines per read (see Figure 4 for an example). However, Hadoop is mainly designed to process line-based text formats where identifying individual records is simple as line boundaries are denoted by newline characters (i.e., one record per line). Although the record delimiter can be changed to use the character that separates single reads in the input file (e.g., '@' for FASTQ), this would not work since such character can also occur in the quality string. In fact, none of the other built-in *FileInputFormat* subclasses provided by Hadoop for processing text-based files (e.g., *KeyValueTextInputFormat*) can handle those sequence formats straightforwardly.

One simple but inefficient way to overcome this issue is to convert the input sequence files into the appropriate line-by-line format required by *TextInputFormat* (i.e., one read per line) and then copy the converted files to HDFS. This has been the preferred approach for many bioinformatics tools based on Hadoop/Spark that process input datasets in FASTQ/FASTA format, such as BigBWA [55], DistMap [66] and CloudEC [39]. However, this preprocessing of the input files incurs high disk overhead and degrades performance significantly [57]. A more advanced approach consists in using specialized libraries that provide specific routines to parse these file formats directly from HDFS. Hadoop-BAM [67], BioPig [68], FASTdoop [69] and HSP [70] are the available alternatives. All these Java-based libraries provide custom implementations of the *FileInputFormat* class that allow to handle single-end datasets in FASTQ/FASTA formats. However, Hadoop-BAM, BioPig and FASTdoop do not provide specific support for paired-end datasets, so a preprocessing step is still required in this case. Furthermore, Hadoop-BAM and FASTdoop do not support compressed datasets. BioPig provides this support, but it has proved to be the most inefficient library among them according to [69]. To the best of our knowledge, HSP is the only library that allows avoiding any preprocessing of the input files both for single- and paired-end datasets. Therefore, SMusket relies on HSP to create the initial RDD from the input files in an efficient and simple way, as briefly described next.

#### *4.2.1. HSP library*

Basically, our Hadoop Sequence Parser (HSP) library [70] provides two classes that extend the Hadoop *FileInputFormat* class for single- and paired-end datasets: *SingleEndSequenceInputFormat* and *PairedEndSequenceInputFormat*, respectively. On the one hand, the former provides specific implementations for FASTQ (*FastQInputFormat*) and FASTA (*FastAInputFormat*) to support single-end datasets (i.e., one input file). On the other hand, the latter supports paired-end datasets, where the two ends of paired reads are distributed in two separate files, one of them containing the forward reads (i.e., the "left" file) and the other one containing the corresponding reverse reads (i.e., the "right" file). Regarding data types, HSP generates <key,value> pairs of type $<Long, Text>$. In single-end mode, the key is a unique self-generated identifier for each read and the value is the text-based content of the read (e.g., read name, bases and qualities for FASTQ). In paired-end mode, the key provides the length (in bytes) of a single read in the pair and the value is the merged content of both reads. If needed, HSP provides static methods that allow programmers to obtain "left" and "right" reads separately as *String* objects (*getLeftRead* and *getRightRead*, respectively).

As illustrative examples of the simplicity of using HSP, Listings 1 and 2 show the Java code required to create RDDs for single- and paired-end datasets, respectively, using the *newAPIHadoopFile* method provided by Spark. This method allows programmers to define an RDD for a given input file that conforms with a custom *FileInputFormat* implementation.

Listing 1: Basic Java code to create an RDD from a single-end dataset in FASTQ format using HSP

```java
SparkSession sparkSession = SparkSession.builder().config(new SparkConf()).getOrCreate();
JavaSparkContext jsc = JavaSparkContext.fromSparkContext(sparkSession.sparkContext());
Configuration config = jsc.hadoopConfiguration();

Class inputFormat = FastQInputFormat.class;
JavaPairRDD<LongWritable,Text> readsRDD = jsc.newAPIHadoopFile('/path/to/file',
inputFormat, LongWritable.class, Text.class, config);
```

Listing 2: Basic Java code to create an RDD from a paired-end dataset in FASTA format using HSP

```java
SparkSession sparkSession = SparkSession.builder().config(new SparkConf()).getOrCreate();
JavaSparkContext jsc = JavaSparkContext.fromSparkContext(sparkSession.sparkContext());
Configuration config = jsc.hadoopConfiguration();

// Set left and right input paths for HSP
Class inputFormat = FastAInputFormat.class;
PairedEndSequenceInputFormat.setLeftInputPath(config, '/path/to/file1', inputFormat);
PairedEndSequenceInputFormat.setRightInputPath(config, '/path/to/file2', inputFormat);

JavaPairRDD<LongWritable,Text> readsRDD = jsc.newAPIHadoopFile('path/to/file1',
PairedEndSequenceInputFormat.class, LongWritable.class, Text.class, config);
```

#### *4.2.2. RDD partitioning*

Once an RDD is created, transformations and actions can be performed in parallel over it. It is important to remark that an RDD is a partitioned collection of elements distributed across the cluster, and such number of RDD partitions can affect performance significantly as Spark can only run a single concurrent task for each partition up to $nc$ tasks, $nc$ being the total number of cores in the cluster. By default, Spark creates one RDD partition per HDFS data block in the file, but partitioning is also configurable. However, there is no straight rule to determine the optimum number of RDD partitions. To ensure a minimum level of parallelism, this value must be equal to (or greater than) the total number of tasks that can be executed by all *Executors* (i.e., equal to $nc$). To increase the level of parallelism, it is generally beneficial to create multiple partitions per core so that the workload gets distributed more evenly among *Executors*, but the optimum value still depends to a great extent on each specific workload. In the case of SMusket, users can easily specify the desired number of RDD partitions via a command-line option, and the impact of varying such value on performance will be assessed in detail in Section 5.1.

### *4.3. Spark algorithm*

This section presents the algorithm performed by SMusket to implement on top of Spark the workflow described in Section 4.1. Figure 5 shows a high-level overview of the algorithm for single-end correction. As can be observed, it can be divided into three main stages: (1) $k$-spectrum construction, (2) $k$-mers broadcast, and (3) error correction. Stages 1 and 3 correspond directly with the Spark implementation of the two phases of the workflow shown in Figure 3. Stage 2 arises as a requirement to perform the error correction in parallel on a distributed-memory system, as will be explained later. The three stages of the algorithm and the operations they perform are detailed in Sections 4.3.1, 4.3.2 and 4.3.3, respectively. Finally, although the algorithm shown in Figure 5 is intended for correcting single-end datasets, paired-end correction requires minor changes that are explained separately in Section 4.3.4.

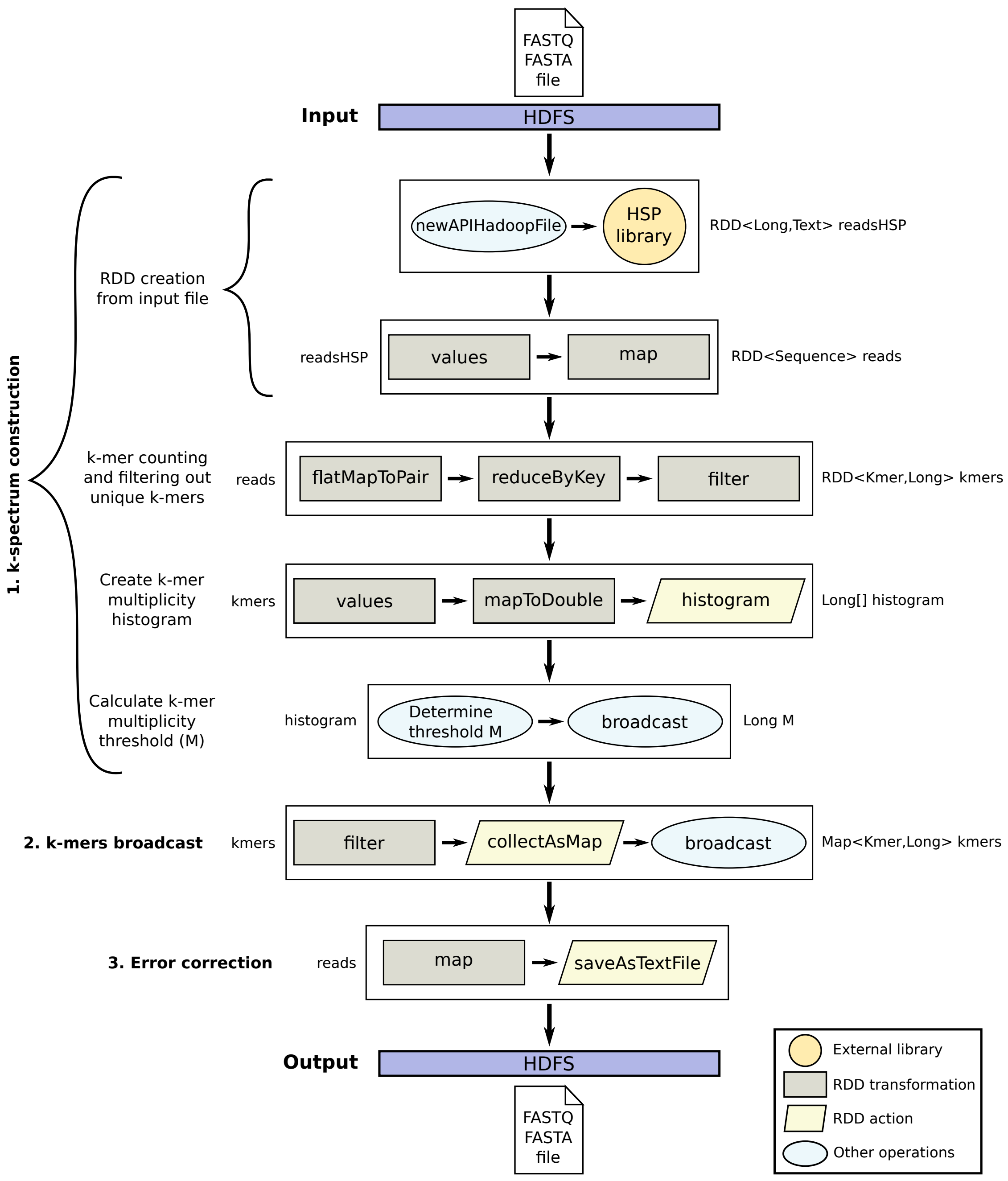


Figure 5: Overview of the parallel algorithm performed by SMusket on top of Spark for single-end correction

*4.3.1. k-spectrum construction*

First of all, an RDD must be created from the input sequence file stored in HDFS. To do so, the *newAPIHadoopFile* method from Spark and the input formats provided by the HSP library are used as described in Section 4.2.1. This step returns a <key,value> RDD of type <*Long*,*Text*> (named `readsHSP` in Figure 5). In our scenario, the read identifier generated by HSP as key is not needed. So, a *values* transformation is then applied on the previously generated RDD, which returns a new RDD with only the values of each pair. As mentioned before, HSP provides as value the whole content of the read as a *Text*

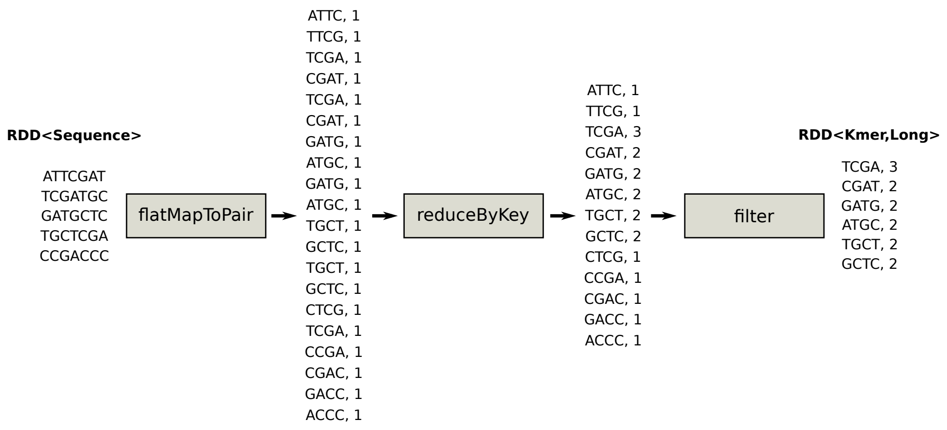


Figure 6: $k$-mer counting and filtering out unique $k$-mers on an RDD with five reads ($k = 4$)

object, which is a Hadoop custom class similar to *String* in Java. To obtain separately the necessary data for each read (e.g., the bases to be corrected), the text contained in the values is parsed accordingly by applying a *map* transformation that converts each RDD element (i.e., each *Text* object) into an object of our custom *Sequence* class with the appropriate attributes. The *map* transformation returns a new RDD formed by passing each element of the source RDD (`readsHSP`) to a user-defined function that performs the conversion between both objects. As a result, a new RDD of *Sequence* objects is returned (named `reads` in the figure).

Once the `reads` RDD has been created from the input file, all the $k$-mers for each read are generated by applying a *flatMapToPair* transformation. This operation is a special version of *map* where each element in the source RDD can be mapped to 0 or more <key,value> pairs in the target RDD, whereas in a *map* operation the correspondence is always one-to-one. Concretely, the function executed by *flatMapToPair* generates all the $k$-mers for each read: $L$ - $k$ + 1 $k$-mers in a read of length $L$ (i.e., $L$ bases or nucleotides). This function outputs each $k$-mer as key using our custom *Kmer* object together with a multiplicity of one as value. Hence, *flatMapToPair* returns a <key,value> RDD of type <*Kmer*,*Long*>. To perform $k$-mer counting, a *reduceByKey* transformation is then applied on this RDD so that the values for each key are aggregated based on a given user-defined function. In this case, our reduce function simply sums all the values (i.e., multiplicities) together for each key (i.e., $k$-mer). Next, unique $k$-mers are filtered out by applying a *filter* transformation that returns a new RDD formed by selecting only those elements of the source RDD on which a user-defined function returns true. This function simply selects those $k$-mers whose multiplicity is greater than one. After the filtering step, the resulting RDD is also of type <*Kmer*,*Long*> (`kmers` in the figure), containing all non-unique $k$-mers and their corresponding accumulated multiplicities. For the sake of clarity, Figure 6 illustrates the procedure of $k$-mer counting and filtering with a simple example that uses a $k$-mer length of four ($k = 4$) on an input RDD that contains five reads of length seven. Four $k$-mers $(7 - 4 + 1)$ are thus generated for each read (20 in total).

After $k$-mer counting, the $k$-mer multiplicity histogram can be easily generated from the `kmers` RDD using the *histogram* action provided by Spark. However, this action can only be executed on RDDs of type *Double*. So, the keys from `kmers` RDD are first discarded using a *values* transformation and then the multiplicities are converted from *Long* to *Double* by chaining a *mapToDouble* transformation. Next, the *histogram* action can be performed. It is worth noting that all RDD operations prior to *histogram* were transformations that are lazily evaluated (i.e., no computation has actually been done yet). As mentioned in Section 2.3, actions are Spark operations that trigger computations over RDDs, and the values returned by these operations are stored in the *Driver* or in external storage (e.g., HDFS). In the case of the *histogram* action, an array representing the histogram computed on the *Executors* is returned to the *Driver*.

Following the example presented in Figure 6, Figure 7 shows how the histogram is generated using the `kmers` RDD, from which the $k$-mer multiplicity threshold $M$ can be determined. Note that $M$ is computed

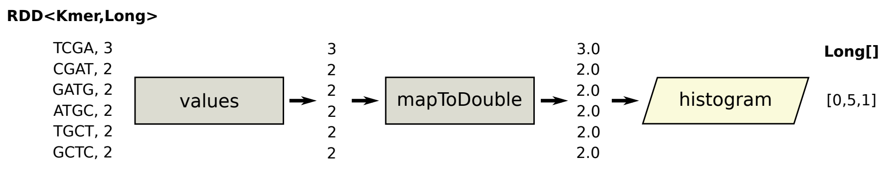


Figure 7: $k$-mer multiplicity histogram generated from non-unique $k$-mers

on the *Driver* (i.e., in a non-parallel way), but the time needed to do so is totally negligible. Once $M$ is computed, its value must also be made available to the *Executors* as it is needed during the error correction stage to separate solid $k$-mers from weak ones. Rather than sending a copy of $M$ together with each task that needs it from *Driver* to *Executors*, which increases network overhead, its value is sent only once using a broadcast variable. Broadcast variables in Spark are a handy mechanism for sharing read-only variables among *Executors*. These variables are sent across the network using efficient broadcast algorithms to reduce communication cost. Hence, the $k$-spectrum construction stage finishes by defining a broadcast variable for $M$ to make it available to *Executors* for subsequent stages.

### *4.3.2. k-mers broadcast*

During the error correction stage, the `reads` RDD is processed to test the solidity of all the $k$-mers for each read by comparing their multiplicity with threshold $M$. Solid $k$-mers (those with multiplicity $>= M$) are deemed to be error free, whereas weak or untrusted $k$-mers are considered for error correction. This means that all *Executors* running in the cluster must be able to check the multiplicity of any possible $k$-mer that can be generated from any input read. However, the multiplicities of non-unique $k$-mers are stored in an RDD (`kmers`) so that their values are actually distributed across *Executors* (i.e., each *Executor* only has part of the data). Note that this issue does not arise in a parallel implementation for shared-memory systems such as Musket, where non-unique $k$-mers are stored in local hash tables shared by all threads.

In a similar way as before for threshold $M$, this stage is in charge of sending non-unique $k$-mers and their multiplicities to the *Executors* by using a broadcast variable. However, this mechanism requires to have all the data to be sent available on the *Driver*. Therefore, the `kmers` RDD must be first sent from the *Executors* to the *Driver* by applying a *collectAsMap* operation. This operation is an RDD action that brings all the RDD elements as a Java *Map* collection to the *Driver* side. Once this action has been completed, the broadcast mechanism can be used to send the *Map* collection to all *Executors* so that they can test the solidity of any $k$-mer by simply querying the *Map* using the *get* method.

In the previous operations, it is crucial to minimize the amount of data to be sent across the network from the *Executors* to the *Driver* during the *collectAsMap* action and the other way around during the broadcast. After analyzing in detail the error correction routine in Musket, we have checked that multiplicity is only actually used for solid $k$-mers. In order to reduce network overhead, a *filter* transformation is first applied to the `kmers` RDD to discard weak $k$-mers (those with multiplicity $< M$) and thus select solid $k$-mers. When testing the solidity of a $k$-mer during error correction, if the *Map* does not contain such $k$-mer it means that is considered weak; otherwise, it is solid and its corresponding multiplicity can be obtained from the *Map*. It is important to note that the filtering operation not only reduces network overhead but also minimizes the amount of memory needed to store the *Map*.

### *4.3.3. Error correction*

Basically, this stage processes each input read to correct potential sequencing errors relying on the data already available on each *Executor* as a result of the previous stages: the threshold $M$ and the *Map* collection that contains solid $k$-mers and their multiplicities. To do so, the RDD that contains the input reads (`reads` in Figure 5) is operated by applying a *map* transformation that processes each element through a function that implements the three error correction techniques from Musket, as mentioned in Section 4.1 (see Figure 3). Another option would be to apply three chained *map* transformations on the `reads` RDD, each one performing one technique. However, this can lead to lower performance as it increases the overhead associated with creating and scheduling a considerably greater number of tasks.

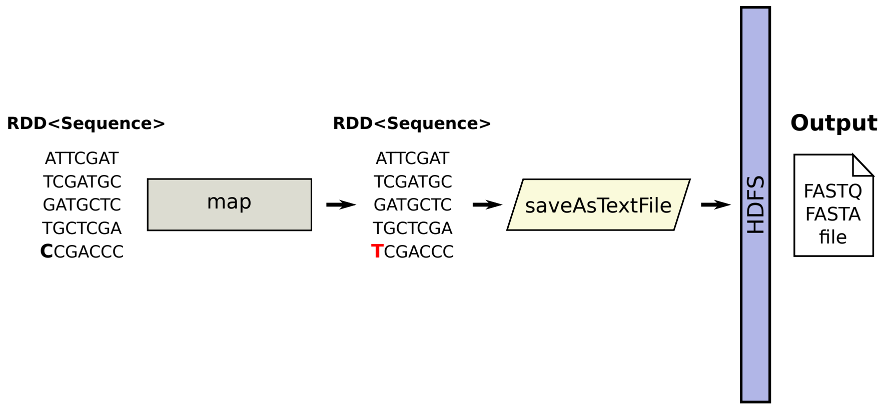


Figure 8: Storing an RDD with five reads in HDFS after performing the error correction routine

Finally, the corrected RDD of *Sequence* objects that is returned from the previous *map* transformation must be stored back in HDFS. Spark provides a specific operation for this purpose: *saveAsTextFile*. This RDD action writes all the RDD elements as a text file in a given path in HDFS (see Figure 8 for an example), or in any other Hadoop-supported file system. More specifically, each RDD element is first converted to a *String* object through the *toString* method before writing it to the file system. Our custom *Sequence* class overrides this method accordingly to provide a specific implementation that conforms with the corresponding sequence format of the reads contained in the RDD (FASTQ/FASTA).

### *4.3.4. Paired-end correction*

The algorithm previously described allows performing single-end correction. For paired-end datasets, two changes are required as depicted in Figure 9. The first one is related with the creation of the RDD that contains the input reads. It is necessary to take into account that the two ends of paired reads are distributed in two separate files, with one of them containing the forward reads (i.e., the "left" file) and the other one containing the corresponding reverse reads (i.e., the "right" file). As explained in Section 4.2.1, HSP supports paired-end datasets straightforwardly by using the *PairedEndSequenceInputFormat* class (see Listing 2). The <key,value> RDD returned from the *newAPIHadoopFile* method is also of type *<Long,Text>* as in the case of single-end correction. However, the *Text* object now contains both ends of a paired read. To convert from this single *Text* object containing both reads into two separate *Sequence* objects, a *mapToPair* transformation is applied to the `readsHSP` RDD. This transformation is similar to *map* as it also creates a one-to-one mapping between elements of the source RDD and the target RDD, but it allows to return a <key,value> RDD. In this case, the function executed by *mapToPair* is in charge of parsing the *Text* object accordingly to create two *Sequence* objects representing both ends of a paired read. Hence, the RDD returned from *mapToPair* is of type *<Sequence,Sequence>* (`reads` in the figure). During $k$-mer counting, a *flatMapToPair* transformation is also applied to the `reads` RDD as in single-end correction, but now the function executed by this transformation must generate all the $k$-mers for each read of the pair.

The second change is needed during the error correction stage. As `reads` is now a <key,value> RDD, both ends contained in each element must be processed to correct potential errors. This can be done in a similar way as before by applying a *map* transformation to the `reads` RDD in order to execute the correction routine on each RDD element, but now separately in each *Sequence* object of the pair. To do so, it is necessary to apply a previous *keys* or *values* transformation to the `reads` RDD to discard forward or reverse reads, respectively. In this way, both ends are corrected and returned in separate RDDs in order to be written back to HDFS.

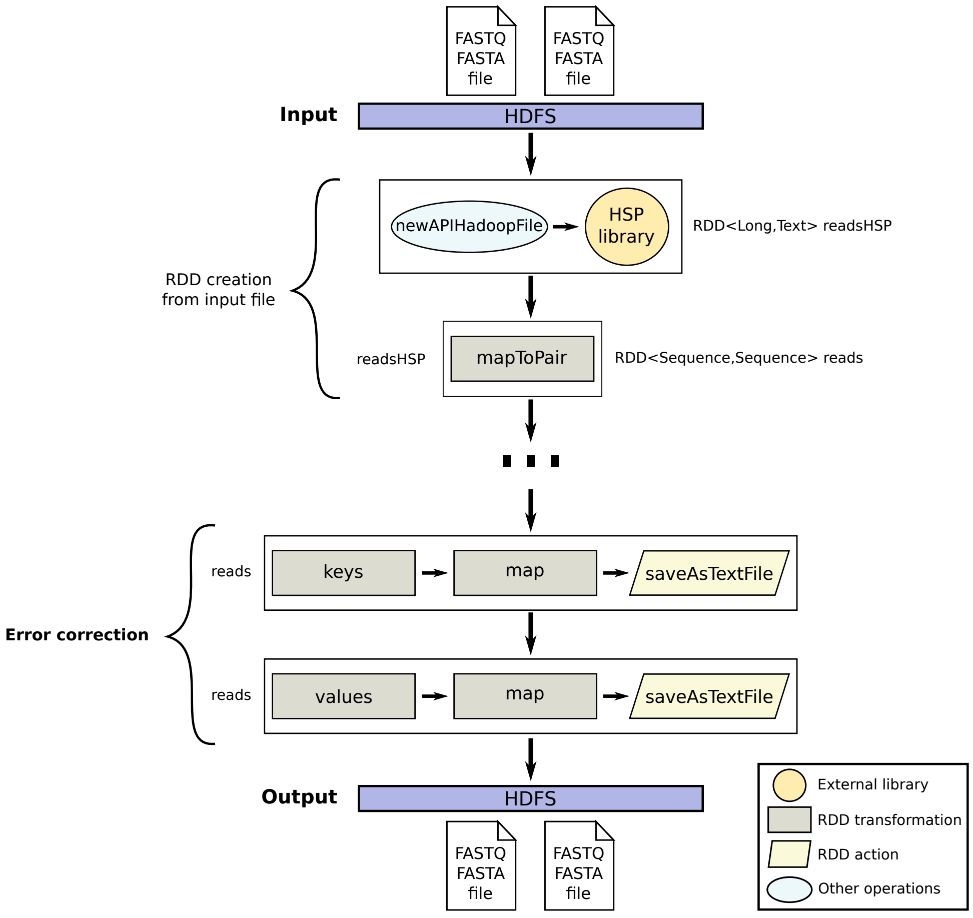


Figure 9: Overview of the parallel algorithm performed by SMusket on top of Spark for paired-end correction (trimmed)

## 5. Performance evaluation

As mentioned in Section 4.1, the quality of error correction provided by SMusket remains the same as that of Musket, whose accuracy has already been thoroughly assessed in multiple previous studies [14, 24, 29, 60]. Therefore, the experimental evaluation of SMusket has focused on performance in terms of execution time, as our main goal is to provide a scalable parallel tool. In order to evaluate performance and scalability, a 16-node commodity cluster running Spark version 2.3.1 has been used as testbed. Each cluster node consists of two Intel Xeon E5-2660 octa-core processors at 2.2 GHz (i.e., 16 cores per node), 64 GiB of memory, and one 800 GiB local disk intended for both HDFS and intermediate data storage during the executions. Nodes are interconnected through Gigabit Ethernet (1 Gbps) and InfiniBand FDR (56 Gbps). The system runs Linux CentOS 6.10 with kernel 2.6.32-754 and the JVM version is Oracle 1.8.0_201. Regarding HDFS settings, the block size and the replication factor were set to 128 MiB and 3, respectively. To deploy Spark on the cluster, the Big Data Evaluator (BDEv) tool [71, 72] has been used running one *Executor* per node and 16 cores per *Executor* (256 cores in total), since our preliminary experiments proved it to be the best configuration for this system. Finally, the *Driver* runs on the master node of the cluster.

As shown in Table 1, four publicly available datasets with different characteristics have been used in the experiments, named after their accession numbers in the European Nucleotide Archive [73, 74] and tagged accordingly for the sake of clarity (see second column). The number of reads (fifth column) refers to the number of input DNA sequences to be corrected, whereas the read length (last column) is expressed in terms of the number of base pairs (bp). All the results shown in this section correspond to the median value for

Table 1: Public datasets used in the experimental evaluation of SMusket

| **Dataset** | **Tag** | **Instrument model** | **Organism** | **#Reads** | **Read length** |
|---|---|---|---|---|---|
| SRR507810 | SRR50 | Illumina HiSeq 2000 | Mus musculus | $96\text{x}10^6$ | 101 bp |
| SRR534301 | SRR53 | Illumina HiSeq 2000 | Homo sapiens | $108\text{x}10^6$ | 101 bp |
| SRR567455 | SRR56 | Illumina HiSeq 2000 | Homo sapiens | $251\text{x}10^6$ | 76 bp |
| SRR317060 | SRR31 | Illumina Genome Analyzer II | Homo sapiens | $110\text{x}10^6$ | 76 bp |

a set of 10 executions for each experiment using the default $k$-mer length ($k = 21$), although the observed variance was not significant.

Regarding the experiments, Section 5.1 first analyzes the impact of varying the number of RDD partitions on the performance of SMusket when using 4, 8, 12 and 16 nodes. This first set of experiments considers both single-end and paired-end correction so that one and two input files, respectively, are processed for each dataset. Next, Section 5.2 presents an in-depth breakdown of the runtimes obtained by SMusket for paired-end correction, which is the most computationally intensive scenario under evaluation. Section 5.3 compares the performance of SMusket with its multithreaded counterpart in order to measure the speedups obtained when distributing the workload across the cluster. Finally, a performance comparison with other state-of-the-art parallel tools is provided in Section 5.4.

### *5.1. Impact of RDD partitioning*

Figure 10 presents the runtimes of SMusket for single-end (left graphs) and paired-end correction (right graphs) when varying the number of partitions of the input RDD, which contains the reads to be corrected. The number of partitions ($NP$) varies from 8 to 48, expressed in terms of partitions created per *Executor* core. Therefore, each *Executor* processes $NP \times 16$ partitions and the total number of partitions can be calculated as $NP \times 16 \times \#nodes$ when using one *Executor* per node as in our testbed.

The most important conclusion that can be drawn from these results is that increasing the number of partitions generally improves performance for all datasets, especially in the range from 8 to 24 partitions per core. The main reason is that some partitions are more computationally intensive to correct than others, which coincide with those regions of the input file that contain reads with more errors. This in turn causes slight workload imbalance among *Executors*. The increase in the total number of partitions reduces the size of each one so that the workload gets distributed more evenly among *Executors*. As can be observed, the performance improvement when creating more partitions decreases as the number of nodes increases since more *Executors* are available for running tasks, which benefits a more balanced workload. In fact, the performance differences are generally small from 24 partitions onwards, especially when using more than 8 nodes. The optimum value is generally in the range from 32 to 48, but it depends on each dataset and number of nodes. We have also checked that performance does not improve from 48 partitions onwards, and it may even worsen slightly due to the overhead of creating and scheduling more tasks. These results also confirm that our tool provides excellent scalability, both for single- and paired-end correction, as performance improves proportionally when using more hardware resources. From here on, all the experimental results shown below have been obtained with the best partitioning configuration for each dataset.

### *5.2. Runtime breakdown*

Figure 11 shows the runtime breakdown of SMusket for paired-end correction when varying the number of nodes from 4 to 16. These graphs allow assessing separately the three stages of the algorithm described in Section 4.3. Roughly speaking, stages 1 and 3 take similar times, especially when using 8 or more nodes. The exception is the SRR50 dataset (see Figure 11(a)), in which the error correction stage is the most computationally demanding: ranging from 60% and 77% of the total runtime. This behaviour can be explained by the number of non-unique $k$-mers that are generated from this dataset, which is an order of magnitude higher than in the others. Consequently, many more weak $k$-mers need to be processed during stage 3 for correcting potential errors on them. This makes SRR50 the most computationally intensive dataset even when it is the smallest one in terms of number of reads (see Table 1).

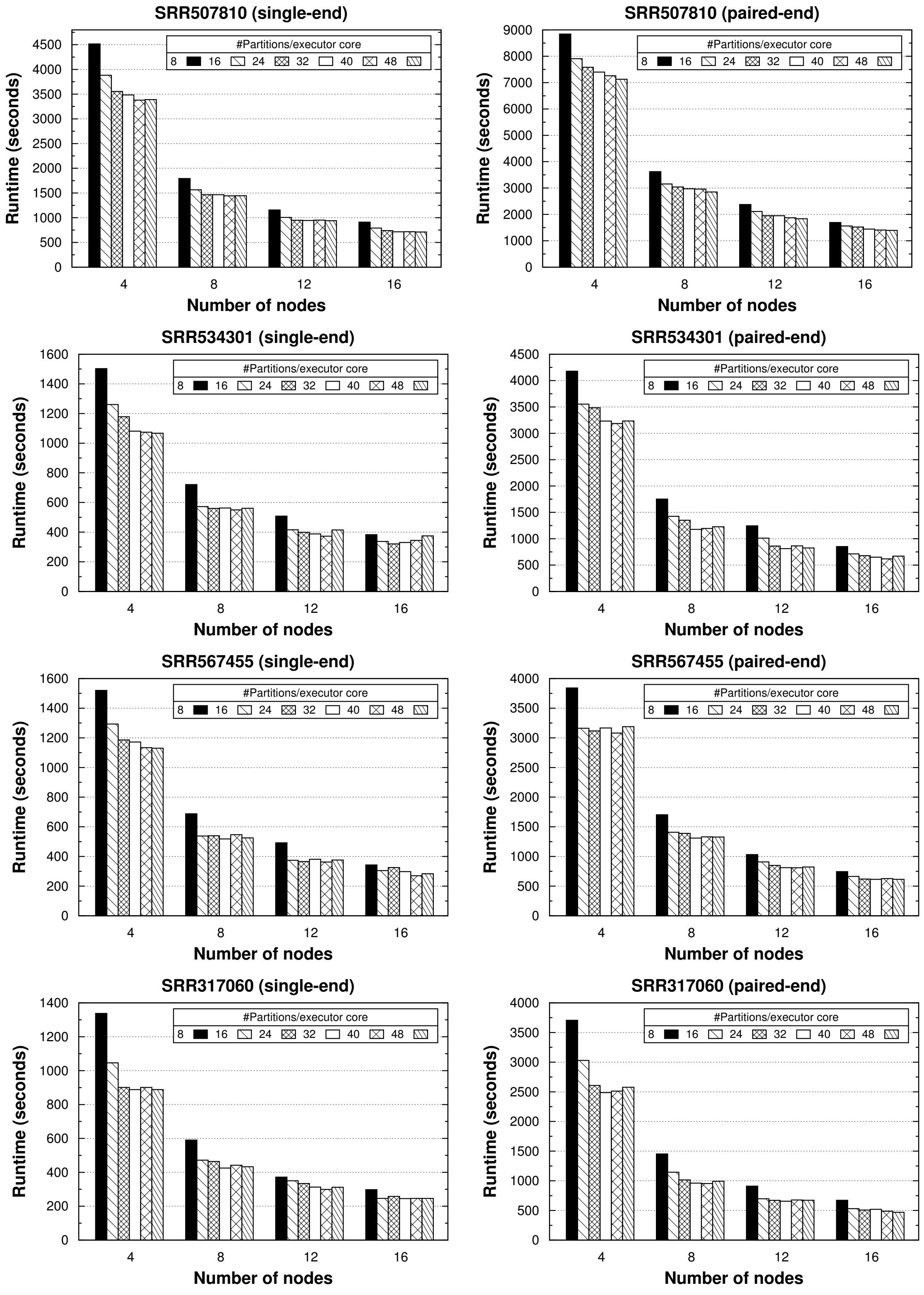
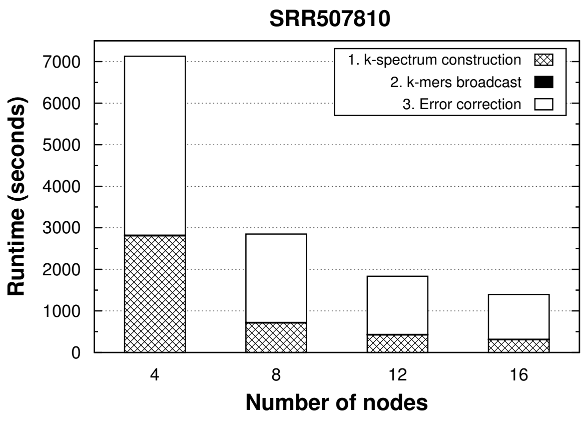
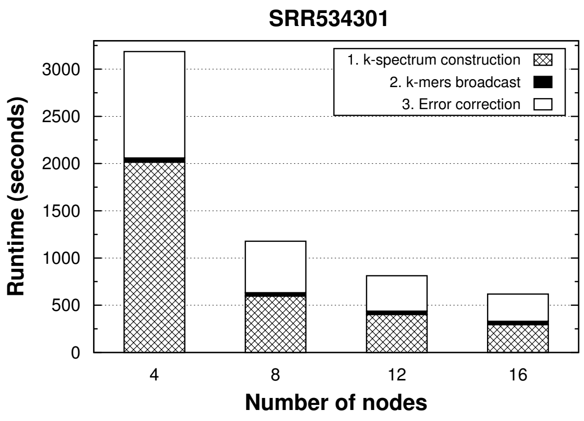


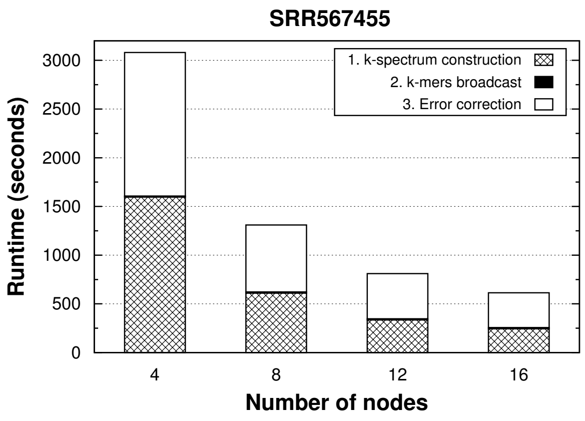
Figure 10: Runtimes of SMusket for single- (left) and paired-end (right) correction when varying the number of RDD partitions



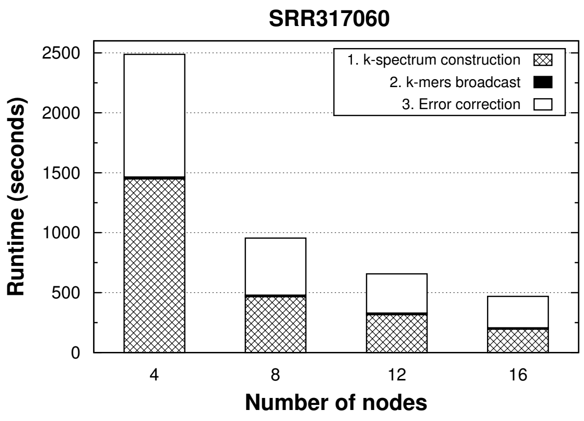
(a)



(b)



(c)



(d)

Figure 11: Runtime breakdown of SMusket for paired-end correction

These results validate our approach for implementing stage 2 based on the broadcast mechanism provided by Spark (see Section 4.3.2), which makes all solid $k$-mers and their multiplicities available to all *Executors*. Note that this is a network-intensive stage that consists of first collecting all solid $k$-mers at the *Driver* side after filtering out the weak ones, which introduces a global synchronization point in the algorithm. Next, solid $k$-mers are broadcast to all *Executors* through the network. As can be seen in Figure 11, its impact on the total runtime can be considered negligible regardless of the dataset and number of nodes. The biggest impact occurs for the SRR53 dataset (see Figure 11(b)). Nevertheless, this stage only represents up to 5.7% of the total runtime when using 16 nodes (about 35 seconds). The SRR53 dataset contains an order of magnitude more solid $k$-mers than the others and thus much more data must be sent across the network.

## *5.3. Performance comparison with Musket*

Table 2 shows the runtimes of SMusket for paired-end correction when using from 1 to 16 nodes, and compares them with those of Musket on a single node using all the available cores (16). This scenario allows measuring the maximum performance benefits of distributing the workload across the cluster. When using the same hardware resources (1 node), SMusket obtains very competitive results taking into account the performance differences between Java and C++, even outperforming Musket for the most computationally intensive dataset (SRR50). From one node onwards, SMusket provides significant speedups over its multi-threaded counterpart: from a minimum of 1.5x on 2 nodes to a maximum of 29.8x on 16. The maximum

Table 2: Runtimes (in seconds) of SMusket and corresponding speedups over Musket for paired-end correction

| SRR50 | | | | SRR53 | | | | SRR56 | | | | SRR31 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Musket** | **SMusket** | | | **Musket** | **SMusket** | | | **Musket** | **SMusket** | | | **Musket** | **SMusket** | | |
| Time | Nodes | Time | Speedup | Time | Nodes | Time | Speedup | Time | Nodes | Time | Speedup | Time | Nodes | Time | Speedup |
| 41506 | 1 | 37185 | 1.1x | 10427 | 1 | 14534 | 0.7x | 12479 | 1 | 13298 | 0.9x | 9626 | 1 | 12399 | 0.8x |
| | 2 | 16493 | 2.5x | | 2 | 7165 | 1.5x | | 2 | 6112 | 2.0x | | 2 | 5322 | 1.8x |
| | 4 | 7127 | 5.8x | | 4 | 3186 | 3.3x | | 4 | 3080 | 4.1x | | 4 | 2487 | 3.9x |
| | 8 | 2850 | 14.6x | | 8 | 1178 | 8.9x | | 8 | 1310 | 9.5x | | 8 | 955 | 10.1x |
| | 12 | 1835 | 22.6x | | 12 | 812 | 12.8x | | 12 | 810 | 15.4x | | 12 | 656 | 14.7x |
| | 16 | 1395 | 29.8x | | 16 | 618 | 16.9x | | 16 | 614 | 20.3x | | 16 | 469 | 20.5x |

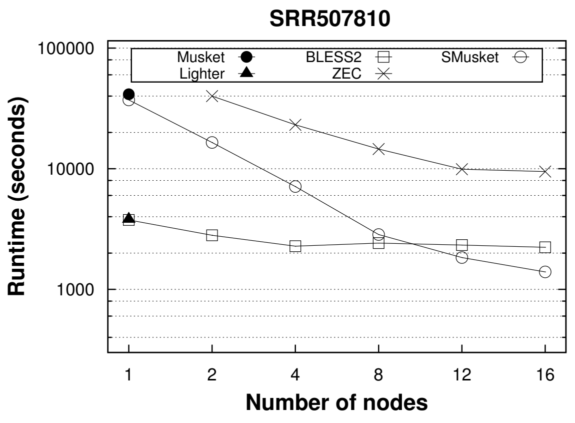
speedups for each dataset and cluster size are always obtained for SRR50, whereas the minimum ones correspond with the most network-intensive dataset (SRR53). The average speedups are 1.9x, 4.3x, 10.8x, 16.4x and 21.9x when using 2, 4, 8, 12 and 16 nodes, respectively, which allows significantly reducing runtimes from several hours to a few minutes.

### *5.4. Performance comparison with other tools*

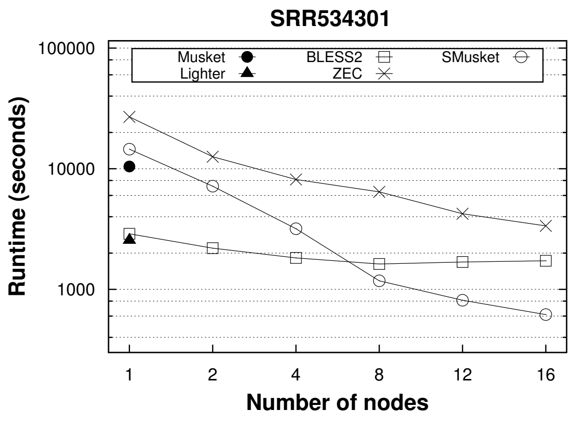
In this section, the performance of SMusket is compared with several publicly available state-of-the-art correctors based on the $k$-spectrum approach (see Section 3): Lighter (v1.1.2) [24], BLESS2 (v1.02) [26] and ZEC (v0.99) [31]. Lighter is a fast multithreaded corrector for shared-memory systems, whose main characteristic is that it uses sampling rather than counting to obtain a set of $k$-mers that are likely from the genome. BLESS2 and ZEC are MPI-based tools that provide support for distributed-memory systems with no specific hardware requirements as SMusket. Both tools perform $k$-mer counting by relying on the KMC library [75]. For all tools, we have used the default settings and disabled read trimming, and the $k$-mer length has been set to 21 ($k = 21$) as in SMusket. Results are shown using one node for Lighter and from 1 to 16 nodes for the distributed-memory tools (BLESS2, ZEC and SMusket). We have also included the results of Musket on one node to be used as reference. It is important to remark when analyzing these results that each tool implements its specific error correction routine, so they can differ greatly in the amount of computation done to correct each read.

Figure 12 reports the measured runtimes for paired-end correction. Note that the results of ZEC on one node for the SRR50 and SRR56 datasets could not be obtained due to runtime failures. On a single node, Lighter clearly outperforms Musket as it avoids counting $k$-mers and implements a lightweight error correction routine, although its accuracy has been proved to be lower [60]. Lighter also outperforms SMusket using up to 4 nodes except for the SRR56 dataset (see Figure 12(c)), where it only outperforms the SMusket single-node configuration. This dataset is the largest one in terms of the total number of input reads (see Table 1), showing the ability of our tool to handle large datasets. Regarding distributed-memory implementations, BLESS2 is generally the fastest tool up to 4 nodes, obtaining very similar performance to Lighter on a single node. However, the scalability of BLESS2 is rather poor since this tool only takes advantage of up to 4 nodes, showing no scalability from that point on, mainly limited by I/O performance when handling large datasets. ZEC is the slowest tool, although it has moderate scalability: up to 8 times faster for the SRR53 and SRR31 datasets when increasing the number of nodes from 1 to 16 (see Figures 12(b) and 12(d)). SMusket is clearly the fastest tool from 8 nodes onwards except for SRR50 (from 12 nodes in this case), showing an excellent scalability trend. More specifically, SMusket is on average around 24.6 times faster on 16 nodes than on a single node.

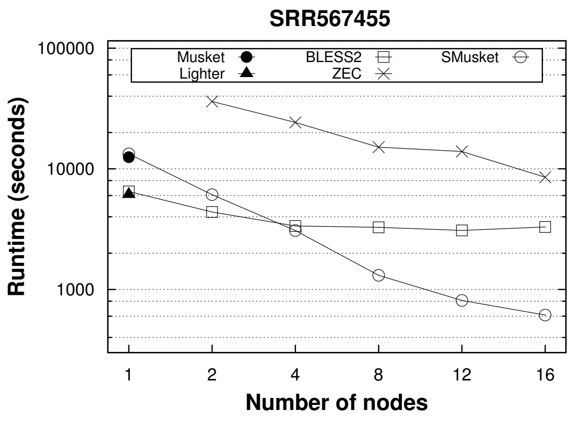
As a summary, Table 3 provides the runtimes of SMusket for single- and paired-end correction using 16 nodes and the corresponding speedups obtained over all the evaluated tools. As can be observed, SMusket achieves significant speedups of up to 6.2x, 15.3x, 10x and 29.8x over BLESS2, ZEC, Lighter and Musket, respectively. These results validate our Spark-based algorithm as they prove that SMusket is on average 3.4 and 10.1 times faster than previous MPI-based tools (BLESS2 and ZEC, respectively). This in turn reinforces the suitability of Big Data technologies such as Spark to tackle the expected growth in the size of genomic datasets.



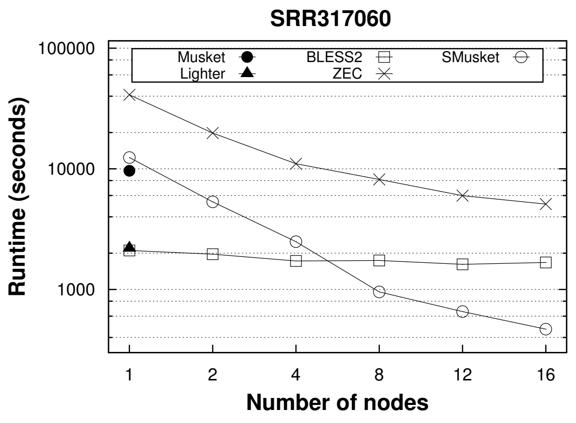
(a)



(b)



(c)



(d)

Figure 12: Runtimes of several state-of-the-art parallel tools for paired-end correction (logarithmic scale)

Table 3: Runtimes (in seconds) of all the tools and corresponding speedups obtained by SMusket

| **Dataset** | | **16 nodes** | | | | | **1 node** | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | **SMusket** | **BLESS2** | | **ZEC** | | **Lighter** | | **Musket** | |
| | | | Runtime | Speedup | Runtime | Speedup | Runtime | Speedup | Runtime | Speedup |
| Single-end | SRR50 | 710 | 1148 | 1.6x | 7020 | 9.9x | 1424 | 2.0x | 14875 | 21.0x |
| | SRR53 | 321 | 893 | 2.8x | 1740 | 5.4x | 846 | 2.6x | 3690 | 11.5x |
| | SRR56 | 270 | 1667 | 6.2x | 4120 | 15.3x | 1987 | 7.4x | 4522 | 16.7x |
| | SRR31 | 245 | 827 | 3.4x | 3240 | 13.2x | 854 | 3.5x | 4410 | 18.0x |
| Paired-end | SRR50 | 1395 | 2234 | 1.6x | 9480 | 6.8x | 3810 | 2.7x | 41506 | 29.8x |
| | SRR53 | 618 | 1725 | 2.8x | 3360 | 5.4x | 2548 | 4.1x | 10427 | 16.9x |
| | SRR56 | 614 | 3300 | 5.4x | 8520 | 13.9x | 6120 | 10.0x | 12479 | 20.3x |
| | SRR31 | 469 | 1673 | 3.6x | 5100 | 10.9x | 2195 | 4.7x | 9626 | 20.5x |
| Average speedup | | | | 3.4x | | 10.1x | | 4.6x | | 19.3x |

## 6. Conclusions

The massive amount of data produced by modern NGS technologies reinforces the need for scalable tools with the ability to perform parallel computations by taking advantage of distributed-memory systems. In this paper we have presented SMusket, a Big Data tool that fully exploits the features of Apache Spark to boost the performance of Musket, a popular and accurate DNA read error corrector. Our tool extends the correction capabilities of Musket to distributed-memory systems obtaining high scalability and providing the same accuracy as its counterpart. Moreover, SMusket is especially intended for clusters based on commodity processing nodes, as it does not require any specific hardware device or feature.

The performance of our tool has been extensively evaluated on a 16-node cluster using four publicly available datasets. The experimental results have shown that SMusket provides significant performance improvements that range from 11.5 to 29.8 times faster than its multithreaded counterpart. Furthermore, a comparison between SMusket and previous MPI-based correctors was also carried out, showing that our tool is the fastest one achieving speedups of up to 6.2x and 15.3x over BLESS2 and ZEC, respectively, when executed on the same hardware. SMusket is distributed as free open-source software released under the GNU GPLv3 license and is publicly available at https://github.com/rreye/smusket.

As future work, we aim to evaluate the performance of our tool on public cloud platforms such as Amazon EMR and Azure HDInsight. Furthermore, we intend to add support for other correction routines based on other $k$-spectrum techniques, turning SMusket into a generic error correction tool.

## Acknowledgments

This work was supported by the Ministry of Economy, Industry and Competitiveness of Spain and FEDER funds of the European Union (project TIN2016-75845-P, AEI/FEDER/EU); and by Xunta de Galicia (projects ED431G/01 and ED431C 2017/04).

## References

[1] K. A. Phillips, Assessing the value of next-generation sequencing technologies: an introduction, Value in Health 21 (9) (2018) 1031–1032.

[2] Z. D. Stephens, et al., Big data: astronomical or genomical?, PLoS Biol 13 (7) (2015) e1002195.

[3] H. Y. K. Lam, et al., Performance comparison of whole-genome sequencing platforms, Nat Biotechnol 30 (1) (2012) 78–82.

[4] C. Alkan, S. Sajjadian, E. E. Eichler, Limitations of next-generation genome sequence assembly, Nat Methods 8 (1) (2011) 61–65.

[5] H. Li, R. Durbin, Fast and accurate short read alignment with Burrows-Wheeler transform, Bioinformatics 25 (14) (2009) 1754–1760.

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

[7] Q. Zou, X.-B. Li, W.-R. Jiang, Z.-Y. Lin, G.-L. Li, K. Chen, Survey of MapReduce frame operation in bioinformatics, Brief Bioinform 15 (4) (2013) 637–647.

[8] J. Cała, E. Marei, Y. Xu, K. Takeda, P. Missier, Scalable and efficient whole-exome data processing using workflows on the cloud, Future Gener Comput Syst 65 (2016) 153–168.

[9] A. O'Driscoll, J. Daugelaite, R. D. Sleator, 'Big data', Hadoop and cloud computing in genomics, J Biomed Inform 46 (5) (2013) 774–781.

[10] C. Smowton, et al., A cost-effective approach to improving performance of big genomic data analyses in clouds, Future Gener Comput Syst 67 (2017) 368–381.

[11] J. Luo, M. Wu, D. Gopukumar, Y. Zhao, Big data application in biomedical research and health care: a literature review, Biomed Inform Insights 8 (2016) 1–10.

[12] M. Zaharia, et al., Apache Spark: a unified engine for Big Data processing, Commun ACM 59 (11) (2016) 56–65.

[13] M. Zaharia, et al., Resilient Distributed Datasets: a fault-tolerant abstraction for in-memory cluster computing, in: Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI'12), San Jose, CA, USA, 2012, pp. 15–28.

[14] Y. Liu, J. Schröder, B. Schmidt, Musket: a multistage k-mer spectrum-based error corrector for Illumina sequence data, Bioinformatics 29 (3) (2013) 308–315.

[15] M. Chaisson, P. Pevzner, H. Tang, Fragment assembly with short reads, Bioinformatics 20 (13) (2004) 2067–2074.

[16] X. Yang, S. Chockalingam, S. Aluru, A survey of error-correction methods for next-generation sequencing, Brief Bioinform 14 (1) (2012) 56–66.

[17] M. Molnar, L. Ilie, Correcting Illumina data, Brief Bioinform 16 (4) (2014) 588–599.

[18] X. Yang, K. Dorman, S. Aluru, Reptile: representative tiling for short read error correction, Bioinformatics 26 (20) (2010) 2526–2533.

[19] D. R. Kelley, M. C. Schatz, S. L. Salzberg, Quake: quality-aware detection and correction of sequencing errors, Genome Biol 11 (11) (2010) R116.

[20] H. Shi, B. Schmidt, W. Liu, W. Müller-Wittig, A parallel algorithm for error correction in high-throughput short-read data on CUDA-enabled graphics hardware, J Comput Biol 17 (4) (2010) 603–615.

[21] Y. Liu, B. Schmidt, D. L. Maskell, DecGPU: distributed error correction on massively parallel graphics processing units using CUDA and MPI, BMC Bioinformatics 12 (1) (2011) 85.

[22] J. T. Simpson, R. Durbin, Efficient de novo assembly of large genomes using compressed data structures, Genome Res 22 (3) (2012) 549–556.

[23] L. Ilie, M. Molnar, RACER: rapid and accurate correction of errors in reads, Bioinformatics 29 (19) (2013) 2490–2493.

[24] L. Song, L. Florea, B. Langmead, Lighter: fast and memory-efficient sequencing error correction without counting, Genome Biol 15 (11) (2014) 509.

[25] Y. Heo, X.-L. Wu, D. Chen, J. Ma, W.-M. Hwu, BLESS: bloom filter-based error correction solution for high-throughput sequencing reads, Bioinformatics 30 (10) (2014) 1354–1362.

[26] Y. Heo, A. Ramachandran, W.-M. Hwu, J. Ma, D. Chen, BLESS 2: accurate, memory-efficient and fast error correction method, Bioinformatics 32 (15) (2016) 2369–2371.

[27] H. Li, BFC: correcting Illumina sequencing errors, Bioinformatics 31 (17) (2015) 2885–2887.

[28] A. Ramachandran, Y. Heo, W.-M. Hwu, J. Ma, D. Chen, FPGA accelerated DNA error correction, in: Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition (DATE'15), Grenoble, France, 2015, pp. 1371–1376.

[29] M. Długosz, S. Deorowicz, RECKONER: read error corrector based on KMC, Bioinformatics 33 (7) (2017) 1086–1089.

[30] K. Xu, et al., SPECTR: scalable parallel short read error correction on multi-core and many-core architectures, in: Proceedings of the 47th International Conference on Parallel Processing (ICPP 2018), Eugene, OR, USA, 2018, pp. 39:1–39:10.

[31] L. Zhao, et al., Mining statistically-solid k-mers for accurate NGS error correction, BMC Genomics 19 (10) (2018) 912.

[32] J. Schröder, H. Schröder, S. J. Puglisi, R. Sinha, B. Schmidt, SHREC: a short-read error correction method, Bioinformatics 25 (17) (2009) 2157–2163.

[33] L. Salmela, Correction of sequencing errors in a mixed set of reads, Bioinformatics 26 (10) (2010) 1284–1290.

[34] L. Ilie, F. Fazayeli, S. Ilie, HiTEC: accurate error correction in high-throughput sequencing data, Bioinformatics 27 (3) (2010) 295–302.

[35] M. H. Schulz, et al., Fiona: a parallel and automatic strategy for read error correction, Bioinformatics 30 (17) (2014) i356–i363.

[36] L. Salmela, J. Schröder, Correcting errors in short reads by multiple alignments, Bioinformatics 27 (11) (2011) 1455–1461.

[37] W.-C. Kao, A. H. Chan, Y. S. Song, ECHO: a reference-free short-read error correction algorithm, Genome Res 21 (7) (2011) 1181–1192.

[38] C.-C. Chen, Y.-J. Chang, W.-C. Chung, D. T. Lee, J.-M. Ho, CloudRS: an error correction algorithm of high-throughput sequencing data based on scalable framework, in: Proceedings of the IEEE International Conference on Big Data (IEEE BigData 2013), Santa Clara, CA, USA, 2013, pp. 717–722.

[39] W.-C. Chung, J.-M. Ho, C.-Y. Lin, D. T. Lee, CloudEC: a MapReduce-based algorithm for correcting errors in next-generation sequencing Big Data, in: Proceedings of the IEEE International Conference on Big Data (IEEE BigData 2017), Boston, MA, USA, 2017, pp. 2836–2842.

[40] L. Zhao, Q. Chen, W. Li, P. Jiang, L. Wong, J. Li, MapReduce for accurate error correction of next-generation sequencing data, Bioinformatics 33 (23) (2017) 3844–3851.

[41] S. Ghemawat, H. Gobioff, S.-T. Leung, The Google file system, SIGOPS Oper Syst Rev 37 (5) (2003) 29–43.

[42] The Apache Software Foundation, Apache Hadoop, http://hadoop.apache.org, [cited 28 August 2019].

[43] K. Shvachko, H. Kuang, S. Radia, R. Chansler, The Hadoop distributed file system, in: Proceedings of the IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST'2010), Incline Village, NV, USA, 2010, pp. 1–10.

[44] V. K. Vavilapalli, et al., Apache Hadoop YARN: Yet Another Resource Negotiator, in: Proceedings of the 4th Annual Symposium on Cloud Computing (SOCC'13), Santa Clara, CA, USA, 2013, pp. 5:1–5:16.

[45] The Apache Software Foundation, Apache Spark: lightning-fast cluster computing, https://spark.apache.org, [cited 28 August 2019].

[46] B. Hindman, et al., Mesos: a platform for fine-grained resource sharing in the data center, in: Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI'11), Boston, MA, USA, 2011, pp. 295–308.

[47] E. A. Brewer, Kubernetes and the path to cloud native, in: Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC'15), Kohala Coast, HI, USA, 2015, p. 167.

[48] V. Chang, Towards data analysis for weather cloud computing, Knowl Based Syst 127 (2017) 29–45.

[49] Y. Wang, L. Kung, T. A. Byrd, Big Data analytics: understanding its capabilities and potential benefits for healthcare organizations, Technol Forecast Soc Change 126 (2018) 3–13.

[50] V. Chang, Computational intelligence for medical imaging simulations, J Med Syst 42 (1) (2018) 10.

[51] S. Peng, S. Yu, P. Mueller, Social networking Big Data: opportunities, solutions, and challenges, Future Gener Comput Syst 86 (2018) 1456–1458.

[52] M. H. ur Rehman, I. Yaqoob, K. Salah, M. Imran, P. P. Jayaraman, C. Perera, The role of Big Data analytics in industrial Internet of Things, Future Gener Comput Syst 99 (2019) 247–259.

[53] S. Min, B. Lee, S. Yoon, Deep learning in bioinformatics, Brief Bioinform 18 (5) (2016) 851–869.

[54] V. Chang, Data analytics and visualization for inspecting cancers and genes, Multimed Tools Appl 77 (14) (2018) 17693–

17707.
[55] 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.
[56] 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.
[57] R. R. Expósito, J. González-Domínguez, J. Touriño, HSRA: Hadoop-based spliced read aligner for RNA sequencing data, PLoS ONE 13 (7) (2018) e0201483.
[58] Y.-J. Chang, C.-C. Chen, C.-L. Chen, J.-M. Ho, A de novo next generation genomic sequence assembler based on string graph and MapReduce cloud computing framework, BMC Genomics 13 (7) (2012) S28.
[59] A. Abu-Doleh, Ü. V. Çatalyürek, Spaler: Spark and GraphX based de novo genome assembler, in: Proceedings of the IEEE International Conference on Big Data (IEEE BigData 2015), Santa Clara, CA, USA, 2015, pp. 1013–1018.
[60] I. Akogwu, N. Wang, C. Zhang, P. Gong, A comparative study of k-spectrum-based error correction methods for next-generation sequencing data analysis, Hum Genomics 10 (2) (2016) 20.
[61] Message Passing Interface Forum, MPI: a Message Passing Interface standard, http://www.mpi-forum.org, [cited 28 August 2019].
[62] L. Dagum, R. Menon, OpenMP: an industry-standard API for shared-memory programming, IEEE Comput Sci Eng 5 (1) (1998) 46–55.
[63] D. Luebke, CUDA: scalable parallel programming for high-performance scientific computing, in: Proceedings of the 5th IEEE International Symposium on Biomedical Imaging: from Nano to Macro (ISBI'08), Paris, France, 2008, pp. 836–838.
[64] M. Hertz, E. D. Berger, Quantifying the performance of garbage collection vs. explicit memory management, in: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'05), San Diego, CA, USA, 2005, pp. 313–326.
[65] M. Fourment, M. R. Gillings, A comparison of common programming languages used in bioinformatics, BMC Bioinformatics 9 (1) (2008) 82.
[66] R. V. Pandey, C. Schlötterer, DistMap: a toolkit for distributed short read mapping on a Hadoop cluster, PLoS ONE 8 (8) (2013) e72614.
[67] M. Niemenmaa, A. Kallio, A. Schumacher, P. Klemelä, E. Korpelainen, K. Heljanko, Hadoop-BAM: directly manipulating next generation sequencing data in the cloud, Bioinformatics 28 (6) (2012) 876–877.
[68] H. Nordberg, K. Bhatia, K. Wang, Z. Wang, BioPig: a Hadoop-based analytic toolkit for large-scale sequence data, Bioinformatics 29 (23) (2013) 3014–3019.
[69] U. Ferraro Petrillo, G. Roscigno, G. Cattaneo, R. Giancarlo, FASTdoop: a versatile and efficient library for the input of FASTA and FASTQ files for MapReduce Hadoop bioinformatics applications, Bioinformatics 33 (10) (2017) 1575–1577.
[70] R. R. Expósito, J. González-Domínguez, J. Touriño, Hadoop Sequence Parser (HSP) library for FASTQ/FASTA datasets, https://github.com/rreye/hsp, [cited 28 August 2019].
[71] J. Veiga, J. Enes, R. R. Expósito, J. Touriño, BDEv 3.0: energy efficiency and microarchitectural characterization of Big Data processing frameworks, Future Gener Comput Syst 86 (2018) 565–581.
[72] J. Veiga, R. R. Expósito, G. L. Taboada, J. Touriño, BDEv: Big Data Evaluator tool, http://bdev.des.udc.es, [cited 28 August 2019].
[73] R. Leinonen, et al., The European Nucleotide Archive, Nucleic Acids Res 39 (suppl_1) (2010) D28–D31.
[74] The European Bioinformatics Institute, The European Nucleotide Archive (ENA), https://www.ebi.ac.uk/ena, [cited 28 August 2019].
[75] S. Deorowicz, M. Kokot, S. Grabowski, A. Debudaj-Grabysz, KMC 2: fast and resource-frugal k-mer counting, Bioinformatics 31 (10) (2015) 1569–1576.