Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

BigDEC: A multi-algorithm Big Data tool based on the *k*-mer spectrum method for scalable short-read error correction

Roberto R. Expósito [*], Jorge González-Domínguez

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

ARTICLE INFO

Dataset link: https://www.ncbi.nlm.nih.gov/sra

Keywords:
Big Data processing
Next-Generation Sequencing (NGS)
Error correction
Apache Spark
Apache Flink

ABSTRACT

Despite the significant improvements in both throughput and cost provided by modern Next-Generation Sequencing (NGS) platforms, sequencing errors in NGS datasets can still degrade the quality of downstream analysis. Although state-of-the-art correction tools can provide high accuracy to improve such analysis, they are limited to apply a single correction algorithm while also requiring long runtimes when processing large NGS datasets. Furthermore, current parallel correctors generally only provide efficient support for shared-memory systems lacking the ability to scale out across a cluster of multicore nodes, or they require the availability of specific hardware devices or features. In this paper we present a Big Data Error Correction (BigDEC) tool that overcomes all those limitations by: (1) implementing three different error correction algorithms based on the widely extended *k*-mer spectrum method; (2) providing scalable performance for large datasets by efficiently exploiting the capabilities of Big Data technologies on multicore clusters based on commodity hardware; (3) supporting two different Big Data processing frameworks (Spark and Flink) to provide greater flexibility to end users; (4) including an efficient, stream-based merge operation to ease downstream processing of the corrected datasets; and (5) significantly outperforming existing parallel tools, being up to 79% faster on a 16-node multicore cluster when using the same underlying correction algorithm. BigDEC is publicly available to download at https://github.com/UDC-GAC/BigDEC.

1. Introduction

Currently, Next-Generation Sequencing (NGS) [1,2] is considered as an extremely useful technology for the prevention, diagnosis and treatment of a wide spectrum of diseases including genetic conditions, chronic pathologies and infectious diseases, or even to reveal the progression of SARS-CoV2 [3–5]. The continuous development of NGS over the years has made it possible to generate hundreds of millions of DNA sequence fragments (the so-called reads) in a single run, while drastically reducing its financial cost. For example, the price of sequencing the human genome had been reduced to less than $1,000 in 2018 [6]. Furthermore, it is expected that the sequencing capabilities will continue to grow in the coming years, with some studies predicting that between 100 million and up to 2 billion human genomes could be sequenced by 2025 [7].

Nowadays, the dominant NGS platform in the market are Illumina sequencers [8], which are characterized by generating short DNA reads (most often 50–150 base pairs). Due to their cost efficiency and relatively high accuracy, Illumina datasets are frequently used for many fundamental applications such as *de novo* genome assembly [9] or cancer mutation discovery. However, NGS platforms are imperfect and can introduce sequencing errors in generated reads which can affect the quality of downstream analysis. For instance, it is estimated that Illumina sequencers introduce approximately one error in every one thousand nucleotides [10]. Therefore, error correction is an important pre-processing step in many NGS pipelines. The underlying idea is that the prior usage of correction tools on raw NGS datasets provides assemblers with a cleaner input and subsequently leads to improved results.

Despite the rich abundance of tools aimed at correcting short-read NGS datasets [11–17], a common drawback is that they are all limited to apply a single correction algorithm over the input data. According to multiple studies found in the literature [18–20], the accuracy of error correction algorithms varies substantially across different types of datasets, genome coverage, error distribution or even the organism from which reads have been obtained, with no single tool performing best on all the scenarios [20]. This fact makes it difficult for bioinformaticians to decide in advance which one to use for a particular NGS pipeline. Therefore, having a single tool capable of correcting errors by applying several algorithms separately can provide better accuracy

[*] Corresponding author.
E-mail addresses: roberto.rey.exposito@udc.es (R.R. Expósito), jgonzalezd@udc.es (J. González-Domínguez).

https://doi.org/10.1016/j.future.2024.01.011
Received 22 March 2023; Received in revised form 6 October 2023; Accepted 10 January 2024
Available online 12 January 2024
0167-739X/© 2024 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

in a greater variety of scenarios. This tool should be easy to use and computationally efficient for very large NGS datasets.

Furthermore, many of the existing correctors can only take advantage of the computational resources of multicore architectures to reduce correction times, thus being limited to support shared-memory systems through multithreading. Some other correctors require the availability of specific hardware devices (e.g GPUs) or CPU features (e.g. AVX-512), which reduces the range of potential users who can use them. Although there are a few tools that can be executed on distributed-memory systems such as multicore clusters, their performance is suboptimal when processing very large datasets. This issue is specially relevant in the NGS context due to the explosion in the amount of genomic data being generated, which demands the use of more efficient scale-out approaches both in terms of computing and storage resources, something that can be tackled through exploiting Big Data technologies [21–24].

In this paper we present BigDEC, a Big Data Error Correction parallel tool intended for short-read NGS datasets that provides the following contributions over the state of the art:

- Up to our knowledge, this is the first corrector that integrates three different approaches within the same tool. The algorithms included in BigDEC are based on the k-mer spectrum method, which is the most extended technique for short-read error correction due to its simplicity and good accuracy. More specifically, the supported algorithms are based on those proposed by Musket [11], BLESS2 [12] and RECKONER [14]. These algorithms can be performed individually over the input dataset in a single execution of the tool, thus enabling potential optimizations in terms of performance due to the sharing of certain computation phases.
- Up to our knowledge, BigDEC is also the first parallel corrector that can be executed using different data processing engines, currently supporting two popular open-source and widely extended Big Data processing frameworks: Apache Spark [25] and Apache Flink [26]. Along with the support for the Hadoop Distributed File System (HDFS) [27] to store large NGS datasets, BigDEC extends the execution of multiple correction algorithms to distributed-memory systems while providing more flexibility to scientists than previous tools.
- The parallel approach included in BigDEC, based on the combination of the aforementioned Big Data technologies, ensures high scalability when correcting large datasets by efficiently exploiting the performance of multicore clusters based on commodity nodes without requiring any specific hardware device or feature. Furthermore, BigDEC implements an efficient stream-based merge operation that combines all the corrected output files generated on the cluster nodes into a single output file. This feature is useful in those scenarios where downstream analyses are performed with tools that do not support distributed processing over HDFS, and its efficiency is key to minimize disk and network overheads.
- BigDEC has been implemented in "pure" Java code (i.e. 100% Java) to maximize cross-platform portability across different systems, supporting both single- and paired-end datasets stored in FASTQ [28], a sequence format widely spread among state-of-the-art correctors. Our tool is publicly available to the scientific community as free open-source software released under the GNU GPLv3 license.

The rest of this paper is organized 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 16-node cluster to assess the performance and scalability of BigDEC comparatively with other state-of-the-art tools. Finally, Section 6 concludes the paper.

2. Background

This section introduces the main concepts needed to understand our proposal, namely background about Big Data technologies, the basics of the k-mer spectrum-based error correction method and some details regarding the FASTQ format.

2.1. Big data technologies

Scientists and researchers are currently facing new challenges when storing and analyzing large datasets. The characteristics of Big Data require powerful and novel parallel approaches to extract meaningful information from such massive datasets in a scalable manner by efficiently exploiting the computational resources of distributed-memory systems. The Google MapReduce programming paradigm [29] and its associated open-source implementation, the Apache Hadoop project [30], were the cornerstone of Big Data processing over the last decade. In fact, the exploitation of such technologies has transformed multiple disciplines including weather forecasting [31], healthcare [32,33], medical imaging simulation [34], deep learning [35] and genome analysis [21,36,37], among others. The key for their success is that the MapReduce model provides a quite convenient way to implement distributed applications, allowing programmers to focus on the task rather than on other low-level parallelization issues such as interprocess communications. This is opposed to other previous models such as MPI, which requires much more effort from programmers to obtain high performance.

Furthermore, Hadoop also provides a distributed storage layer in order to efficiently support the MapReduce model: the Hadoop Distributed File System (HDFS) [27]. HDFS is a block-oriented file system implemented in Java that is specifically designed to provide high bandwidth by distributing and replicating data across a cluster of commodity machines. This file system has built-in fault tolerance by using a data block replication scheme, and the number of times that each block is replicated over the cluster is known as the replication factor. Relying on HDFS, Hadoop MapReduce attempts to schedule processing tasks on the cluster machines where the input data blocks reside, thus minimizing data movements across the network.

Nowadays, Hadoop MapReduce has been largely superseded by more advanced frameworks such as Apache Spark [25] and Apache Flink [26], which also take advantage of HDFS features. These frameworks overcome the disadvantages of MapReduce by providing a richer programming API to allow for more flexible data-parallel operations over distributed datasets, by reducing disk-based processing through in-memory computations to improve overall performance, by supporting streaming and interactive data processing, and by offering interfaces to implement applications in multiple languages (e.g. Java, Scala, Python, R) in order to ease software development. At the highest level of abstraction, both Spark and Flink provide programmers with appropriate interfaces and data structures that allow to process a large dataset as a collection of data elements distributed over a cluster of machines, which can be efficiently operated in parallel. More details about these frameworks are provided next, mostly focused on their support for batch processing as this is the relevant context for this work.

2.1.1. Spark and Flink overview

Spark is an open-source, general-purpose Big Data framework that supports both batch and streaming processing models. The fundamental data structure in Spark is based on the Resilient Distributed Dataset (RDD) [38], which provides an immutable, distributed collection of data elements partitioned across the cluster that can be created in different ways, for instance by loading an external dataset from supported file systems such as HDFS. Once created, an RDD can be manipulated using a rich set of data-parallel operations (e.g. *map*, *filter*). For instance, the *map* transformation processes each RDD element through a user-defined function and returns a new RDD representing the results,

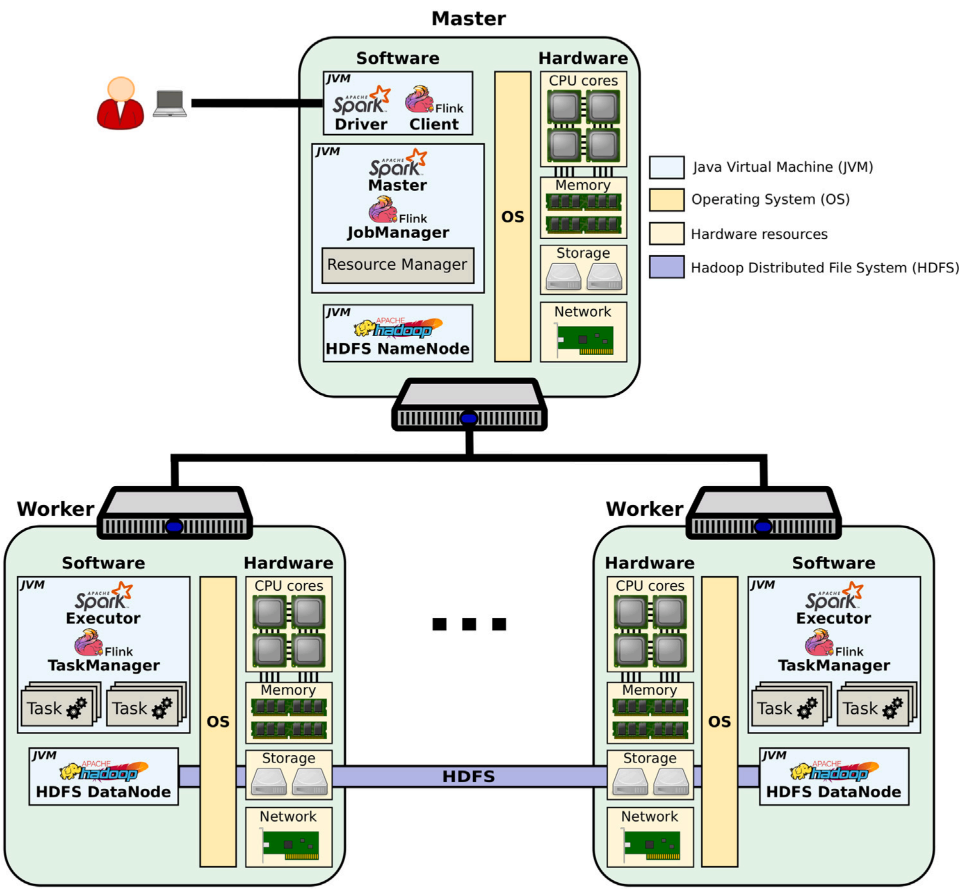


Fig. 1. Standalone cluster deployment for Spark and Flink for batch processing on HDFS.

while *filter* returns a new RDD formed by selecting only those elements of the source RDD on which a user-defined function returns true.

Although RDD was the primary user-facing API since the inception of Spark, new higher-level APIs have been introduced later to make large-scale processing even easier: Dataframes/Datasets. On the one hand, Dataframes are, like RDDs, immutable distributed collections of data but, differently to RDDs, such data are organized into named columns like a table in a relational database. They allow developers to impose a structure onto a distributed dataset, providing a domain specific language to manipulate it. Dataframe API is built on top of the Spark SQL engine [39], which takes advantage of Catalyst to generate an optimized logical and physical query plan to improve performance [40]. On the other hand, Dataset API is just an extension of Dataframes which provides the functionality of type-safe, object-oriented interface of the RDD API together with the performance benefits of the Catalyst optimizer.

While Spark is mainly a batch processing engine that also supports streaming computations, Flink is a native, record-by-record stream engine that supports batch processing by simply considering batches to be bounded data streams. The basic building blocks of Flink applications are composed of streaming dataflows that can be transformed by user-defined operators. These dataflows form directed graphs that start with one or more data sources and finish in one or more data sinks. For batch processing, the Flink Dataset API allows programmers to perform transformations (e.g. *map*, *filter*) on distributed data collections in a similar way to Spark. These Datasets are initially created from certain data sources (e.g. by reading files, or from local collections), and results are returned via data sinks which may, for example, write the data to distributed files.

From now on, we will use the term “Dataset” regardless of the framework/API to refer to a distributed collection of data elements, whereas “dataset” will be used for genomic data (i.e. FASTQ reads in our context). The term for a specific data structure or API (e.g. Spark RDD) will be used to explain or clarify something specifically related to it.

2.1.2. *Cluster deployment*

Both frameworks are designed to be deployed on a cluster of commodity machines without requiring any specific hardware on them, by following a master/worker architecture as depicted in Fig. 1. It is important to remark that these frameworks are implemented in programming languages that run on top of the Java Virtual Machine (JVM): Spark in Scala and Flink in Java, just like HDFS (also in Java). So, their runtime systems are composed of software components in the form of JVM processes deployed across the cluster machines. More specifically, these systems consist of two types of JVMs: one that runs on the master node, and one (or more) executed on the worker nodes. For Spark, such JVMs are called *Master/Executor*, respectively, and *JobManager/TaskManager* in the case of Flink.

As can be seen Fig. 1, users launch their applications through the master node. Spark/Flink applications run as independent sets of processes across the cluster that are coordinated by the main function of the program, which is executed within a JVM that usually also runs on the master node. This main program, called Spark *Driver* or Flink *Client*, defines the Datasets and declares the parallel transformations to be carried out over them, submitting such requests to the Spark *Master* or Flink *JobManager* to be processed across the cluster. The worker nodes are the places where all the processing tasks are scheduled among the *Executors/TaskManagers*. Their computational resources determine the maximum parallelism for the applications, and they are generally configured to allow the execution of as many processing tasks as CPU cores available on each worker node. Their resources are allocated

```
@ERR580958.25 DHKW5DQ1:300:C233YACXX:7:1101:1564:2227/1
CTAGGACTGCAGGAATGAAATCCACGAGACGCAGACCTCTGGGACAATCTACGAACAGAATCGAGACTCTTTTTT
+
;??DDDDDDBFHFGIIEGIIIGG3FGBEHGIDHDHIGGEFHIGI@DFGIII@G8EEAA??C:@9=?<;;ACCCCB
@ERR580958.26 DHKW5DQ1:300:C233YACXX:7:1101:1949:2047/1
TCTCTCATGCAGGTATTCCCACCTGGACGGAGATCCCAAAGAGGTGTCCCCGGTCATCGATCAGTTCCTGGAGTG
+
<?@DBDDDFHHHHGDFG<BFEHGGICHGGIG:DFHBFFHGGGHIFCDHGFBEH<A<ECDC=BACDD((6;;5=(5
```

Fig. 2. Example of two DNA reads in FASTQ format (75 bases).

to Spark/Flink applications through a cluster manager. Although both frameworks support external cluster managers such as YARN [41], they can also run in standalone mode by providing a built-in resource manager and scheduler. For simplicity, we will focus on this standalone mode in this work.

Regarding distributed storage, HDFS also relies on a master/worker architecture, where the deployment consists of running a JVM process on the master node (*NameNode*) and one JVM per worker node (*DataNode*). On the one hand, the *NameNode* is in charge of storing the metadata, so it knows for any given file in HDFS the list of data blocks, their location on the workers and the number of replicas, among other information. On the other hand, the *DataNodes* are responsible for storing the actual data blocks using local disks physically attached to worker nodes. They perform the low-level read and write requests from the HDFS clients, which are the Spark/Flink applications in our context.

2.2. *k-mer spectrum-based error correction*

Multiple methods have been proposed to correct sequencing errors in NGS datasets [18,19,42,43]. Among them, the most popular one for correcting short reads is the k-mer spectrum-based (or k-spectrum-based) approach, mainly due to its simplicity, competitive accuracy and good performance compared to alignment-based approaches. The general idea of the k-spectrum-based method consists in generating the set of distinct overlapping substrings of a fixed length k (so-called k-mers) from the input reads. The corresponding frequency for each k-mer (i.e. its multiplicity) is determined during 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 k-mer is very low we can assume that it contains erroneous bases. Those k-mers with a multiplicity equal to or higher than a certain threshold are considered as solid (trusted) k-mers, whereas the remaining ones are considered weak (or untrusted). Solid k-mers are highly likely to occur in the genome, being unlikely to have been altered by sequencing errors. Therefore, input reads that only contain solid k-mers are assumed to be error-free, whereas those reads that contain weak k-mers are corrected by repeatedly trying to replace them by similar solid k-mers. The k-mer multiplicity threshold that separates solid k-mers from weak ones can be often specified by the user, but most correctors can automatically determine an appropriate value from the k-mer multiplicity histogram. Overall, many k-spectrum-based tools mainly differ in the specific strategy used for implementing the error correction step, which ultimately determines its accuracy and computational efficiency.

2.3. *FASTQ sequence format*

The FASTQ format [28] is a human-readable, text-based file format intended for storing both a biological sequence (usually nucleotide sequence) and its corresponding quality scores. Each of the nucleotides and quality scores are encoded using a single ASCII character in order to minimize the file size. This format is currently considered de facto standard for storing the output of high-throughput NGS platforms such as Illumina sequencers, among many others.

FASTQ represents each DNA sequence using four lines, as can be seen in Fig. 2 which shows an extract from a FASTQ file containing two DNA reads. The first line begins with a '@' character that marks the beginning of the sequence. Everything from the leading '@' to the first whitespace character is considered the sequence identifier (e.g. ERR580958.25). After the first space, an optional sequence description can appear in this first line, which usually contains specific information from the NGS sequencer. The second line contains the raw nucleotide bases: adenine ('A'), cytosine ('C'), guanine ('G') or thymine ('T'). An 'N' base in this line means that the NGS sequencer was not able to make a basecall for this base. The third line begins with a '+' character that marks the end of the nucleotides and is optionally followed by the same sequence identifier and description (if any) from the first line. The last line encodes the quality values for the nucleotide sequence in line 2, containing the same number of characters as bases. These scores represent the likelihood of the base being called wrong by the NGS sequencer, also encoded using ASCII characters to represent the numerical quality scores. These scores can be taken into account by the correction algorithms in order to make decisions about correcting sequencing errors within the bases.

3. Related work

We can find in the literature previous works that take advantage of parallel architectures to increase the performance of k-spectrum-based error correctors: Musket [11], BLESS2 [12], RECKONER [14], Lighter [15], SPECTR [16], Quake [44], CUDA-EC [45], DecGPU [46], SGA-EC [47], RACER [48], Blue [49], BFC [50], QuorUM [51], FADE [52], ZEC [53] and SMusket [54]. Most of them only provide a parallel implementation through multithreading, so that their scalability is limited to exploit the computational resources of a single machine. Some of them (SPECTR, CUDA-EC and DecGPU) are designed to be executed on specific hardware (e.g. NVIDIA GPUs, AVX-512), whereas BLESS2, ZEC and SMusket are the only ones that can be executed on distributed-memory systems and without requiring specific hardware. On the one hand, both BLESS2 and ZEC combine MPI to work on different machines with a multithreading approach based on OpenMP [55]. On the other hand, SMusket is our previous work based on Spark that extends the Musket correction algorithm to be executed on clusters of commodity machines.

A common drawback for all the previous tools is that they are limited to implement a single error correction algorithm. According to previous benchmarking studies on error correction methods [18–20, 42,56], there is no one-fits-all corrector. For instance, authors in [20] evaluate the ability of several error correction algorithms to fix errors across different types of datasets with various levels of heterogeneity and using various coverage settings. According to their results, the best method varies substantially across different types of datasets depending on several factors (e.g. genome coverage). In [18], authors compare six k-spectrum-based correction algorithms using multiple paired-end datasets varying their coverage depth, read length and genome size. Their experimental results suggest that good performance of a certain algorithm for a specific dataset does not guarantee its ability to perform as well for another type of dataset. Therefore, the availability of a corrector that integrates multiple algorithms into a single tool can be of great interest for bioinformaticians, especially if the procedure of applying all of them over the input dataset is simple, fast and scalable over a cluster of commodity machines to correct large datasets efficiently. Furthermore, existing correctors are restricted to be executed with a specific computing library of processing framework (e.g. MPI, Spark), so providing more flexibility in this regards can expand the potential range of users of the tool. These are all the challenges we intend to address with the proposal of the BigDEC tool in this paper.

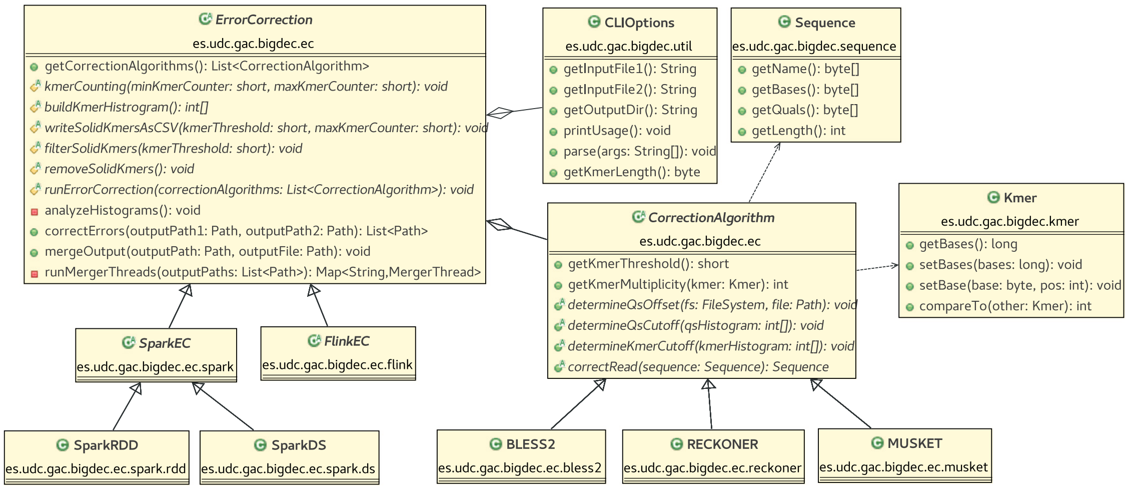

ErrorCorrection
es.udc.gac.bigdec.ec
getCorrectionAlgorithms(): List<CorrectionAlgorithm>
kmerCounting(minKmerCounter: short, maxKmerCounter: short): void
buildKmerHistrogram(): int[]
writeSolidKmersAsCSV(kmerThreshold: short, maxKmerCounter: short): void
filterSolidKmers(kmerThreshold: short): void
removeSolidKmers(): void
runErrorCorrection(correctionAlgorithms: List<CorrectionAlgorithm>): void
analyzeHistograms(): void
correctErrors(outputPath1: Path, outputPath2: Path): List<Path>
mergeOutput(outputPath: Path, outputFile: Path): void
runMergerThreads(outputPaths: List<Path>): Map<String,MergerThread>
CLIOptions
es.udc.gac.bigdec.util
getInputFile1(): String
getInputFile2(): String
getOutputDir(): String
printUsage(): void
parse(args: String[]): void
getKmerLength(): byte
Sequence
es.udc.gac.bigdec.sequence
getName(): byte[]
getBases(): byte[]
getQuals(): byte[]
getLength(): int
Kmer
es.udc.gac.bigdec.kmer
getBases(): long
setBases(bases: long): void
setBase(base: byte, pos: int): void
compareTo(other: Kmer): int
CorrectionAlgorithm
es.udc.gac.bigdec.ec
getKmerThreshold(): short
getKmerMultiplicity(kmer: Kmer): int
determineQsOffset(fs: FileSystem, file: Path): void
determineQsCutoff(qsHistogram: int[]): void
determineKmerCutoff(kmerHistogram: int[]): void
correctRead(sequence: Sequence): Sequence
SparkEC
es.udc.gac.bigdec.ec.spark
FlinkEC
es.udc.gac.bigdec.ec.flink
SparkRDD
es.udc.gac.bigdec.ec.spark.rdd
SparkDS
es.udc.gac.bigdec.ec.spark.ds
BLESS2
es.udc.gac.bigdec.ec.bless2
RECKONER
es.udc.gac.bigdec.ec.reckoner
MUSKET
es.udc.gac.bigdec.ec.musket

Fig. 3. Streamlined UML diagram of the overall BigDEC design with the main classes.

4. Implementation

BigDEC is a novel Big Data tool to speed up the correction of short-read NGS datasets by taking advantage of the computational capabilities provided by distributed-memory systems such as multi-core clusters. Unlike previous correctors, our tool provides support for three different algorithms that are based on the k-spectrum method: Musket [11], BLESS2 [12] and RECKONER [14]. These algorithms have been selected due to their popularity, good accuracy [9,20] and great similarities in their overall workflow, which not only eases the implementation of our tool but also allows to introduce performance optimizations. It is important to remark that BigDEC provides the same output (i.e. error correction accuracy) as its counterpart standalone tools but in a more scalable way thanks to relying on several Big Data technologies, namely Spark, Flink and HDFS. Similar to the standalone tools, BigDEC requires as arguments the path to the input FASTQ file for correcting single-end datasets (two paths for paired-end correction), as well as the k-mer length to be used, among other optional arguments. Moreover, it accepts the path to a configuration file where other parameters can be set, such as the algorithms to be used during the execution, some other options related with Spark, Flink and HDFS, and other more advanced, algorithm-specific settings that are intended for expert users. Furthermore, our tool is designed following a modular, object-oriented approach to enhance code reusability and ease its future extension with new algorithms or frameworks through class inheritance, as depicted in Fig. 3.

Unlike the standalone tools, which are all implemented in C++, BigDEC has been implemented in Java since Spark and Flink frameworks do not provide C++-based APIs to transform RDD/Datasets. Although these frameworks do allow to execute C++ code, wrapping the standalone tools is not feasible without source code modifications. Moreover, these tools must also be modified to take advantage of HDFS features (e.g. data locality). Even though the use of Java may penalize the overall performance of BigDEC as the computational efficiency of the JVM is lower than that of languages compiled to native code (e.g. C/C++), it enhances cross-platform portability across different systems (BigDEC is 100% Java code).

4.1. Overall workflow

At the highest level of abstraction, the overall workflow of BigDEC can be simplified into four main phases: (1) k-mer generation and counting from the input DNA reads; (2) creation of the k-mer histogram to determine the multiplicity thresholds (one per correction algorithm); (3) filtering out weak k-mers to obtain solid ones; and (4) execution of the specific correction routine for each algorithm over the input reads using solid k-mers and writing of the corrected reads to the corresponding output files.

Algorithm 1 shows the pseudocode to correct a single-end dataset that illustrates the previously described workflow. Note that this pseudocode does not take into account those specific aspects related with parallelization or with Big Data technologies for simplicity purposes, which are later explained in following sections. The execution starts by reading the input file in order to generate all the k-mers from each read (lines 5–10). In this code block, $(L - K + 1)$ k-mers are generated in a read of length L (i.e. L bases or nucleotides), being K the k-mer length specified as input argument. The number of k-mer occurrences (i.e. its multiplicity) is also computed so that unique k-mers can be filtered out in the next phase since they are considered to be largely untrusted (see line 11). Next, the k-mer histogram is created from non-unique k-mers and multiplicity thresholds are determined from such histogram for each correction algorithm (lines 12–14). Next phase is in charge of filtering out weak k-mers according to the minimum threshold value to remove as few k-mers as possible. To do so, firstly the algorithms are sorted in ascending order according to their corresponding thresholds and then k-mer filtering is performed (lines 15–16). The correction algorithms are then applied over the input reads in that same threshold order so that they use exclusively the set of solid k-mers that correspond to them during the correction phase (lines 18–24). In this way, we ensure that their accuracy is not affected since the code routines to correct the reads in BigDEC are exactly the same as that of their counterpart standalone tools, but implemented in Java, and they use the same solid k-mers during correction. Note also that the set of solid k-mers for each algorithm is filtered out according to its corresponding threshold only when strictly necessary (see lines 17 and 19–21). Finally, each algorithm writes the corrected reads in the corresponding output file (line 24).

Although Algorithm 1 is intended for single-end datasets, paired-end correction requires minor changes. In this case, the two ends of paired reads are distributed into two separate input files, with one of them containing the forward reads and the other one containing the corresponding reverse reads. Therefore, BigDEC must generate the k-mers for both forward and reverse reads during the k-mer counting phase. During the error correction phase, both ends of paired reads must be processed through the correction routine and the corrected reads written to separate output files (i.e. two output files per algorithm).

The workflow described in this section illustrates some of the potential benefits of our proposal, not only in terms of usability by allowing to apply multiple correction algorithms built within the same tool, but also in terms of performance since the three first phases are common to all of them.

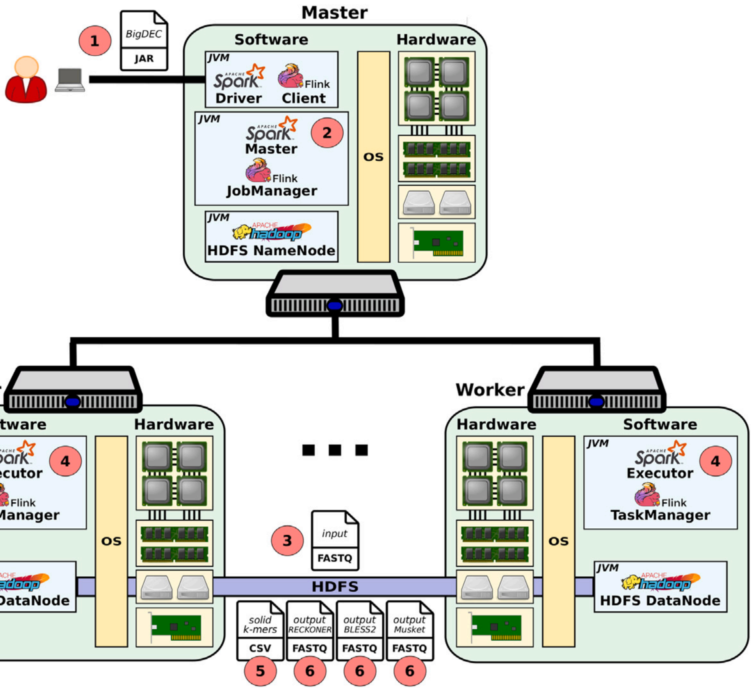


Fig. 4. High-level overview of BigDEC on Spark/Flink clusters.

4.2. BigDEC deployment overview

Fig. 4 depicts the high-level overview of the tool on Spark and Flink clusters. First of all, the user must submit the Java application to the cluster through the master node using the packaged JAR file of BigDEC (see 1 in the figure). The main function of the application, which represents the Spark *Driver* or Flink *Client*, will connect to the Spark *Master* or Flink *JobManager* (see 2) to obtain computational resources from the worker nodes to process the input FASTQ file stored in HDFS (see 3, two files in case of paired-end correction). Next, distributed computing among the workers begins (see 4), each one in charge of processing a certain chunk of the input file/s to perform the k-mer counting phase. It is important to remark that each *Executor/TaskManager* will require the full set of solid k-mers during the error correction phase, as the solidity of all the k-mers for each input read assigned to them must be tested by comparing their multiplicity with the corresponding threshold. However, solid k-mers are actually distributed across multiple workers in our cluster environment. To solve this issue, solid k-mers are written to HDFS in CSV format to make them available to all workers (see 5). In this way, *Executors/TaskManagers* can load those k-mers from HDFS in a local Java *Map* collection before performing error correction. Finally, corrected reads are written back to HDFS (see 6). The BigDEC overview just briefly outlined here is detailed in the following sections.

4.3. Spark/Flink parallelization

The parallel approach used to obtain the set of solid k-mers from the input dataset stored in HDFS is illustrated in Fig. 5. As explained in Section 2.1.2, HDFS files are actually distributed across the worker nodes and stored in small data chunks or blocks of a fixed length (see left part of the figure), which are managed by the *DataNodes* processes. When processing a file from HDFS, logical splits are created from HDFS data blocks (one split per block by default). Remark that a split is just a logical reference to data, whereas an HDFS block is a physical location where actual data are stored. Generally, splits are preferably scheduled to be processed by Big Data frameworks to a worker node that hosts the data referred by the split for performance reasons. BigDEC allows configuring the number of logical splits that are created from each HDFS block through a command-line option (two splits per block are used in the figure as an example).

The total number of splits (NS from now on) directly affects the way Spark partitions an RDD/Dataset: it creates a single partition for each input split. In the example shown in Fig. 5, NS is equal to 4 and thus $N = 4$, N being the number of partitions of the RDD/Dataset created from the input file. When processing an RDD/Dataset, Spark runs a single concurrent task for each partition up to P tasks, P being the total number of CPU cores available on worker nodes (i.e. P is the maximum parallelism). Therefore, NS should be equal to (or higher than) P to ensure that all the available cores are used by the Spark *Executors* for processing tasks. Generally, it is beneficial to create more than one split/partition per CPU core (e.g. $NS = 2 \times P = N$) so that the workload is more evenly distributed among the *Executors*. However, Flink relies on a different approach compared to Spark by running a fixed number of data source tasks that depends on the configured parallelism of those Flink operators and not on the NS value. These data source tasks request splits from the Flink *JobManager* for processing them on the *TaskManagers*, and the split assignment tries to exploit locality preference (splits are fetched from the *JobManager* one after the other, local ones first and remote ones later). As the default parallelism for Flink operators is usually configured to P to exploit all the available resources (i.e. P data source tasks are executed), the maximum number of partitions for any Flink Dataset is also P regardless of the number of splits (i.e. $N <= P$). In terms of efficiency, BigDEC ensures that $NS >= P$ in order to feed all the data source tasks with input data.

Once the Dataset that contains the input reads has been created, k-mers can be generated within each partition to perform their counting and then proceed to filter out unique ones. Details for these steps are not shown in Fig. 5 for simplicity purposes, but the next section describes them together with the specific operations that are required for Spark and Flink (all the steps prior to the red asterisk that can be seen in Fig. 5 will be detailed). The result of these steps is a new Dataset containing only non-unique k-mers, from which the histogram can be generated and thresholds computed for each algorithm. Algorithms are then sorted according to their thresholds and the Dataset containing non-unique k-mers is filtered using the minimum value (see also lines 15–16 in Algorithm 1). This step creates a new Dataset that represents

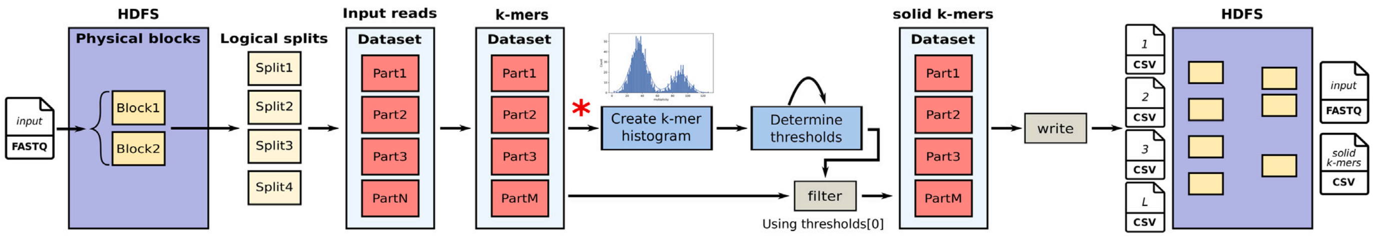


Fig. 5. Simplified parallel approach in BigDEC to obtain solid k-mers from input reads on HDFS.

Algorithm 1: Pseudocode of BigDEC to correct a single-end FASTQ dataset

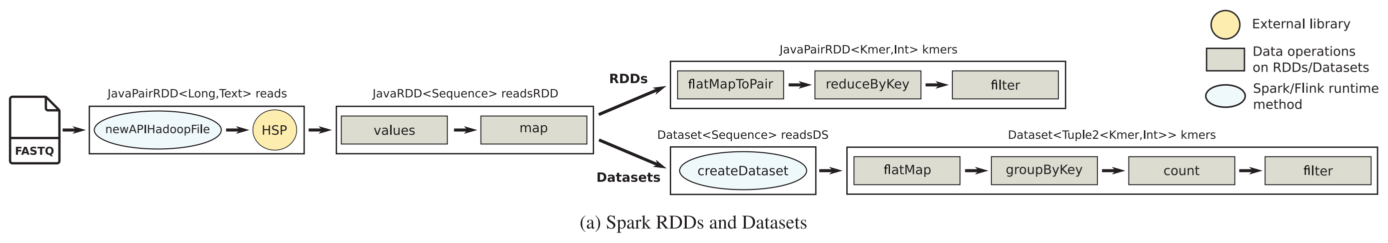
```
1  INPUT: Path to sequence file ($ifile$) with $N$ reads; $K$ as $k$-mer length;
     List $algorithms$ with $C$ algorithms to apply
2  OUTPUT: Sequence files ($ofile_i$) with $N$ corrected reads each, with
     $0 < i < C$
   /* Reading of input reads and $k$-mer counting */
3  Initialize empty list $sequences$
4  Initialize empty map $kmers$ // Map-like data structure to
     store <$k$-mer,multiplicity> pairs
5  for $0 < i < N$ do
6      Read sequence $S_i$ from input file $ifile$
7      Add $S_i$ to $sequences$
8      $L$ = Length($S_i$)
9      $seqKmers$ = CreateKmers($S_i$, $L$, $K$) // Generate $(L - K + 1)$
         $k$-mers with multiplicity 1
10     Put $seqKmers$ into $kmers$ map updating their multiplicities
   end
   /* Filtering out unique $k$-mers and determining
      thresholds from the $k$-mer histogram */
11 $kmers$ = RemoveKmers($kmers$, 1) // Filter out $k$-mers with
     multiplicity 1
12 $histogram$ = CreateKmerHistogram($kmers$)
13 for $0 < i < C$ do
       /* $thresholds[i]$ represents $k$-mer threshold for
          $algorithms(i)$ */
14     $thresholds[i]$ = DetermineKmerThreshold($histogram$, $algorithms(i)$)
   end
   /* Filtering out weak $k$-mers using the minimum
      threshold value */
15 Sort($algorithms$, $thresholds$) // Sort list in ascending order
     according to their thresholds
16 $solidKmers$ = RemoveKmers($kmers$, $thresholds[0]$)
   /* Performing error correction for each algorithm and
      writing the output files */
17 $prevThreshold$ = INT_MAX // Integer maximum value
18 for $0 < i < C$ do
19     if $thresholds[i] > prevThreshold$ then
20         $solidKmers$ = RemoveKmers($solidKmers$, $thresholds[i]$)
       end
21     $prevThreshold$ = $thresholds[i]$
22     for $0 < j < N$ do
23         $O_j$ = CorrectErrors($sequences(j)$, $algorithms(i)$, $solidKmers$, $K$)
24         Write corrected sequence $O_j$ into output file $ofile_i$
       end
   end
```

the set of solid k-mers for the first correction algorithm. As previously explained in Section 4.2, solid k-mers are written to HDFS in CSV format to make them available to all worker nodes during the error correction phase. It is important to remark that when writing a Dataset to HDFS, multiple output files are usually created. On the one hand, Flink writes as many files as the parallelism configured for data sink tasks, which are in charge of writing the Dataset to HDFS. Similar to data sources, these Flink operators are configured with maximum parallelism, so P data sinks are running (i.e. $L = P$ in Fig. 5, L being the number of output CSV files). On the other hand, Spark writes as many files as the number of RDD/Dataset partitions (M in the figure). Our tool ensures that the RDD/Dataset of solid k-mers is created with at least P partitions (i.e. $M >= P$), but the user can modify this setting through the BigDEC configuration file. Therefore, Spark writes M output CSV files to HDFS (i.e. $L = M$ in the figure). After writing the Dataset to HDFS, BigDEC merges all output files into a single one to ease the reading of solid k-mers on *Executors/TaskManagers* before starting the error correction phase. The overhead of this copy-merge operation is negligible, as the resulting CSV file is typically of the order of a few megabytes in size.

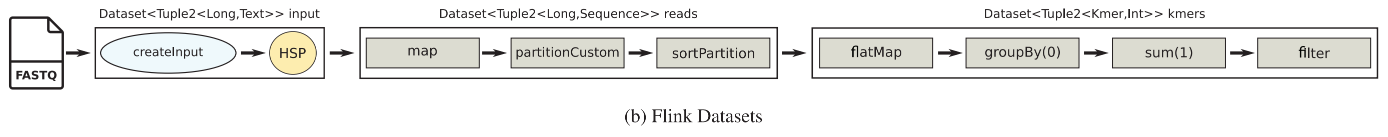
4.3.1. Input reading and k-mer counting

To create the initial Dataset from the input file, BigDEC does not rely on general-purpose methods provided by Spark and Flink for reading text-based files. These built-in methods cannot handle the FASTQ format straightforwardly since they process text files on a line-by-line basis by default (i.e. one record per line). However, FASTQ is a text-based sequence format that involves multiple lines per DNA read, as previously explained in Section 2.3. Even though the default record delimiter in those reading methods can be replaced by the character that separates FASTQ reads (‘@’) within the file, this would not work since such character can also appear in the quality string (see Fig. 2). One simple solution is to preprocess the input file to convert FASTQ reads into an appropriate line-by-line format (i.e. one DNA read per line). This approach, which is used by some previous Hadoop/Spark tools [57–59], incurs significant overhead. Instead, BigDEC relies on the Hadoop Sequence Parser (HSP) [60] to avoid any preprocessing. HSP is a Java library that allows reading unaligned sequence formats (FASTQ/FASTA) directly from any Hadoop-compatible file system (e.g. local file system, HDFS). Using HSP, Spark RDDs and Flink Datasets can be created in an efficient and simple way, supporting both single- and paired-end reads.

Spark implementation. Fig. 6(a) shows the BigDEC implementation of the input reading and k-mer counting steps using Spark RDDs/Datasets. An initial RDD is created from the input file using the *newAPIHadoopFile* method provided by Spark together with the *FastQInputFormat* class from HSP, which extends the Hadoop *FileInputFormat* class to support FASTQ files. HSP generates *<key,value>* pairs of type *<Long, Text>*, where the key is the byte offset in the input file for each read and the value is the text-based content of the read itself: identifier, bases and qualities. Offsets are not needed so that keys are discarded by applying a *values* transformation on the initial RDD. The text contained in the values is parsed accordingly by chaining a *map* transformation that converts each RDD element (i.e. each *Text* object) into an custom *Sequence* object. As a result, a new RDD of type *Sequence* is obtained (named `readsRDD` in Fig. 6(a)). When using Spark Datasets, this RDD is converted into a Dataset of type *Sequence* using the *createDataset* method (named `readsDS` in the figure). From this RDD/Dataset that contains the input reads, k-mers are then generated by applying the appropriate transformation depending on the API: *flatMapToPair* and *flatMap* for RDDs and Datasets, respectively. The function executed


External library
Data operations on RDDs/Datasets
Spark/Flink runtime method
FASTQ
JavaPairRDD<Long,Text> reads
newAPIHadoopFile
HSP
JavaRDD<Sequence> readsRDD
values
map
RDDs
JavaPairRDD<Kmer,Int> kmers
flatMapToPair
reduceByKey
filter
Datasets
Dataset<Sequence> readsDS
createDataset
Dataset<Tuple2<Kmer,Int>> kmers
flatMap
groupByKey
count
filter

(a) Spark RDDs and Datasets


FASTQ
Dataset<Tuple2<Long,Text>> input
createInput
HSP
Dataset<Tuple2<Long,Sequence>> reads
map
partitionCustom
sortPartition
Dataset<Tuple2<Kmer,Int>> kmers
flatMap
groupBy(0)
sum(1)
filter

(b) Flink Datasets

Fig. 6. Reading of input sequences, k-mer counting and filtering out unique k-mers using Spark and Flink.

by these transformations outputs each k-mer as key together with a multiplicity of one as value. To perform k-mer counting, the values for each key are aggregated by just adding all the values (i.e. multiplicities) together for each key (i.e. k-mer). When using RDDs, this can be done with just a *reduceByKey* transformation, whereas Datasets require chaining *groupByKey* and *count*. Next, unique k-mers are filtered out by applying a *filter* transformation that returns a new RDD/Dataset formed by selecting those k-mers whose multiplicity is higher than one. The resulting RDD/Dataset contains non-unique k-mers and their corresponding multiplicities.

Flink implementation. Fig. 6(b) shows the same steps implemented using Flink. The initial Dataset is created using HSP and the *createInput* method provided by Flink. This Dataset contains *<key,value>* tuples of type *<Long, Text>* with the same meaning as in Spark. The text-based content of each read is parsed by applying a *map* transformation to obtain a Dataset of type *<Long, Sequence>*. Unlike Spark, the offsets (i.e. first tuple field) are not discarded as they are needed to keep corrected reads in the output file in the same input order. In Spark, the partitions of the initial RDD are created from the splits so that the ith partition contains the data from the ith split. Therefore, the RDD partition order reflects the order of the reads within the input file. Since all the transformations applied by BigDEC on this RDD do not change such partitioning, the order when writing the corrected reads to HDFS is ensured. However, the split assignment performed by the Flink *JobManager* does not ensure any particular order, so a certain data source task can process, for example, splits 15, 102, 3 y 24 (in that order). To solve this issue, the Flink Dataset is partitioned by key using a custom range partitioner so that all keys falling in the same range will land in the same partition (see *partitionCustom* transformation in Fig. 6(b)). As the keys represent the byte offset for each read within the input file, the partition number can be calculated as the division between the byte offset and the partition size. Partition size is simply obtained by dividing the input file size by the number of Dataset partitions, which is equal to the maximum parallelism P. To ensure the order within each partition, a *sortPartition* transformation is applied after partitioning to sort reads in ascending order based on their offset. To perform k-mer counting, the procedure is very similar to Spark Datasets. Using *flatMap*, k-mers are generated for each read and then aggregated by applying a *groupBy* transformation to group the Dataset by key (i.e. the first tuple field). Multiplicities are then computed through an aggregate transformation on the second tuple field (*sum(1)*). Finally, unique k-mers are removed with a *filter* transformation as in the case of Spark.

4.3.2. Error correction

As a result of all the previous steps, solid k-mers and their multiplicities can be loaded from HDFS on each *Executor/TaskManager* to perform error correction (see Fig. 7). In this last phase, each read in the input Dataset must be processed to correct potential sequencing errors using the algorithms specified by the user. Their execution order depends on their corresponding multiplicity thresholds, as explained in Section 4.1. Basically, the Dataset containing the input reads is operated by applying a *map* transformation that processes each element through a function that implements the specific correction routine for a certain algorithm. These routines haven been reimplemented in Java from their original counterparts to keep their quality of error correction. The corrected Dataset is finally written to HDFS, and this correction–writing loop is repeated until all algorithms selected by the user have been applied. Note that, if required, the set of solid k-mers is filtered accordingly for each algorithm before applying the *map* transformation to correct the input reads. In case of paired-end correction, the only difference would be that there are two input Datasets to be operated through the *map* transformation, with one of them containing forward reads and the other one containing the reverse reads. The two corrected Datasets are written to HDFS on separate directories.

Once the correction phase has finished, Q output files have been created on HDFS for each algorithm ($2 \times Q$ in case of paired-end correction). The number of files depends on the number of RDD/Dataset partitions for Spark (i.e. $Q = N$) and the configured parallelism for data sinks in the case of Flink (i.e. $Q = P$), as previously explained when writing solid k-mers to HDFS (see Section 4.3). For Spark, it is generally beneficial to create the input Dataset with multiple partitions per CPU core so that the workload gets distributed more evenly among *Executors* (e.g. $N = 4 \times P$).

Optionally, the output files for each algorithm can be merged by BigDEC into a single FASTQ file and copied it to the local file system of the master node for further downstream analyses (e.g. mapping, assembly). The most simple approach is to perform this copy-merge operation within the Spark/Flink *Driver/Client* process on the master node just after error correction has finished. Due to specific organization of the Dataset partitions for both frameworks (see Section 4.3.1), combining the output files by their numbering order is enough to guarantee that the order of the corrected reads is the same as in the input file. Nevertheless, this feature of the merge operation can be disabled when output ordering is not needed, which may slightly improve performance especially in the case of Flink since the Dataset partitions would not need to be sorted (see Fig. 6(b)).

Finally, it is important to remark that the copy-merge operation can be avoided when downstream analyses are performed using tools that support distributed processing on HDFS [61,62]. Otherwise, it can be enabled by the user through a command-line option that also allows specifying the destination path on the master node where final output files are copied. Although this operation incurs disk and network overhead, next section describes a more efficient approach by merging files on a stream basis as they are written to HDFS.

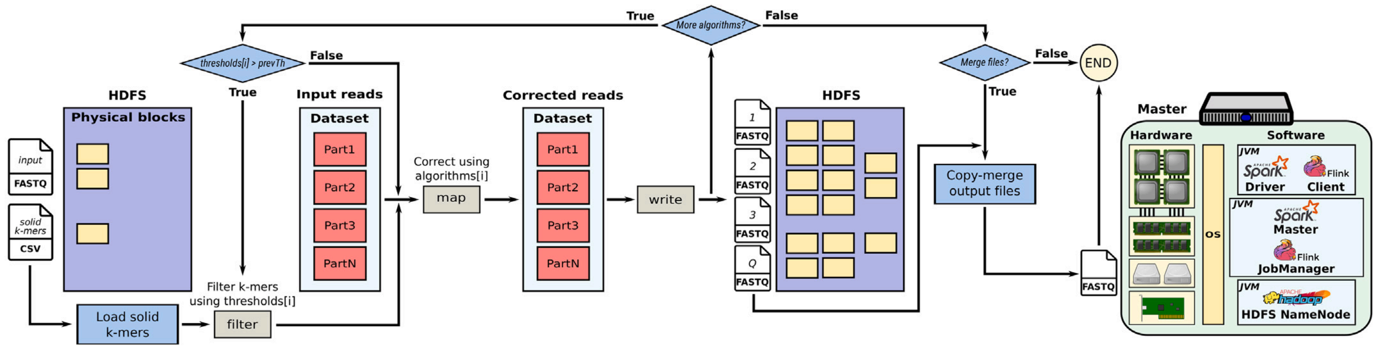


Fig. 7. Overview of the error correction phase and copy-merge operation.

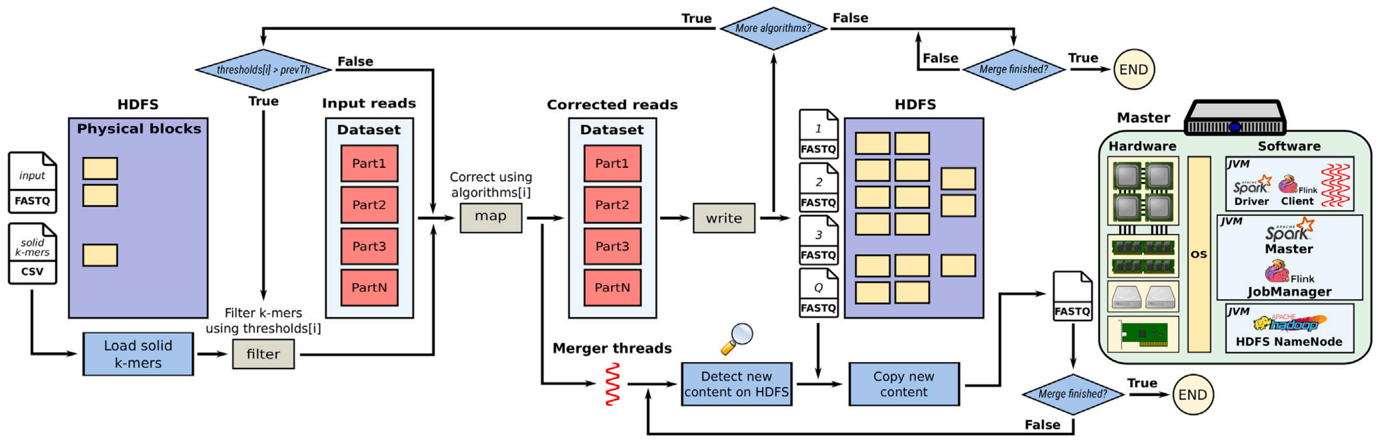


Fig. 8. Error correction using a helper thread to merge output files on a stream basis.

4.4. Stream-based merge operation

To ease downstream analyses of the corrected reads, the copy-merge operation would be needed when HDFS is not supported by the next steps of the bioinformatics pipeline. To improve the performance of this operation, BigDEC includes a stream-based, multithreaded approach that can be enabled to merge and copy output files to the master node as they are written to HDFS. As can be seen in Fig. 8, the basic idea is to launch a new merger thread within the Spark/Flink *Driver/Client* process each time the input Dataset is operated using one of the correction algorithms. Each thread will be in charge of monitoring the HDFS directory where output files are written for a certain algorithm in order to detect new content and copy it to the final output file stored in the master node. However, the merger thread must operate differently depending on the framework being used, since the writing of the corrected Dataset to HDFS creates a different number of output files (Q in the figure) for Spark and Flink, as previously explained.

On the one hand, Spark writes as many output files as the number of RDD/Dataset partitions ($Q = N$), which in turns N depends on the number of input splits (NS) as explained before in Section 4.3 ($Q = N = NS$). For performance reasons, it is usually beneficial for Spark to create the input RDD/Dataset with much more partitions than available CPU cores (i.e. $N > P$). In this scenario, the merger thread monitors HDFS to detect new files written by the *Executors*. Such files represent already corrected partitions and can be copied by the merger thread to the final output file on the master node while the *Executors* are correcting the remaining partitions. To ensure the order of the corrected reads within the final merged file, new detected files must be copied according to their numbering (i.e. ith output file must be copied before jth output file, with $i < j$). This approach allows effectively overlapping computation (error correction) and I/O (copy-merge operation). Increasing NS potentially enhances the chances for overlapping: more but smaller partitions are created, which can be corrected and written to HDFS faster. It is worth noting that the benefits of this approach also apply between the execution of different algorithms thanks to multithreading. When a certain algorithm has finished, its merger thread will continue to copy-merge the remaining output files while the next algorithm and its corresponding merger thread are already being executed. The same reasoning can also be applied for paired-end correction, where the two input Datasets to be corrected use separate merger threads to copy-merge their corresponding output files.

On the other hand, Flink writes as many output files as the configured parallelism for the data sink operators regardless of the number of splits, as explained in Section 4.3. BigDEC configures such parallelism to the maximum level for performance reasons, so that P output files are created per each algorithm ($Q = P = N$). The custom range partitioner used by BigDEC (see Section 4.3.1) ensures that the ith data sink writes the corrected reads from the ith Dataset partition, with $1 < i <= P$. To ensure the order of the corrected reads within the final merged file, output files must be also copied according to their numbering. Another consequence of the custom partitioner is that almost equal-sized partitions are created, so the number of reads within each one is roughly the same, which favors a more balanced workload. In this scenario, the merger thread monitors HDFS to detect new content written in the first output file and copy it to the final output file on the master node. Only once the first file has been fully copied, it can proceed with the next one to ensure the order of the corrected reads within the merged file. Due to having a balanced workload, all the data sinks will finish writing at about the same time, so the benefits for overlapping in single-algorithm executions are limited to just copy the first file. Nevertheless, this approach still remains useful when executing multiple algorithms and/or in paired-end correction,

Table 1
FASTQ datasets used in the experimental evaluation of BigDEC.

Dataset	Tag	Organism	#Reads	Length
SRR8655541	SRR86	Mus musculus	2 × 150.2 M	75 bp
SRR567455	SRR56	Homo sapiens	2 × 251.9 M	75 bp
ERR580958	ERR58	Labrus bergylta	2 × 148.7 M	100 bp
SRR8908980	SRR89	Homo sapiens	2 × 199.1 M	100 bp

as the chances for overlapping computation and I/O increase, as will be shown in the performance evaluation.

To experimentally evaluate the benefits of the stream-based merge operation, Section 5.1 will assess its impact on BigDEC performance compared to performing the copy-merge just after error correction has finished.

5. Performance evaluation

The experimental evaluation has been focused on performance (i.e. execution time) since the quality of error correction provided by BigDEC remains the same as that of the counterpart standalone tools, as explained in Section 4.1. To perform such evaluation, the experiments were carried out on a 16-worker commodity cluster running Spark 3.1.2, Flink 1.14.0 and Hadoop HDFS 3.3.1. Each worker node consists of two Intel Xeon E5-2660 octa-core processors, 64 GiB of memory and one 1 TiB local disk intended for HDFS data storage (i.e. 16 cores per node and $P = 256$ cores in total). Nodes are interconnected through Gigabit Ethernet and InfiniBand FDR. The cluster runs GNU/Linux CentOS 7.9.2009 with kernel 3.10.0–1160 and the JVM version is Oracle JDK 11.0.11. Regarding HDFS settings, the block size and the replication factor were set to 64 MiB and 3, respectively. To deploy Spark, Flink and HDFS on the cluster nodes, the Big Data Evaluator (BDEv) tool [63] has been used running one *Executor/TaskManager* JVM process per node configured with 16 cores and 54 GiB of memory. Both the Spark *Driver* and Flink *Client* are executed on the master node of the cluster together with the Spark *Master* and the Flink *JobManager* processes (see Fig. 1).

Four publicly available datasets with different characteristics have been evaluated, named after their accession numbers in the European Nucleotide Archive [64] and tagged accordingly for the sake of clarity (see second column in Table 1). The fifth column in this table refers to the total number of input FASTQ reads for paired-end correction, 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 a set of five executions for each experiment using a fixed k-mer length of 25, although the observed variance in runtimes was not significant. Finally, eight splits per CPU core have been used, so the total number of splits for a certain experiment can be calculated as $NS = 8 \times 16 \times \#nodes$, being $\#nodes$ the number of worker nodes used in the experiment.

5.1. Impact of the stream-based merge

This first set of experiments analyzes the benefits provided by the stream-based copy-merge operation described in Section 4.4. To do so, its performance is compared with that of the naive approach where the copy-merge is performed after error correction (see Section 4.3.2) and with that obtained when the copy-merge is disabled. The results are shown for paired-end correction, which is the most computationally intensive scenario under evaluation, using ERR58 and SRR89 as representative datasets (the results for other datasets are very similar).

Fig. 9 shows the runtimes of BigDEC with Spark (labeled SP in the figure) and Flink (FL) for single-algorithm performance using Musket (M), varying the number of nodes from 4 to 16. In the case of Spark, results using the Dataset API are shown (labeled SP-DS-M), although results for the RDD API are very similar. As can be seen in Fig. 9(a), Spark takes full advantage of the stream-based merge as runtimes are nearly identical to those obtained when the merge is not performed, especially for 8 and 16 nodes. For instance, runtime is reduced by 21% and 23% on 16 nodes when using the stream-based merge compared to the naive approach for ERR58 and SRR89, respectively. As expected, the performance benefits for Flink (see Fig. 9(b)) are lower than for Spark due to less chances for overlapping, as previously explained in Section 4.4. However, runtimes reductions of about 6% and 10% are obtained when using 16 nodes for ERR58 and SRR89, respectively, which shows that our approach is also useful for single-algorithm performance.

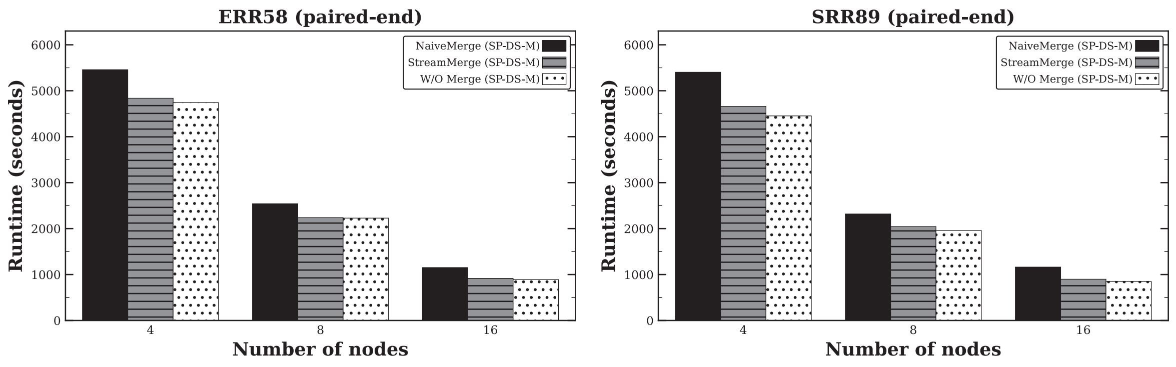
Fig. 10 shows the runtimes for multi-algorithm performance executing the three correction algorithms currently supported by BigDEC. In this scenario, the performance of this operation is key as the amount of output data to copy-merge is increased by a factor of 3x. The overhead incurred using the naive approach compared to not performing the merge is very significant. For example, the merge operation for the ERR58 dataset represents from 30% to 45% of the total runtime for Spark when using the naive approach (see Fig. 10(a)), and from 28% to 60% in the case of Flink (Fig. 10(b)). Nevertheless, the stream-based merge clearly shows its full effectiveness by reducing runtimes significantly, especially for Spark where merge overheads are virtually removed when using 8 and 16 nodes. Although the benefits for Flink are again lower than for Spark, these results confirm that multi-algorithm execution takes more advantage of the stream-based merge, with runtimes reductions of 10%, 36% and 51% for ERR58 when using 4, 8 and 16 nodes, respectively, and 15%, 41% and 55% for SRR89. Therefore, the larger the dataset, the higher the benefit provided our stream-based merge (SRR89 contains 33% more reads than ERR58, see Table 1).

These results also allow to state that BigDEC provides very good scalability overall, as performance improves proportionally when using more hardware resources. All the experimental results shown hereafter have been obtained using the stream-based copy-merge operation due to its better performance.

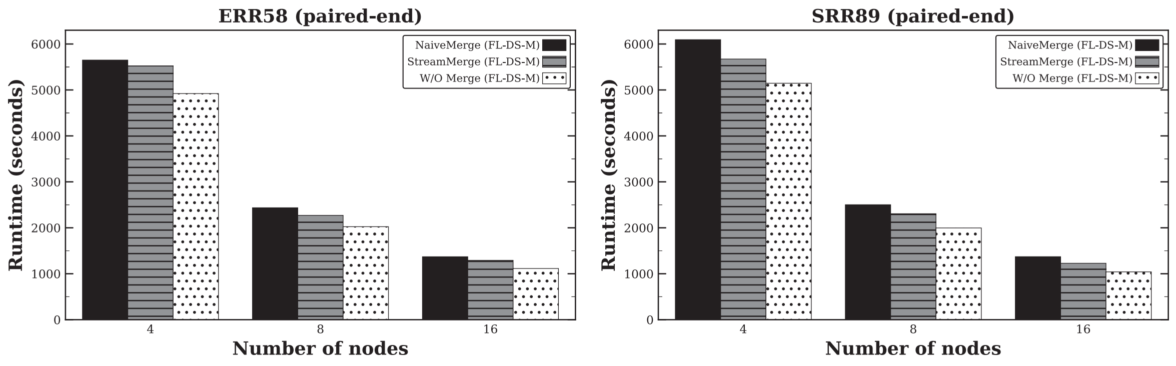
5.2. Spark vs Flink

The second set of experiments compares the performance of BigDEC when using Spark RDDs, Spark Datasets and Flink Datasets, considering both single- and paired-end correction so that one and two input files, respectively, are processed for each experiment. For comparison purposes, results for SMusket are also provided in this section, as this tool is the most similar k-spectrum-based parallel corrector to BigDEC in the state of the art. As mentioned in Section 3, SMusket is implemented on top of the Spark RDD API but only limited to implement the Musket algorithm [11]. Another limitation is that the copy-merge operation in SMusket can only be performed after error correction has finished. To ease the comparison, BigDEC is configured to apply only the Musket algorithm as SMusket does, so that the computations performed during the error correction phase remain exactly the same for both tools.

Fig. 11 shows the runtimes obtained for SRR56, ERR58 and SRR89, varying the number of nodes from 4 to 16. Several analyses can be done based on these results. When comparing SMusket with BigDEC using the same Big Data processing framework (Spark), we can conclude that our tool is significantly faster in all the scenarios under evaluation, even when comparing them using the same Spark API (i.e. RDD). In fact, BigDEC using RDDs (labeled SP-RDD-M in the figure) clearly outperforms SMusket, providing speedups of up to 1.5x, 1.3x and 1.4x for SRR56, ERR58 and SRR89, respectively, when using 16 nodes. The performance improvements increase even more when using the Spark Dataset API (labeled SP-DS-M), which allows stating that this API can provide an extra performance boost over RDDs in our scenario. For instance, BigDEC using Spark Datasets is up to 1.7 and 1.8 times faster than SMusket on 16 nodes for single- and paired-end correction of SRR56, respectively, while also providing an average runtime reduction of around 11% over BigDEC using RDDs.

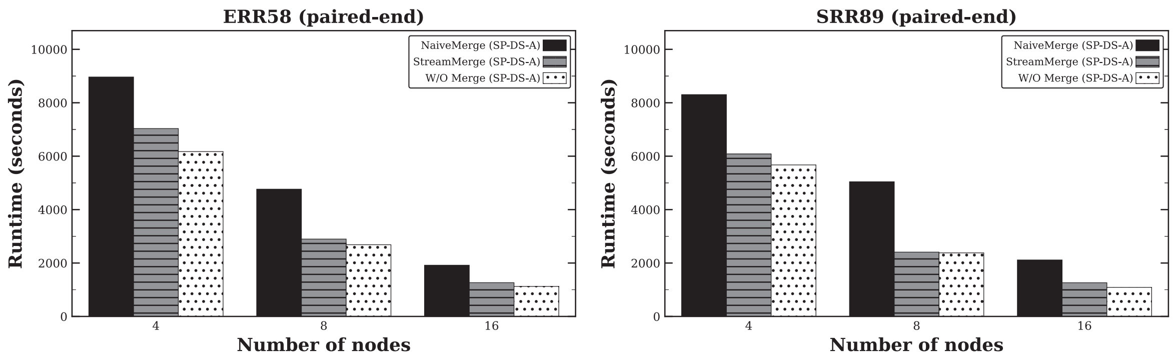


(a) Spark Datasets running Musket (SP-DS-M)

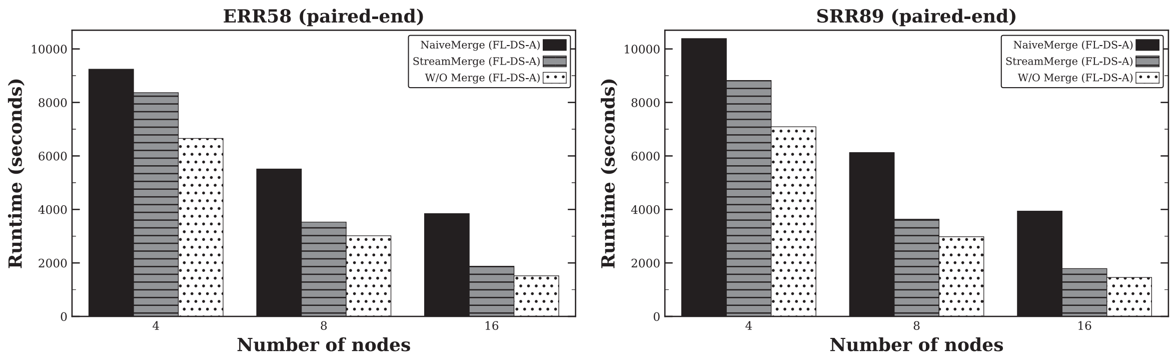


(b) Flink Datasets running Musket (FL-DS-M)

Fig. 9. Runtimes of BigDEC using Spark and Flink Datasets running the Musket algorithm.



(a) Spark Datasets running all the algorithms (SP-DS-A)



(b) Flink Datasets running all the algorithms (FL-DS-A)

Fig. 10. Runtimes of BigDEC using Spark and Flink Datasets running BLESS2, Musket and RECKONER algorithms.

When comparing BigDEC using Spark Datasets with Flink Datasets (labeled FL-DS-M), we can conclude that the former outperforms the latter in all the experiments, being up to 1.9 and 1.7 times faster on 16 nodes for single- and paired-end correction of SRR56, respectively.

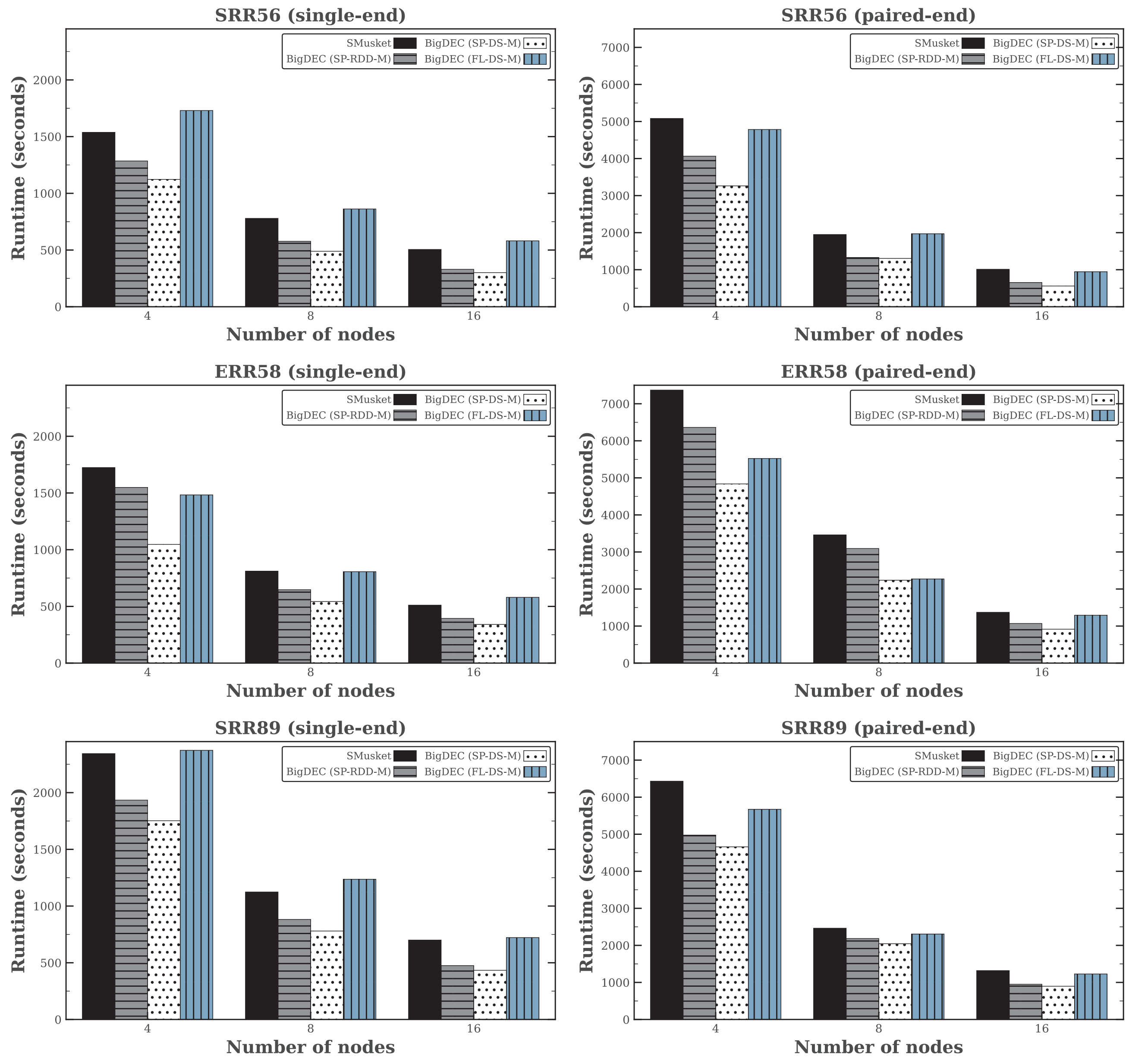


Fig. 11. Runtimes of SMusket and BigDEC tools using the Musket algorithm for single- and paired-end correction.

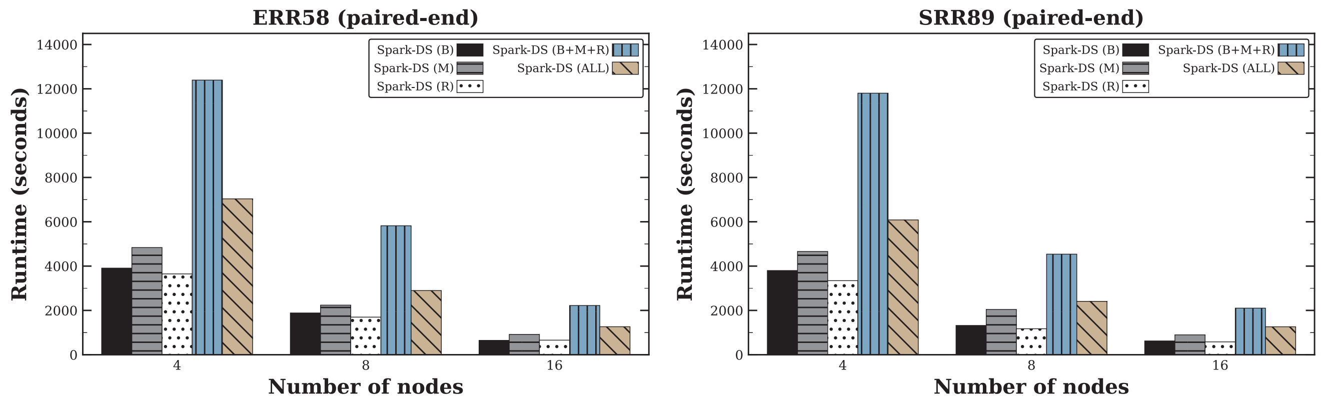
Nevertheless, Flink performance remains very competitive when compared to both Spark-based implementations using RDDs, especially for paired-end correction. Roughly speaking, Flink provides very similar overall performance to SMusket, even outperforming it in most of the paired-end experiments.

5.3. Benefits of multi-algorithm execution

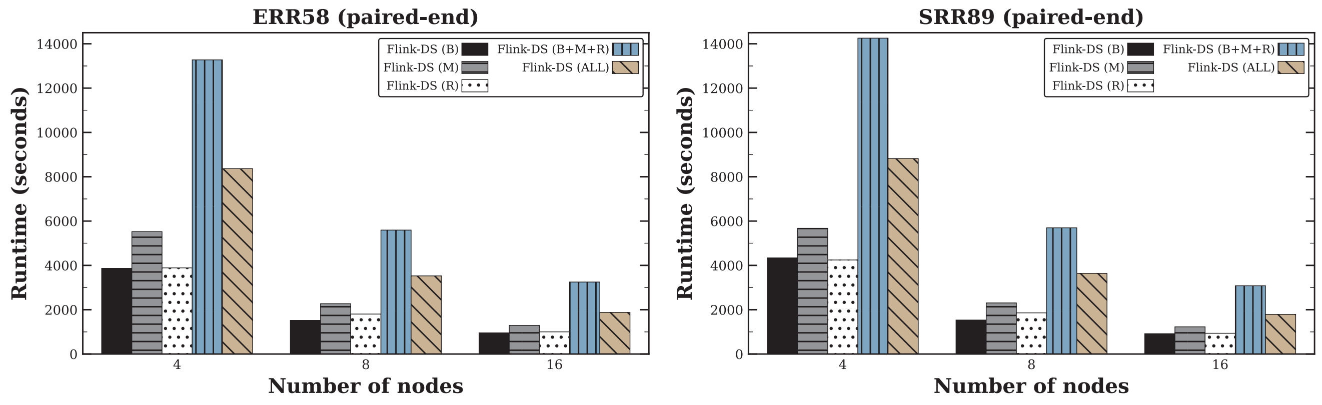
These experiments analyze the benefits of providing specific support for multi-algorithm executions in BigDEC due to sharing some of the workflow phases (e.g. input reading, k-mer counting). To do so, this section compares the total runtime of BigDEC running the three algorithms in separate executions with that of running all of them in a single one (i.e. using the multi-algorithm feature). For comparison purposes, the individual runtime for each algorithm is also included, which represents the single-algorithm performance. The results are shown using Spark and Flink Datasets for paired-end correction of ERR58 and SRR89, but the overall conclusions obtained would be mostly the same if results for the other datasets or single-end correction were included.

Fig. 12 shows the runtimes for ERR58 and SRR89, varying the number of nodes from 4 to 16. The total runtime of BigDEC running the three algorithms in separate executions is labeled as "(B+M+R)" in the figure, which is calculated as the sum of the runtime for each algorithm separately, whereas "(ALL)" represents the runtime when running all of them in a single BigDEC execution. Generally speaking, both Spark and Flink take clear advantage of the multi-algorithm feature, obtaining important performance benefits in all the scenarios and number of nodes for both datasets. The average runtime reductions for Spark when using the multi-algorithm mode are 46%, 49% and 42% when using 4, 8 and 16 nodes, respectively, as can be seen in Fig. 12(a). For Flink, those runtimes reductions are on average 38%, 37% and 42%, respectively (see Fig. 12(b)), so slightly lower values than Spark for 4 and 8 nodes, but still very significant.

These results also allows comparing the performance at the single-algorithm level, although this is not the focus of this work. We can conclude that BLESS2 and RECKONER perform very similar overall,



(a) Spark Datasets



(b) Flink Datasets

Fig. 12. Runtimes of BigDEC using Spark and Flink Datasets for single- and multi-algorithm performance.

whereas Musket is clearly the most computationally intensive algorithm, probably due to its multi-stage correction routine which corrects each read based on three different techniques that are conducted iteratively [11].

5.4. Performance comparison with standalone tools

The last set of experiments are focused on measuring the maximum performance benefits provided by BigDEC when distributing the workload across the whole cluster, so using all the hardware resources (16 nodes in our testbed). The performance of BigDEC using Spark and Flink Datasets is compared with their counterpart standalone tools when support for distributed-memory systems is available. This is possible for Musket using its Spark-based implementation (SMusket) as in Section 5.2, and also for BLESS2 that provides an MPI implementation. We also include the results for their corresponding multithreaded versions to be used as reference, but limited to take advantage of a whole single node (16 cores). Furthermore, this section presents results for all the datasets shown in Table 1.

Table 2 provides the runtimes for paired-end correction obtained by BigDEC using the Musket algorithm with Spark and Flink Datasets, together with the results for SMusket and its multithreaded version. As can be observed, BigDEC using Spark Datasets achieves significant speedups of up to 1.7x and 1.8x over the Flink-based version and SMusket, respectively (1.5x and 1.7x on average). These results confirm two facts: (1) our Spark-based implementation is clearly the fastest option, providing a runtime reduction of up to 45% over SMusket; and (2) our Flink-based version is still able to outperform the performance of SMusket for paired-end correction, as previously mentioned in Section 5.2, even when using a Big Data framework that has shown a lower overall performance than Spark according to our evaluation. Compared to the multitheaded version, BigDEC provides an average runtime reduction of around 95% and 93% for Spark and Flink, respectively, with a maximum speedup of 26x for the Spark version, which allows significantly reducing runtimes from more than 5 h to less than 15 min when correcting the SRR89 dataset.

Table 3 shows the same results for BigDEC but using the BLESS2 algorithm, and compares them to the MPI version of BLESS2 on 16 nodes and its multithreaded counterpart on one node. On the one hand, the performance benefits of the Spark version over the Flink one practically remain the same as before, with an average speedup of around 1.5x. On the other hand, BigDEC using Spark and Flink provides significant runtime reductions over BLESS2 using MPI: up to 79% and 70% when correcting the SRR56 dataset, respectively, on the same hardware resources. Overall, both Spark and Flink clearly outperform the MPI-based implementation, achieving runtime reductions of around 73% and 60% on average. The performance of the multithreaded version is rather good at least compared to Musket, since the maximum speedup for the Spark version is reduced to 12.8x for SRR86, with average runtime reductions of around 91% and 86% for Spark and Flink, respectively.

Table 4 provides the runtimes for BigDEC using the RECKONER algorithm. In this case, there is no counterpart standalone tool for distributed-memory systems, so the results can only be compared to the multithreaded version. Nevertheless, these results are still useful to confirm that BigDEC is the suitable replacement of standalone tools to correct large datasets on a cluster, whereas highly optimized multithreaded implementations such as the one provided by RECKONER would be the preferred choice otherwise. Basically, Big Data processing frameworks are specifically designed to provide scalability on distributed-memory systems, and not to replace parallel paradigms intended for shared-memory systems. In fact, RECKONER is the fastest

Table 2
Runtimes (in seconds) for paired-end correction obtained by BigDEC using the Musket algorithm with Spark Datasets (SP-DS-M) and Flink Datasets (FL-DS-M) on 16 nodes. Results are compared to those obtained by SMusket using Spark on 16 nodes and its multithreaded counterpart (Musket) on one whole node. The percentage values shown in brackets are calculated using BigDEC runtimes as a reference for Spark/Flink, respectively, in this order.

Dataset	16 nodes (256 cores)			1 node (16 cores)
	BigDEC (SP-DS-M)	BigDEC (FL-DS-M)	SMusket (Spark)	Musket (threads)
SRR86	369	607	674 (−45%/−10%)	6589 (−94%/−91%)
SRR56	560	945	1011 (−45%/−7%)	14642 (−96%/−94%)
ERR58	915	1292	1371 (−33%/−6%)	16195 (−94%/−92%)
SRR89	898	1228	1319 (−32%/−7%)	19285 (−95%/−94%)
Average			−38.8%/−7.5%	−94.8%/−92.8%

Table 3
Runtimes (in seconds) for paired-end correction obtained by BigDEC using the BLESS2 algorithm with Spark Datasets (SP-DS-B) and Flink Datasets (FL-DS-B) on 16 nodes. Results are compared to those obtained by BLESS2 using MPI on 16 nodes and its multithreaded support on one whole node. The percentage values shown in brackets are calculated using BigDEC runtimes as a reference for Spark/Flink, respectively, in this order.

Dataset	16 nodes (256 cores)			1 node (16 cores)
	BigDEC (SP-DS-B)	BigDEC (FL-DS-B)	BLESS2 (MPI)	BLESS2 (threads)
SRR86	340	535	1232 (−72%/−57%)	4343 (−92%/−88%)
SRR56	538	780	2601 (−79%/−70%)	5908 (−91%/−87%)
ERR58	648	956	1798 (−64%/−47%)	4798 (−87%/−80%)
SRR89	623	918	2513 (−75%/−64%)	7569 (−92%/−88%)
Average			−72.5%/−59.5%	−90.5%/−85.8%

Table 4
Runtimes (in seconds) for paired-end correction obtained by BigDEC using the RECKONER algorithm with Spark Datasets (SP-DS-R) and Flink Datasets (FL-DS-R) on 16 nodes. Results are compared to those obtained by RECKONER using its multithreaded support on one whole node. The percentage values shown in brackets are calculated using BigDEC runtimes as a reference for Spark/Flink, respectively, in this order.

Dataset	16 nodes (256 cores)		1 node (16 cores)
	BigDEC (SP-DS-B)	BigDEC (FL-DS-B)	RECKONER (threads)
SRR86	315	514	2079 (−85%/−75%)
SRR56	906	1163	8849 (−90%/−87%)
ERR58	658	1001	4474 (−85%/−78%)
SRR89	581	936	3489 (−83%/−73%)
Average			−85.8%/−78.3%

multithreaded tool in our experiments, and probably the preferred option in terms of performance when hardware resources are limited. However, the performance benefits of BigDEC when distributing the workload across the cluster are still good. The maximum speedups are around 9.8x and 7.6x for Spark and Flink, respectively, providing runtime reductions of around 86% and 78% on average. The maximum speedups are obtained when correcting the largest dataset in terms of the total number of reads (i.e. SRR56, see Table 1), showing the ability of our tool to handle large datasets.

Finally, Table 5 provides the runtimes obtained by BigDEC for both single- and paired-end correction modes when using the multi-algorithm feature, so running the three supported algorithms. The results using Spark (SP-DS-A) and Flink Datasets (FL-DS-A) are compared to those obtained by SMusket (Spark) and BLESS2 (MPI), which are limited to a single algorithm, using the same hardware resources (16 nodes). On the one hand, BigDEC using Spark is able to match or even improve the performance of SMusket for three of the four datasets regardless of using single- or paired-end correction, being up to 13% faster for SRR89. On the other hand, the runtimes for the Flink-based version compared to SMusket are slightly worse, but still competitive taking into account the lower raw performance of Flink compared to Spark and that the correction is performed running three algorithms instead of just only one. When comparing BigDEC to BLESS2, the results are noticeable better. As can be seen, the average runtime reductions using Spark are around 43% for single- and paired-correction, being up to two times faster for SRR86 and SRR89. In this case, even the Flink-based version clearly outperforms BLESS2, except for the paired-end correction of the ERR58 dataset. For instance, BigDEC using Flink is approximately more than 30% faster than BLESS2 for SRR86 and SRR56, respectively, using paired-end correction. Overall, these results validate our approach as they prove that BigDEC using the multi-algorithm mode can be even faster than previous single-algorithm tools for distributed-memory systems.

Table 5
Runtimes (in seconds) obtained by BigDEC using the three supported correction algorithms with Spark Datasets (SP-DS-A) and Flink Datasets (FL-DS-A) on 16 nodes. Results are compared to those obtained by SMusket and BLESS2 using Spark and MPI, respectively, on the same hardware resources. The percentage values shown in brackets are calculated using BigDEC runtimes as a reference for Spark/Flink, respectively, in this order.

	Dataset	BigDEC (SP-DS-A)	BigDEC (FL-DS-A)	SMusket (Spark)	BLESS2 (MPI)
Single-end	SRR86	321	512	295 (+9%/+74%)	574 (−44%/−11%)
	SRR56	757	1068	505 (+50%/+111%)	1405 (−46%/−24%)
	ERR58	510	711	511 (0%/+39%)	843 (−40%/−16%)
	SRR89	610	914	700 (−13%/+31%)	1071 (−43%/−15%)
Average				+11.4%/+63.8%	−43.3%/−16.5%
Paired-end	SRR86	625	833	674 (−7%/+23%)	1232 (−49%/−32%)
	SRR56	1536	1798	1011 (+52%/+78%)	2601 (−41%/−31%)
	ERR58	1264	1877	1371 (−8%/+37%)	1798 (−30%/+4%)
	SRR89	1262	1789	1319 (−4%/+36%)	2513 (−50%/−29%)
Average				+8.2%/+43.5%	−42.5%/−21.9%

6. Conclusions

The need for speed is increasing as fast as the NGS datasets swell, which demands bioinformatics tools providing the ability to scale out across a cluster of multicore nodes to reduce runtimes when processing such large datasets. In this paper we have presented BigDEC, a parallel tool implemented in Java intended for correcting large NGS datasets across a cluster of commodity machines that is able to perform error correction using two different Big Data processing frameworks (Spark and Flink) and three different *k*-spectrum-based algorithms from the state of the art (Musket, BLESS2 and RECKONER).

The comprehensive experimental evaluation of BigDEC using four publicly available FASTQ datasets has shown that Spark clearly outperforms Flink for single- and multi-algorithm performance, being around 1.4 times faster on average. When comparing BigDEC to previous parallel correctors for distributed-memory systems, our tool provides significant performance benefits for single-algorithm execution, being up to 45% and 79% faster than SMusket and BLESS2, respectively, when using a 16-node cluster. Moreover, BigDEC is able to correct a dataset with 199 million 100-bp reads using the three supported algorithms up to two times faster than BLESS2, a single-algorithm tool based on a hybrid parallel approach (MPI+OpenMP). Our BigDEC implementation is distributed as free open-source software released under the GNU GPLv3 license and is available to download at https://github.com/UDC-GAC/BigDEC.

As future work, we will explore the possibility of porting BigDEC to an unified programming model using Apache Beam in order to add support for other distributed data processing backends.

CRediT authorship contribution statement

Roberto R. Expósito: Conceptualization, Methodology, Software, Investigation, Validation, Writing – original draft. **Jorge González-Domínguez:** Conceptualization, Investigation, Validation, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The real datasets analyzed during this study are publicly available at the NCBI SRA repository (https://www.ncbi.nlm.nih.gov/sra) using the accession numbers listed in the paper.

Acknowledgments

This work was supported by grants PID2019-104184RB-I00 and PID2022-136435NB-I00, funded by the Ministry of Science and Innovation of Spain, MCIN/AEI/10.13039/501100011033 (PID2022 also funded by “ERDF A way of making Europe”, EU). It was also funded by Xunta de Galicia [Consolidation Program of Competitive Reference Groups, grant ED431C 2021/30]. Funding for open access charge: Universidade da Coruña/CISUG.

References

[1] S. Goodwin, J.D. McPherson, W.R. McCombie, Coming of age: Ten years of next-generation sequencing technologies, Nat. Rev. Genet. 17 (6) (2016) 333–351.
[2] K.A. Phillips, Assessing the value of next-generation sequencing technologies: An introduction, Value Health 21 (9) (2018) 1031–1032.
[3] C. Di Resta, S. Galbiati, P. Carrera, M. Ferrari, Next-generation sequencing approach for the diagnosis of human diseases: Open challenges and new opportunities, EJIFCC 29 (1) (2018) 4–14.
[4] F. Faita, C. Vecoli, I. Foffa, M.G. Andreassi, Next generation sequencing in cardiovascular diseases, World. J. Cardiol. 4 (10) (2012) 288–295.
[5] X. Chen, et al., Next-generation sequencing reveals the progression of COVID-19, Front. Cell Infect. Microbiol. 11 (2021) 632490.
[6] K. Wetterstrand, DNA sequencing costs: data from the NHGRI genome sequencing program, https://www.genome.gov/sequencingcostsdata. [Visited March 2023].
[7] Z.D. Stephens, et al., Big data: Astronomical or genomical? PLoS Biol. 13 (7) (2015) e1002195.
[8] S.A. Jeon, et al., Comparison between MGI and Illumina sequencing platforms for whole genome sequencing, Genes. Genom. 43 (7) (2021) 713–724.
[9] M. Heydari, G. Miclotte, P. Demeester, Y. Van de Peer, J. Fostier, Evaluation of the impact of Illumina error correction tools on de novo genome assembly, BMC Bioinformatics 18 (1) (2017) 1–13.
[10] A. Ratan, et al., Comparison of sequencing platforms for single nucleotide variant calls in a human sample, PLoS One 8 (2) (2013) e55089.
[11] 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.
[12] 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.
[13] A. Allam, P. Kalnis, V. Solovyev, Karect: Accurate correction of substitution, insertion and deletion errors for next-generation sequencing data, Bioinformatics 31 (21) (2015) 3421–3428.
[14] M. Długosz, S. Deorowicz, RECKONER: Read error corrector based on KMC, Bioinformatics 33 (7) (2017) 1086–1089.
[15] L. Song, L. Florea, B. Langmead, Lighter: Fast and memory-efficient sequencing error correction without counting, Genome Biol. 15 (11) (2014) 509.
[16] 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.
[17] F. Kallenborn, A. Hildebrandt, B. Schmidt, CARE: Context-aware sequencing read error correction, Bioinformatics 37 (7) (2021) 889–895.
[18] 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. Genom. 10 (2) (2016) 20.
[19] A.S. Alic, D. Ruzafa, J. Dopazo, I. Blanquer, Objective review of de novo stand-alone error correction methods for NGS data, WIREs Comput. Mol. Sci. 6 (2) (2016) 111–146.
[20] K. Mitchell, et al., Benchmarking of computational error-correction methods for next-generation sequencing data, Genome Biol. 21 (1) (2020) 71.
[21] A. O'Driscoll, J. Daugelaite, R.D. Sleator, 'Big data', Hadoop and cloud computing in genomics, J. Biomed. Inform. 46 (5) (2013) 774–781.
[22] 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.
[23] 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.
[24] R.R. Expósito, J. González-Domínguez, J. Touriño, HSRA: Hadoop-based spliced read aligner for RNA sequencing data, PLoS One 13 (7) (2018) e0201483.
[25] M. Zaharia, et al., Apache spark: A unified engine for big data processing, Commun. ACM 59 (11) (2016) 56–65.
[26] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, K. Tzoumas, Apache Flink: Stream and batch processing in a single engine, IEEE Data Eng. Bull. 38 (4) (2015) 28–38.
[27] 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.
[28] 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, Nucleic Acids Res. 38 (6) (2010) 1767–1771.
[29] J. Dean, S. Ghemawat, MapReduce: Simplified data processing on large clusters, in: Proceedings of the 6th Symposium on Operating System Design and Implementation, OSDI'04, San Francisco, CA, USA, 2004, pp. 137–150.
[30] The Apache Software Foundation, Apache Hadoop, https://hadoop.apache.org. [Visited March 2023].
[31] V. Chang, Towards data analysis for weather cloud computing, Knowl. Based. Syst. 127 (2017) 29–45.

[32] 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.
[33] 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) BII.S31559.
[34] V. Chang, Computational intelligence for medical imaging simulations, J. Med. Syst. 42 (1) (2018) 10.
[35] S. Min, B. Lee, S. Yoon, Deep learning in bioinformatics, Brief. Bioinform. 18 (5) (2016) 851–869.
[36] V. Chang, Data analytics and visualization for inspecting cancers and genes, Multimed. Tools Appl. 77 (14) (2018) 17693–17707.
[37] S.K. Shandilya, S. Sountharrajan, S. Shandilya, E. Suganya, Big Data analytics framework for real-time genome analysis: A comprehensive approach, J. Comput. Theor. Nanosci. 16 (8) (2019) 3419–3427.
[38] 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.
[39] M. Armbrust, et al., Spark SQL: relational data processing in Spark, in: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, SIGMOD'15, Melbourne, Australia, 2015, pp. 1383–1394.
[40] Z. Ren, et al., How good is query optimizer in Spark? in: Proceedings of the 14th International Conference on Collaborative Computing: Networking, Applications and Worksharing, CollaborateCom 2018, Shanghai, China, 2018, pp. 595–609.
[41] 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.
[42] M. Molnar, L. Ilie, Correcting Illumina data, Brief. Bioinform. 16 (4) (2014) 588–599.
[43] D. Laehnemann, A. Borkhardt, A.C. McHardy, Denoising DNA deep sequencing data–high-throughput sequencing errors and their correction, Brief. Bioinform. 17 (1) (2016) 154–179.
[44] D.R. Kelley, M.C. Schatz, S.L. Salzberg, Quake: Quality-aware detection and correction of sequencing errors, Genome Biol. 11 (11) (2010) R116.
[45] 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.
[46] 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.
[47] J.T. Simpson, R. Durbin, Efficient de novo assembly of large genomes using compressed data structures, Genome Res. 22 (3) (2012) 549–556.
[48] L. Ilie, M. Molnar, RACER: Rapid and accurate correction of errors in reads, Bioinformatics 29 (19) (2013) 2490–2493.
[49] P. Greenfield, K. Duesing, A. Papanicolaou, D.C. Bauer, Blue: Correcting sequencing errors using consensus and context, Bioinformatics 30 (19) (2014) 2723–2732.
[50] H. Li, BFC: Correcting Illumina sequencing errors, Bioinformatics 31 (17) (2015) 2885–2887.
[51] G. Marçais, J.A. Yorke, A. Zimin, QuorUM: An error corrector for Illumina reads, PLoS One 10 (6) (2015) e0130821.
[52] 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.
[53] L. Zhao, et al., Mining statistically-solid k-mers for accurate NGS error correction, BMC Genom. 19 (10) (2018) 912.
[54] R.R. Expósito, J. González-Domínguez, J. Touriño, SMusket: Spark-based DNA error correction on distributed-memory systems, Future Gener. Comput. Syst. 111 (2020) 698–713.
[55] L. Dagum, R. Menon, OpenMP: An industry-standard API for shared-memory programming, IEEE Comput. Sci. Eng. 5 (1) (1998) 46–55.
[56] X. Yang, S. Chockalingam, S. Aluru, A survey of error-correction methods for next-generation sequencing, Brief. Bioinform. 14 (1) (2013) 56–66.
[57] 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.
[58] R.V. Pandey, C. Schlötterer, DistMap: A toolkit for distributed short read mapping on a Hadoop cluster, PLoS One 8 (8) (2013) e72614.
[59] 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.
[60] R.R. Expósito, J. González-Domínguez, J. Touriño, Hadoop Sequence Parser (HSP) library for FASTQ/FASTA datasets, https://github.com/rreye/hsp. [Visited March 2023].
[61] X. Li, G. Tan, C. Zhang, X. Li, Z. Zhang, N. Sun, Accelerating large-scale genomic analysis with Spark, in: Proceedings of the 2016 IEEE International Conference on Bioinformatics and Biomedicine, IEEE BIBM 2016, Shenzhen, China, 2016, pp. 747–751.
[62] F.A. Nothaft, et al., Rethinking data-intensive science using scalable analytics systems, in: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, SIGMOD'15, Melbourne, Australia, 2015, pp. 631–646.
[63] 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.
[64] R. Leinonen, et al., The European nucleotide archive, Nucleic Acids Res. 39 (suppl_1) (2010) D28–D31.

Roberto R. Expósito received the B.S. (2010), M.S. (2011) and Ph.D. (2014) degrees in computer science from the Universidade da Coruña (UDC), Spain, where he is currently an Associate Professor in the Department of Computer Engineering (UDC). His main research interests are in the areas of HPC and Big Data computing, focused on the performance optimization of distributed processing models in cluster and cloud infrastructures, and the parallelization of bioinformatics and data mining applications. His homepage is https://gac.udc.es/∼rober.

Jorge González-Domíngue received the B.S. (2008), M.S. (2009) and Ph.D. (2013) degrees in computer science from the Universidade da Coruña (UDC), Spain, where he is currently an Associate Professor in the Department of Computer Engineering (UDC). His main research interests include the development of parallel applications on multiple fields, such as bioinformatics, data mining and machine learning, focused on different architectures (multicore systems, GPUs, clusters, etc.). His homepage is https://gac.udc.es/∼jorgeg.