Future Generation Computer Systems 176 (2026) 108169

Contents lists available at ScienceDirect

Future Generation Computer Systems

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

Distributed compressive genomics: Fundamental pattern matching primitives via spark

Lorenzo Di Rocco [a,1], Umberto Ferraro Petrillo [a,1], Raffaele Giancarlo [b,2], Giuseppe Cattaneo [c,3,*]

[a] Dipartimento di Scienze Statistiche, Università di Roma - La Sapienza, Rome, Italy
[b] Dipartimento di Matematica ed Informatica, Università di Palermo, Palermo, Italy
[c] Dipartimento di Informatica, Università di Salerno, Fisciano (SA), Italy

ARTICLE INFO

Keywords:
Distributed systems
Compressive genomics
Spark
Pattern matching
FM-Index
Data compression

ABSTRACT

Compressive genomics leverages compressed data representations to enhance the efficiency of bioinformatics tasks like sequence comparison and search. Surprisingly, the fundamental operation of pattern matching on large DNA sequence collections remains unexplored in the realm of genomic analysis. However, distributed systems like Spark offer the scalability necessary to process increasingly large genomic datasets efficiently. We present the first Spark-based implementation of the FM-Index and Compressed Boyer-Moore (CBM) algorithms, evaluating their performance and providing insights into their advantages for large-scale bioinformatics applications. A comprehensive experimental study demonstrates clear performance gains over uncompressed approaches. Furthermore, we introduce *SparkGeco*, a distributed compressive genomics software library designed to simplify the integration of FM-Index and CBM algorithms into DNA sequence analysis pipelines within Apache Spark, thus supporting the development of efficient and scalable genomic analysis workflows. This work provides a concrete step towards high-performance, data-centric eScience solutions in computational biology.

1. Introduction

The growing volume of genetic data available, due to advancements in DNA sequencing technologies, presents both opportunities and challenges for bioinformatics [1]. On the one hand, the wealth of genetic information enables a more comprehensive understanding of biological systems, facilitating breakthroughs in fields such as genomics, evolutionary biology, and personalized medicine. On the other hand, processing large amounts of data poses staggering challenges in terms of computational resources (specifically, storage and time) especially during the data analysis phase.

To address those challenges, considerable progress has been made over the years in the design and deployment of efficient computational methods for bioinformatics tasks, as reviewed in [2], and for effective storage of genomic data, as also presented in [3]. High performance computing solutions have also been actively pursued and developed, as shown in [4–9]. Among these solutions addressing the mentioned challenges, two are relevant to place our contribution in the proper perspective: compressive genomics and distributed computing genomics.

The term compressive genomics was introduced in [10] to describe approaches that enable the management and analysis of genomic data in compressed form. Proof of principle methods were also proposed. The advantage of such an approach lies in the reduction in computational time by leveraging the succinct representation of a given dataset; the savings in space are an additional bonus.

Instead, distributed computing genomics refers to the possibility of using the computational capabilities of multiple machines to timely analyze large sets of genomic data in a parallel way.

* Corresponding author.
E-mail address: umberto.ferraro@uniroma1.it (G. Cattaneo).

[1] U.F.P. and L.D.R. partially supported by the INdAM -GNCS Project 2025 (CUP: E53C24001950001), and by the Università di Roma "La Sapienza" Research Project 2023, "Advancing Precision Public Health Medicine: High-Performance Computing for Efficient Solutions to Bioinformatics and Biostatistical Challenges."

[2] R.G. is partially supported by Projects funded by the European Union under the Italian National Recovery and Resilience Plan (PNRR) of NextGenerationEU: SPRINT, partnership on "Telecommunications of the Future" (PE00000001 - program "RESTART" - CUP E83C22004640001); "Models and Algorithms relying on knowledge Graphs for sustainable Development goals monitoring and Accomplishment - MAGDA" (CUP: B77G24000050001), under the "Future Artificial Intelligence - FAIR" program.

[3] G.C. is partially supported by Fondi di Ateneo per la Ricerca di Base (FARB ex 60%) 2023 "Nuovi sviluppi della ricerca nel campo della Cybersecurity, Networking, Intelligent Systems, Algorithms, Advanced Data Processing."

https://doi.org/10.1016/j.future.2025.108169
Received 2 April 2025; Received in revised form 28 August 2025; Accepted 23 September 2025
Available online 29 September 2025

1.1. Our contributions: Fundamental primitives for distributed compressive genomics

Despite significant progress in both compressive genomics and distributed computing genomics, the point of contact between these two areas has been very limited so far. To the best of our knowledge, the only available contributions concern the efficient distributed implementations of data compression algorithms for genomic data, e.g., [11–16]. However, these solutions focus only on compression and decompression, lacking support even for the simplest of the operations such as counting the number of occurrences of a pattern sequence within a longer text sequence. That is, they do not even support a fundamental and pervasive operation such as exact string searching [17], here referred to as pattern matching.

An exception worth mentioning is [18]. Here, the authors introduce a distributed hybrid-indexing method that uses compression to allow for efficient alignment of sequencing reads to multiple genomes simultaneously, addressing the challenges posed by the increasing volume of genomic data. The proposed solution is specialized for a specific application (i.e., aligning sequencing reads to large collections of genomes) and its extension to a more general scenario may be non-trivial to achieve.

However, algorithms and data structures supporting the execution of the pattern matching primitives on compressed data are available for sequential computer architectures, in particular the FM-Index [19], and Compressed Boyer-Moore Pattern Matching (CBM) [20–25]. More relevantly for this research, efficient implementations of algorithms supporting pattern matching in large genomic collections are still an active area of research [26,27]

Unfortunately, despite the well known capabilities of distributed computing frameworks like [28] to handle large genomic tasks [4], no implementation of the fundamental primitive is available in that setting.

Here, we close this gap by proposing a distributed version of the FM-Index and of the CBM, where the first is to be regarded as the central part of this research and the second as a complement.

A more detailed account of our contributions is as follows.

- **Convenience of Compressed Distributed Pattern Matching.** We conduct a performance comparison of non-distributed versus distributed versions of the algorithms we consider. The relevant parameters for the comparisons are specified in Section 5. Moreover, as is well known in the literature ([17]), the pattern matching problem solved by an FM-Index can also be solved by the CBM algorithms. Therefore, it is natural to compare these two approaches within distributed frameworks. We consider the following cases.
  - **Distributed FM-Index.** We consider the following main application scenario, where it is required to compress a collection of genomic sequences, to process later batches of pattern matching queries. The pattern does not require compression. In order to assess the validity of our proposal, we compare the performances of a classic Java implementation of the FM-Index [29] with our distributed one. Moreover, we consider two scenarios: (1) *light workload*, involving a few gigabytes of genomic sequences, and (2) *heavy workload*, involving hundreds of gigabytes of genomic sequences. Our experiments show that the distributed version of the FM-Index is nearly always superior to its non-distributed version, except in light workload scenarios. Moreover, our solution scales well with the number of computational resources available which, in turn, varies with input data size.
  - **Distributed CBM.** We use three versions of the CBM algorithm. The first, denoted CBM-2bit and due to [22], uses the straightforward 2-bit based compression. The second and third follow the approach proposed in [25] (but see also [23,24]), with byte-pair based compression [30] (denoted CBM-bp) and with LZW based compression [31] (denoted CBM-LZW). For each of these versions, we provide our implementation for both the sequential and the distributed versions. At this point, given a set of text and pattern queries simultaneously, the three versions are evaluated in analogy with what described for the FM-Index. Our experiments show that the distributed algorithms perform better than their sequential counterparts and scale well as a function of the computational resources available. Surprisingly, the preferred method among the three is CBM-2bit which, to the best of our knowledge, was not accounted for in the experiments conducted in [23,24].
  - **Comparison between Distributed FM-Index and CBM.** As is well known in the literature, the FM-Index can also be used to solve the same problem as CBM. Therefore, it is natural to compare the two approaches in a distributed setting. Our experiments show that the FM-Index should be preferred over to all versions of CBM, except when the number of queries is very small compared to the size of the text sequences.
- **Software.** The Spark code developed for this study is provided in a publicly available single software library, allowing users with no distributed programming skills to incorporate easily the compressed distributed FM-Index and CBM versions into sequence analysis pipelines.

2. Background

2.1. Compressive genomics

Compressive genomics is a computational paradigm [10] exploiting the high redundancy existing in genomic data. This redundancy across genomes can be leveraged by encoding them in a compressed format preserving the similarity patterns needed for later analysis. The resulting representation would then accelerate the required analysis by performing direct computation on its compressed form using methods known as “compressive algorithms”.

Before genomics, this area was studied in abstract algorithmic terms to design computational methods that work on generic compressed data (e.g., [20]), with self-indexes being among the most profound and useful discoveries [32]. In addition to the ones mentioned in [33], fundamental contributions that have shaped compressive sequence analysis are reported in [34]. Although compressive genomics is a fascinating and actively studied area [35], apart from self-indexes, the solutions obtained so far are too specialized for wide use and therefore have limited impact in genomics.

2.2. Pattern matching on compressed genomic sequences

Pattern matching is a fundamental operation in many computational biology pipelines, including tasks such as sequence alignment and variant detection. Despite its apparent simplicity, the computational cost of pattern matching becomes critical when processing large-scale genomic data. Two major challenges arise in this setting. First, input datasets may exceed the capacity of main memory, forcing reliance on slower secondary storage and incurring high I/O costs. Second, the sheer volume of data imposes substantial computational overhead, even when using optimized pattern matching algorithms.

To address these challenges, we observe that the standard format used to encode genomic sequences is inefficient: it typically uses one byte per character, even though the DNA alphabet consists of only four symbols (A, C, G, T). The key idea behind compressive genomics is, first of all, to use compression for reducing the amount of memory required storing genomes and their derivative data. Moreover, it encourages the development of compression formats able to natively support frequent genome operations in the bioinformatics field without requiring input genomes to be previously decompressed.

Following the classification introduced in [36], genomic compression methods can be categorized in two different modes

- **Horizontal Mode**: compressing S using the information contained only in that sequence. Methods having this property can be further classified as follows:
  - Substitutional based: S is compressed by substituting some of its longer fragments with shorter ones, while maintaining the correspondence using a dictionary. This technique is usually implemented using the *Context Tree Weighting* algorithm [37] or a variant of the *LZ* algorithms [38–40].
  - Statistical based: S is compressed considering an empirically estimated probability distribution of each symbol of the alphabet.
  - Substitutional and Statistical Based: S is compressed using a combination of the substitutional and of the statistical approaches.
  - Transformational based: S is transformed in a different format before the compression phase, in order to improve the compression ratio. One of the most known and used transformations for DNA sequences is the *Burrows-Wheeler Transformation* (*BWT*) [41].
  - Grammar based: the compression is carried out by first generating a context-free grammar from S. The resulting grammar rules are then used to compress the input sequence.
  - Two-bits based: S is compressed exploiting the small DNA alphabet. By using a 2 bits representation for each of the four standard DNA characters, up to 4 characters are packed in a single byte.
- **Vertical Mode**: compressing S using the information contained in a larger set of sequences. This mode is generally used to manage a large set of sequences and it works particularly well when all these sequences come from similar organisms. Usually, the compression is carried out using a sequence as reference for other sequences of the set.

Among the different strategies described above, we now focus on the two compression approaches that are most relevant to efficient pattern matching in genomic applications.

2.2.1. Text preprocessing: FM-Index

The FM-Index belongs to the class of transformational compression methods. It is built from the Burrows-Wheeler Transform (BWT). The FM-Index is expected to work very well with genomic sequences because it is able to exploit the presence of repeating patterns of nucleotide fragments. The BWT plays a crucial role in enabling pattern matching operations, such as counting the occurrences of a pattern within a string, through a technique called backward search. Thus, pattern searching queries on the FM-Index are extremely fast, but the index construction requires a larger upfront investment in terms of time and space resources.

2.2.2. Pattern preprocessing: D-BM compressed

Compressed Boyer-Moore (CBM) methods belong to the class of substitutional compression methods. Here, we consider two variants of this algorithm. The first, called **d-BM** and described in [22], was developed specifically for processing genomes compressed using the two-bit coding scheme. In this scheme, first presented in [22], each of the four nucleotides is encoded with two bits. This reduces the space required to store a nucleotide sequence by a factor of about 4.

The second variant is an implementation of the general pattern matching algorithm described in [25], which can apply the logic of the Boyer-Moore algorithm to any dictionary-based compression algorithm. In our case, we consider two popular encoding schemes. The first is the Lempel-Ziv-Welch encoding scheme. First presented in [38], is a variation of the original Lempel-Ziv algorithm described in [39]. It uses a translation table to map strings into fixed-length codes. In the second encoding scheme, the Byte-Pair encoding scheme, frequent pairs of adjacent characters that occur in an input string are replaced by a single byte that does not occur in that string (see [30] for more details).

2.3. Distributed computing genomics

Distributed computing systems are designed to solve large-scale problems by coordinating the execution of tasks across multiple interconnected machines. These systems include multiple *computing nodes*, each representing a physical or virtual machine. Within each computing node, there may be one or more *computing units*, such as CPU cores or executor processes, responsible for performing the actual computations in parallel.

The appeal of distributed computing for computational biologists lies in the apparent ease with which, as the dataset size grows, additional computational resources (namely processors and memory) can be employed "on demand". Although such scalability is not easy to achieve, quite a few classic bioinformatics tasks have been implemented in such a setting (see, e.g., [42–44]). However, distributed computing infrastructures can be costly if not used efficiently and, in fact, the design of efficient algorithms for genomic tasks is actively pursued, e.g., [45,46]. Adapting an algorithmic approach to a distributed system involves several significant challenges, including:

- Efficient partitioning and distribution of input data across computing nodes to minimize unnecessary transmissions;
- Scalable solutions to ensure all the nodes of a distributed system process an equivalent workload;
- Effective memory management to maximize in-memory data storage, to reduce the usage of slower external devices and I/O performance bottlenecks.

2.4. Apache spark

High-level programming paradigms have emerged to simplify the design of distributed algorithms, allowing developers to focus more on algorithmic logic than on low-level details. Among these, the *MapReduce* (MR) paradigm [47] represents the most prominent one, with *Apache Spark* being its most widely adopted implementation framework.

Apache Spark, Spark from now on, organizes data using *Resilient Distributed Datasets* (RDDs), which are collections of key-value pairs partitioned across a cluster of computing nodes, called *worker nodes*. Each worker node can host one or more *executors*, where typically each executor is tied to one computing unit. Executors are responsible for the execution of tasks, where each task processes a partition of an RDD. At the core of a MR application is the *driver* program, which coordinates the execution flow. Specifically, an input RDD can be processed using a *map* or a *reduce* function.

A map function transforms each input pair into zero or more output pairs, potentially changing the size and domain of the input RDD. Conversely, a reduce function aggregates all values associated with the same key using, thereby reducing, the number of key-value pairs while preserving the key domain.

A complete MR program is typically expressed as a nested sequence of map and reduce operations applied to an initial RDD, transforming the dataset progressively toward the final desired output.

Finally, to efficiently share read-only data such as query sets across all worker nodes without repeatedly sending it, Spark provides *broadcast variables*, which cache these data locally on each node.

2.4.1. Limitations and performance overheads

Although Apache Spark offers several advantages over lower-level frameworks for parallel and distributed computing, it also introduces potential performance limitations that must be taken into account before opting for this solution. One of the most relevant ones is data partitioning. Unlike other solutions where partitioning is implemented by the developer, Apache Spark automatically partitions input datasets across the nodes of a distributed system. While this alleviates the developer from a complex task, it may also lead to severe performance limitations when the resulting partitions are imbalanced, yielding an uneven workload distribution.

Another relevant performance issue is related to I/O bottlenecks, especially during the execution of reduce functions. These operations requires a preliminary *shuffling* phase where input data is partitioned and aggregated across the nodes of a distributed system according to their keys. On a side, this implies that the computation is stalled until this (mostly I/O-based) aggregation is finished. On the other side, the sheer amount of data exchanged during this phase may likely saturate the communication networks, leading to relevant delays in the overall execution time. More details the limitations and the performance overhead that may affect a distributed application based on Apache Spark, and the possible solutions, can be found in [4].

3. Methodology

In this section, we present a formal description of the new algorithms and distributed data structures introduced in this work. Our contribution allows to perform exact pattern matching directly on compressed representations of large-scale genomic sequences, while leveraging the scalability and fault-tolerance of distributed systems.

3.1. Problem definition

Let $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$ be a collection of genomic sequences. Given a pattern P, our goal is to count the total number of times P occurs in $\mathcal{S}$, by processing each sequence of $\mathcal{S}$ in its compressed representation under a distributed computing environment.

3.2. The proposed algorithms

We describe here two classes of distributed compression-aware pattern matching algorithms: FM-Index and CBM. The proposed algorithms are designed to work on compressed data and to efficiently operate on a Spark-based distributed environment.

Each algorithm relies on one of the following data distribution strategies, used to define how the input collection $\mathcal{S}$ of genomic sequences is initially partitioned over the computing nodes:

- **Scattering.** Each sequence in $\mathcal{S}$ is entirely assigned to a single node of the distributed system, where it is compressed using the chosen algorithm and stored.

  This strategy is preferable when analyzing large collections of relatively small sequences.
- **Partitioning.** Each sequence in $\mathcal{S}$ is split into smaller chunks, sent to different nodes of the distributed system. Each chunk does not overlap with the others, except for a prefix of the following one. This overlap ensures that each subsequence with a length up to k is entirely contained in at least one chunk.

  This strategy is likely to be preferred when analyzing collections of very long sequences.

3.2.1. FM-Index Based distributed pattern matching

Let $\mathcal{S}$ be a collection of genomic sequences. Assuming the scattering data distribution strategy is used, for each sequence s in $\mathcal{S}$:

- s is sent to one of the available computing nodes (scattering operation), as described in Algorithm 1.
- The FM-Index is built by the hosting node on s using suffix array and BWT, as described again in Algorithm 1.
- Pattern matching queries are executed in parallel on each node, where multiple executors divide the set of locally stored FM-Indexes and perform backward search concurrently, as described in Algorithm 3.
- Local results are sent for aggregation to the Spark driver program using a reduce function, as described again in Algorithm 3.

Algorithm 1: Building a distributed compressed representation of a collection of sequences using FM-Index and scattering strategy.

Input:
$\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$ `// input sequences`
Output:
`// RDD of FM-Index compressed sequences`
$\hat{\mathcal{R}}_{FM} = \{(i, \hat{s}_i)\}_{i=1}^{n}$

`// Step 1: Load the sequences into an RDD`
1 $\mathcal{R} = \{(i, s_i)\}_{i=1}^{n}$

`// Step 2: Build the distributed FM-Index`
2 $\hat{\mathcal{R}}_{FM} \leftarrow \mathcal{R}.\texttt{map}((i, s) \rightarrow (i, \texttt{Compress}_{FM}(s)))$

3 **return** $\hat{\mathcal{R}}_{FM}$

3.2.2. CBM Based distributed pattern matching

Let $\mathcal{S}$ be a collection of genomic sequences. Assuming the partitioning data distribution strategy is used, for each sequence s in $\mathcal{S}$:

- s is split into chunks of equal size, where each chunk includes also a small prefix of the following one. The resulting collection of chunks is stored by means of a Spark distributed RDD data structure (partitioning operation), as described in Algorithm 2.
- Each block c of s is compressed using one of the supported schemes, as described again in Algorithm 2:
  - **CBM-2bit**: fixed-width encoding exploiting the four-symbol DNA alphabet.
  - **CBM-LZW**: dictionary-based Lempel-Ziv-Welch compression.
  - **CBM-BPE**: substitution-based Byte-Pair Encoding.
- Pattern matching queries are executed in parallel on each node, by running a variant of the Boyer-Moore pattern matching algorithm on each compressed chunk available locally, as described in Algorithm 3.
- Local results are sent for aggregation to the Spark driver program using a reduce function, as described again in Algorithm 3.

Algorithm 2: Building a distributed compressed representation of a collection of sequences using CBM and partitioning strategy.

Input:
$\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$ `// Input Sequences`
ℓ `// Chunk size`
δ `// Overlap buffer length`
Output:
`// RDD of CBM compressed chunks`
$\hat{\mathcal{R}}_{CBM} = \{(i, \hat{s}_i)\}$

`// Step 1: Load the sequences into an RDD`
1 $\mathcal{R} = \{(i, s_i)\}_{i=1}^{n}$

`// Step 2: Split each sequence into chunks`
2 $\mathcal{C} \leftarrow \mathcal{R}.\texttt{map}(s \rightarrow \texttt{split}(s, \ell, \delta))$

`// Step 3: Compress each chunk using CBM`
3 $\hat{\mathcal{R}}_{CBM} \leftarrow \mathcal{C}.\texttt{map}((i, c) \rightarrow (i, \texttt{Compress}_{CBM}(c)))$

4 **return** $\hat{\mathcal{R}}_{CBM}$

4. System design and implementation

In this section we introduce *SparkGeco*, a software library allowing for compression-aware distributed-pattern matching on large-scale genomic data using Spark.

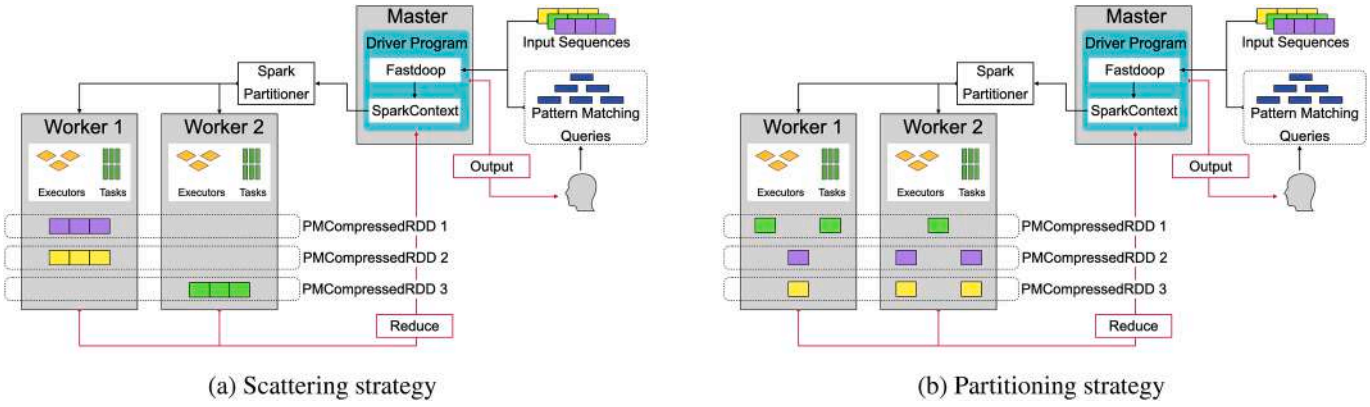

Fig. 1. System architecture of *SparkGeco* showing the two supported data distribution strategies: (a) scattering and (b) partitioning. Input genomic sequences are loaded into the memory of the distributed system using the Fastdoop library. The driver node coordinates the distribution of data across the cluster. In the scattering strategy, each full sequence is assigned to a single node; in the partitioning strategy, sequences are split into overlapping fixed-length chunks and distributed across nodes. Each worker compresses its assigned data in parallel via Spark executors. Pattern matching queries are sent by the driver, locally processed by the workers, and the partial results are aggregated and returned to the driver for final output.

Algorithm 3: Performing pattern matching queries on a compressed distributed representation of a collection of sequences.

Input:
$\hat{\mathcal{R}}_\tau = \{(i, \hat{s}_i)\}$ `// RDD of τ-compressed sequences`
$\mathcal{Q} = \{q_1, q_2, \ldots, q_m\}$ `// Pattern matching queries`
Output:
$\mathcal{C} = \{(q_j, c_j)\}$ `// Occurrences for each query`

`// Step 1: Broadcast the set of queries to all nodes`
$\mathcal{Q}_{bc} \leftarrow$ `broadcast`$(\mathcal{Q})$
`// Step 2: For each compressed sequence, execute all queries`
$\mathcal{M} \leftarrow \hat{\mathcal{R}}_\tau$.`map`$((i, \hat{s}) \rightarrow$
  `[for each` $q_j \in \mathcal{Q}_{bc}$: `match`$_\tau(\hat{s}, q_j)$`]`
)
`// Step 3: Aggregate match counts by query`
$\mathcal{C} \leftarrow \mathcal{M}$.`reduce`$((a, b) \rightarrow a + b)$
return $\mathcal{C}$

4.1. Architectural overview

From the technical viewpoint, *SparkGeco* introduces a novel type of distributed compressed data structure, referred to as `PMCompressedRDD`, which is derived from the standard RDD type, available with Spark (see Section 2.4). At a high level, all operations involving the distributed compressed representation of genomic sequences are orchestrated by this class. It implements a standard interface including primitives for:

- **Compression.** Converts input sequences into compressed representations using one of the supported methods.
- **Pattern matching.** Executes exact pattern matching queries on the compressed sequences.
- **Decompression.** Converts compressed sequences to their decompressed form using one of the supported methods.

The actual low-level implementation of these primitives is delegated by `PMCompressedRDD` to specialized secondary classes for each supported compression method, to ensure modularity and extensibility of the code. Fig. 1 depicts diagram summarizing graphically the system architecture.

4.2. Supported compression schemes

SparkGeco currently supports four compression-aware pattern matching methods:

- FM-Index: FM-Index construction and backward search-based querying.
- CBM-2bit: 2-bit encoding of nucleotides, with a CBM-style search algorithm.
- CBM-bp: Byte-Pair Encoding (BPE)-based compression and CBM search.
- CBM-LZW: Lempel-Ziv-Welch (LZW) compression and CBM search.

By default, the FM-Index implementation uses the scattering data distribution strategy, while all CBM variants use the partitioning data distribution strategy.

4.3. Architectural engineering details

This section provides a detailed description of the architectural engineering of the main primitives coming with the `PMCompressedRDD` class. These primitives are executed on a distributed system and, for each instance of the `PMCompressedRDD` class, rely on the availability of a secondary class in charge of providing the low-level primitives corresponding to the chosen compression method.

4.3.1. Compression

Compression is automatically performed when a new instance of `PMCompressedRDD` is created. It is implemented as follows:

- **Data Acquisition:** Input sequences are loaded from FASTA or FASTQ files using the Fastdoop library [48].
- **Data Distribution:** The implementation of this step depends on the chosen data distribution strategy.
  - *Scattering*: each sequence is assigned as a whole to one node of the distributed system, which becomes responsible for its compression and storage;
  - *Partitioning*: each sequence is split into non-overlapping chunks of fixed length, except for a short k-length prefix from the subsequent chunk, with each chunk being sent to a node of the distributed system. This prefix, also referred to as overlap buffer, is required to guarantee that each subsequence with a length up to k is entirely contained in at least one chunk.

- **Data Compression:** the process is carried out independently and in parallel by Spark executors on each partition. Compression is delegated to the corresponding primitive implemented in the secondary class associated with the selected compression scheme.

4.3.2. Pattern matching query

A pattern matching query can be executed by invoking the `search()` method available on the `PMCompressedRDD` class, while providing the pattern P as input. It is implemented as follows.

- **Pattern Broadcast:** The pattern P is broadcast to all executors using the Spark `broadcast` communication primitive, so as to reduce network overhead.
- **Local Query Execution:**
  - *Scattering*: each node applies the pattern matching routine to the full compressed sequences it holds. When multiple compressed sequences are on the same node, they are processed in parallel by distinct executors.
  - *Partitioning*: the input pattern is matched independently against each compressed chunk of the same global sequence. When multiple compressed objects are on the same node, they are processed in parallel by distinct executors.
- **Aggregation:**
  Each executor evaluates a count of the pattern occurrences in the processed sequences. These partial counts are sent back to the driver node and summed through a Spark aggregation primitive.

4.3.3. Decompression

Decompression is available to the user through the optional `decompress()` method of the `PMCompressedRDD` class. In its implementation,each node independently decompresses its assigned sequences, and in parallel, using multiple executors.

4.4. Our library: A user point of view

We now provide a short example on how our library can be used in a genomic analysis pipeline. Assume the input consists of a collection S of genomic sequences, initially stored on an external storage in either FASTA or FASTQ format, and a batch $\mathcal{B}$ of patterns to be matched.

Suppose one is interested in performing this task using `FmRDD`, i.e., the FM-Index implementation available with *SparkGeco*. As a first step, one has to create a new object of this type, by using the provided `read()` method, which requires the input path of S. Once invoked, this method loads the input sequences using the `Fastdoop` library [48] and, then, compresses them using the algorithm described in Section 3.2.1. The result is a distributed collection of FM-Indexes.

After the data is loaded, pattern matching is done by executing the `search()` method, which returns the number of occurrences of each pattern x in $\mathcal{B}$.

Switching to a different compression algorithm requires only to replace the `FmRDD` with the desired algorithm (e.g., `BpeRDD`). Users are not required to understand the technical details regarding compression and query processing, as these are handled transparently by the library.

5. Experimental evaluation

5.1. Experimental design

In general, given a compressive genomic algorithm, its distributed implementation can be evaluated according to the following metrics: 1) compression effectiveness, 2) algorithm execution (including a possible preprocessing step) and 3) scalability of the proposed algorithm, i.e., its ability to scale its execution times in proportion with the size of the underlying distributed system. We designed experiments, detailed next, that account for those metrics, within the two scenarios outlined in the Introduction.

Table 1
Details about the datasets used for our experiments.

ID	Genome	Size (GB)
PL1	Picea Abies	11.2
PL2	Picea Glauca	18.8
PL3	Pinus Taeda	19.1
SARSCOV	SARS-CoV-2	198.6

5.1.1. Datasets

With reference to the two scenarios introduced in Section 1.1, we use the following datasets,[1] whose sizes have been chosen according to what stated in that section.

- **Light workload scenario: Plant Genomes.** This scenario uses as the collection of sequences to be queried three plant genomes that have already been considered in distributed computing bioinformatics applications [49]. Namely, Picea Abies, Picea Glauca and Pinus Taeda (respectively, PL1, PL2 and PL3). As for the queries, we generate batches of pattern matching operations targeting random patterns in these sequences.
- **Heavy workload scenario: SARSCOV Dataset.** This scenario uses as sequences to be queried the collection of the different SARS-CoV-2 genomes available from the National Center for Biotechnology Information (NCBI) database. As for the queries, we generate batches of pattern matching operations targeting random patterns in these sequences.

Details about the size of these datasets are available in Table 1.

5.1.2. Computational environment

We use a Spark 3.5.1 installation running on a cluster with 5 computing nodes. One node acts as the master and the rest act as workers. Each worker node is equipped with 64 GB of RAM and 32 processing cores, available by Spark as computational units, for an overall of 128 computing units.

5.2. Experiments: Aim and scope

Given the *SparkGeco* library, we now detail the experiments we have conducted, indicating for each its objective and motivation[2].

Experiment 1: Measuring the compression memory saving, i.e., compression effectiveness.

- **Objective.** Analyze the memory savings achieved by compressing the genomes of our datasets using *SparkGeco* distributed implementations of compressed data structures (e.g., FM-Index). Notice that the memory occupation of our compressed distributed data structures is not influenced by the actual number of computing units being used. For this reason, we take such a measure on the Plant Genomes and the SARSCOV datasets using just one computational unit (see Section 5.1.2).
- **Motivation.** The efficiency of the compression has a cascading effect on the computational efficiency of subsequent analyses. Specifically, efficient compression reduces the amount of data that needs to be processed and transferred across nodes in a distributed computing environment. This, in turn, can lead to faster analysis times and more efficient use of computational resources.

[1] The dataset links are provided in the official GitHub repository at https://github.com/ldirocco/SparkGeco

[2] A copy of the scripts used for these experiments, along with all other materials needed to reproduce them, can be found in the official GitHub repository at https://github.com/ldirocco/SparkGeco

Experiment 2: Evaluation of the compression operation cost, i.e., algorithm execution - preprocessing.

- **Objective.** Measure the computational cost involved in compressing input datasets using *SparkGeco*. We take such a measure on the Plant Genomes and the SARSCOV datasets with a fixed number of computing units at work, set to 32, as scalability is evaluated in Experiment 5.
- **Motivation.** A high cost to be paid in compression may discourage its use, even if later processing tasks may take advantage of it.

Experiment 3: Evaluation of the pattern matching query cost, i.e., algorithm execution - query processing.

- **Objective.** Compare the time required to perform a large batch of pattern matching queries on the SARSCOV dataset compressed using *SparkGeco*, with the time required for the same operations on the uncompressed dataset. We take such a measure with a fixed number of computing units at work, set to 32. The same evaluation was also carried out on the Plant Genomes datasets, to ensure consistency in analysis across different scenarios.
- **Motivation.** Assess the efficiency gains in execution time when querying compressed distributed data compared to distributed uncompressed data. Specifically, we are interested in assessing how many pattern matching operations are needed to repay the initial execution cost for each of the supported compression schemes.

Experiment 4: Evaluation of the trade-off between initial compression and following pattern matching , i.e., algorithm execution - preprocessing and query processing.

- **Objective.** Evaluate the time required to initially compress the SARSCOV dataset. It also focuses on the time needed to carry out an increasing number of pattern matching operations targeting random patterns, comparing the proposed compressive algorithms with the corresponding algorithms using uncompressed data. We take such a measure with a fixed number of computing units at work, set to 32. The same evaluation is also carried out on the Plant Genomes datasets.
- **Motivation:** Determine the minimum number of pattern matching operations required to balance the initial execution cost for each of the supported compression schemes.

Experiment 5: Evaluation of the scalability.

- **Objective.** Measure the execution time of compression and pattern matching operations on the SARSCOV dataset using the different compression schemes supported by *SparkGeco* with an increasing number of computing units. The same evaluation is also carried out on the Plant Genomes datasets.
- **Motivation:** Determine the scalability of the proposed algorithms. Specifically, we are interested in assessing if the execution time of these algorithms decreases linearly with the increase in the number of computing units.

6. Results and discussion

We present and analyze the results of the experiments introduced in Section 5. As far as the Plant Genomes are concerned, we report only the results regarding PL3, since the remaining ones have analogous outcomes and can be omitted for brevity. They are available upon request.

Experiment 1: Measuring the compression memory saving, i.e., compression effectiveness.

- **Indications to be drawn.** The results regarding this experiment show that the use of compression leads to significant savings in the size of the analyzed datasets. This is true even when constrained to support pattern matching queries. This is especially true for the 2-bit and FM-Index cases. Their compression efficiency is similar, so the most favorable option between them must be determined by the indications given by the remaining experiments.

Table 2
Overall execution times, in seconds, required to compress the input datasets using the supported compression methods (**comp**) and to perform batches of 10,000 pattern matching queries targeting random patterns on these datasets (**pm**), along with their sum (**total**), on a distributed system with 32 computing units. It is reported also the time required to query the input uncompressed datasets (**Uncompressed**) using a distributed linear scan algorithm.

Method	Operation	Execution times (seconds)			
		PL1	PL2	PL3	SARSCOV
CBM-2bit	**comp**	58	71	73	695
	pm	2170	2570	2650	5670
	total	**2,228**	**2,641**	**2,723**	**6,365**
CBM-bp	**comp**	87	102	103	1183
	pm	2210	2830	2890	5730
	total	**2,297**	**2,932**	**2,993**	**6,913**
CBM-LZW	**comp**	83	105	107	1021
	pm	2240	2990	3010	5840
	total	**2,323**	**3,095**	**3,117**	**6,861**
FM-Index	**comp**	69	83	86	834
	pm	1280	1570	1630	2750
	total	**1,349**	**1,653**	**1,716**	**3,584**
Uncompressed	**comp**	–	–	–	–
	pm	6210	6820	4150	7780

Experiment 2: Evaluation of the compression operation cost, i.e., algorithm execution - preprocessing.

- **Indications to be drawn.** The results regarding this experiment, reported in Table 2, clearly show that the 2-bit and FM-Index primitives outperform BPE and LZW due to their lower computational overhead and optimized data structures.

Experiment 3: Evaluation of the pattern matching query cost, i.e., algorithm execution - query processing.

- **Indications to be drawn.** The results regarding this experiment, reported in Table 2, suggesting significant performance gains from using compressive algorithms. We also note that the performance of the FM-Index in this case is much better than that of all other compression algorithms, including 2-bit. Another relevant fact is that the performance improvement experienced by using compressive algorithms is similar for all datasets. This suggests that the observed trends are not specific to a particular dataset, but rather characteristics of the algorithms themselves.

Experiment 4: Evaluation of the trade-off between initial compression and following pattern matching , i.e., algorithm execution - preprocessing and query processing.

- **Indications to be drawn.** The results regarding this experiment, reported in Figs. 2–3, show that compressive genomics offers no advantage when the number of pattern matching queries is relatively small. For the Plant Genomes, we report results only for PL3, as the results for the others are similar and omitted for brevity (available upon request). When increasing this number (i.e., $\geq 2,000$), FM-Index quickly becomes the fastest option. This applies both to the SARSCOV dataset and the Plant Genome datasets (data for plant genomes other than PL3 are omitted for brevity but are available upon request). We also notice that the 2-bit encoding scheme does not outperform the other in terms of cumulative execution time.

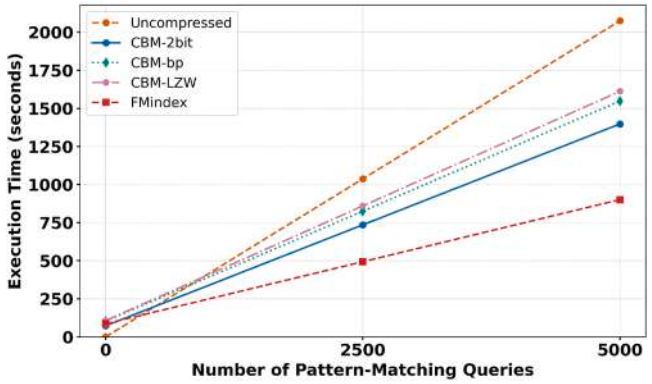

Fig. 2. Cumulative execution times, in minutes, required to compress the PL3 input dataset and executing an increasing number of pattern matching queries, using the encoding schemes available with *SparkGeco* and 32 computing units.

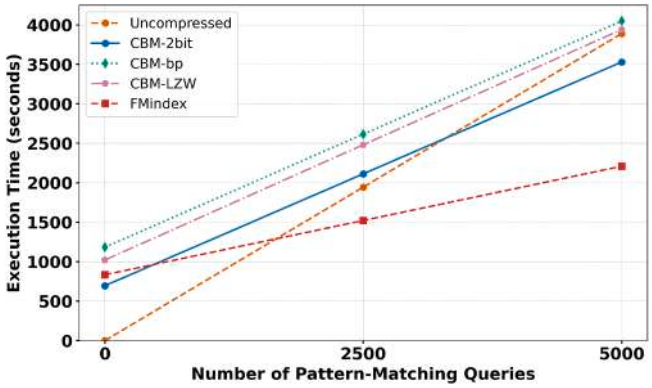

Fig. 3. Cumulative execution times, in minutes, required to compress the SARSCOV input dataset and executing an increasing number of pattern matching operations, using the encoding schemes available with SparkGeco and 32 computing units.

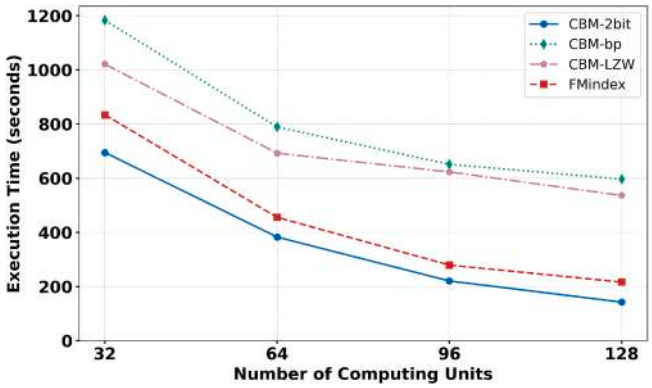

Fig. 4. Execution times, in minutes, required to compress the SARSCOV dataset using the encoding schemes available with *SparkGeco*, and increasing computing units.

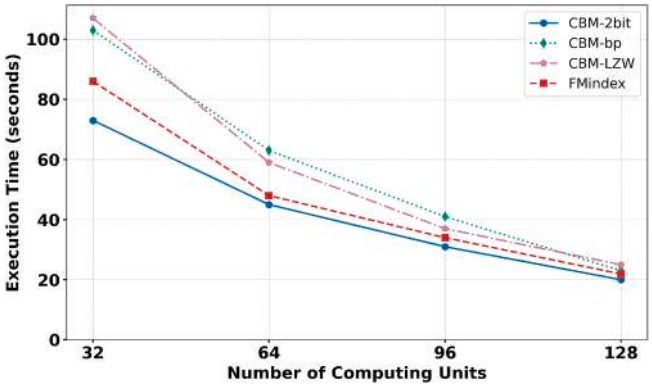

Fig. 5. Execution times, in minutes, required to compress the PL3 dataset using the encoding schemes available with *SparkGeco*, and increasing computing units.

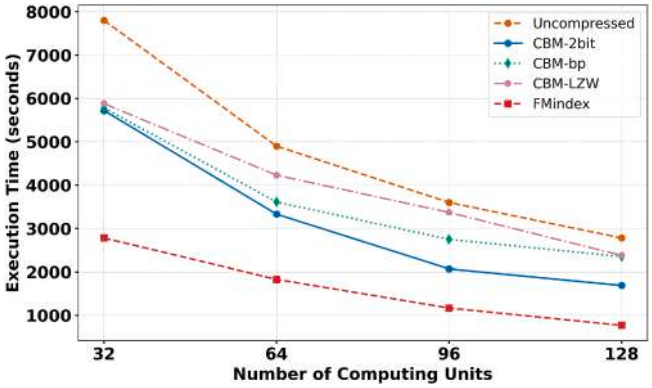

Fig. 6. Execution times, in minutes, required to perform a batch of 10,000 pattern matching queries targeting random patterns on the SARSCOV dataset, using the encoding schemes available with *SparkGeco*, and increasing computing units, against the same operations carried out on an uncompressed version of the same dataset.

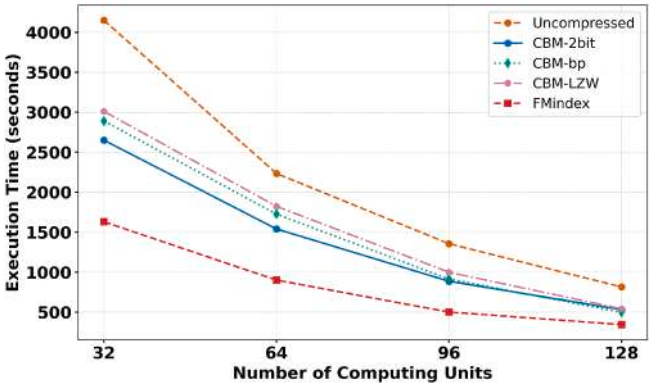

Fig. 7. Execution times, in minutes, required to perform a batch of 10,000 pattern matching queries targeting random patterns on the PL3 dataset, using the encoding schemes available with *SparkGeco*, and increasing computing units, against the same operations carried out on an uncompressed version of the same dataset.

Experiment 5: Evaluation of the scalability.

- **Indications to be drawn.** The results regarding this experiment, reported in Figs. 4–7, show that all considered algorithms are able to scale well with the number of computing units, although not reaching a perfect linear scalability, likely because of increased network communication overhead. Moreover, scalability decreases as the number of computing units increases, likely due to external factors like the communication overhead on the shared network. This applies both to the SARSCOV dataset and to the Plant Genome datasets (data for plant genomes other than PL3 are omitted for brevity but are available upon request).

7. Conclusion

In this paper we have presented a comprehensive evaluation of the advantages of compressive genomics for pattern matching, compared to more conventional approaches. We have achieved this by developing

and exploiting the first known distributed implementation of an FM-Index data structure, showing its effectiveness in handling large-scale genomic datasets on distributed systems.

Our study also included simpler approaches for doing pattern matching on compressed data, that are well-suited for handling more basic application scenarios, such as those based on the Compressed Boyer-Moore pattern matching algorithm. These alternative approaches were also implemented as distributed algorithms, being the first of this type to be developed.

Our findings show that performing pattern matching on compressed data is convenient (see Table 2). Furthermore, we have shown that our approach is scalable with respect to the size of the input datasets and the number of computational units. Finally, as expected, we have seen that for small-scale tasks, simpler compression schemes, like the two-bit encoding scheme based pattern matching algorithm, are favorable, thanks to their small compression overhead and their fast execution times.

Apart from the considered use-cases, our framework can be applied to many other relevant tasks, either coming from bioinformatics or more general data science fields. To name a few, we cite the need to efficiently query compressed genomic datasets for the rapid identification of genetic mutations, as well as the need for fast and scalable querying of large text corpora in natural language processing tasks.

CRediT authorship contribution statement

Lorenzo Di Rocco: Writing – original draft, Software, Methodology, Conceptualization; **Umberto Ferraro Petrillo:** Writing – review & editing, Methodology, Investigation, Conceptualization; **Raffaele Giancarlo:** Writing – review & editing, Supervision, Conceptualization; **Giuseppe Cattaneo:** Writing – review & editing.

Data availability

A copy of all materials, included the data used for the experiments, can be found in the official GitHub repository at https://github.com/ldirocco/SparkGeco

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.

References

[1] V. Marx, The big challenges of big data, Nature 498 (2013) 255–260. https://doi.org/10.1038/498255a

[2] B. Berger, J. Peng, M. Singh, Computational solutions for omics data, Nat. Rev. Genet. 14 (2013) 333–346. https://doi.org/10.1038/nrg3433

[3] I. Numanagić, J.K. Bonfield, F. Hach, J. Voges, J. Ostermann, C. Alberti, M. Mattavelli, S.C. Sahinalp, Comparison of high-throughput sequencing data compression tools, Nat. Methods 13 (2016) 1005–1008. https://doi.org/10.1038/nmeth.4037

[4] D. Chicco, U.F. Petrillo, G. Cattaneo, Ten quick tips for bioinformatics analyses using an apache spark distributed computing environment, PLoS Comput. Biol. 19 (2023) 1011272. https://doi.org/10.1371/journal.pcbi.1011272

[5] J.P. Courneya, A. Mayo, High-performance computing service for bioinformatics and data science, J. Med. Libr. Assoc. 106 (2018) 494–495. https://doi.org/10.5195/jmla.2018.512

[6] L.D. Rocco, U.F. Petrillo, A distributed alignment-free pipeline for human SNPS genotyping, in: Proceedings of the 14th ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics, the 14th ACM International Conference on Bioinformatics, Computational Biology, and Health InformaticsNew York, NY, USA, Association for Computing Machinery, 2023, pp. 1–10. https://doi.org/10.1145/3584371.3612990

[7] L.D. Rocco, U.F. Petrillo, Efficient and scalable alignment-free distributed genotyping of snps and short indels, IEEE Trans. Comput. Bio. Bioinf. 22 (2025) 101–115. https://doi.org/10.1109/TCBB.2025.3525547

[8] L.D. Rocco, U.F. Petrillo, S.E. Rombo, DIAMIN: a software library for the distributed analysis of large-scale molecular interaction networks, BMC Bioinf. 23 (1) (2022) 474. https://doi.org/10.1186/s12859-022-05026-w

[9] M.S. Nobile, P. Cazzaniga, A. Tangherloni, D. Besozzi, Graphics processing units in bioinformatics, computational biology and systems biology, Brief. Bioinf. 18 (2016) 870–885. https://doi.org/10.1093/bib/bbw058

[10] P.R. Loh, M. Baym, B. Berger, Compressive genomics, Nat. Biotechnol. 30 (2012) 627–630. https://doi.org/10.1038/nbt.2241

[11] Y. Ji, H. Fa, H. Yao, S. Shao, M. Wu, H. Fang, Q. Liu, S. Liu, FQCSpark: efficient spark-based parallel compression algorithm for FASTQ genome sequences, in: Proceedings of the 2022 IEEE 25th International Conference on Computer Supported Cooperative Work in Design, the 2022 IEEE 25th International Conference on Computer Supported Cooperative Work in Design, IEEE, 2022, pp. 671–676. https://doi.org/10.1109/CSCWD54268.2022.9776028

[12] Y. Ji, H. Fang, H. Yao, J. He, S. Chen, K. Li, S. Liu, FastDRC: fast and scalable genome compression based on distributed and parallel processing, in: International Conference on Algorithms and Architectures for Parallel Processing, Springer, 2019, pp. 313–319.

[13] S. Roy, A. Mukhopadhyay, A randomized optimal k-mer indexing approach for efficient parallel genome sequence compression, Gene 907 (2024) 148235.

[14] S. Vargas-Pérez, F. Saeed, A hybrid MPI-OpenMP strategy to speedup the compression of big next-generation sequencing datasets, IEEE Trans. Parallel Distrib. Syst. 28 (2017) 2760–2769. https://doi.org/10.1109/TPDS.2017.2692782

[15] H. Yao, S. Chen, S. Liu, K. Li, Y. Ji, G. Hu, R. Wang, A parallel compression for large collections of genomes, Concurrency Comput. Pract Exper. 34 (2022) 6339.

[16] H. Yao, G. Hu, S. Liu, H. Fang, Y. Ji, SparkGC: spark based genome compression for large collections of genomes, BMC Bioinf. 23 (2022) 297. https://doi.org/10.1186/s12859-022-04825-5

[17] D. Gusfield, Algorithms on Strings, Trees, and Sequences: computer Science and Computational Biology, Cambridge University Press, 1997. https://doi.org/10.1017/CBO9780511574931

[18] A.I. Maarala, O. Arasalo, D. Valenzuela, V. Mäkinen, K. Heljanko, Distributed hybrid-indexing of compressed pan-genomes for scalable and fast sequence alignment, PLoS ONE 16 (2021) 255260.

[19] P. Ferragina, G. Manzini, Indexing compressed text, J. ACM 52 (2005) 552–581. https://doi.org/10.1145/1082036.1082039

[20] A. Amir, G. Benson, M. Farach, Let sleeping files lie: pattern matching in z-compressed files, J. Comput. Syst. Sci. 52 (1006/jcss.1996.0023) (1996) 299–307.

[21] R.S. Boyer, J.S. Moore, A fast string searching algorithm, Commun. ACM 20 (1977) 762–772. https://doi.org/10.1145/359842.359859

[22] L. Chen, S. Lu, J. Ram, Compressed pattern matching in DNA sequences, in: Proceedings. 2004 IEEE Computational Systems Bioinformatics Conference, 2004 IEEE Computational Systems Bioinformatics Conference, IEEE, 2004, pp. 62–68. https://doi.org/10.1109/csb.2004.1332418

[23] G. Navarro, J. Tarhio, Boyer-Moore string matching over Ziv-Lempel compressed text, in: R. Giancarlo, D. Sankoff (Eds.), Combinatorial Pattern Matching, 1007/3-540-45123-4_16 in Berlin Heidelberg; Berlin, Heidelberg, Springer, 2000, pp. 166–180.

[24] G. Navarro, J. Tarhio, LZGrep: a Boyer-Moore string matching tool for Ziv-Lempel compressed text, Softw. Pract. Exp. 35 (2005) 1107–1130. https://doi.org/10.1002/spe.663

[25] Y. Shibata, T. Matsumoto, M. Takeda, A. Shinohara, S. Arikawa, A Boyer-Moore type algorithm for compressed pattern matching, in: R. Giancarlo, D. Sankoff (Eds.), Combinatorial Pattern Matching - 11th Annual Symposium, CPM 2000, Proceedings, Germany, Springer Verlag, 2000, pp. 181–194. https://doi.org/10.1007/3-540-45123-4_17

[26] H. Huo, P. Liu, C. Wang, H. Jiang, J.S. Vitter, Cindex: compressed indexes for fast retrieval of FASTQ files, Bioinformatics 38 (2021) 335–343. https://doi.org/10.1093/bioinformatics/btab655

[27] P. Neamatollahi, M. Hadi, M. Naghibzadeh, Simple and efficient pattern matching algorithms for biological sequences, IEEE Access. 8 (2020) 23838–23846. https://doi.org/10.1109/ACCESS.2020.2969038

[28] M. Zaharia, R.S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M.J. Franklin, et al., Apache spark: a unified engine for big data processing, Commun. ACM 59 (2016) 56–65.

[29] A. Groove, FM Index and Backward Search, 2014. https://github.com/agroves333/fm-index/tree/master.

[30] P. Gage, A new algorithm for data compression, The C Users J. 12 (1994) 23–38.

[31] J. Ziv, A. Lempel, Compression of individual sequences via variable-rate coding, IEEE Trans. Inf. Theory 24 (1978) 530–536. https://doi.org/10.1109/TIT.1978.1055934

[32] G. Navarro, V. Mäkinen, Compressed full-text indexes, ACM Comput. Surv. 39 (2007) 2. https://doi.org/10.1145/1216370.1216372

[33] P.R. Loh, Algorithms for Genomics and Genetics: Compression-Accelerated Search and Admixture Analysis, Technical Report, MIT, 2013.

[34] R. Giancarlo, S.E. Rombo, F. Utro, Compressive biological sequence analysis and archival in the era of high-throughput sequencing technologies, Brief. Bioinf. 15 (2013) 390–406. https://doi.org/10.1093/bib/bbt088

[35] R. Wertenbroek, S. Rubinacci, I. Xenarios, Y. Thoma, O. Delaneau, XSI-A genotype compression tool for compressive genomics in large biobanks, Bioinformatics 38 (2022) 3778–3784. https://doi.org/10.1093/bioinformatics/btac413

[36] S. Grumbach, F. Tahi, Compression of DNA sequences, in: DCC93: Data Compression Conference, IEEE, 1993, pp. 340–350. https://doi.org/10.1109/DCC.1993.253115

[37] F. Willems, Y. Shtarkov, T. Tjalkens, The context-tree weighting method: basic properties, IEEE Trans. Inf. Theory 41 (1995) 653–664. https://doi.org/10.1109/18.382012

[38] T.A. Welch, A technique for high-performance data compression, 1984, pp. 8–19. https://doi.org/10.1109/MC.1984.1659158

[39] J. Ziv, A. Lempel, A universal algorithm for sequential data compression, IEEE Trans. Inf. Theory 23 (1977) 337–343. https://doi.org/10.1109/TIT.1977.1055714

[40] J. Ziv, A. Lempel, Compression of individual sequences via variable-rate coding, IEEE Trans. Inf. Theory 24 (1978) 530–536. https://doi.org/10.1109/TIT.1978.1055934

[41] M. Burrows, D.J. Wheeler, A Block-Sorting Lossless Data Compression Algorithm, 1994.

[42] G. Cattaneo, R. Giancarlo, U.F. Petrillo, G. Roscigno, Mapreduce in computational biology via Hadoop and spark, in: S. Ranganathan, M. Gribskov, K. Nakai, C. Schönbach (Eds.), Encyclopedia of Bioinformatics and Computational Biology, Oxford, Academic Press, 2019, pp. 221–229. https://doi.org/10.1016/B978-0-12-809633-8.20371-3

[43] S. Pal, S. Mondal, G. Das, S. Khatua, Z. Ghosh, Big data in biology: the hope and present-day challenges in it, Gene. Reports 21 (2020) 100869. https://doi.org/10.1016/j.genrep.2020.100869

[44] V. Vineetha, C.L. Biji, A.S. Nair, Spark-MSNA: efficient algorithm on apache spark for aligning multiple similar DNA/RNA sequences with supervised learning, Sci. Rep. 9 (2019) 6631. https://doi.org/10.1038/s41598-019-42966-5

[45] U.F. Petrillo, F. Palini, G. Cattaneo, R. Giancarlo, Alignment-free genomic analysis via a big data spark platform, Bioinformatics 37 (2021) 1658–1665. https://doi.org/10.1093/bioinformatics/btab014

[46] J. Nyström-Persson, G. Keeble-Gagnère, N. Zawad, Compact and evenly distributed k-mer binning for genomic sequences, Bioinformatics 37 (2021) 2563–2569. https://doi.org/10.1093/bioinformatics/btab156

[47] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, Commun. ACM 51 (2008) 107–113. https://doi.org/10.1145/1327452.1327492

[48] U.F. Petrillo, G. Roscigno, G. Cattaneo, R. Giancarlo, Fastdoop: a versatile and efficient library for the input of FASTA and FASTQ files for mapreduce Hadoop bioinformatics applications, Bioinformatics 33 (2017) 1575–1577. https://doi.org/10.1093/bioinformatics/btx010

[49] U.F. Petrillo, G. Roscigno, G. Cattaneo, R. Giancarlo, Informational and linguistic analysis of large genomic sequence collections via efficient Hadoop cluster algorithms, Bioinformatics 34 (1093/bioinformatics/bty018) (2018) 1826–1833.