Distributed Compressive Genomics: Fundamental Pattern Matching Primitives via Spark

Supplementary Material

Lorenzo Di Rocco* Umberto Ferraro Petrillo* Raffaele Giancarlo†
Giuseppe Cattaneo‡

Abstract

1 Spark and Distributed Architectures

Apache Spark [2] is a framework used mainly to support programs with in-memory computing and acyclic data-flow model to be executed on a distributed computing architecture. In simple and intuitive terms, this latter can be described as a set of computing nodes that cooperate in order to solve a problem via local computing and by exchange of messages [4].

1.1 The Spark Architecture

In the standard Spark distributed architecture (see Figure 1, the driver program serves as the central point of control, responsible for defining the distributed behavior of the application being executed. Through the creation of a *SparkContext*, it coordinates the execution of tasks, and manages the overall workflow of the application. The *cluster manager*, on the other hand, manages the resources of the cluster and allocates them to applications.

Each computing node in the cluster, here referred to as *worker node*, runs one or more executor processes, which are responsible for running individual tasks and storing data for the application. *Executors* are used by worker nodes to execute *tasks*; these are the smallest units of work in Spark, represent a single operation applied to a block of data. They are distributed across the worker nodes for parallel execution.

Spark allows data to be cached in memory, which is particularly beneficial for applications that reuse a working set of data across multiple parallel operations (e.g., iterative algorithms). By caching data, Spark reduces the need to recompute it, leading to significant performance improvements.

Spark can be used for applications that require a combination of streaming and batch processing. This capability is in contrast to Hadoop [1], which is primarily suited for batch applications. Spark's ability to cache data and execute tasks in memory makes it highly efficient for iterative algorithms and interactive data analysis.

In addition, Spark is not limited to support only the MapReduce [3] paradigm, although it preserves all the facilities of MapReduce-based frameworks, and it can have the Hadoop as the underlying middleware.

1.2 The Spark API

It provides high-level APIs which support multiple languages (Scala, its native language, Java and Python) and is made of two fundamental logic blocks: a programming model that creates a task dependency graph, and an optimized runtime system which exploits this graph to deploy code and data

*Dipartimento di Scienze Statistiche, Università di Roma - La Sapienza, Rome, 00185, Italy
†Dipartimento di Matematica ed Informatica, Università di Palermo, Palermo, 90133, Italy
‡Dipartimento di Informatica, Università di Salerno, Fisciano (SA), 84084, Italy

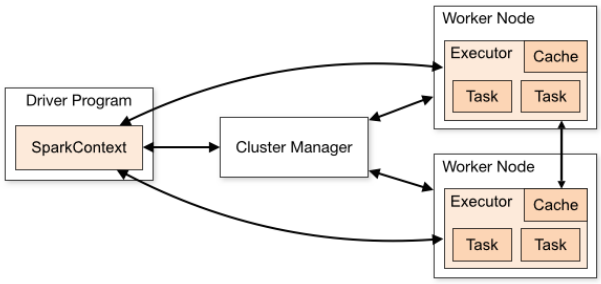

Figure 1: Apache Spark distributed architecture overview.

and schedule work units on the distributed system nodes. At the core of the Spark programming model is the *Resilient Distributed Dataset* (RDD) abstraction, a fault-tolerant, distributed data structure that can be created and manipulated using a rich set of operators: programmers start by defining one or more RDDs through *transformations* of data that originally resides on stable storage or other RDDs.

2 The MapReduce Paradigm

The MapReduce (MR) paradigm, introduced in [3], is a fundamental programming model for processing large datasets in a distributed manner. In Apache Spark, the MR paradigm is employed for developing distributed algorithms that operate on Resilient Distributed Datasets (RDDs). These algorithms are expressed using two types of functions:

- **Map functions:** These functions process each key-value pair in an input RDD and generate a new RDD, which may be empty.
- **Reduce functions:** These functions aggregate values associated with the same key from an input RDD and produce a new RDD, which may also be empty.

By combining map and reduce functions in a sequence, developers can construct distributed algorithms in Apache Spark. These algorithms are fed input RDDs and produce output RDDs.

Reduce functions are essential for many Spark applications, but their implementation can impact performance due to *data shuffling*, a process that gathers pairs with the same key onto the same computing node. To mitigate this performance concern, Apache Spark provides *broadcast variables*, which can hold read-only data. By broadcasting large datasets to each computing node, network traffic can be significantly reduced.

3 *SparkGeco* usage example

We provide here an usage example for our library. Namely, we show in Listing 1 how to compress sequences and search for specific patterns using *SparkGeco*.

In this code, we first set the path of sequences to be compressed from the command-line arguments. Then, a list of queries is defined, each representing a specific pattern to search for within the sequences. Next, the Spark environment is initialized. The input sequences are then read from file and used to create a new object of type `BpeRDD`. Finally, for each query in the list, the code performs a search and aggregates the results. The total number of occurrences found is then printed.

Notice that in order to switch to another compression technique, the developer has only to change the `BpeRDD` with the class implementing the compression technique of choice. Currently, *SparkGeco* provides the following specialized classes: `BpeRDD`, `ChenRDD`, `LzwRDD` and `FmRDD`.

```
]
public class Main {
    public static void main(String[] args) {

        // Path of the sequences to compress
        String input_file = args[0];

        // Queries
        List<String> Ps = new ArrayList<>();
        Ps.add("TTCCTTAGGAAAAGGGGAAGACCACCAATC");
        Ps.add("AGAGGATTATGTACATCAGCACAGGATGCA");
        Ps.add("GAAGGACTTAGGGGAGTCCTCATGAAAAAT");
        Ps.add("GTATTAGTACAGTAGAGCCTTCACCGGCAT");
        Ps.add("TCTGTTTATTAAGTTATTTCTACAGCAAAA");
        Ps.add("CGATCATATGCAGATCCGCAGTGCGCGGTA");

        SparkConf conf = new SparkConf().setMaster("yarn");
        JavaSparkContext sc;

        BpeRDD sequence = BpeRDD.read(input_file, sc);

        long found = 0;

        for (String P : Ps)
            found += sequence
                .search(P)
                .aggregate(0L, (v, arr) -> arr.length + v, Long::sum);

        System.out.println("Found: " + found);
    }
}
```

Listing 1: Java code for compressing a collection of input genomic sequences and searching for specific patterns using the `BpeRDD` class available with our library

Dataset	Plain	CBM-2bit	CBM-bp	CBM-LZW	FM-Index
PL1	10.8	2.8	3.8	6.4	2.9
PL2	17.4	4.7	6.3	10.3	5.2
PL3	18.6	4.8	6.8	10.7	5.4
SARSCOV	197.9	49.7	67.8	113.2	50.1

Table 1: Memory savings achieved by compressing the datasets used in this study when using the distributed compressed data structures available with *SparkGeco*. The test is performed with the use of one computing unit. For each option we report the size, in Gigabytes, of the corresponding compressed data structure and the compression option, as well as the original size.

References

[1] Apache Software Foundation. Apache Hadoop. (Available from: `http://hadoop.apache.org`), 2006.

[2] Apache Software Foundation. Apache Spark. (Available from: `http://spark.apache.org`), 2016.

[3] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[4] Nancy A. Lynch. *Distributed algorithms*. Morgan Kaufmann, 1996.