PASTASpark: multiple sequence alignment meets Big Data

José M. Abuín, Tomás F. Pena and Juan C. Pichel

Supplementary Material

1 Apache Spark

PASTASpark allows to execute PASTA on a distributed memory cluster using Apache Spark [1]. Spark is a cluster computing framework designed to support iterative jobs and interactive analytics, which includes automatic parallelization and task distribution as well as fault tolerance. Spark was originally developed at the University of California, Berkeley, becoming a Top-Level Apache Project in 2014. It uses a master/slave architecture with one central coordinator, named Spark Driver, and many distributed workers or Spark Executors, as Figure 1 illustrates. Spark applications run as independent sets of processes on a cluster coordinated by an object called SparkContext, which is a defined object in the Driver. The SparkContext can connect to several types of cluster managers (either Spark’s own standalone cluster manager, Mesos [2] or YARN [3]). Through the cluster manager, Spark acquires Executors on the cluster nodes, which are processes that run computations and store data for a Spark user program.

The Spark Driver is responsible for converting an user program into units of physical execution called tasks. A Spark program implicitly creates a logical directed acyclic graph (DAG) of operations, which the Driver converts into a physical execution plan consisting in a set of stages, where each stage is composed of multiple tasks. This plan is then optimized, for example, merging several transformations, and individual tasks are bundled up and prepared to be sent to the cluster. The Spark Driver tries to schedule each task in an appropriate location based on the data placement.

On the other hand, Spark Executors run the tasks that make up the application into a Spark Container (a Java Virtual Machine), returning results to the Driver. Besides, when tasks execute, their associated data can be cached in the Executors local memory. The Driver tracks the location of cached data and uses it to schedule future tasks that access that data.

Spark handle fault tolerance by introducing the concept of Resilient Distributed Datasets (RDDs) [4]. An RDD represents a read-only collection of objects partitioned across the cluster nodes that can be rebuilt if a partition is lost. Users can explicitly
cache an RDD in memory across machines and reuse it in multiple MapReduce-like parallel operations. By using RDDs, programmers can perform iterative operations on their data without writing intermediary results to disk. In this way, Spark is well-suited, for example, to machine learning algorithms.

RDDs can be created from a collection of objects (e.g., a list or set) or by loading an external dataset. Note that Spark can process input data from HDFS, HBase [5], Cassandra [6], Hive [7], Parquet [8], and any Hadoop InputFormat. On created RDDs, Spark supports two types of parallel operations: transformations and actions. Transformations are operations on RDDs that return a new RDD, such as `map`, `filter`, `join`, `groupByKey`, etc. The resulting RDD will be stored in memory by default, but Spark also supports the option of writing RDDs to disk whenever necessary. On the other hand, actions are operations that kick off a computation, returning a result to the Driver program or writing it to storage. Examples of actions are `collect`, `count`, `take`, etc. Note that transformations on RDDs are lazily evaluated, meaning that Spark will not begin to execute until it sees an action.

When running on a cluster, Spark allows two deploy modes namely `client` and `cluster` mode. In client mode, the Driver program runs on the same machine where it is itself being invoked, usually the user workstation. In cluster mode, the Driver will be shipped to execute on a worker node in the cluster, freeing the user computer to do other jobs.

Apache Spark provides Python, Scala, and R interactive shells, which let the user interact with data that is distributed on disk or in memory across many machines. Apart from running interactively, Spark can also be linked into applications in either Java, Scala, or Python. As the original PASTA is written in Python, we have decided to use the Spark Python API (also known as PySpark) to implement PASTASpark.
1.1 Big Data and Bioinformatics

Due to the fast expansion of Big Data technologies, we can find several bioinformatics works that exploit the capabilities of different Big Data frameworks such as Apache Hadoop or Apache Spark on distributed memory clusters. However, to the best of our knowledge, none of those works deal with Multiple Sequence Alignment (MSA), being most of them focused on the short read alignment problem. Some relevant examples are the following:

- **SparkBWA** [9]: this tool exploits the capabilities of the Apache Spark engine to boost the performance of one of the most widely adopted aligner, the Burrows-Wheeler Aligner (BWA). The design of SparkBWA uses two independent software layers in such a way that no modifications to the original BWA source code are required, which assures its compatibility with any BWA version (future or legacy).

- **BigBWA** [10]: BigBWA is a tool that uses Apache Hadoop to boost the performance of the Burrows-Wheeler aligner (BWA). BigBWA is fault tolerant and it does not require any modification of the original BWA source code.

- **SEAL** [11]: SEAL is a scalable tool that also uses Apache Hadoop for short read pair mapping and duplicate removal. It computes mappings that are consistent with those produced by BWA and removes duplicates according to the same criteria employed by Picard’s MarkDuplicates.

- **Halvade** [12]: Halvade is a framework that enables sequencing pipelines to be executed in parallel on a multi-node and/or multi-core Hadoop compute infrastructure in a highly efficient manner. As an example, a DNA sequencing analysis pipeline for variant calling has been implemented according to the GATK Best Practices recommendations, supporting both whole genome and whole exome sequencing.

Additionally, **MSAProbs-MPI** [13] is a distributed-memory parallel version of the multithreaded MSAProbs [14] tool that reduce runtimes by exploiting the compute capabilities of distributed memory clusters by using the MPI [15] library. MSAProbs is a state-of-the-art protein multiple sequence alignment tool based on hidden Markov models. It can achieve high alignment accuracy at the expense of relatively long runtimes for large-scale input datasets. Note that unlike traditional parallel programming paradigms as MPI, code developing in Spark is largely simplified with characteristics as the automatic input splitting, task scheduling or fault tolerance mechanism.

2 PASTASpark in more detail

Next we describe the design and implementation of PASTASpark, but first, some background on the original PASTA application is provided.
2.1 PASTA Workflow

As it is described in [16], PASTA uses an iterative strategy. The first iteration begins with a starting tree, and subsequent iterations use as input the tree estimated in the previous one. In each step, the guide tree is used to divide the sequence set $S$ into smaller subsets, and to build a spanning tree with these subsets as nodes. Then multiple sequence alignments (MSAs) for all the sequence subsets are independently estimated. After that, pairs of MSAs corresponding to subset that are adjacent in the spanning tree are aligned together. The resulting collection of MSAs overlap each other and are compatible where they overlap. These properties enable PASTA to merge these overlapping MSAs using transitivity and generate an MSA on the entire set of sequences. Finally, a maximum likelihood (ML) tree is estimated on the final alignment.

The starting tree can be provided by the user as an input parameter or it can be computed from an alignment $A$ of a random subset $X$ of 100 sequences from $S$. PASTA uses HMMER [17, 18] to compute an Hidden Markov Model on $A$, and to align all sequences in $S - X$ one by one to $A$. Then, an ML tree on this alignment is constructed using FastTree-2 [19].

Considering this starting tree as input, the PASTA workflow consist in an iterative method that we can summarize in four phases (see Figure 2):

**Phase 1 (P1):**
- The sequence set $S$ is divided into disjoint sets $S_1, \ldots, S_m$, each with at most 200 sequences, using the current guided tree and the centroid decomposition technique in SATé-II [20]. In this technique, if the tree has at most 200 leaves, the set of sequences is returned; otherwise, an edge in the tree that splits the set of leaves into roughly equal sizes is found and removed from the tree. Then, the algorithm recurses on each subtree.
A spanning tree $T^*$ on the subsets is obtained. To do this, all leaves are labeled by their subset and, for every node $v$ in the guide tree that is on a path between two leaves that both belong to $S_i$, it is labeled by $S_i$. If after this process there are unlabeled nodes, labels are propagated from nodes to unlabeled neighbors (breaking ties by using the closest neighbor according to branch lengths in the guide tree) until all nodes are labeled. Then, edges edges that have the same label at the endpoints are collapsed.

**Phase 2 (P2):**

- In this step, MSAs on each $S_i$ are obtained using an existing MSA tool. Each such alignment is denoted as a type 1 subalignment. By default, MAFFT [21] with the L-INS-i settings is used. It is based on the iterative refinement method incorporating local pairwise alignment information.

**Phase 3 (P3):**

- Every node in $T^*$ is labeled by an alignment subset for which we have a type 1 subalignment from previous step. For every edge $(v, w)$ in $T^*$, OPAL [22] is employed to align the type 1 subalignments at $v$ and $w$; this produces a new set of alignments, each containing at most $2k$ sequences, which are called type 2 subalignments. The merger technique used to compute type 2 subalignments is required not to change the alignments on the type 1 subalignments; therefore, type 2 subalignments induce the type 1 subalignments computed in the phase 2 and are all compatible with each other.

- The final alignment is computed through a sequence of pairwise mergers using transitivity, as described in [16]. Using the concept of transitivity merger, the spanning tree is used to merge all the type 2 subalignments through a sequence of pairwise transitivity mergers into a multiple sequence alignment on the entire set of sequences. The final transitivity merger produces an alignment that includes all the sequences.

**Phase 4 (P4):**

- If an additional iteration (or a tree on the alignment) is desired, FastTree-2 is used to estimate a maximum likelihood tree on the MSA produced in the previous phase and the process is repeated.

As it was stated in [16], P1 and P3 require a small amount of time, so their parallelization/optimization is not necessary. The most expensive phase in terms of computational time is P2 (subsets alignment), which may imply more than 70% of the total execution time. Finally, P4 (tree estimation) requires more time than P1 and P3 but it is much faster than P2.

P2 in PASTA is parallelized using a multithreaded approach. By default, PASTA set the number of threads to be used equal to the number of available cores in the shared
memory system, although the user can also specify a particular value. Figure 3 shows an example of how P2 is implemented in PASTA considering five subsequences and a quad-core computer. First, the subsequences are stored in the hard disk because the input to the aligner (MAFFT in this case) can only be a regular file. Afterwards, each Python thread forks a new child process, which execute MAFFT using Python’s class `subprocess.Popen`. These child processes run in parallel in the four cores, and their outputs are again stored to disk to be used in the next phase (P3). Note that in the example of Figure 3 since there are five subsequences and only four cores, one of the cores executes sequentially two MAFFT subprocesses.

The PASTA parallelization strategy has three main drawbacks:

1. Both the Python reference interpreter (CPython) and the alternative interpreter that offers the fastest single-threaded performance for pure Python code (PyPy), use a Global Interpreter Lock (GIL) to avoid various problems that arise when using threading models that implicitly allowing concurrent access to objects from multiple threads of execution. This makes Python unsuitable for using shared memory threading to exploit multiple cores on a single machine. To overcome this problem, the adopted solution in PASTA was to lunch a child process in each thread using the Python multiprocessing library. These processes can run in parallel but with the extra overhead of creating them.

2. The second drawback is that, with the adopted approach, each subprocess runs on a single core and call the external alignment program (MAFFT by default) with one of the input subsequences. MAFFT supports multithreading with OpenMP, but, as each subsequence is aligned in a single core, MAFFT parallelism cannot be harnessed.

3. The third drawback and the most important one is that PASTA can only be exe-
cuted on shared memory computers, which limits its scalability to a few number of cores. In this way, PASTA is limited to process small input datasets because the memory and time requirements of large datasets exceed the computing power of any shared memory system.

2.2 PASTASpark

The objective of PASTASpark is to reduce the execution time of PASTA using Apache Spark as engine to run it on a distributed memory cluster. Current version of PASTASpark focused only in the subset alignment step (P2) of the PASTA workflow. So, other steps are executed by the Spark Driver program.

Figure 4 shows how P2 is modified in PASTASpark. This example considers that we have again five subsequences and Spark is running on a distributed memory cluster with two dual core nodes. If the code is launched using spark_submit, the output of the first phase is not spilled to disk but a pair RDD is created. A pair RDD is an RDD where the elements are (key, value) pairs. In this case, the key is a job identifier and the value is one of the subsequences created in P1 and which resides in the Driver memory. Afterwards, a map() transformation is applied to the RDD in such a way that MAFFT is executed concurrently on each of the subsequences. We must highlight that each piece of the RDD is stored in the local disk of a worker node (not in the Driver disk), and the writing is performed in parallel reducing the I/O cost with respect to PASTA.

The output of the map transformation is a new RDD with the aligned subsequences (stored in the local memory of the worker nodes). As transformations on RDDs are lazily evaluated, Spark will not start to execute the map until it detects an action. In this case, a collect() action is used to retrieve the output RDD as a list, which is finally stored in the local disk of the Driver. From this point, the rest of the PASTA phases are executed in the Driver.
<table>
<thead>
<tr>
<th>Tag</th>
<th>Name</th>
<th>No. of sequences</th>
<th>Avg. sequence length</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1</td>
<td>16S.B.ALL</td>
<td>27,643</td>
<td>1,371.9</td>
<td>184.1 MB</td>
</tr>
<tr>
<td>D2</td>
<td>50k RNASim</td>
<td>50,000</td>
<td>1,556</td>
<td>591.8 MB</td>
</tr>
<tr>
<td>D3</td>
<td>200k RNASim</td>
<td>200,000</td>
<td>1,556</td>
<td>3.4 GB</td>
</tr>
</tbody>
</table>

Table 1: Datasets used for the experimental evaluation of PASTASpark.

3 Experimental setup

3.1 Input Datasets

In the original PASTA paper \[16\] several input datasets were used. In particular, datasets are from the million-sequence RNASim \[24\] with subsamples of 10k, 20k, 50k, 100k and 200k sequences, and biological datasets from the comparative ribosomal website (CRW) \[25\]. In order to test PASTASpark datasets from both groups have been chosen (see Table 1 for details).

According to the original PASTA paper and regarding the RNASim datasets, PASTA was able to complete two iterations with the 100k subsample and only one iteration with the 200k subsample. However, it completes the three iterations that PASTA runs by default with the 50k sample. For this reason we have chosen as illustrative examples the 50K and 200k datasets (\(D_2\) and \(D_3\) respectively). \(D_1\) was selected since it is the biggest dataset among the CRW examples.

3.2 Computational Platforms

We have used two different clusters in the performance tests. The first one is a Big Data cluster installed at CESGA\[1\] (Galicia Supercomputing Center in Spain). This cluster has 12 computing nodes with 54.5 GB of RAM memory each one, and 19 cores available for Spark containers. However, in our case the maximum number of cores that can be used per container is 8 due to some restrictions set by the system administrators. CPUs are Intel Xeon CPU E5-2620-v3 at 2.40GHz. Hadoop, Spark and Java versions are 2.7.1 (HDP), 1.6.1, and 1.8, respectively.

The second platform is a 9-node AWS (Amazon Web Services) EC2 cluster\[2\] with 16 cores (Intel Xeon E5-2670 at 2.5GHz CPUs) and 122 GB of RAM memory per node. In particular, each computing node corresponds to a r3.4xlarge EC2 instance. In this case, Hadoop, Spark and Java versions are 2.7.2, 1.6.2, and 1.8 respectively.

It should be taken into account that PASTASpark runs P1, P3 and P4 in the Spark Driver, while P2 is performed by the Spark Executors. In this way, depending on the input dataset some hardware parameters should be set: the number of cores and RAM memory used by the Driver, the memory assigned to the Executors, etc. For example, the

---

[1] www.cesga.es
Driver considering D2 uses 20 GB, while for D1 15 GB is an adequate value. Regarding the Executors, 5 GB are enough in both cases.

4 Future work

The first topic to be addressed in the future work is the problem related to the parallelization efficiency of FastTree-2 tool (P4). As the number of cores used in the computation increases, P4 becomes the major bottleneck in the performance of PASTASpark. In this way, it is necessary to completely redesign the parallelization strategy of FastTree-2 or Fast-Tree-2 should be replaced by a different tool with a better scalability. The goal is also to integrate P4 in the Spark framework. In this way, we believe that the scalability of PASTASpark will improve noticeably.

Acknowledgements

We would like to thank CESGA (Galicia Supercomputing Center, Spain) for granting us access to their supercomputing facilities, and the Computer Science Department of the University of Illinois at Urbana-Champaign for helping us with the PASTA code.

References


