A distributed sorting framework for SAM files using Spark

Pramod Rao
Dheeraja Rajreddygari

INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY
HYDERABAD

May 15, 2023

1 Hardware configuration

We used a cluster with one master driver node (which does not take part in any computation tasks) and 4 worker nodes, each with a Intel Xeon Gold 6226R CPU and over 180 GB of RAM. Three of the workers have two NUMA nodes while one worker has 4 NUMA nodes. The cluster has a network bandwidth of 950 Mbps.

Table 1: Spark submit configuration

Property	**Value**
Executors	2 per node (8 in total)
Cores	4 per executor (32 in total)
Memory	8GB per executor (64GB in total)

2 Samtools baseline

We use a total of 32 cores and 64 GB of memory as mentioned in the configuration mentioned in the previous section. So, we obtain the baseline to our experiments by timing Samtools run with varying number of threads such that the total memory remains 64 GB. We benchmark the experiments on two datasets, namely *Data1.sam* of size 6.5 GB and *Data4.sam* of size 20 GB.

We obtain the following result:

Table 2: Samtools performance

File	**Size**	**Time taken**
Data1.sam	6.5 GB	82 s
Data4.sam	20 GB	249 s

3 Experiments

In the following section we describe the experiments that we performed and their results. We use the following subroutine for parsing the input file throughout the experiments: **PARSE:**

1. Read the input SAM file
2. Extract the SAM records by applying a filter that discards the header lines
3. Extract just the position as a new column

The above routine, returns a DF/RDD with each entry being a tuple of the position (Int) and the entire SAM record (String). In the case of an RDD, this results in a "PairedRDD". The rationale behind this instead of opting to read the SAM file and store in a native data-structure (such as an array of maps) was that doing the latter resulted in a huge performance drop owing to the complex data-structures. We separately benchmark DFs and RDDs for each experiment.

Each of the experiments described below, we write the RDD/DF to disk using the `saveAsTextFiile` method. This saves the DF/RDD on HDFS and creates one file per partition. This output can be combined to generate either one single global text file, or one file per node which contains all entries pertaining to a section of the genome. In order to generate a single output file, we use the `getmerge`, which takes all the files on HDFS, concatenates them and puts it on local storage. In order to generate a single file per node, we have written a script that takes as arguments the total number of partitons, total number of nodes, and ID number of the current node to combine the relevant section and writes it to that node's local disk.

3.1 Experiment 0

We use experiment 0 as the Spark baseline. In this experiment, we simply perform a whole sorting on the input after parsing it. The exact methods called vary depending on if we're working with DFs or RDDs.

3.2 Experiment 0.5

This experiment focuses on minimizing the data transfer between nodes. The idea here is that it is redundant to copy over the entire SAM record in memory when we only use the position for sorting. We can store a reference to the record and only copy it after all the computation has been done. To study the extent of benefit from doing this, we created a DF which contains the position and a dummy 4-byte string that acts as the place holder for the pointer.

We found that the shuffle read/write size was reduced by over 90X (from 14GB to 155MB for a 30 GB file). This reduced the time taken by the last stage (where the shuffle read happens). However, the improvement in overall sorting time was not significant, especially since we must then use the reference to fetch the original record which may be on a different node.

3.3 Experiment 1

In this experiment, we focus on the effects of manually re-partitioning the data before performing a partition-wise sorting on it. Formally, in this experiment, we parse the input, repartition it based on ranges of the read position and then perform partition-wise sorting on it.

In the case of RDDs we perform a bucketing partition based on the maximum position value (we assume it is equal to the size of the genome, which is known). We use the method `rdd.repartitionAndSortWithinPartitions()` to define a custom partitioner that splits the range of position values into equal ranges.

In the case of DFs however, supplying a custom re-partitioning function isn't possible. So, we use the inbuilt `df.repartitionByRange()` function, which first samples the data to determine the range boundaries for each partition. Here, the ranges themselves are not equal, but each range contains a similar number of records.

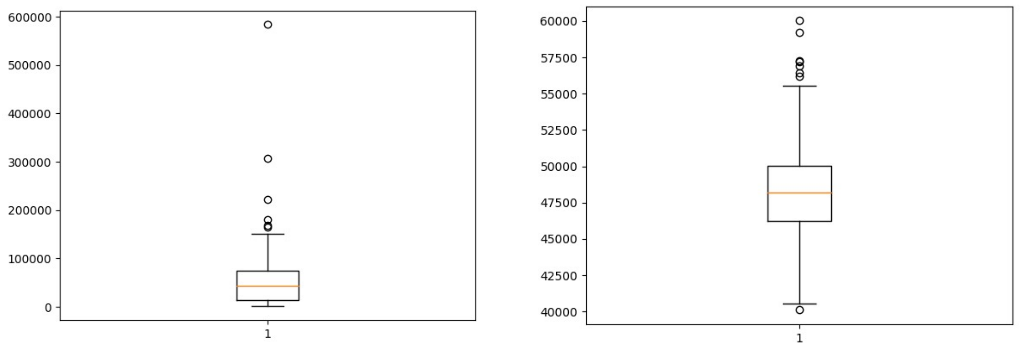

Figure 1: Distribution of partition sizes using a custom partitioner on RDD(left) and `df.partitionByRange()` (right)

We had expected that the reads would be distributed fairly well across the genome, which would mean that splitting the genome into equally sized ranges should result in sufficient load balancing. However, we found that this was not the case, which resulted in the partition sizes being rather skewed for the RDDs (Fig 1). On the other hand, DF partitions were well-balanced, but the sampling step incurred some overhead.

3.4 Final Experiment

In the final experiment, we changed the way the records are sorted. Previously the records were sorted solely based on the position of the entry. Since we wanted the resulting files to agree with the output of Samtools, we decided to perform sorting in a similar manner. Formally, we sorted the records first on the basis of their RName and then within a particular RName we sorted them in the increasing order of their position.

To achieve this, we first parse the header and non-header lines into RDDs by using the parsing subroutine. Then, we collect the header lines and process them sequentially. This is acceptable since the number of header lines is extremely small compared to number of records in a SAM file. As we iterate through the headers, in a python dictionary, we store the name and the current cumulative length. Note that the SAM format specification allows for unmapped reads in the SAM file which have the RName set to '*'. Similar to Samtools, we add an entry for such entries at the end of our dictionary which makes the records to be placed at the end of the output file.

With the dictionary built, we now map the record lines into a pair RDD where the key is now modified as compared to the previous experiments. The key is now a sum of the record position as well as the corresponding cumulative prefix length obtained from the dictionary corresponding to its RName. Once the RDD is built, the rest of the sorting routine is similar to the previous experiments.

3.5 Results

The below table summarizes the bench-marking results obtained. In order to force spark to perform a computation, we call the method `df.rdd.count()` in the case of DFs and `rdd.`

`count()` in the case of RDDs.

Table 3: Time taken by each experiment in seconds

Experiment	Data1.sam	Data4.sam
exp0_df	36	98
exp0_rdd	44	113
exp1_df	40	91
exp1_rdd	21	68

The Spark UI shows that the sampling in *exp0_rdd* launches additional jobs which are absent in case of DF. This contributed to RDD sorting being slower than DF.This is a known side-effect of spark launching a job to sample the RDD upon calling the `rdd.sortByKey()` method as can be found in the issue here (which was closed with the resolution “won’t fix”).

4 Further findings

4.1 Speed-up with caching

We experimented with caching the RDD/DF just before the sorting step. Doing so, we saw improvements in the execution time of the experiments as follows:

Table 4: Time taken by each experiment (with caching) in seconds

Experiment	Data1.sam	Data4.sam	Improvement factor
exp0_df	29	70	1.24 - 1.4
exp0_rdd	28	71	1.6
exp1_df	30	70	1.3
exp1_rdd	26	71	0.8 - 0.96

We see an improvement in the run-time of every experiment other than `exp1_rdd` through caching.

4.2 Experiments with storage and locality

Across all experiments we found that the compute times of tasks within all executors were skewed (Fig 2). This was surprising since each partition at this stage was roughly equal in size (128 MiB). We discovered that the locality level of each task was set to *ANY*, which essentially gives no importance to locality. The locality preference of spark must be adjusted to exploit the benefits of data locality.

4.3 Data serialization

We implemented some of the earlier experiments in Scala to see the performance parallels and noticed that it performed poorly. This was a result of poor caching of the bloated JVM

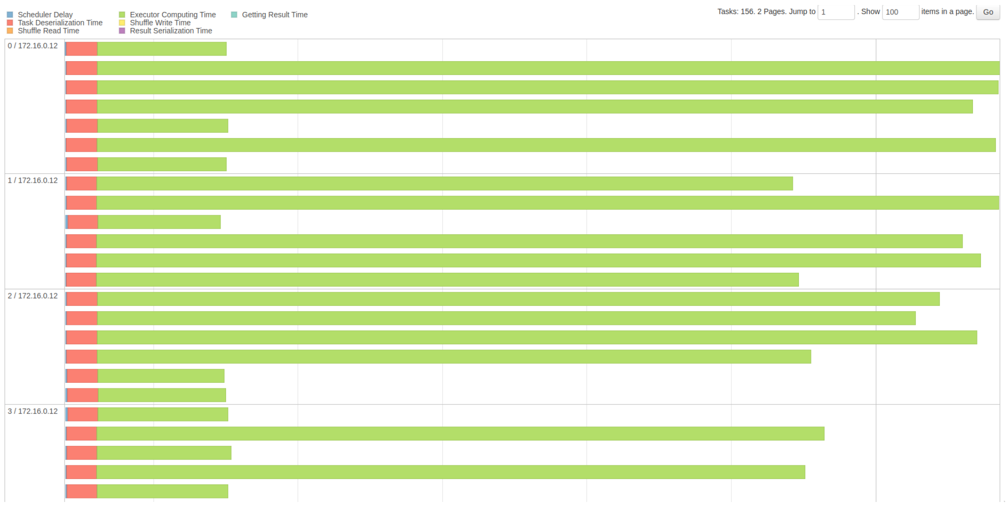

Figure 2: Skew in execution times for practically equal sized tasks

objects (as noted on the Spark page on tuning). While tuning some of the compression and serialization parameters, the execution time turned out equal to the PySpark parallels. After experimenting with the Kryo Serializer, we arrived at the conclusion that there is not muhc improvement that can be achieved. This is because, Spark inherently uses the Kryo Serializer for native data-structures like Scala Tuples for example which we use in our code. If however we use more complex data-structures in the future, then there is some scope for improvement by switching out the default Serializer.

4.4 Disabling AQE

Adaptive Query Execution is a spark optimization that uses runtime statistics to generate an adaptive execution plan. AQE was introduced in Spark 3.0 and enabled by default since Apache Spark 3.2.0 Although this optimization doesn't make any difference in our program, it can improve some spark queries. However, as an effect of this, the order of records within a partition is not guaranteed in case a spill occurs. In case we use a version of spark that enables this by default, we need to disable AQE in our program so our sort order is not broken in case of spill. It can be done by adding `--conf spark.sql.adaptive.enabled=false` to the end of the spark-submit command.