Design notes for a distributed sorting application

Chengxin Ma

February 4, 2020

1 Introduction

This report describes the overall design of a distributed sorting application (Section 2), some key decisions in implementation (Section 3), and a test run (including setup, result, and analysis) to see the performance of the application (Section 4). In addition, it contains the next steps of the work and predicted results (Section 5).

2 Design

2.1 Test data

The final goal of designing this application is to integrate it into a genomic data process pipeline, where data in the SAM format [1] is sorted.

From the perspective of sorting, the most interesting fields are `RNAME` (the name of the references sequence) and `POS` (position). They together determine the order in which records are sorted.

Thus, to simplify the prototyping work, we design a data structure with three fields: `GROUP`, `SEQ`, and `DATA`. Each record belongs to a group and has a sequence number in that group. Its data is placed in the `DATA` field.

Here is an example input file: *Input records on Node 0*, and here is an example file containing expected sorted records: *Expected sorted records on Node 0*. Note that we do not necessarily need to write the output into files. It is only for experimenting purpose. After integration we could use in-memory storage instead.

2.2 Functional decomposition and phases

The following functional components must be implemented to complete the application.

- sorter
- sender and receiver

[1] `https://en.wikipedia.org/wiki/SAM_(file_format)#Format`

- partitioner and merger
- storage (optional)

The *sorter* is responsible for sorting the data in our desired order: records with smaller Group IDs are placed before those with large Group IDs. If the Group IDs of two records are the same, the secondary criteria is the sequence number.

The *sender* and *receiver* are responsible for sending data to destination nodes and receiving data from source node respectively.

The responsibility of the *partitioner* is to partition the data into different groups that would be sent to different destinations, while the *merger* is to merge the partitioned data to a complete set.

Storage is needed when we want to temporarily store the data before further processing.

Based on if data is shuffled or not during execution, we can divide the overall execution time into three phases.

The first phase is the *partitioning* phase. In this phase, data is partitioned to subsets according to the final destination where it is going to be sent to. Beginning of this phase is marked by the earliest start time on all the nodes, while the end of this phase is marked by the time when all nodes have finished partitioning and serializing the input data.

The second phase is the *communication* phase. In this phase, data is shuffled to destination nodes in parallel. As soon as one node has started sending data out, it is the time that marks the beginning of the communication phase. End of this phase is marked by the time when all nodes have received all data belonging to them.

The third phase is the *sorting* phase. In this phase, data from all other nodes and on the local node is merged and sorted. Beginning and end of this phase is marked by the earliest start time of deserialization on all nodes and the latest finish time of sorting on all nodes respectively.

3 Implementation

3.1 Implementation choices

We choose `pandas`, a Python library for data manipulation and analysis, for the partitioning and sorting phase of the application. For the communication phase, we have two alternatives: Python's `socket` module and `Apache Arrow Flight`. The interface to the partitioning phase and sorting phase is thus also different. We use `pickle` to serialize/deserialize the data before/after the communication phase in the `socket` approach, while `Plasma` and `Arrow RecordBatch` are used when we use `Apache Arrow Flight` for communication.

To achieve parallelism in communication, in the `socket` approach, we use the `multiprocessing` module (instead of `threading` which cannot take advantage of multiple cores due to the existence of `CPython`'s Global Interpreter

Lock), while for the `Apache Arrow Flight` approach we use `Arrow`'s internal `ThreadPool`.

Can experiment result show if they are correctly implemented?

3.2 Flowchart

Figure 1 shows the overall flowchart of the distributed sorting application.

4 Test run

4.1 Setup

To see the performance of the application, a test run has been performed. 4 nodes on Cartesius (`tcn798, tcn799, tcn804, tcn807`, we refer them as from Node 0 to 3 hereafter) were allocated for this test run. The data for the test run contains records of 40 groups (from `GROUP0` to `GROUP39`), each having 2 million records (i.e. the total number of records is 80 million). Each node had 20 million records in random order before the application ran, and we expected that when the application finishes, Node 0 stores records from `GROUP0` to `GROUP9`, Node 1 stores records from `GROUP10` to `GROUP19`, Node 2 stores records from `GROUP20` to `GROUP29`,and Node 3 stores records from `GROUP30` to `GROUP39`, in the ascending order.

The size of the input and output files on each node is around 540 MB.

The test run was started one by one manually, in the order of from Node 0 to 3.

4.2 Results

The tables in Appendix A show the original timestamp of tasks on these four nodes.

To have a more straightforward view, we have adjusted the original timestamp by setting the starting time to 0:00:00.

As can be seen in Table 1 and 2, both implementations can complete the tasks with in 1 minute. The `Flight` implementation (53 seconds) is 6 seconds faster than the `socket` implementation (59 seconds).

4.3 Discussion: Where does the difference come from?

The difference comes from (de)serialization and communication. In the `socket` approach, serializing `DataFrame` to `pickle` files took 13.75 seconds on average, while in the `Flight` approach, putting partitioned `DataFrame` to `Plasma` only took 3.75 second on average. Unpickling `pickle` files to `DataFrame` took 3 seconds, while converting data from `Plasma` to `DataFrame` took 8 seconds. Therefore, using `Arrow` saves us 5 seconds in (de)serialization from using `socket`.

The above average time of (de)serialization was calculated by $\frac{1}{4}\sum_{i=0}^{3}(finish\_time_i - start\_time_i)$. We cannot apply the same formula for calculating communication

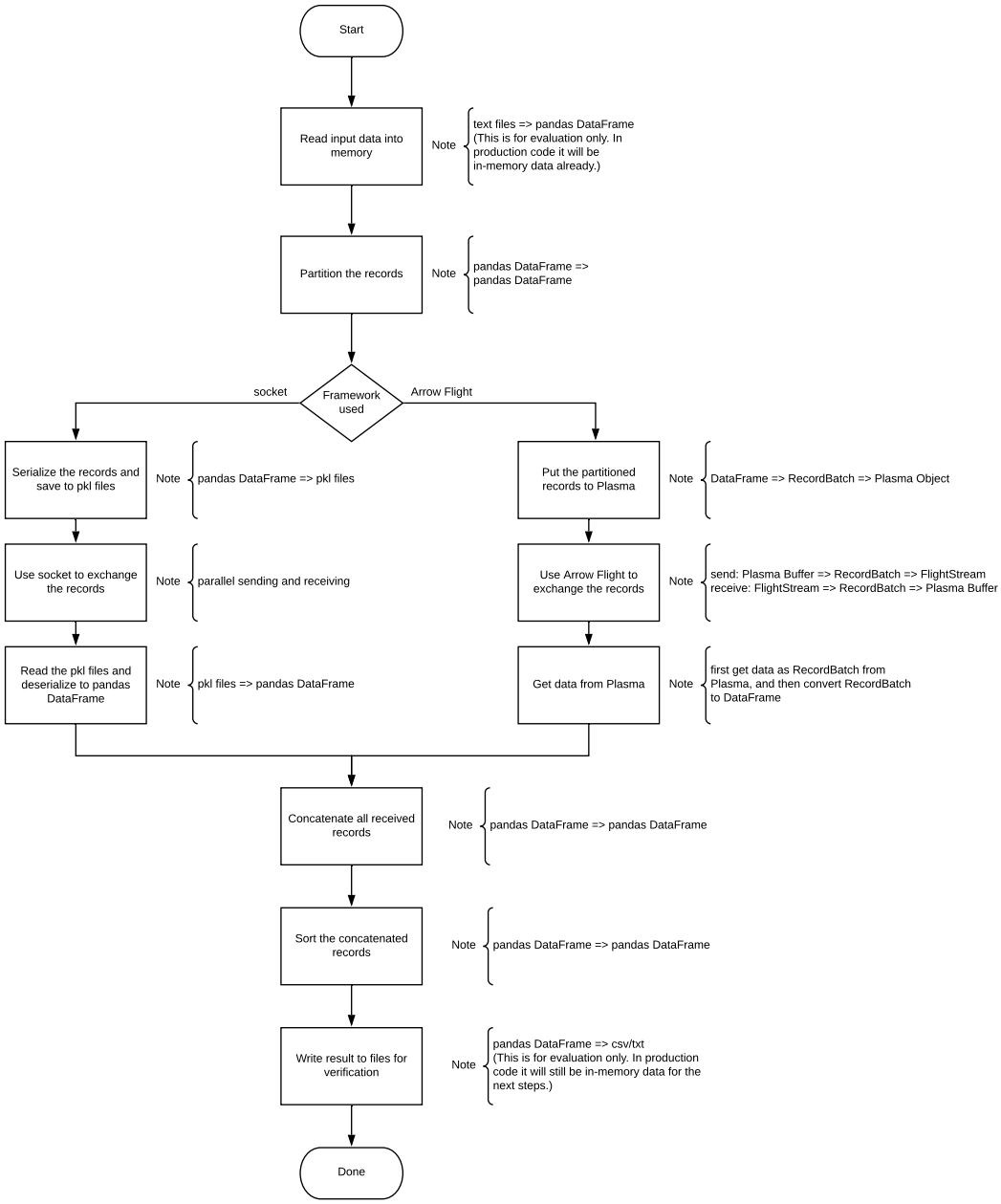

Figure 1: Flowchart of the application, with two alternatives for the communication phase

	tcn798	**tcn799**	**tcn804**	**tcn807**
started partitioning the records	0:00:00	0:00:00	0:00:00	0:00:02
finished partitioning the records, started serializing the data to pkl files	0:00:15	0:00:14	0:00:15	0:00:18
finished serializing the data to pkl files, started sending to destination nodes	0:00:29	0:00:26	0:00:29	0:00:33
received last file	0:00:35	0:00:35	0:00:35	0:00:35
started unpickling the records from pkl files	0:00:36	0:00:36	0:00:36	0:00:36
finished unpickling the records from pkl files, started concatenating	0:00:39	0:00:39	0:00:39	0:00:39
finished concatenating, started sorting the records	0:00:43	0:00:43	0:00:43	0:00:43
finished sorting the records	0:00:58	0:00:59	0:00:58	0:00:59

Table 1: Adjusted timestamp of the distributed sorting application (using socket) on 4 nodes

	tcn798	**tcn799**	**tcn804**	**tcn807**
started partitioning the records	0:00:01	0:00:01	0:00:00	0:00:00
finished partitioning the records, started putting them to Plasma	0:00:16	0:00:20	0:00:15	0:00:15
finished putting partitioned records to Plasma	0:00:21	0:00:23	0:00:18	0:00:19
send-to-dest started	0:00:23	0:00:26	0:00:20	0:00:22
received last piece of data	0:00:26	0:00:26	0:00:26	0:00:26
started retrieving the records from Plasma	0:00:27	0:00:27	0:00:27	0:00:27
finished retrieving the records from Plasma, started converting to DataFrame	0:00:27	0:00:27	0:00:27	0:00:27
finished converting to DataFrame, started concatenating	0:00:35	0:00:35	0:00:35	0:00:35
finished concatenating, started sorting the records	0:00:38	0:00:37	0:00:37	0:00:38
finished sorting the records	0:00:53	0:00:52	0:00:52	0:00:53

Table 2: Adjusted timestamp of the distributed sorting application (using Apache Arrow Flight) on 4 nodes

time, since for one sending from one node there are multiple receiving on multiple nodes. We thus use the following formula to calculate the communication time: $\frac{1}{4}\sum_{i=0}^{3}(max(finish\_time_{ij}) - start\_time_i)$, where $finish\_time_{ij}$ denotes the time when node j finished receiving from node i.

In the `socket` implementation, these four nodes sent out data at `13:08:15`, `13:08:18`, `13:08:18`, and `13:08:22`. The receiver logs (see Figure 4 as an example) show that these nodes received first batch of files at `13:08:16`, `13:08:17`, or `13:08:18`, the second and third batch of files at `13:08:22`, and the last batch at `13:08:24`. The average communication time is thus 3.25 seconds. We also noticed that sending the second and third batches started at the same time, and

receiving them finished at the same time as well. These two fully overlapped communication took 4 seconds, which is the longest communication time we have observed. This might be a sign that at this point the application is already IO bounded and multiprocessing won't help much in acceleration.

For the `Flight` implementation, after examining the logs (see Figure 5 as an example) of these nodes, we found that as soon as one node sent out the data, all nodes (including the sender itself) would receive it immediately. Since the granularity of logging is 1 second, we assume that communication is 0.5 second.

Based on the analysis above, we estimate that using `Arrow Flight` saves approximately 3 seconds than using `socket` for communication.

4.4 Discussion: How much does each task take up in the overall execution?

It is also important to see how much time each task take in the overall execution, such that we can determine the direction for optimization.

We have plotted two pie charts of the amount of time each task took in the `socket` approach and in the `Flight` approach. (We used the C++ version of `Arrow Flight` and we noticed that there is a "switch time" from Python to C++. See Appendix B for more details.)

We see that most of the time was spent on partitioning, sorting, adn SerDes. The communication phase does not take up much time in both cases.

4.5 Discussion: Load balancing

So far, load balancing looks good. See Figure 3 for a visualized result.

5 Running in a larger scale

This part still in progress. Here is my predication.

If we use more nodes and more data to perform distributed sorting, does the conclusions we made in Section 4 still hold?

We expect that:

- in both approaches (`socket` and `Flight`), the percentage of time spent on partitioning and sorting would decrease;
- in both approaches, the percentage of time spent on communication would increase;
- the time save by using `Arrow` and `Flight` instead of `pickle` and `socket` would increase.

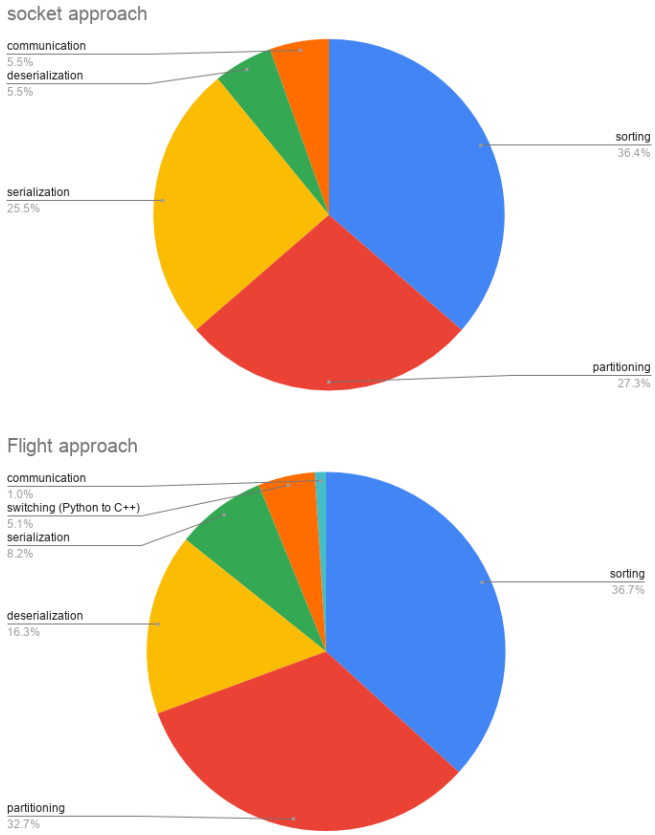

Figure 2: Comparison of the percentage each task took in two approaches

Appendices

A Test run results

See Table 3 and 4 for the original timestamp of the distributed sorting application.

B Known issues

B.1 Building the project

On macOS, the `grpc` library installed via `Homebrew` (as a dependency of `apache-arrow`) seems to be problematic. The Flight server would incur a segmentation fault due to the current version (stable 1.26.0) of `grpc`.

Figure 3: Timestamp of tasks in two approaches

	tcn798	**tcn799**	**tcn804**	**tcn807**
started partitioning the records	13:07:49	13:07:49	13:07:49	13:07:51
finished partitioning the records, started serializing the data to pkl files	13:08:04	13:08:03	13:08:04	13:08:07
finished serializing the data to pkl files, started sending to destination nodes	13:08:18	13:08:15	13:08:18	13:08:22
received last file	13:08:24	13:08:24	13:08:24	13:08:24
started unpickling the records from pkl files	13:08:25	13:08:25	13:08:25	13:08:25
finished unpickling the records from pkl files, started concatenating	13:08:28	13:08:28	13:08:28	13:08:28
finished concatenating, started sorting the records	13:08:32	13:08:32	13:08:32	13:08:32
finished sorting the records	13:08:47	13:08:48	13:08:47	13:08:48

Table 3: Original timestamp of the distributed sorting application (using socket) on 4 nodes

We can make use of the existing build system of arrow to build `grpc` from source. (The build system is capable of building any missing dependency from

	tcn798	**tcn799**	**tcn804**	**tcn807**
started partitioning the records	12:49:49	12:49:49	12:49:48	12:49:48
finished partitioning the records, started putting them to Plasma	12:50:04	12:50:08	12:50:03	12:50:03
finished putting partitioned records to Plasma	12:50:09	12:50:11	12:50:06	12:50:07
send-to-dest started	12:50:11	12:50:14	12:50:08	12:50:10
received last piece of data	12:50:14	12:50:14	12:50:14	12:50:14
started retrieving the records from Plasma	12:50:15	12:50:15	12:50:15	12:50:15
finished retrieving the records from Plasma, started converting to DataFrame	12:50:15	12:50:15	12:50:15	12:50:15
finished converting to DataFrame, started concatenating	12:50:23	12:50:23	12:50:23	12:50:23
finished concatenating, started sorting the records	12:50:26	12:50:25	12:50:25	12:50:26
finished sorting the records	12:50:41	12:50:40	12:50:40	12:50:41

Table 4: Original timestamp of the distributed sorting application (using Apache Arrow Flight) on 4 nodes

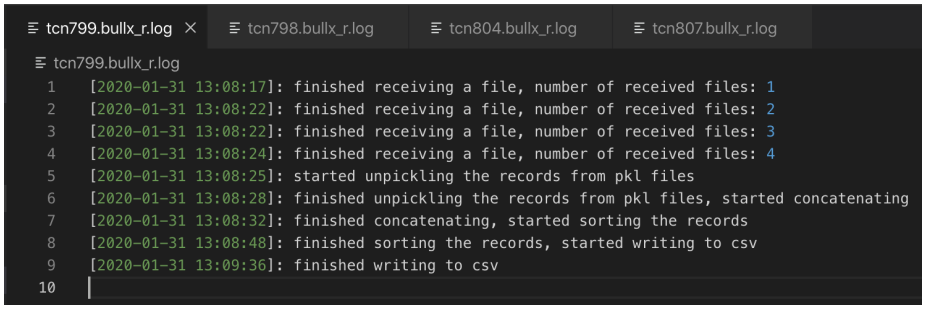

Figure 4: log (receiver part) of node `tcn799`, socket approach

source.) This also saves us from building missing dependencies manually on Cartesius.

B.2 Switch time

Since we used the C++ implementation of `Arrow Flight`, we need to switch from the Python process performing the partitioning task after it is done. The time used for switching from the Python process to C++ process is around 3 seconds, which is more than expected. Since our concern is computation (including partitioning, sorting, and SerDes) and communication, we ignore this issue for now. For analysis we can simply remove this part from the total execution time.

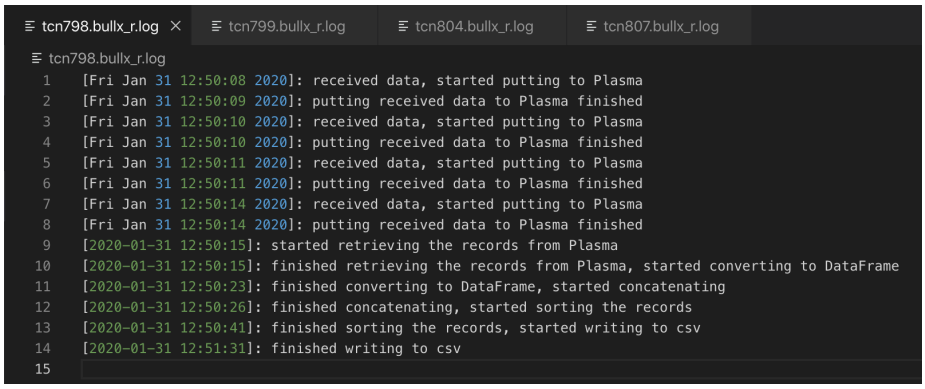

Figure 5: log (receiver part) of node `tcn798`, Flight approach