Design notes for a distributed sorting application

Chengxin Ma

February 1, 2020

1 Introduction

TODO

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
- partitioner and merger
- storage

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

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 the input data.

We denote the start time of partitioning on each node as $part\_s_i$ and the finish time as $part\_f_i$. Thus, the overall time of the partitioning phase is $max(part\_f_i) - min(part\_s_i)$.

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. We denote the time when node i starts to send data to node j as $comm\_send_{ij}$, and the time when node j receives all the data from node i as as $comm\_recv_{ij}$. Assume that sending and receiving take a constant time (i.e. $comm\_recv_{ij} - comm\_send_{ij} = T$ for a given pair of (i, j)), then time of the communication phase $max(comm\_recv_{ij}) - min(comm\_send_{ij})$ would become $T + max(comm\_send_{ij}) - min(comm\_send_{ij})$.

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 merging on all nodes and the latest finish time of sorting on all nodes respectively. We denote the start time of the sorting phase on each node as $sort\_s_i$ and finish time as $sort\_f_i$. Thus, the overall time of the sorting phase is $max(sort\_f_i) - min(sort\_s_i)$.

3 Implementation

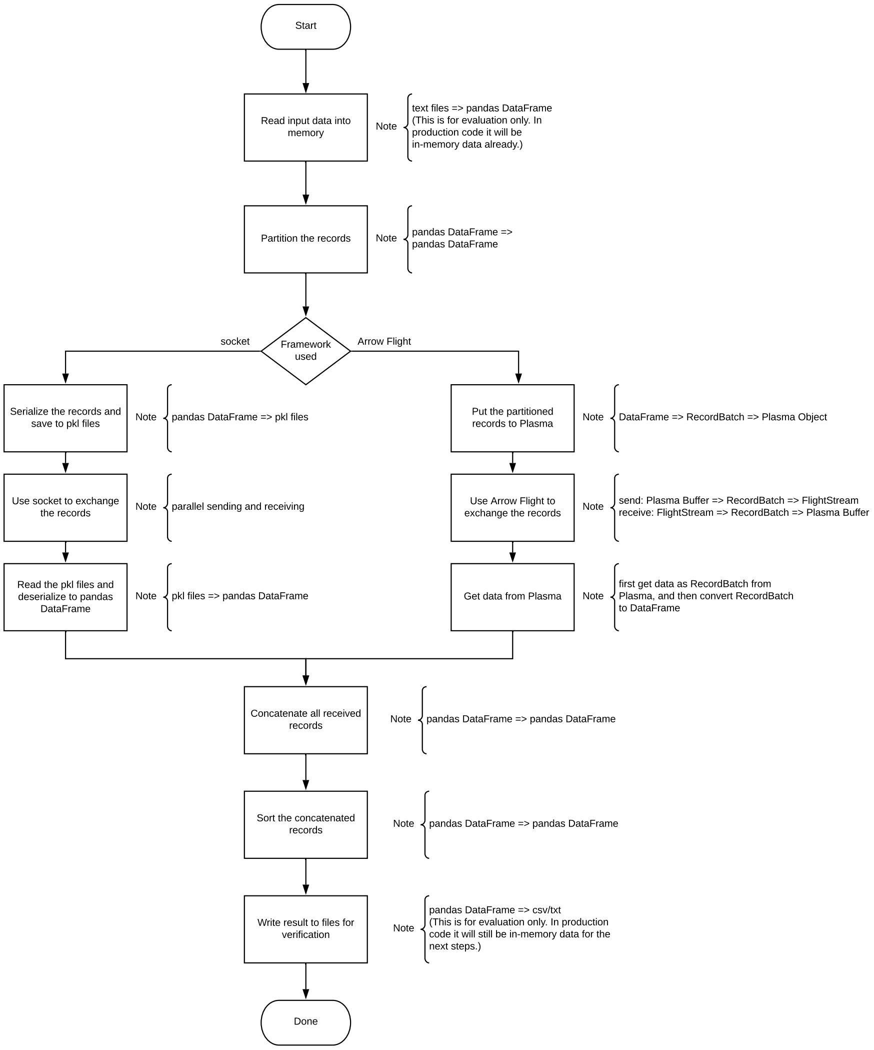
3.1 Implementation choices

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: `socket` (Python version) and `Apache Arrow`

Flight. The interface to the partitioning phase and sorting phase is also different. We use pickle to serialize/deserialize the data before/after the communication phase in the socket version, while Plasma and Arrow RecordBatch are used when Apache Arrow Flight is used for communication.

3.2 Flowchart

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


Start
Read input data into memory
Note
text files => pandas DataFrame
(This is for evaluation only. In production code it will be in-memory data already.)
Partition the records
Note
pandas DataFrame => pandas DataFrame
Framework used
socket
Arrow Flight
Serialize the records and save to pkl files
Note
pandas DataFrame => pkl files
Use socket to exchange the records
Note
parallel sending and receiving
Read the pkl files and deserialize to pandas DataFrame
Note
pkl files => pandas DataFrame
Put the partitioned records to Plasma
Note
DataFrame => RecordBatch => Plasma Object
Use Arrow Flight to exchange the records
Note
send: Plasma Buffer => RecordBatch => FlightStream
receive: FlightStream => RecordBatch => Plasma Buffer
Get data from Plasma
Note
first get data as RecordBatch from Plasma, and then convert RecordBatch to DataFrame
Concatenate all received records
Note
pandas DataFrame => pandas DataFrame
Sort the concatenated records
Note
pandas DataFrame => pandas DataFrame
Write result to files for verification
Note
pandas DataFrame => csv/txt
(This is for evaluation only. In production code it will still be in-memory data for the next steps.)
Done

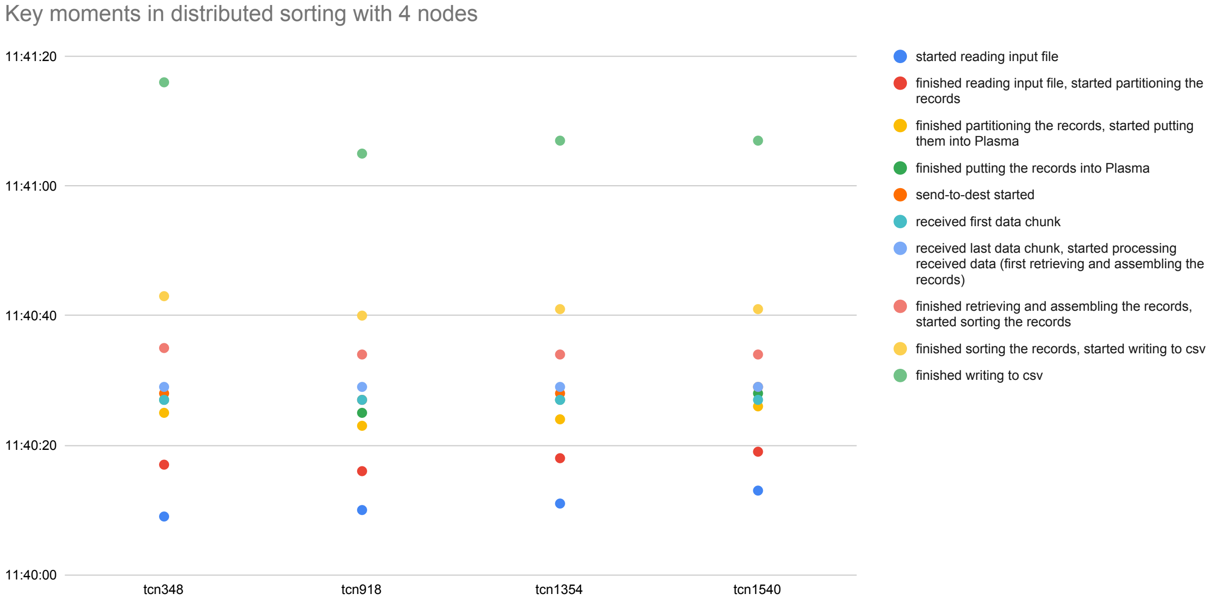
4 Test run

To see the performance of the application, a test run has been performed. 4 nodes on Cartesius (`tcn348, tcn918, tcn1354, tcn1540`, we name them 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 1 million records (i.e. the total number of records is 40 million). Each node had 10 million records in random order before the application ran, and we expected that after 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 270 MB.

The test run was started one by one manually, in the order of from Node 0 to 3. The following picture shows the key moments on these four nodes. [2]

Figure 2: Key moments in distributed sorting test run with 4 nodes



We can see that the performance on the four nodes is more or less the same, except that on `tcn348` it took a bit longer to write to the output files.

In this test run, the first task was to read the data from the input file. It took around 7 seconds. Note that there is no such step after the distributed

[2] Data source: `https://docs.google.com/spreadsheets/d/1xP0lDKG_e8G8zPHNXfgWH1iLae2SFBsvTdGjZgrteWA/edit?usp=sharing`

sorting application has been integrated to the whole genomic pipeline, as its previous step could store the temporary results in memory.

After the input data had been loaded into memory as `pandas DataFrame`, the next task is to partition it according to some criteria. In this case, since we have four nodes, the data was partitioned to 4 subsets, and later they were stored to `Plasma Object Store`. Partition took about 8 seconds while putting the partitioned data to the in-memory store took about 2 seconds.

At this moment, the *sender* was ready to send away to the data to the destination nodes.

The receiving process on every node would put received data to `Plasma` and increment a counter counting how many datasets had arrived. If the *receiver* had observed that it had got all the data, the data processing part (merge and sort) would begin.

On each node, starting sending data to other nodes marks the beginning of the communication phase, while receiving the last piece of data marks the end of the communication phase. Note that the receiving part is dependent on other nodes.

In this test run, the communication phase on each node took roughly 2 seconds. The overall communication phase (marked by earliest sending and latest receiving on **all** nodes) is also about 2 seconds since each node's communication phase overlapped with other nodes'.

To process the received data, first it was needed to be retrieved from `Plasma`. It took about 5 seconds.

After that it was the sorting task, which took approximately 6 seconds.

Finally, the sorted records were written to `csv` files for verification. It took about 26 seconds on 3 nodes, and on the other node it took 33 seconds. Like reading from input data, these step in not needed after integration.

Ignoring the abnormal writing to disk time on `tcn348`, we can see that the overall execution time is about 56 seconds, in which:

- file I/O (reading from and writing to `csv` files) took 32 seconds;
- in-memory store access took 7 seconds;
- communication (sending and receiving records) took 3 seconds;
- partitioning took 8 seconds;
- sorting took 6 seconds.

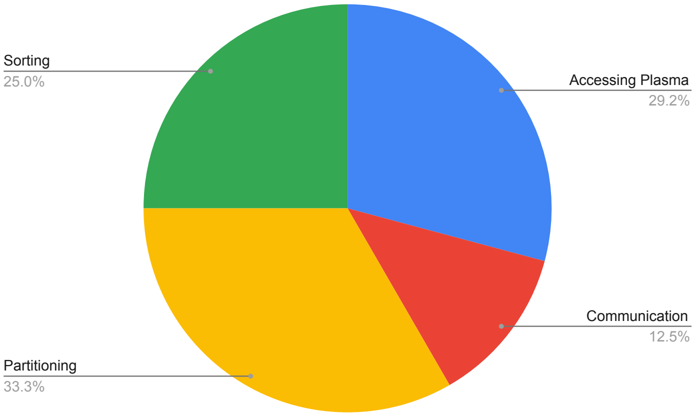
Since disk I/O can be removed after integration, we remove this task and plot the remaining ones in Figure 3.

We see that only a quarter of the time is spent on sorting. The other tasks are required to perform **distributed** sorting. It will be interesting to ask:

Is it worth the effort to distribute the sorting task to multiple nodes? [3]

[3] In the test run, we can predict that if we use only one node to sort all the data, the time of partitioning, accessing Plasma, and communication will be all gone, but the sorting time will be a bit more than 4 times than the current sorting time (assuming $O(n \log n)$ complexity). So the overall time might be a bit more than distributed sorting.

Figure 3: Time spent on different tasks (disk I/O removed)



5 Running in a larger scale

A Known issues

A.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`.

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