Project #4 - Analysis of COVID-19 Data

Problem Presentation and Dataset

In the following project, we need to implement an MPI program that analyzes the evolution of COVID19 data over time. We decided to use a ready-made dataset with worldwide data, starting from the 31st of December 2019 until the 14th of December 2020.
The goal of the project is to compute different metrics in order to track the spreading of COVID19 in each country (for every day):

- Seven-days moving average (7DMA) of new reported cases;
- Percentage increase (with respect to the day before) of the 7DMA;
- Top 10 countries with the highest percentage increase of the 7DMA.

The dataset already had all the needed data (date, name of the country, cases), so no further processing was needed. Also, it's ordered by country (alphabetically) and by date (descending order).

Assumptions

- Static dataset: the dataset is not updated once the program starts.
- Dataset ordered by country name (need all data from a country to be together) and date (descending order).
- When there is missing data (sometimes it happens that a day (or more) is not present in the dataset), we assume the 7DMA remains the same as the day before.
- There are at least 2 available processes, in order to correctly run the program.

Algorithm Design

Since MPI is a protocol designed for parallel computing, we decided to use a master-slave approach, using 1 process as master (the one with rank=0) and all the others as slaves. The master displays the metrics and computes the top 10, while the slaves have the data for each country and process it to compute the seven-days moving average.
The program is actually divided into two parts:

- Data distribution: all the data is first distributed among all the available slaves by the master (such that all the data from a country goes to the same slave).

- Computations: the master sends everyday the new date and the slaves compute, for all the countries they have in memory, the new metrics.

Data Structures

The main data of the program is enclosed in 4 structs:

- `CountryResults`: stores the aggregated data required by the program (`countryName`, `percentageIncreaseMA` (Moving Average), `cases`).
    - This is the data used by the master in the “Computations” part.
- `SlaveData`: saves the number of countries saved inside the slave and has an array of “`Country`” (1 element for each country saved inside it).
    - It’s the main data structure of the slaves.
- `Country`: for each country in the dataset saves the name, an array of “`data`” (1 element for each day in the dataset), the 7DMA, percentage increase of the 7DMA, the total cases for the seven-days, a window array with the last 7 considered values (it works as a sliding window).
    - The two indexes present in the structure are used to know what is the next position to be used inside the corresponding array.
- `data`: saves the actual data from the dataset (date, number of cases).
    - It’s the data used to do the computations.

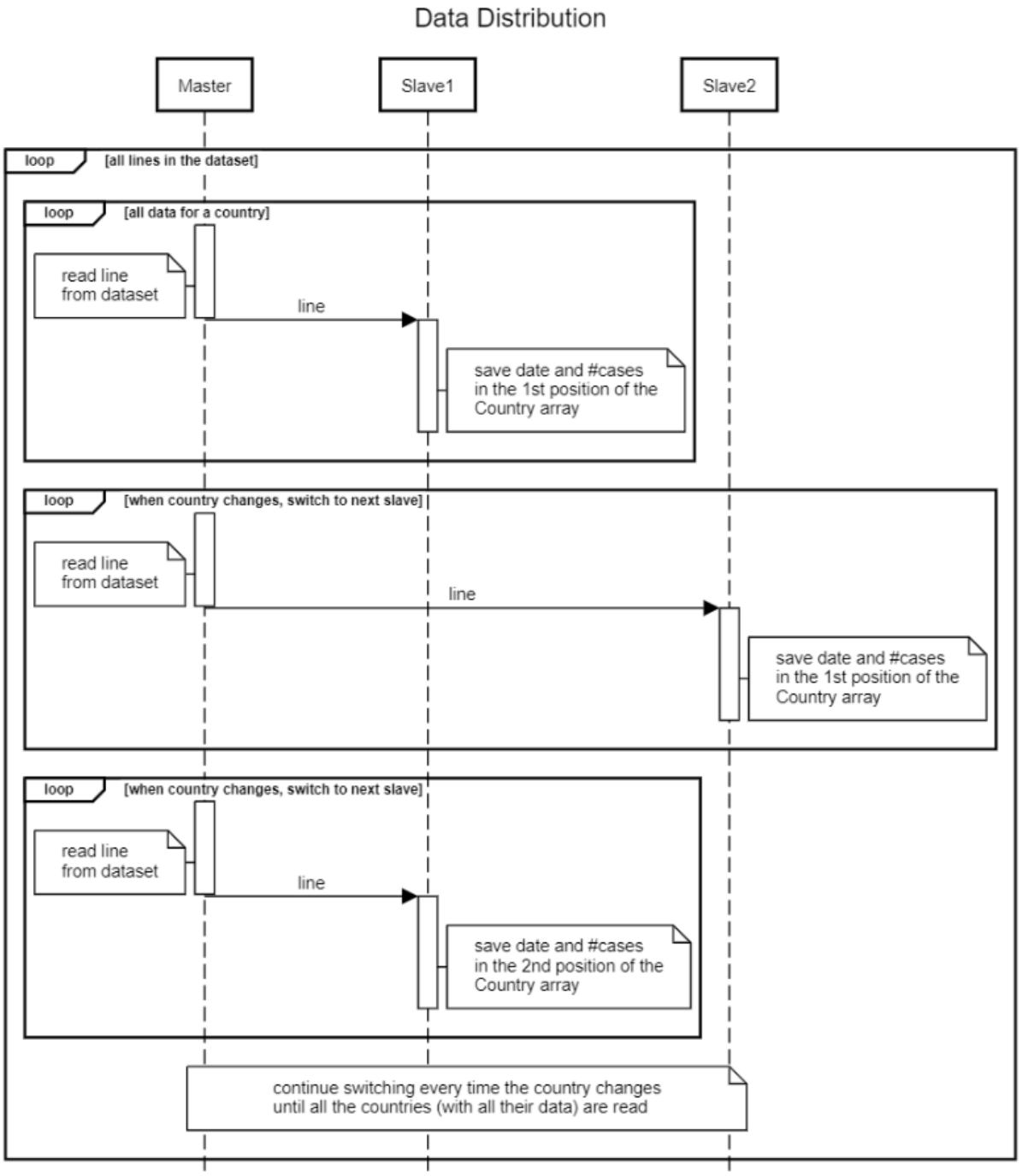
Implementation

The program starts with the master reading the dataset and immediately sending each line it reads to a slave. It keeps sending data to the same slave until the `countryName` changes: then the master switches the destination to the next slave. This was done to allow as much data locality as possible while trying to maintain a balanced system (i.e. giving the same number of countries to each slave). Also, the master saves the latest and earliest dates present in the dataset (this was done to make the program more flexible, allowing it to also eventually work with an updated dataset).
The slaves receive the raw data, they check if they are receiving a new `countryName` (if so, a new position in the array `Country` is used), then extract the date and number of cases and save them in the dedicated data structure.
Since the distribution part is over, an `MPI_Barrier` is used to synchronize all the nodes: this was done to allow all the nodes to “synchronously” start the second phase.
Furthermore, the computation part proceeds almost synchronously among all the nodes: in order to compute the metrics for each day, the slaves wait for the

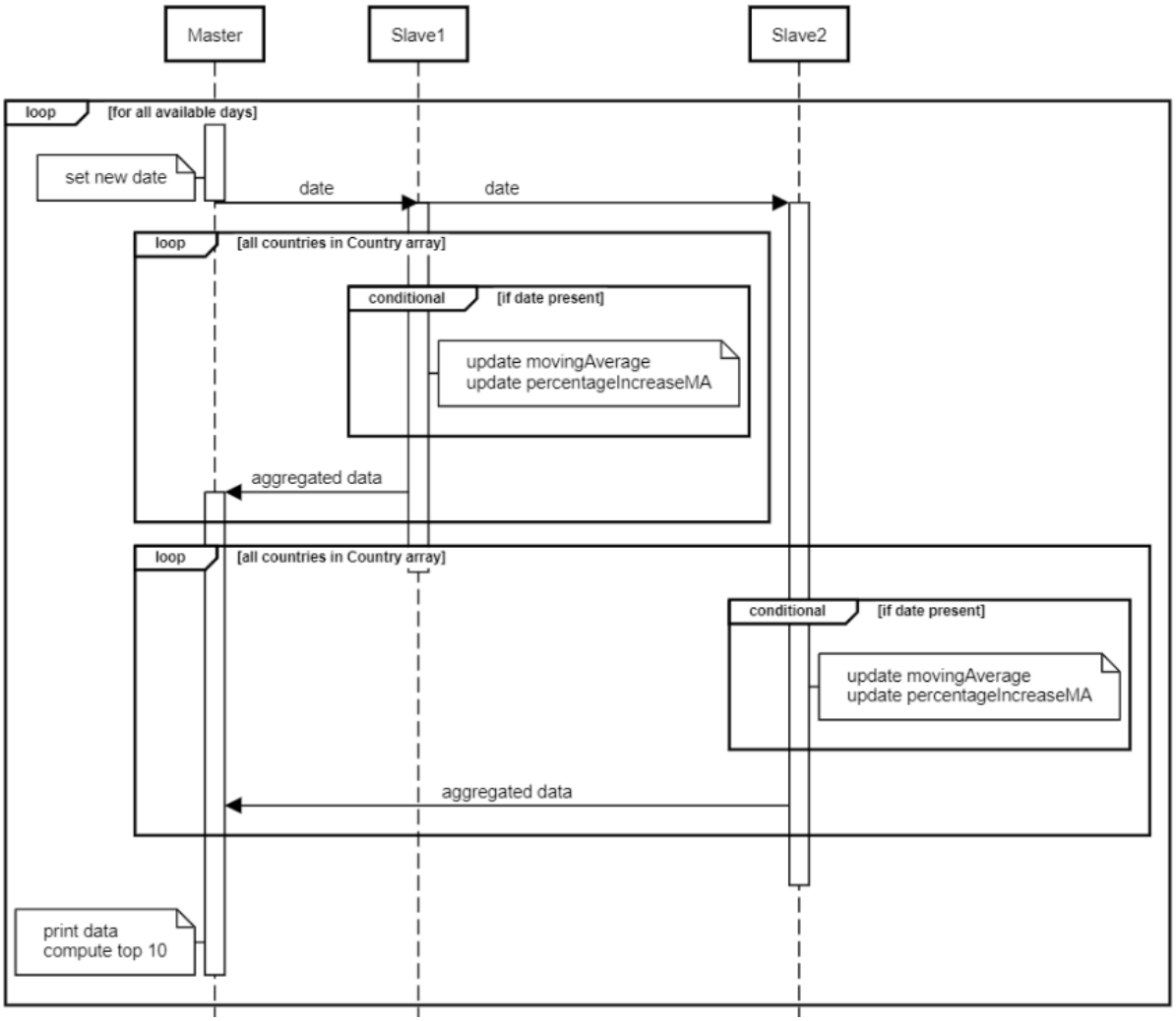
master to send them the date to be considered for the current computation. This was done because the program needs to print the metrics for each day (and this allowed to do it in a tidy fashion) and because the master has to compute a top 10 for every day, so the clients need to send only the corresponding data without “jumping ahead” to the next days.
Here the master starts by sending to all the slaves the start and end dates, then the main cycle takes place: in each iteration the server sends a different date (starting from the first date present in the dataset until the last one) and waits to receive all the data from all countries (for that day). Then computes and prints the top 10 (we decided to compute it using an insertion sort algorithm: it’s not the best in terms of complexity, but since the data to sort is at most the number of countries in the world (around 200), we figured it wouldn’t be too computationally demanding).
The slaves instead, in the computation part, start by receiving the start and end dates, then enter their main loop. Once they receive the a new date, they check in all the countries saved inside their data structures if that date is present: if so, they save the number of cases in the sliding window, update the metrics and send them to the master, otherwise they send to the master the same metrics that were computed previously, since we assume the moving average is the same as before.

Sequence Diagrams

Below, there are two sequence diagrams to better explain the interactions between master and slaves during the program, using only two slaves for readability.


Data Distribution
Master
Slave1
Slave2
loop
[all lines in the dataset]
loop
[all data for a country]
read line from dataset
line
save date and #cases in the 1st position of the Country array
loop
[when country changes, switch to next slave]
read line from dataset
line
save date and #cases in the 1st position of the Country array
loop
[when country changes, switch to next slave]
read line from dataset
line
save date and #cases in the 2nd position of the Country array
continue switching every time the country changes until all the countries (with all their data) are read

Computation



Performance Tests

Tests were performed on the implemented program using 2 virtual machines running Ubuntu 20.04.6 LTS, both on the same computer and subnet (ssh configured using local IPs).
Tests were made by only printing the time taken to perform the computations, not also printing the actual data: this was done to get a better estimation of the performance, without messing with the prints on the terminal.
Different datasets and configurations for the machines were used.
Datasets:

- super-reduced-dataset: 10 countries, 7 days
- reduced-dataset: 100 countries, 150 days
- input: complete dataset; 213 countries, 350 days

We arrived at the conclusion that running this program on one machine is the best option, given the computation times (it's basically twice as fast as running in a distributed fashion). This is probably due to overheads in the MPI and ssh communications (since many messages need to be exchanged in the program) and it also might be due to the virtual machines running on the same machine (thus probably sharing the physical resources).

Single machine with 3 cores, 2GB of memory, n total cores used in the computation (m,k)
- m is the local machine (the one running the command)
- k is the ssh machine

super-reduced-dataset
- 2 cores
  - 5, 3, 3 milliseconds
  - Average: 4 ms
- 3 cores
  - 2, 3, 2 milliseconds
  - Average: 2 ms

reduced-dataset
- 2 cores
  - 305, 304, 303 milliseconds
  - Average: 304 ms
- 3 cores
  - 324, 327, 339 milliseconds
  - Average: 330 ms

input
- 2 cores
  - 1244, 1230, 1238 milliseconds
  - Average: 1237 ms
- 3 cores
  - 1352, 1322, 1332 milliseconds
  - Average: 1335 ms

Machines with 2 cores each, 2GB of memory each, n total cores used in the computation (m,k)
- m is the local machine (the one running the command)
- k is the ssh machine

super-reduced-dataset
- 2 total cores (1,1)
  - 26, 37, 30 milliseconds
  - Average: 31 ms
- 3 total cores (2,1)
  - 31, 26, 37 milliseconds
  - Average: 31 ms
- 4 total cores (2,2)
  - 57, 36, 49 milliseconds
  - Average: 47 ms

reduced-dataset
- 2 total cores (1,1)
  - 809, 771, 766 milliseconds
  - Average: 782 ms
- 3 total cores (2,1)
  - 675, 599, 651 milliseconds
  - Average: 641 ms
- 4 total cores (2,2)
  - 809, 714, 700 milliseconds
  - Average: 741 ms

input
- 2 total cores (1,1)
  - 2784, 2716, 2743 milliseconds

  - Average: 2747 ms
- 3 total cores (2,1)
  - 2270, 2274, 2443 milliseconds
  - Average: 2329 ms
- 4 total cores (2,2)
  - 2640, 3022, 2969 milliseconds
  - Average: 2877 ms

Machines with 3 cores each, 2GB of memory each, n total cores used in the computation (m,k)
- m is the local machine (the one running the command)
- k is the ssh machine

<u>super-reduced-dataset</u>
- 2 total cores (1,1)
  - 34, 25, 25 milliseconds
  - Average: 28 ms
- 4 total cores (2,2)
  - 45, 60, 47 milliseconds
  - Average: 50 ms
- 4 total cores (3,1)
  - 35, 24, 33 milliseconds
  - Average: 30 ms
- 5 total cores (3,2)
  - 47, 58, 65 milliseconds
  - Average: 56 ms
- 6 total cores (3,3)
  - 62, 62, 73 milliseconds
  - Average: 65 ms

<u>reduced-dataset</u>
- 2 total cores (1,1)
  - 739, 742, 707 milliseconds
  - Average: 729 ms
- 4 total cores (2,2)
  - 860, 725, 747 milliseconds
  - Average: 777 ms
- 4 total cores (3,1)
  - 699, 703, 698 milliseconds
  - Average: 700 ms

- 5 total cores (3,2)
  - 832, 908, 870 milliseconds
  - Average: 870 ms
- 6 total cores (3,3)
  - 1119, 1014, 1126 milliseconds
  - Average: 1086 ms

input
- 2 total cores (1,1)
  - 2993, 2941, 2918 milliseconds
  - Average: 2950 ms
- 4 total cores (2,2)
  - 2938, 2932, 2976 milliseconds
  - Average: 2948 ms
- 4 total cores (3,1)
  - 2555, 2295, 2533 milliseconds
  - Average: 2461 ms
- 5 total cores (3,2)
  - 2732, 2757, 2812 milliseconds
  - Average: 2767 ms
- 6 total cores (3,3)
  - 4260, 4466, 4447 milliseconds
  - Average: 4391 ms