MapReduce on IBM Cloud

Distributed Systems Task 1

German Telmo Eizaguirre Suarez

URV, Double Degree in Biotechnology and Computer Engineering

Tarragona, 23-04-2019

Source code: https://github.com/GEizaguirreURV/MapReduce-SD.git

Specification

We had to implement a MapReduce model of a CountingWords and WordCounting application over a text file. We created a serverless function-based application using the IBM Cloud Functions service. IBM Cloud Functions is a FaaS platform for serverless computing. In our program we used serverless functions for distributing the computational work among different numbers of workers.

We used the IBM COS service for sharing results between workers and the central node. The central node is called orchestrator and monitors the whole computation. Workers upload their mapping results to a IBM COS bucket and the orchestrator reduces the partial results read from the bucket.

For improving the implementations' communication we used a push model using the IBM CloudAMQP message queue service. It avoids overloading the orchestrator with apull communication model by subscribing to a queue where workers put notifications of their partial results.

We also included a control to avoid cutting words when assigning chunks to each worker for the CountWords and the WordCount.

In our program, workers are the message producers of the queue and the orchestrator is the consumer.

Dependencies

Our program is implemented in Python 3.6 and requires having the following Python packages installed in the computer or IDE before execution:

- **ibm_botocore** to connect to IBM COS.
- **ibm_boto3** to connect to IBM COS.
- **ibm_cf_connection** to connect to IBM Cloud Functions.
- **requests** to connect to the IBM Cloud.
- **pika** to use the IBM CloudAMQP queue service.
- **json** to put and get dictionaries in/from a COS bucket.
- **yaml** from **pyyaml** for the external file with configuration parameters.
- **zipfile** for putting the mapping function into the IBM Cloud Functions service.

Design decisions

To execute our orchestrator a configuration yaml file named *'cloud_config'* must be located in the program's path. This file includes the names of the two IBM COS buckets used in the

execution and the configuration parameters for IBM COS, IBM Cloud Functions and IBM CloudAMQP. A model of the *'cloud_config'* file is included in the GitHub repository.

The file orchestrator.py file accepts the following arguments:

- The name of a text file dataset. The dataset must exist in the *'bucket_name'* bucket of the configuration file.
- A number of chunks. It is adjusted to 19 as a maximum. This refers to the number of parallel workers that the program will call for processing the dataset, which is equal to the number of chunks in which the dataset will be divided. It is limited to 19 as our RabbitAMQ accepts 20 connections as a maximum, according to its web page.
- An optional initial *'-p'* argument for printing the whole resulting dictionary of the WordCount.

Execution examples:

python3 orchestrator.py gutenberg-100M.txt 7

python3 orchestrator.py -p DonQuijote.txt 2

Workers execute a map_dataset function remotely. They do the WordCount and the CountWords of their assigned chunk and upload the resulting dictionary to the *'chunks_bucket'*. They then publish a message into the connection queue with the name of the object. The orchestrator receives the message, gets the named object from the bucket, reduces the dictionary and finally deletes the partial result from the COS.

The map_dataset function is uploaded to COS Cloud Functions by the orchestrator with 1024 MB memory and 600s timeout. 600s of timeout are particularly necessary for the single worker processing of big datasets.

We used two COS buckets in the implementation for clarity. The "clean" bucket ('*bucket_name*') stores the datasets and it is where the orchestrator uploads the final result of the execution. The "dirty" bucket (*'chunks_bucket'*) is where workers upload the partial results of their chunks and from where the orchestrator reads and reduces these results.

We added a not-cutting-words code so that when dividing datasets into chunks words are not cut in the middle. Basically, each worker displaces its chunk to the left from the start until the beginning of the first word of the chunk. They do the same with the end of the chunk. As every worker executes this displacement, chunks are not overlapped.

Although when testing our code with small datasets the WordCount and CountWord functionalities worked well, we found some inconsistencies when working with big datasets. In these cases the total count of words varied in a two unit range when using different numbers of chunks. Our implementation does not consider words including not-alphanumerical symbols (such as "sugar-free" or webpage urls), which are cut. Chunks starting or ending with this sort of words could be the source of our problem.

When working with big datasets our map_dataset function generated memory errors due to big dictionaries. We solved this in two ways. On the one hand, we split the chunks into iterator-like word generators rather than word lists, which are a lot less memory demanding. Word lists store the whole of their data structure into physical memory, whereas iterators generate one item at a time. On the other hand, we did a subdivision of chunks inside the map_dataset function. Each

worker, regardless of the size of it chunk, processes the text in blocks of 40 MB. The worker loads the result of each block to IBM COS and publishes its name into the queue. When it ends with the chunk, publishes a "final" message into the queue, so that the orchestrator knows it has finished.

Besides, the second part of the solution improves the performance of our program, as the orchestrator can start reducing results before workers have completed their chunks. It allows an additional level of parallelization. We stablished the size of our chunks in 40 MB as it showed the best performance after several trials.

The program handles various exceptions concerning errors in connections credentials and arguments. However there are some not handled errors such as the loss of Internet connection of the machine.

Performance

We did a speedup test against a sequential version of our program. (orchestratorSequential.py in the GitHub repository). Results can be seen in Figure 1. Numerical results are included in the annex.

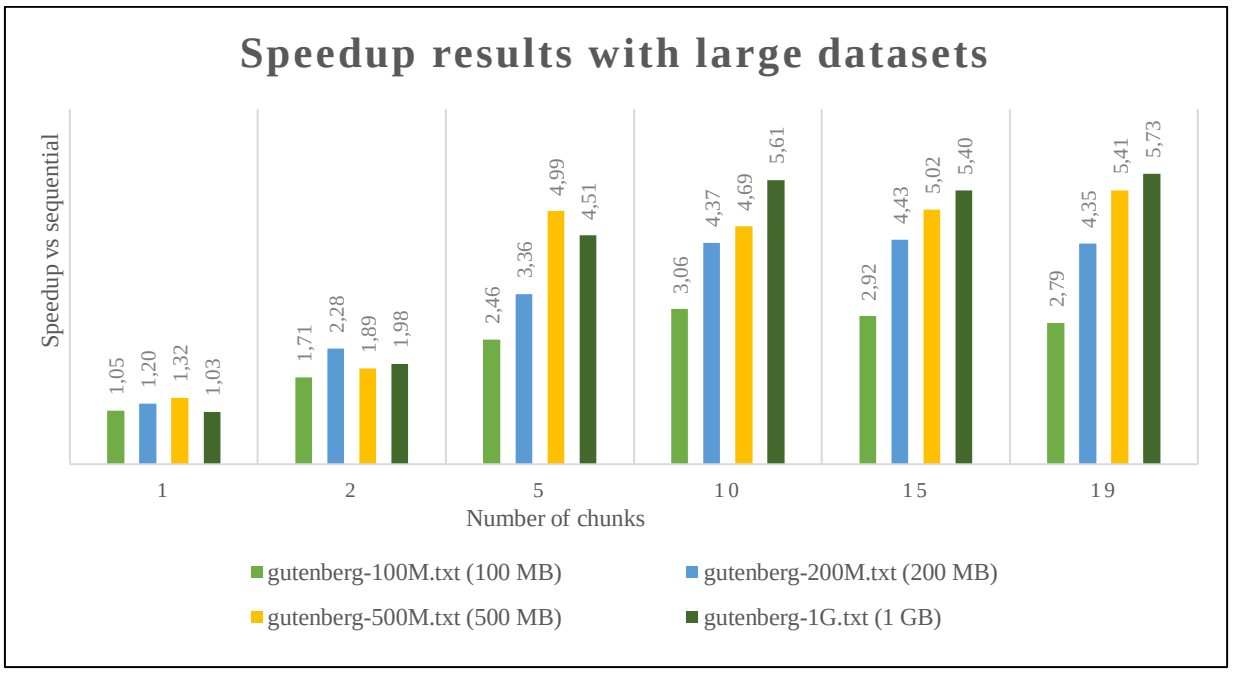

Figure 1 Speedup results for our program.

We distinguish an improvement in performance in all executions. Large datasets (500 MB and 1 GB) show great speedup when dividing them in 10 chunks or more. However, 10, 15 and 19 chunks' executions have similar results. Our possible chunk number range (1-19) shows good behaviour with our target datasets. As with an increasing number of chunks speedup does not clearly grow, more parallel workers are not likely to increase the performance. Generally speedups improve as the size of datasets increases.

We also tested our program with small datasets (less than 10 MB). However speedup was almost inexistent and we decided not to include their graphics in this document. Numerical results can also be checked in the annex.

Annex

Table 1 Numeric results for our tests.

Benchmark	Chunk number	Real time (s)	Speedup
DonQuijote.txt (2,1 MB)	sequential	3,309	1,00
	1	3,6199	0,91
	2	3,4858	0,95
	5	3,8658	0,86
	10	4,4405	0,75
	15	4,8364	0,68
	19	6,0836	0,54
TheBible.txt (4,2 MB)	sequential	5,0009	1,00
	1	2,7208	1,84
	2	2,8973	1,73
	5	3,0548	1,64
	10	3,8747	1,29
	15	4,3461	1,15
	19	5,2346	0,96
SherlockHolmes.txt (6,2 MB)	sequential	5,399	1,00
	1	3,6752	1,47
	2	3,5362	1,53
	5	4,0725	1,33
	10	4,9228	1,10
	15	6,2586	0,86
	19	7,5245	0,72
gutenberg-100M.txt (100 MB)	sequential	37,6777	1,00
	1	35,7159	1,05
	2	21,9703	1,71
	5	15,3031	2,46
	10	12,3231	3,06
	15	12,9058	2,92
	19	13,5276	2,79
gutenberg-200M.txt (200 MB)	sequential	85,7202	1,00
	1	71,5955	1,20
	2	37,5751	2,28
	5	25,5345	3,36
	10	19,6208	4,37
	15	19,356	4,43
	19	19,7214	4,35
gutenberg-500M.txt (500 MB)	sequential	193,2622	1,00
	1	146,9512	1,32
	2	102,5188	1,89
	5	38,7158	4,99
	10	41,2024	4,69
	15	38,5155	5,02
	19	35,7524	5,41

gutenberg-1G.txt (1 GB)	sequential	342,888	1,00
	1	332,2255	1,03
	2	172,819	1,98
	5	76,0005	4,51
	10	61,1627	5,61
	15	63,4941	5,40
	19	59,885	5,73