MAP REDUCE

Distributed Systems

Aaron Carricondo Sánchez

Nikolay Dimitrov Vasilev

Index

Design decisions

Punctuations

The first design decision was the replacement of nearly all the punctuation signs:

```
!()-[]{};:'"\,<>.?@#$%^&*_~=\n\r\t
```

This decision may cause lots of retard on the largest files, but some characters like '\n' must be replaced for a proper behaviour of the programs.

Cut words

To solve this problem we made two things, both are made after replacing the punctuations signs:

- First, if the last character of the text we took is a blank, we take that word if not, we remove it from the partition text.

- After that, we first check if it is the first partition (start byte = 0). If it is not, we get the previous byte of the starting one (the last byte from the previous partition), and we check if it's a punctuation sign or a blank. If it is not, we add that character to an auxiliary string that stores the word. We repeat this procedure until one of the two are found (blank or punctuation sign). When it is found, then we put the new word in the beginning of the text.
  A pseudocode of this will be:

```
start--
new_word = ‘’
char = get byte(start)
while (True) do

      char = get byte(start)
      if (char != punctuation_sign && char != ‘ ‘) then
            new_word = char + new_word
      else
            break;
      fi
      start--

fwhile

text = new_word + text
```

RabbitAMQP

To use the AMQP queues instead of pulling the Object Storage periodically, we used queues with a publisher / consumer model. In this case the orchestrator will be the consumer in both tasks (map and reduce), and the publishers will be map and reduce in it' s corresponding tasks.

We finally used two queues, one for map and one for reduce, because we think it's more efficient use a queue than make constant pulls to the COS IBM service.

To implement this, we just have to declare the channel to connect to the IBM service and the two queues in the orchestrator. Then in both the reduce and map we just connect to the channel connected to the IBM service and when the function ends, we publish a message to its corresponding queue.

For counting the messages received by both the mapper and reducer, we implemented two *callback* function for each:

- The map *callback* stops consuming messages when the counter arrives to the number of partitions, because it has to wait until all the maps are done.

- The reduce *callback* stops consuming messages when the counter arrives to one, because we only execute one reducer for each Counting Words and Word Count.

The program prints every message received from each queue.

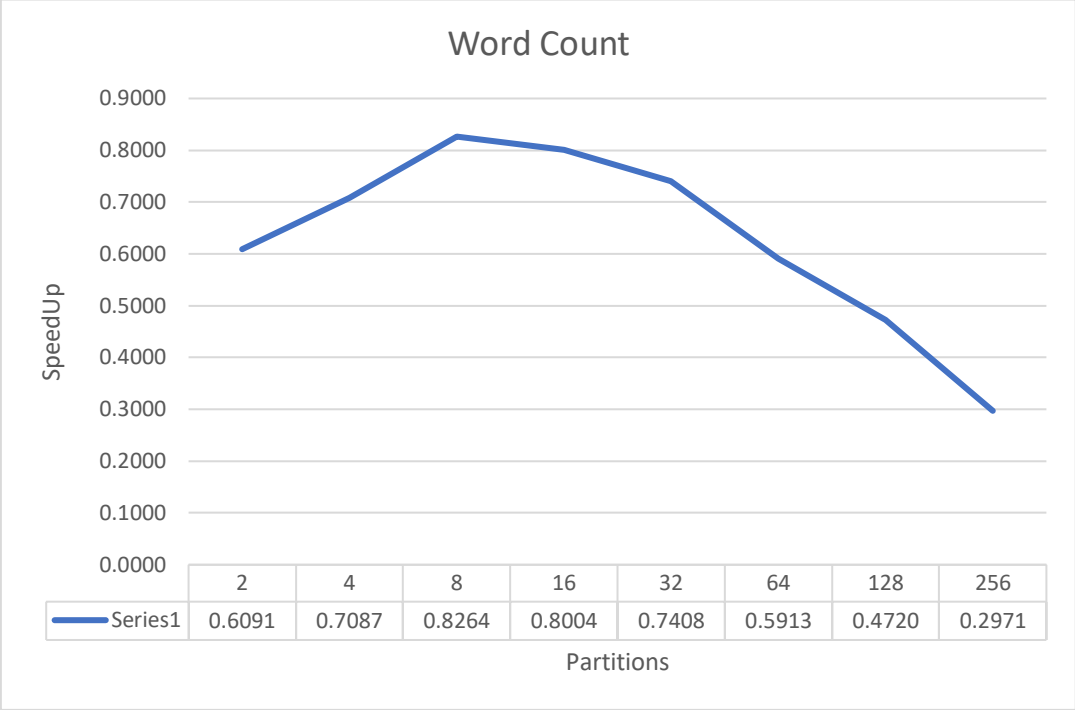
SpeedUps

In the next pages we will see two graphics for every large file, one for the Word Count version and one for the Counting Words version. They both will show the speedup between the IBM Functions version and the local lineal version.

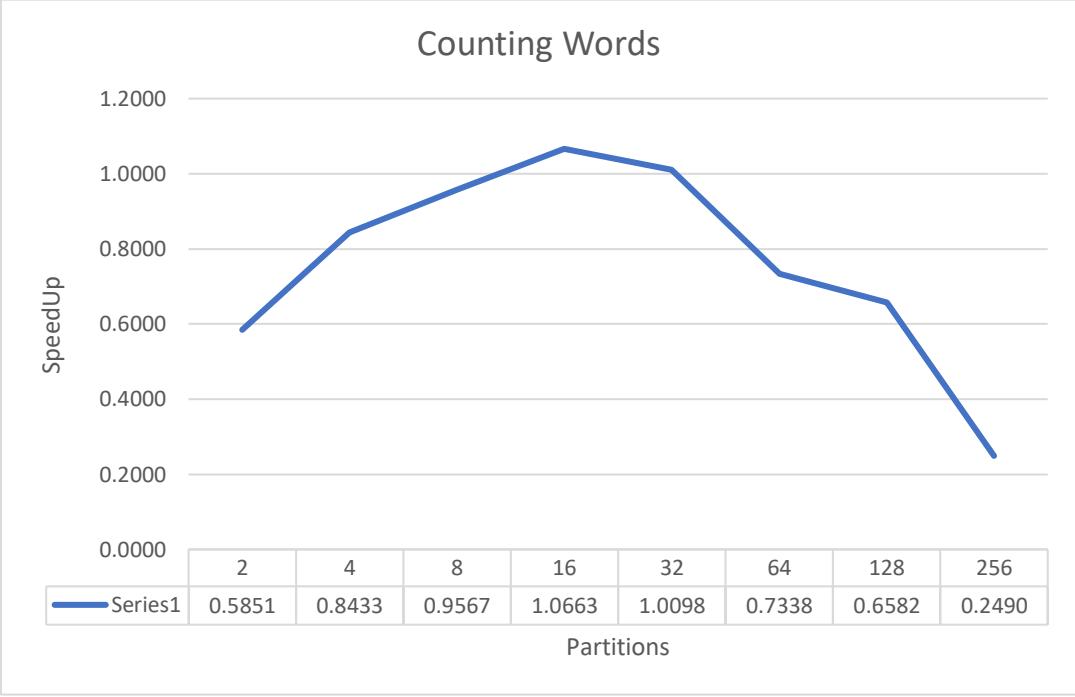
gutenberg-100M.txt (104,9 MB)

As we can see in the graphics, our implementation of the Word Count does not take a better performance than in the local version. Although, the Counting Words is slightly better with 16 and 32 partitions.

- Word Count local time = 16,52 seconds

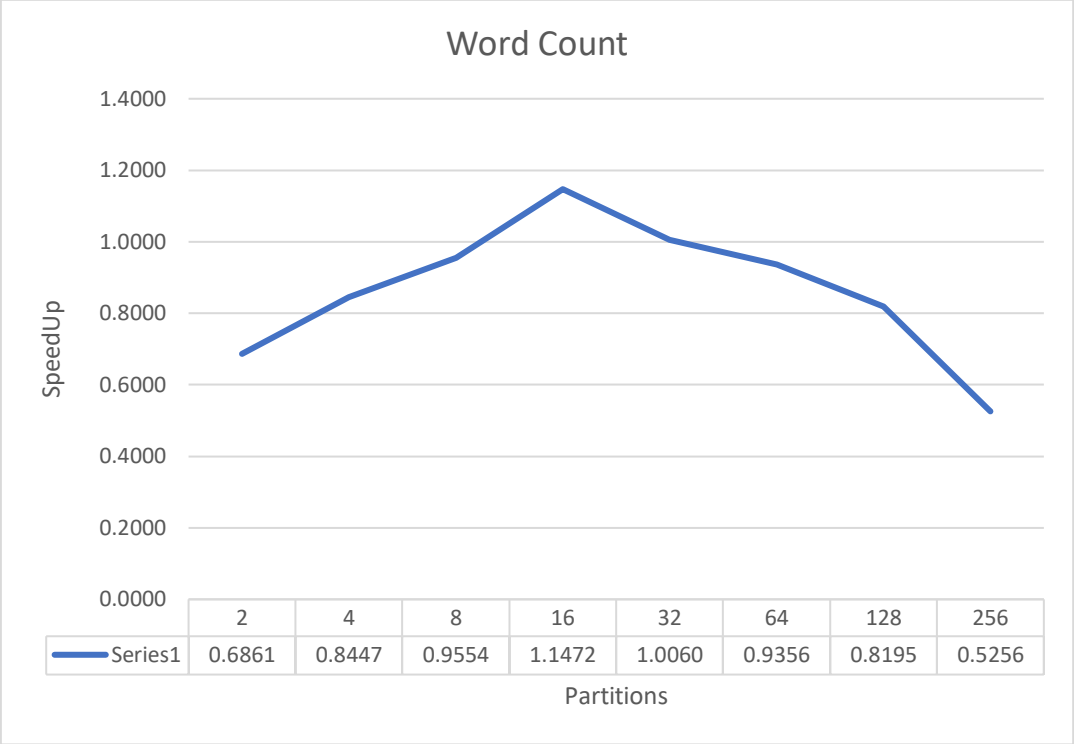


- Counting Words local time = 12,38 seconds

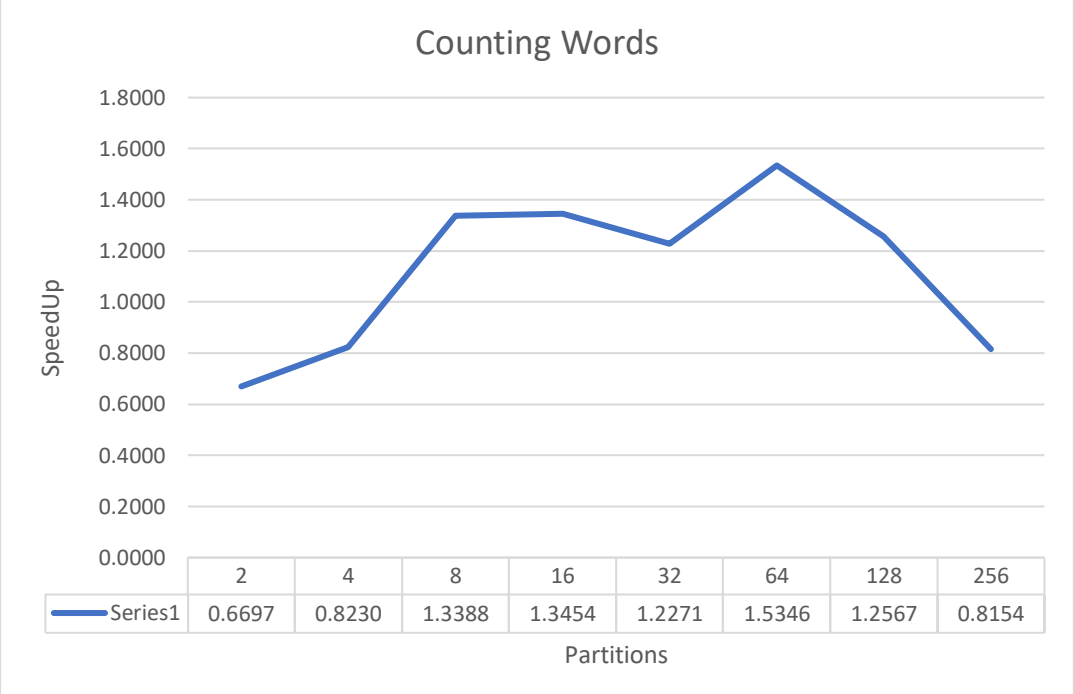


gutenberg-200M.txt (209,7 MB)

- Word Count local time = 35,15 seconds



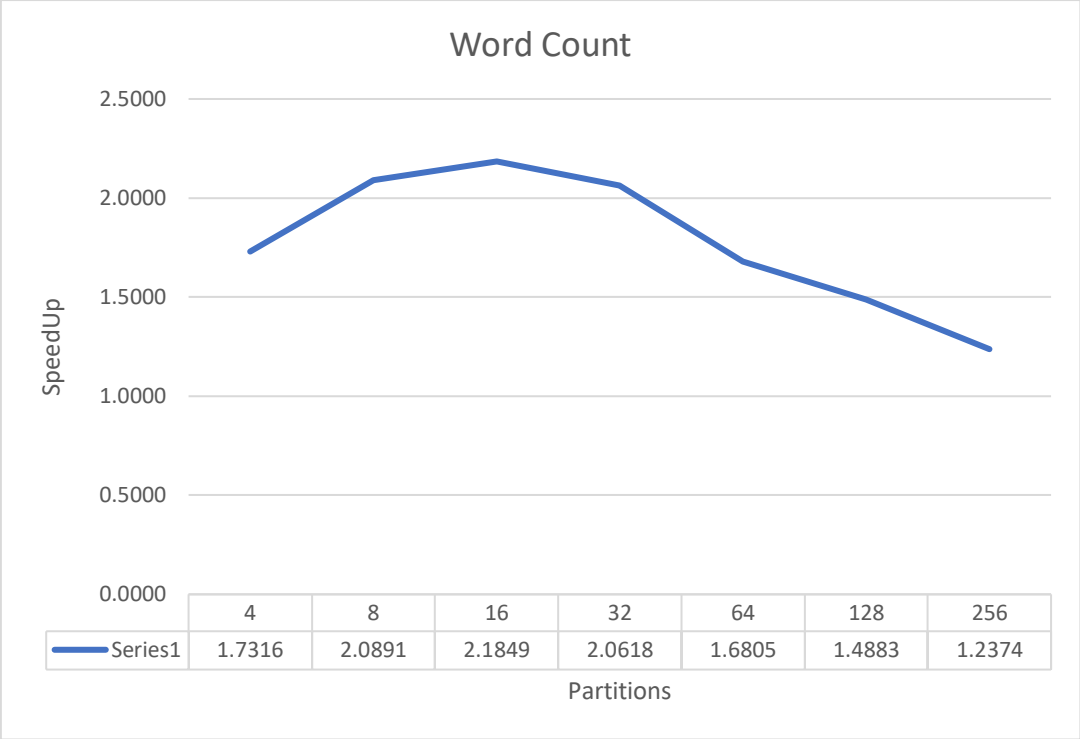
- Counting Words local time = 27,07 seconds



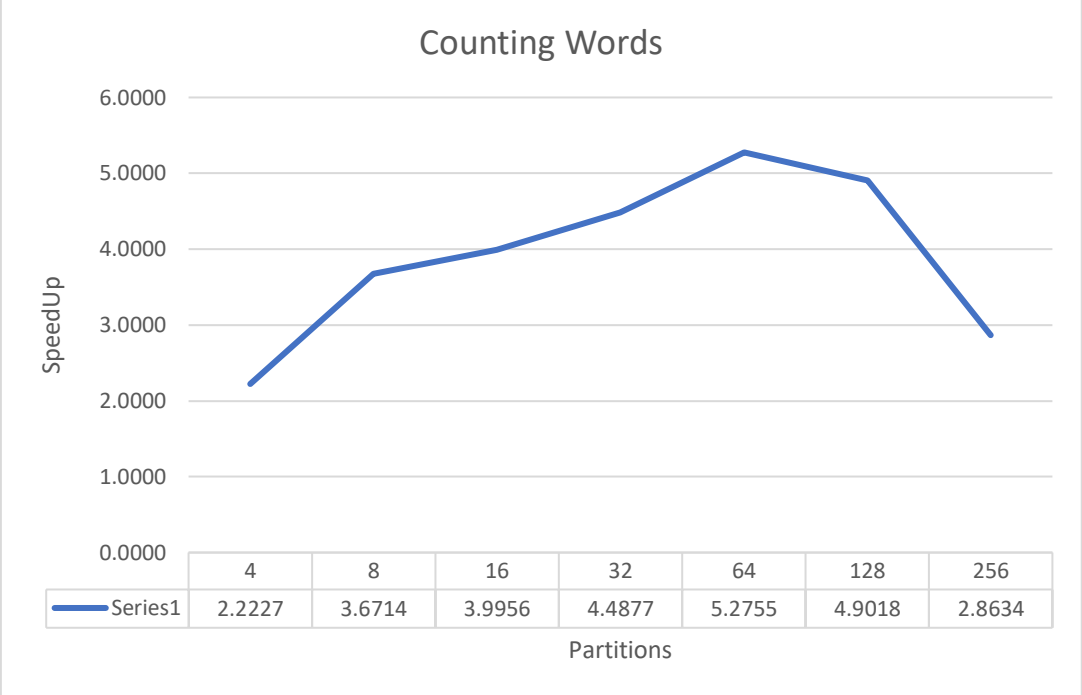
gutenberg-500M.txt (524,3 MB)

For this file we begin with 4 partitions because when we tried it with 2 partitions, we get a memory exhaust error. We will get this error in the largest file too.

- Word Count local time = 99,15 seconds



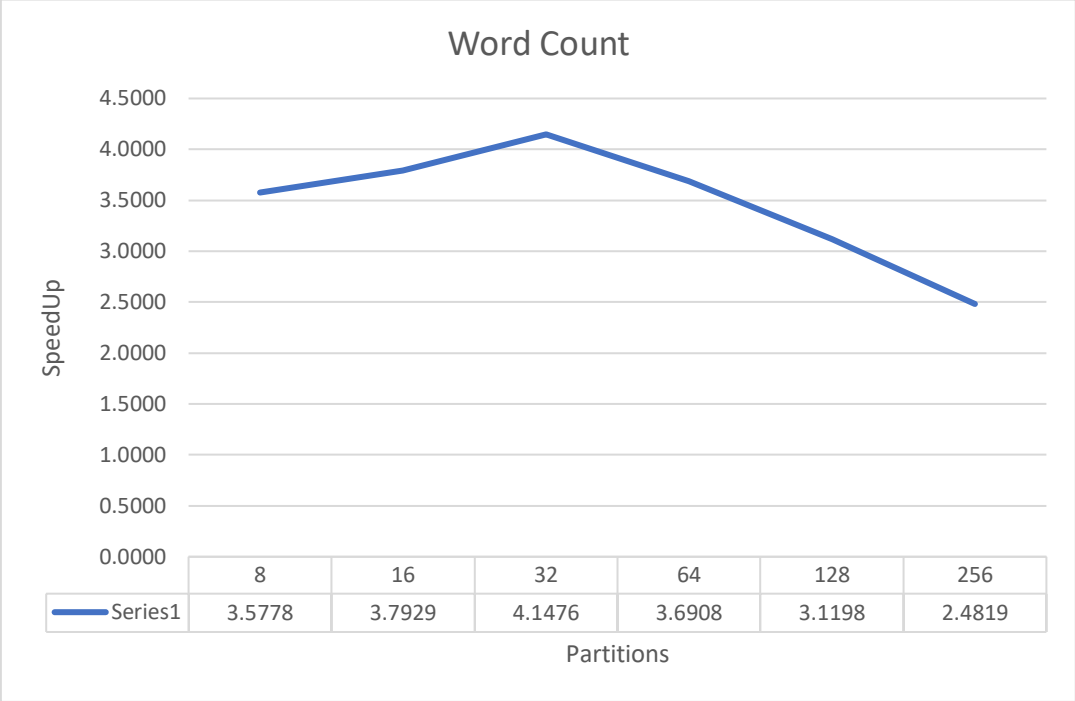
- Counting Words local time = 82,35 seconds



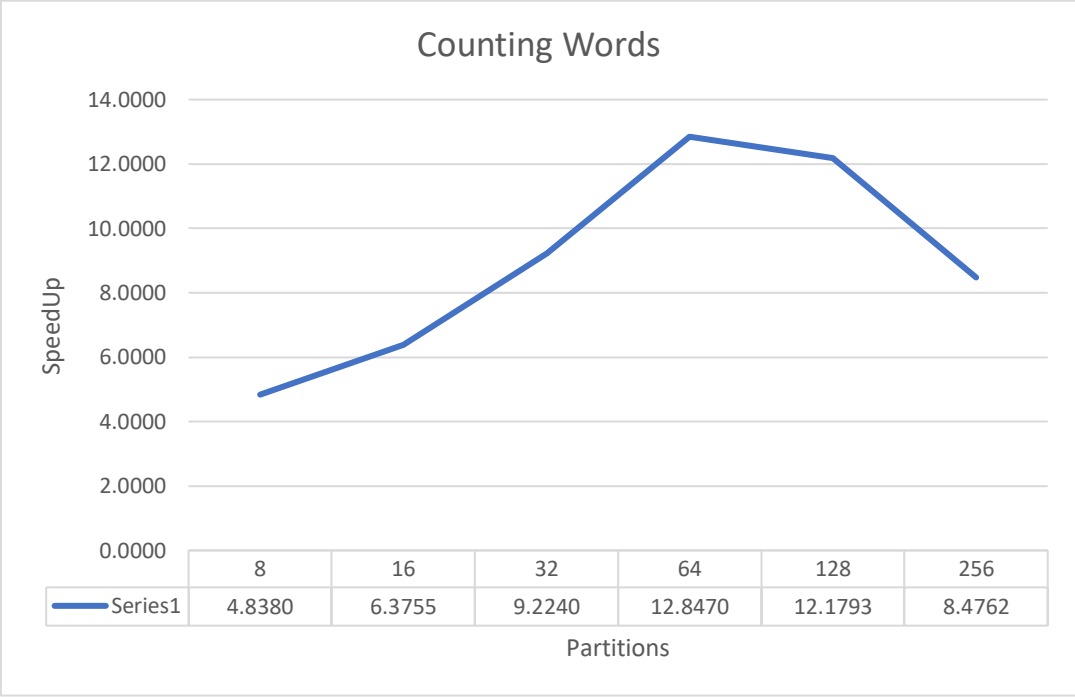
gutenberg-1G.txt (1,07 GB)

For this file we begin with 8 partitions and finish with 128. We tried to make 2 and 4 partitions but both of them exhausted the 2048 MB maximum memory like the previous file.

- Word Count local time = 285,15 seconds



- Counting Words local time = 213,26 seconds



Conclusion

As you could see, in the largest files the Counting Words program takes the best speedup when executing with 64 partitions, and it decreases with more than that. Executing with 256 partitions takes a very similar speedup to the smallest number of partitions permitted for the file (4 and 8 for the 500MB and 1GB respectively).

In the other hand, the Word Count behaves little different for every file, but it has the best performance when executing with 16-32 partitions.

All the local executions for both programs have been made with an Intel Core i5-8259U.

Other aspects

We have made 3 different main programs, one that can execute both WordCount and CountingWords (reads the option you want to execute) and the other two execute WordCount and CountingWords separately.

Their names are:

- orchestrator.py → both
- wordcount.py → WordCount
- countword.py → CountingWords

In order to the good execution of the programs, the parameters have to be:

- First: name of the file
- Second: number of partitions

Finally, there has to be a configuration file with the following information (in the same directory where the python script is executed) in a dictionary format:

ibm_cf :
  endpoint : link to IBM Functions
  namespace : namespace
  api_key : API secret key

ibm_cos :
  endpoint : link to IBM COS
  access_key : COS access key
  secret_key : COS secret key

ibm_rabbit : link with the key to IBM RABBITAMQP

To access the GitHub repository to the project, click here.