Escola Tècnica Superior d'Enginyeria

Universitat Rovira i Virgili

Departament d'Enginyeria Informàtica i Matemàtiques

Task 1

Communication models and Middleware

Distribuited Systems

Francesc Ferré Tarrés

Cristina Ionela Nistor

Distribuited Systems

23 / 04 / 2019

Contents

1. How to use project?

This documentation is for this repository in Francesc's github:

- CescFT/Task1AMQPSistemesDistribuits: https://github.com/CescFT/Task1AMQPSistemesDistribuits

The solution of this project is using IBM Cloud: Cloud Object Storage and CloudAMQP Please make sure that you have hired these services.

The content of the repository is: the implementation of the sequential algorithm as AlgorismeSequencial.py, the connector of IBM Cloud Functions as ibm_cf_connector.py, the connector of COS backend as cos_backend.py, Orchestrator.py, different test files for testing the projects (pg10.txt, pg2000.txt and test.txt), the functions executed in IBM Cloud (wordcount, countwords and reducer) and finally we have a configuration file to put the personal credentials of IBM Cloud.

The steps to use one of the solutions are:

1. Clone the project using git:

```
$ git clone https://github.com/CescFT/ Task1AMQPSistemesDistribuits
```

NOTE: You can download this as a zip file. If you do this, you don't have to do the command in git.

2. Open a new command prompt and go to directory where you cloned or downloaded the project (if you downloaded the zip, ensure that you unzipped this!).

```
$ cd <YOUR PATH>
```

3. Now you can execute the code if you have Python v3.x. If you don't have this version of Python, please download this.
   3.1. If you want to execute sequential algorithm execute this command:[1]

```
$ AlgorismeSequencial.py <file name>
```

   3.2. If you want to execute Orchestrator.py first of all you need to modify ibm_cloud_config.txt and put here all information. Ensure that you write well all information required. Then execute this command:

```
$ Orchestrator.py <file name> <number of wordcounts>
```

[1] In Linux the command is: python3 AlgorismeSequencial.py <file name>

4. When it is executed it generates a zip file with the solution if you executed Orchestrator.py. It contains two text files resultat_final.txt that contains the final dictionary and paraules_totals.txt that contains the total words. This solution is stored in COS backend too. Otherwise when you executed sequential algorithm (AlgorismeSequencial.py) it generates two text files with the two solutions res_final_sequencial.txt have wordcount and par_totals_sequencial.txt have countwords.

2. MapReduce Architecture

The idea of the project is the implementation of MapReduce architecture that it enable the parallel processing of big data.

First you have huge amounts of data and it is broken into smaller datasets to be processed separately on different worker nodes (in this project are wordcounts) whose results are merged into only one result.

This architecture have different steps:

1. The program receive amounts of data.
2. It splits data in different parts.
3. These parts are processed in parallel (this work is performed for wordcount workers).
4. Finally partial results are merged in only one result that is shown to final user.

In the real MapReduce Architecture we have shuffing step but we implemented a simplified MapReduce Architecture and this step is not implemented.

To show the performance of this architecture, we implemented the processing of the words in a text file. The idea is generate a final dictionary with all words that contains the original text file and count all words. For example, if we have *example_file.txt*:

```
Hello Hello GoodBye Hello
```

The solution should be:

Wordcount: {“Hello”: 3, “GoodBye”: 1}.

CountWords: 4.

In this solution we have performed the steps of this architecture with:

- **Wordcount**: It represents the parallel processing of smalled datasets.
- **Reducer**: It represents the merge of the different partial solutions done by wordcount workers.
- **CountWords**: It only count the total words in the final result and show it.

The information is shown by Orchestrator.py or AlgorismeSequencial.py (explained in the following sections).

3. Sequential Implementation

3.1 Architecture explanation

This solution is performed by AlgorismeSequencial.py file. This represents a simple MapReduce Architecture with: 1 wordcount, 1 reducer and 1 countwords. The objective of this solution is see the speedup.

This file contains a very simple main that reads a text file and it calls sequentially wordcount, merger and countwords.

When the file is read, as we said above, the text is processed by wordCount function that returns a dictionary with the solution (like the example). Then, this solution is processed by reducer function that returns the merged solution.

Finally, main calls the countwords function to count all words in the text file.

The results are written in a two text files:

- ***res_final_sequencial.txt***: This text file contains the result of work performed by wordcount and reducer functions.

- ***par_totals_sequencial.txt***: This text file contains the result of work performed by countwords function.

3.1.1 WordCount Implementation

```
def wordCount(text):
    global temps_total
    ti=time.time()
    diccionari={}
    words=""
    punctuation2 = "['-]"
    regex2 = r"(\s)"

    for value in text:
        if re.search(regex2, value):
            value = ' '
            words = words + value
        else:
            if not re.search(punctuation2, value) and
            re.search('['+string.punctuation+']', value):
                value = ' '
            words = words + value

    words = words.split(' ')
    for par in words:
        if par != "":
            if par in diccionari:
                diccionari[par]=diccionari.get(par)+1
            else:
                diccionari[par]=1
    tf=time.time()
    dif=tf-ti
    temps_total=temps_total + dif
    return diccionari
```

This is the code of wordcount function. As you can see it receive the content of the file (all words read) and it process all words. As you can see, this function have regular expressions that help us to have good solution of wordcount:

- `r"(\s)"` : This regular expresion matches whitespaces characters, which includes the \t, \n, \r and space characters.
- `"['-]"` : This regular expressions pretends delete all punctuation marks except "'" and "-".

When text is filtered, it generates a list separating words with spaces. So, if text is "hello hello", this list have: ["hello", "hello"].

Then, this list is treated and returns a dictionary with words as keys and the aparences of that words in the text as value, explained above.

3.1.2 Reducer Implementation

```
def merger(dicc2merge, final):
    global temps_total
    ti=time.time()
    for key in dicc2merge:
        if key in final:
            final[key]=final.get(key)+dicc2merge.get(key)
        else:
            final[key]=dicc2merge.get(key)
    tf=time.time()
    dif= tf - ti
    temps_total+=dif
```

The code of the reducer is very simple. It have two parameters. The first parameter is the dictionary to merge in the solution and second parameter have the final result. The core of this function handle the words that receive in the dictionary to join. It checks if the words are in the final result. If so, the function add in final result the previous counter and the counter of the dictionary that the function receive.

Otherwise, the function store appears of the words in the dictionary that it receive as a parameter.

3.1.3 CountWords Implementation

```
def countWords(merged):
    global temps_total
    ti=time.time()
    cont=0

    contador=0
    keys=merged.keys()
    for val in keys:
            contador+=merged[val]

    tf=time.time()
    dif=tf - ti
    temps_total+= dif
    return contador
```

The code of countwords is very simple. It receives merged solution and counts the words in this dictionary retrieving first keys (words in merged dictionary). Then it only adds appearances (values) of all words in merged solution.

3.1.4 Other funcions

```
def esborrar_pantalla():
    os.system("cls")
```

Function **esborrar_pantalla()** is very simple. It only clears command prompt to see more clearly the text shown on it.[2]

```
def inicialitzar_carpeta():
    resfinaltxt='res_final_sequencial.txt'
    respartotalstxt='par_totals_sequencial.txt'
    if os.path.exists(resfinaltxt):
        os.remove(resfinaltxt)
    if os.path.exists(respartotalstxt):
        os.remove(respartotalstxt)
```

Function **inicialitzar_carpeta()** prepares your execution directory. The idea of this function is no accumulate solutions and if you runned previously the program it removes old solutions. If not, it don't do nothing.

4. Implementation of Orchestrator solution

4.1 Architecture explanation

The repository that you could find this solution is here. This represents a simple MapReduce Architecture with: N wordcount, 1 reducer and 1 countwords.

The solution is performed using Orchestrator.py, cos_backend.py (library that connects code with IBM Cloud Object Storage), ibm_cf_connector.py (library that connects code with IBM Cloud Functions), reducercountwordsAMQP.zip, wordCountAMQP.zip and finally ibm_cloud_config.txt that you need to fill up with your credentials.

So, the first step you have to do is fill up this file that right now have this information:

```
ibm_cf:
    endpoint     : <YOUR ENDPOINT>
    namespace    : <YOUR NAMESPACE>
    api_key      : <YOUR API_KEY>

ibm_cos:
    endpoint    : <YOUR ENDPOINT>
    access_key : <YOUR ACCESS_KEY>
    secret_key : <YOUR SECRET_KEY>
amqp:
    url  : <YOUR RABBITMQURL>
```

[2] If you run this program in Linux, change os.system("cls") to os.system("clear").

NOTE: Ensure that you have good verion of pika because of IBM Cloud uses a specific version of this python module.

The user only works with Orchestrator.py that monitors all movements of wordcount workers, reducer worker and countwords worker.

This file contains a main code that receive the file that you want to read and the number of wordcount workers, hence is the number of splits of the information inside the file.

It splits the file following next idea:

We have number of partitions and the size of the file (previously saved in COS backend) so we divided this size with number of partitions. The result is the part that corresponds to process a wordcount.

For example if we have a file which size is 10MB and user puts 2 wordcounts. With this algorism first word count will process [0-5]MB and the second will process [6-10]MB.

This moment we check if word is splitted. We use another time one regular expression explained above which matches whitespaces characters (\s) and this is used except the last time. With this regular expression Orchestrator knows where are whitespaces and if we have whitespaces have calculated maximum bytes of partition and it is sent to get_object function from cos_backend.py with minimum and this part of bytes of file is sent to wordcount worker. To treat special characters, we decode bytes with latin1 charset.

At the same time Orchestrator is invoking all wordcounts. When it is completed, Orchestrator invokes only one reducer too and it is locked waiting that reducer worker sends a message to Orchestrator to unlock it.

Finally, when the execution is finished and the final results are stored in IBM Cloud Object Storage, Orchestrator retrieves the solutions and store this solutions in a zip file. So, the final solutions are in IBM Cloud Object Storage and in your computer.

4.1.1 Orchestrator Implementation

The basic idea of Orchestrator is explained above, but in this section we will explain with more detail.

As we said previously, it reads parameters that you put in command prompt (see How to use project).

Then, if you insert well parameters it reads ibm_cloud_config.txt and generates three dictionaries. The first one is with credentials for IBM Cloud Object Storage, the second one is with credentials of IBM Cloud Functions and the other dictionary is with credentials of RabbitMQ. With this dictionaries initialize an object to communicate Orchestrator with IBM Cloud Object Storage and other object to communicate Orchestrator with IBM Cloud Functions, using library cos_backend.py and ibm_cf_connector.py.

The other one is for generate a queue communication between reducer and Orchestrator.

The next step is initializations of IBM Cloud and your computer and divide data with smaller parts which will process the different word counts invoking at the same time this functions.

NOTE: When we call functions implemented in cos_backend.py we pass our bucket name. Change this for your bucket name.

Then, Orchestrator waits that reducer have the solution. This is achieved with this code in main:

```
        url = amqp.get("url")
        params = pika.URLParameters(url)
        connection = pika.BlockingConnection(params)
        channel = connection.channel() # start a channel
        channel.queue_declare(queue='orchestrator')
        channel.basic_consume(callback, queue='orchestrator', no_ack=True)
        channel.start_consuming()
```

With this code, Orchestrator is locked waiting that IBM Cloud finish doing work (wordcount, reducer and countwords).

The code that Orchestrator executes in "lock" period is:

```
def callback(ch, method, properties, body):
    string=""
    for i in body:
        if i < 128:
            val=str(chr(i))
            string+=val
        else:
            lletra=tractar_lletra(i)
            string+=lletra
    if string == "stop":
        channel.stop_consuming()
```

Here we see function **callback(ch,method, properities, body)**, we only use the final argument, body. In body we have the message that sends reducer. When reducer says "stop", callback function automatically unlock Orchestrator and it can continue the execution.

Finally, generates a zip file with the final results stored in IBM Cloud Object Storage and delete queues used in this solution.

Orchestrator.py file contains auxiliary functions that we explain here:

```
def generar_funcions(nom_zip, invocador, nom_funcio):
    with open(nom_zip, 'rb') as funcio:
        bytes_f=funcio.read()
    v=invocador.create_action(nom_funcio, bytes_f, 'blackbox',
'ibmfunctions/action-python-v3.6', is_binary=True, overwrite=True)
    return v
```

Function **generar_funcions(nom_zip, invocador, nom_funcio)** is responsible of the generation of functions in IBM Cloud Functions.

As you can see it receives the name of zip file which contains the code of function, the function invocator (an object of CloudFunctions implemented in ibm_cf_connector.py) and the name of function in IBM Cloud Functions.

The code is very simple, it reads a zip file in binary mode and creates action calling **create_action()** which code is implemented in ibm_cf_connector.py.

```
def inicialitzar_objectStorage(storage_backend, elem=None):
    ll=storage_backend.list_objects('contenidortask1', elem)
    if len(ll) != 0:
        for i in ll:
                nomFitx=i['Key']
                storage_backend.delete_object('contenidortask1',
nomFitx)
```

Function **inicialitzar_objectStorage(storage_backend, elem=None)** is responsible of the inicialization of IBM Cloud Object Storage. The idea of this function is delete objects generated in previous executions. If not, this don't delete nothing.

```
def ini_cloud_dades(object_storage):
    inicialitzar_objectStorage(object_storage, 'wordcount')
    inicialitzar_objectStorage(object_storage, 'countwords')
    inicialitzar_objectStorage(object_storage, 'resultat_finalAMQP.txt')
```

Function **ini_cloud_dades(object_storage)** calls the previoous function to initialize the IBM Cloud Object Storage.

```
def ini_cloud_funcions(invo_funcions):

    tot_be=generar_funcions('wordCountAMQP.zip',invo_funcions,
'wordCountAMQP')
    if tot_be == 1:
        esborrar_pantalla()
        print('Sembla ser que no hi ha el fitxer wordCount.zip, no
podem procedir a executar res. Abortar.')
        return 1
    tot_be=generar_funcions('reducercountwordsAMQP.zip',invo_funcions,
'reducer-countwordsAMQP')
    if tot_be == 1:
        esborrar_pantalla()
        print('Sembla ser que no hi ha el fitxer reducer.zip, no podem
procedir a executar res. Abortar.')
        return 1
    return 0
```

Function **ini_cloud_funcions(invo_funcions)** generates functions calling the previous fuction. If you don't have function as a zipped files it return 1 and Orchestrator stops the execution beacause of you don't have wordCount.zip or reducer.zip or countWords.zip.

```
def esborrar_pantalla():
    os.system("cls")
```

Function **esborrar_pantalla()** is very simple. It only clears command prompt to see more clearly the text shown on it.[3]

[3] If you run this program in Linux, change os.system("cls") to os.system("clear").

```
def ini_repositori_local(opcio):
    resfinaltxt='resultat_final.txt'
    respartotalstxt='paraules_totals.txt'
    zip_res_final='resultats_OrchestratorCues_Cristina_Francesc.zip'
    if opcio == 'inici':
        if os.path.exists(resfinaltxt):
            os.remove(resfinaltxt)
        if os.path.exists(respartotalstxt):
            os.remove(respartotalstxt)
        if os.path.exists(zip_res_final):
            os.remove(zip_res_final)
    elif opcio == 'final':
        if os.path.exists(resfinaltxt):
            os.remove(resfinaltxt)
        if os.path.exists(respartotalstxt):
            os.remove(respartotalstxt)
```

Function **ini_repositori_local(opcio)** only removes previous results if you executed previously Orchestrator.

```
def netejarObjectStorage(cloudOS):
    ll=cloudOS.list_objects('contenidortask1', 'wordcount')
    for diccinfo in ll:
        key=diccinfo['Key']
        cloudOS.delete_object('contenidortask1', key)
```

Function **netejarObjectStorage(cloudOS)** deletes partial results of wordcount workers and this involve that in IBM Cloud Object Storage only will have final results.

4.1.2 WordCount Implementation

The implementation of word count function now is represented in a zip file called "wordCountAMQP.zip".

This zip contains: __main__.py, cos_backend.py and WordCount.py.

The first file is the main program of the function implemented in WordCount.py and is executed when in Orchestrator we do this:

```
params = {'credentials':conf['ibm_cos'], 'text':decoded_text,
'index':str(n+1), "url":conf['amqp']}
functionInvocator.invoke('wordCountAMQP', params)
```

With this line in Orchestrator's main invokes the function in IBM Cloud Functions and the code executed by IBM is first __main__.py which contains main code of function. The code is very simple, it only gets parameters that we passed in the line of Orchestrator's main and calls the function wordCount implemented in WordCount.py zipped in wordCountAMQP.zip.

It incorpores code to use queue which service is provided by RabbitMQ. The code that makes this possible is:

```
url = amqp['url']
params = pika.URLParameters(url)
connection = pika.BlockingConnection(params)
channel = connection.channel()
channel.queue_declare(queue='hello')
message='wordcount'+str(index)+'.txt'
print (message)
channel.basic_publish(exchange='', routing_key='hello', body=message)
connection.close()
```

With this code in __main__.py wordcount worker sends a message to reducer when it have partial result. It sends the name of file that contains partial result.

The code is very similar with the wordcount function in AlgorismeSequencial.py.

When wordcount function finalized, __main__.py store partial result in IBM Cloud Object Storage.

4.1.4 Reducer - Countwords Implementation

We decided to generate two functions in only one zip. This function is zipped in "reducercountwordsAMQP.zip".

This file contains: __main__.py, cos_backend.py, CountWords.py and reducer.py.

As we said in previous sections in __main__.py file we have the main of the function that is the first that IBM Cloud Function executes. This is executed when in Orchestrator's main we invoke this function and cloud invoke this. The line that invoke reducer function is:

```
functionInvocator.invoke('reducer-countwordsAMQP',
{"credentials":conf['ibm_cos'], "url":amqp, "num_maps":num_maps})
```

First of all, in __main__.py we start consuming of the queue that wordcounts sends the name of partial result. When reducer obtains this result it consumes this message, recollects partial result with the name sent by wordcount getting object stored in IBM Cloud Object Storage and we proceed to change bytestring to string and it change type to dictionary for reduce this partial results to only one result and saves this partial dictionary appending it in a list.

When IBM Cloud Object Storage have all partial results, it stops consuming and executes reducer and countwords, in this order.

When reducer stops consuming, the list that have partial dictionaries is passed inside a dictionary to reducer function implemented in reducer.py.

Here it retrieve the list from input dictionary called args and just execute the same code like AlgorismeSequencial.py.

As we said before, then is executed countwords code that is very equal with the code of AlgorismeSequencial.py.

When this all work is done, reducer stop consuming of the queue that communicates it and wordcounts and make a message to orchestrator with the purpose of unlock it. Then, Orchestrator can download final results and zip this results. This is the line that unlocks orchestrator:

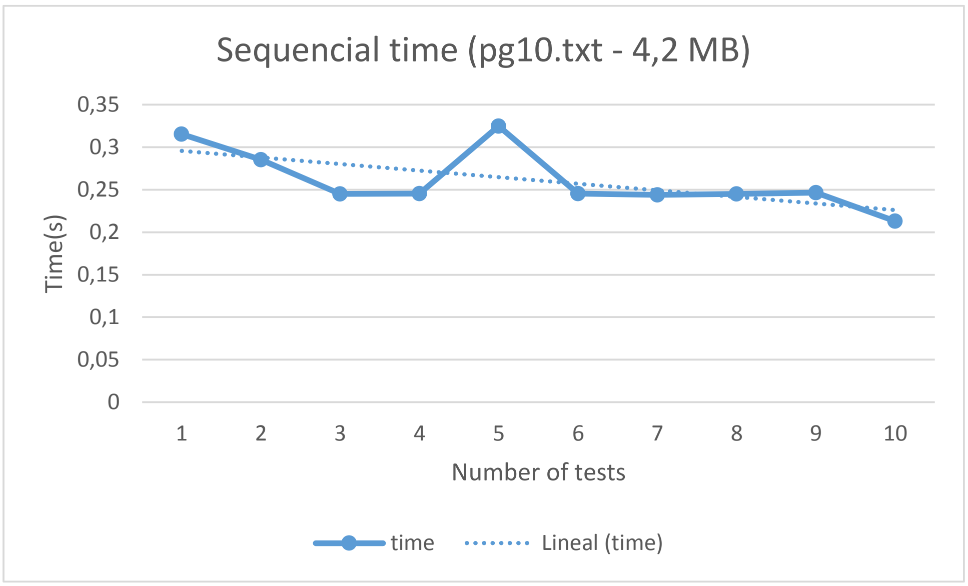
```
channel.basic_publish(exchange='', routing_key='orchestrator', body='stop')
```

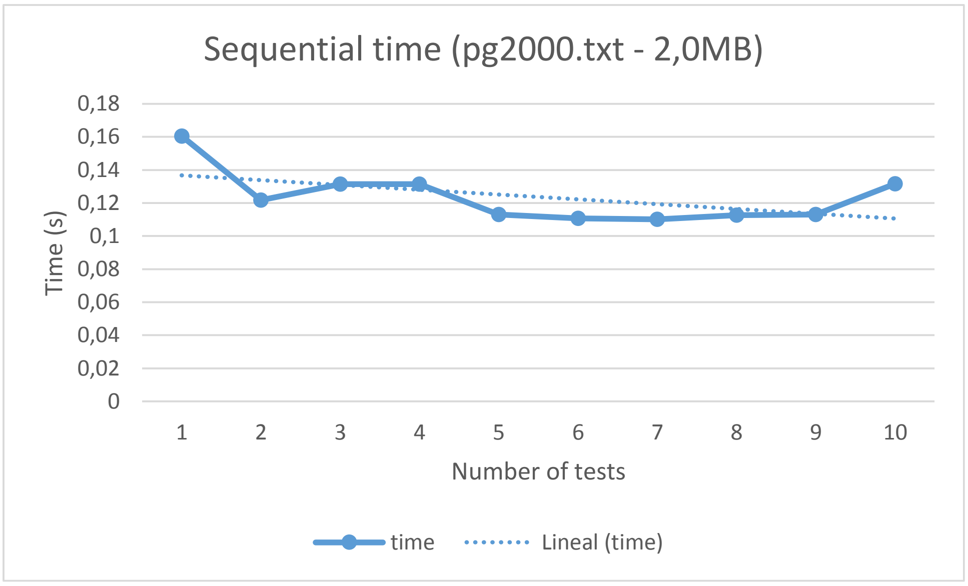
Finally, main function close connection and return.

5. Speed-up, graphics and conclusions

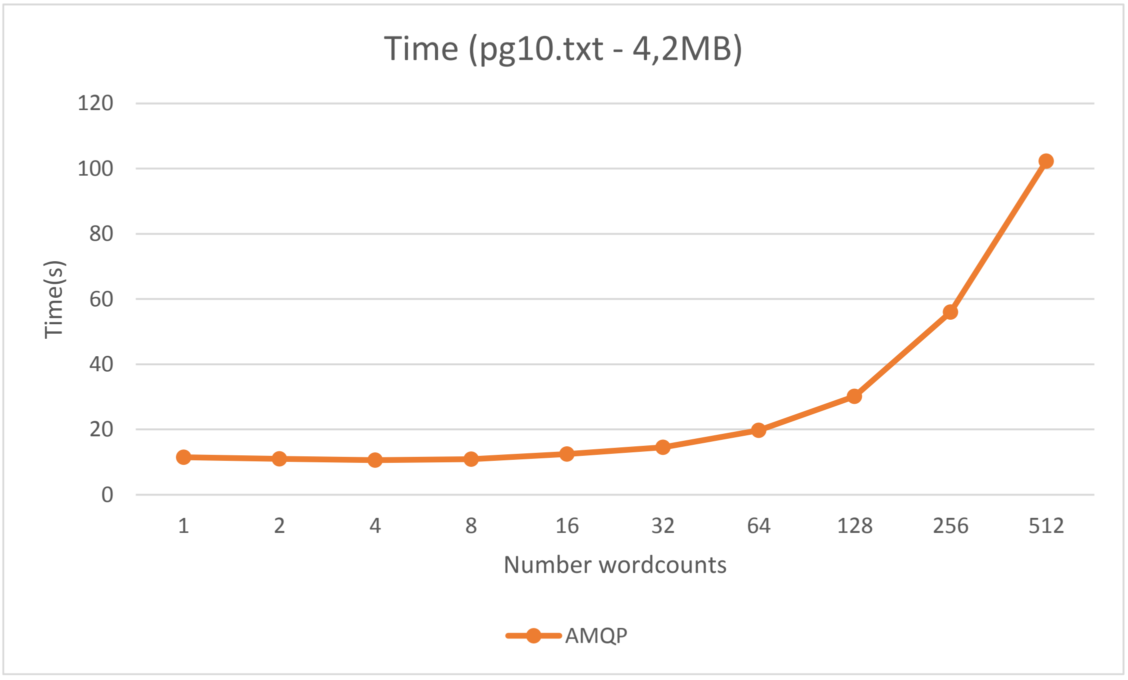
We tested our solutions with files: pg10.txt, pg2000.txt and test.txt, but we make graphics executing sequential solution with pg10.txt (4,2 MB) and pg2000.txt (2,0MB) and this are the results.

We tested 10 times three solutions to have more accurate results. To test Orchestrator we used different number of wordcounts (1,2,4,8,16,32,64,128,256,512).

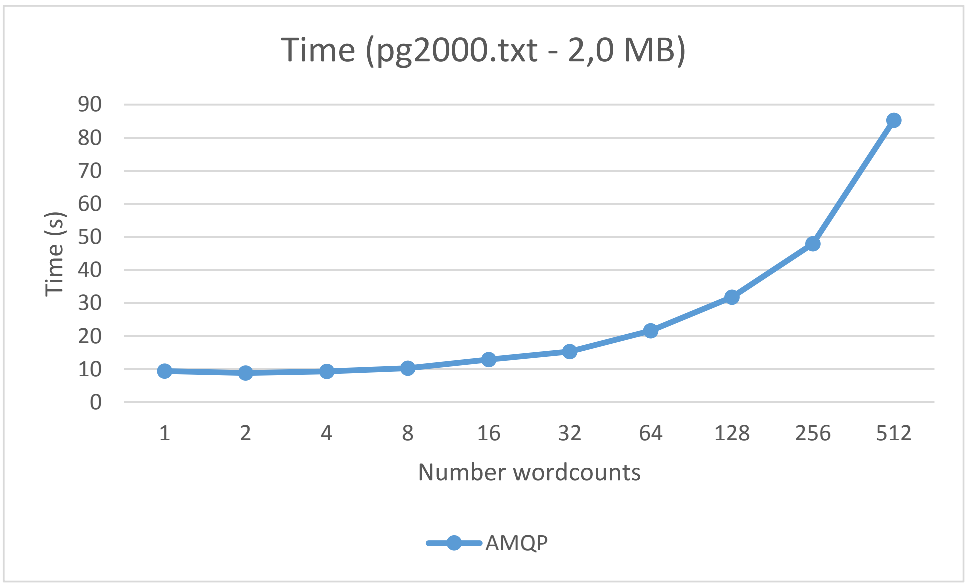




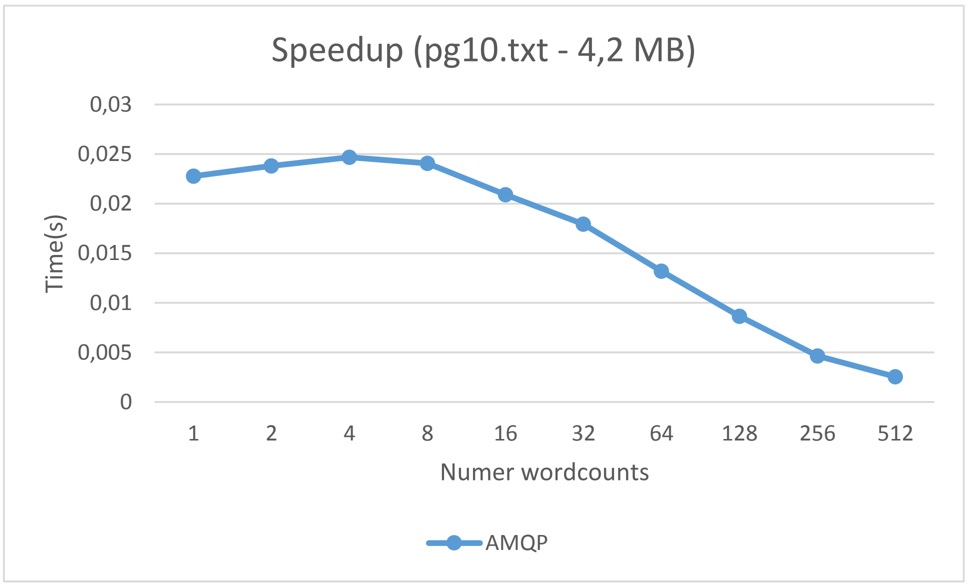
As you can see, the differences between these two graphics are that with more information it cost more to have solution. Specifically, as we can see, a file weighs twice as much as the other, which causes the processing time to be double the one of the other. However, this conclusion alone does not say anything because it depends primarily on the architecture of the computer where it is executed.



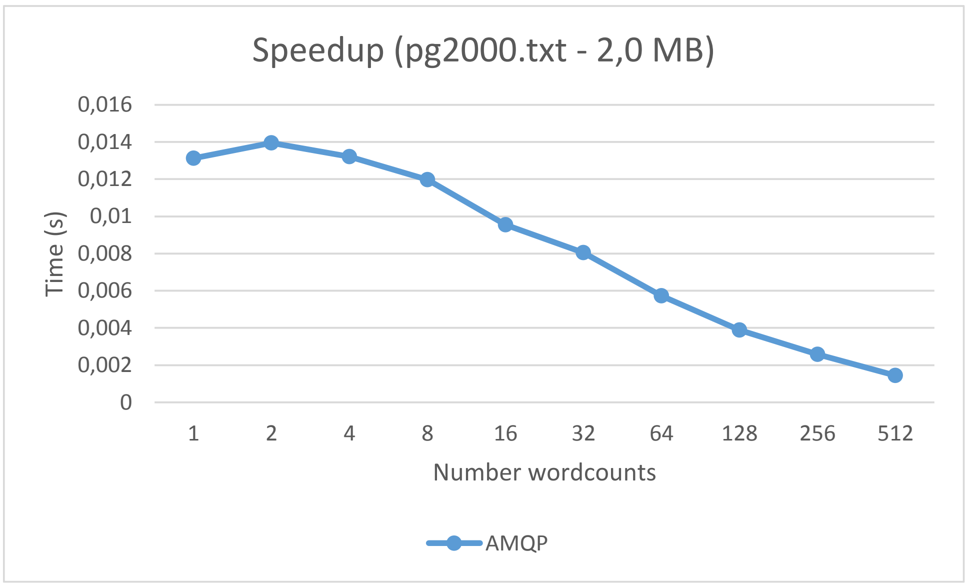
This graphic is time using AMQP solution with pg10.txt file (4,2 MB).



This graphic represents time using AMQP solution with pg2000.txt file (2,0 MB).



This graphic represents speed-up AMQP solution using pg10.txt (4,2MB).



This graphic represents the speed-up AMQP solution using pg2000.txt (2,0MB).

To conclude, as you can see when we execute Orchestrator with 32 - 64 wordcounts it begins to cost. It can be seen in speed-up graphics.

The fact that the wordcount are working in parallel, increments possibility that more than one wordcount attempt to send a message to queue. It means that there are many requests at the same time and it can create a bottle neck.

We think that the use of RabbitMQ, is more slow than no use this independently of file size.

Invoking a lot of functions at the same time causes a high computational load and wordcount workers are consuming limited resources.

Therefore, as long as we try to parallel the solution invoking many workers at the same time these are occupying limited resources that can be completed and until some resources are released, another wordcount can’t be executed.

Finally, we observed that more than 6 MB size file text can explodes because of we have a limit of size that wordcount can handle. So, we can’t know if the execution of file which size is more than 6MB Orchestrator finish well for the previous reason.