Task 1: Communication models and Middleware

ALEIX SANCHO I ERIC PI

23-04-2019

Índex

1. Arquitectura

1.1. Versió seqüencial

1.1.1. Orchestrator seqüencial

- count_words

```python
def count_words(dicc):
    num_words = 0
    words = dicc.keys()

    for word in words:
        num_words = num_words + dicc[word]

    return {"Number of words": num_words}
```

La funció count_words rep per paràmetre el diccionari final, i mitjançant la funció keys recorrem tot el diccionari sumant el nombre de vegades que es repeteix cadascuna de les paraules.

- Reducer

```python
def reducer(dicc):
    dictionary = {}
    for word in dicc:
        if word in dictionary:
            dictionary[word] = dictionary.get(word) + words.get(word)
        else:
            dictionary[word] = dicc.get(word)

    return dictionary
```

La funció reducer rep per paràmetre el diccionari solució de la funció word_count, i el que fem és passar aquest diccionari al diccionari final el qual passarem a la funció count_words. En el codi seqüencial realment no té una funció essencial però volem que tant la versió seqüencial com la versió paral·lelitzada siguin com més semblants millor.

- word_count

```python
def word_count(dicc):
    words = ""
    punctuation2 = "['-]"
    regex2 = r"(\s)"

    for value in dicc:
        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(' ')

    dictionary = {}
    for word in words:
        if not word == "":
            if word in dictionary:
                dictionary[word] = dictionary.get(word) + 1
            else:
                dictionary[word] = 1

    return dictionary
```

La funció word_count rep per paràmetre un diccionari que conté tot el text el qual volem analitzar lletra per lletra i tractem amb la llibreria re que la lletra no sigui un caràcter d'espai en blanc, si ho és, el substituïm per un espai en blanc. A més si aquest caràcter és un caràcter especial, convertim aquest caràcter en un espai i obtenim dues paraules diferents respectant els apòstrofs i els guions.

Mitjançant el mètode split s'aconsegueix separar les diferents paraules per espais i així tractar aquesta cadena com una cadena de paraules independents.

Per últim, per tota la cadena creem un diccionari relacionant cada paraula amb el nombre de cops que surt en el text donat.

- main

```
if __name__ == '__main__':
    if len(sys.argv) == 2:
        file = open(sys.argv[1])
        dictionary = word_count(file.read('r'))
        word_number = count_words(dictionary)
        final_dictionary = reducer(dictionary)
        f1 = open('resultat_count_words_sequencial.txt', 'w+')
        f1.write(str(word_number))
        f2 = open('resultat_reducer_sequencial.txt', 'w+')
        f2.write(str(final_dictionary))
        f1.close()
        f2.close()
    else:
        print("Necessites introduir dos parametres")
```

La funció main s'encarrega de llegir el fitxer que volem tractar i cridem les tres funcions esmentades anteriorment, el word_count, el count_words i el reducer.

Per últim guardem el resultat de les dues funcions, word_count i count_words als fitxers resultat_count_words_sequencial.txt i resultat_reducer_sequencial.txt.

1.2. Versió paral·lelitzada

1.2.1. Introducció

El projecte tracta de realitzar una versió de l'arquitectura del Map Reduce aplicant-ho a la solució del problema que se'ns ha donat. El problema tracta de, donat un text i un nombre de particions, hem de relacionar el nombre de cops que apareix cada paraula en aquest text i dividir aquesta tasca en diferents processos per tal de paral·lelitzar aquesta feina. Un cop totes les particions han estat realitzades unim la informació extreta de cadascuna de les particions ajuntant-t'ho així en una única solució final.

Per tal de realitzar aquesta tasca utilitzarem tres funcions principals:

- word_count: S'encarrega de fer un diccionari relacionant cada paraula que apareix en un segment de text i el nombre de cops que apareix.

- reducer: S'encarrega d'unir els resultats de les diferents particions i posar-los tots en una mateixa solució final(un únic diccionari).

- count_words S'encarrega de contar el nombre de paraules totals que hi ha en el text donat. Per facilitar la feina utilitzarem la solució del reducer.

1.2.2. Orchestrator

```
if __name__ == '__main__':
    if len(sys.argv) == 3:
        with open('ibm_cloud_config.txt', 'r') as config_file:
            res = yaml.safe_load(config_file)

        config_file = res['ibm_cos']
        config_file_cloud = res['ibm_cf']
        invocador = CloudFunctions(config_file_cloud)
        cos_backend = COS_backend(config_file)
```

L'orchestrator té una única funció que és el main, i necessitarà tres arguments d'entrada per funcionar (nom del codi que executa, el text a analitzar i el nombre de particions que volem). A l'inici del main farem la validació del nombre d'arguments d'entrada i carregarem l'invocador per poder crear accions i invocar-les al cloud, per altra banda també carregarem el cos_backend que ens permet obtenir les dades guardades al nostre contenidor o guardar-hi dades.

```
    ll = cos_backend.list_objects("sanchoericbucket", "count_words")

    for elem in ll:
        file = elem["Key"]
        cos_backend.delete_object("sanchoericbucket", file)

    ll = cos_backend.list_objects("sanchoericbucket", "reducer")

    for elem in ll:
        file = elem["Key"]
        cos_backend.delete_object("sanchoericbucket", file)

    ll = cos_backend.list_objects("sanchoericbucket", "word_count_")

    for elem in ll:
        file = elem["Key"]
        cos_backend.delete_object("sanchoericbucket", file)
```

Per assegurar-nos del bon funcionament del programa, eliminem totes les solucions existents guardades al cloud.

```
with open('count_words.zip', 'rb') as count_words:
    codi = count_words.read()

invocador.create_action('count_words', codi)

with open('reducer.zip', 'rb') as reducer:
    codi = reducer.read()

invocador.create_action('reducer', codi)

with open('word_count.zip', 'rb') as word_count:
    codi = word_count.read()
invocador.create_action('word_count', codi)
```

Definim les tres accions que necessitarem invocar més tard. Utilitzem ‘rb’ en obrir els fitxers en zip, ja que així obtindrem els arxius binaris.

```
num_workers = int(sys.argv[2])
index = 0
tamany_totalB = cos_backend.head_object('sanchoericbucket', sys.argv[1])
mida = tamany_totalB['content-length']
tamany_particionat = int(mida) / num_workers
inici_bytes = int(tamany_particionat * index)
regex = r"(\s)"
initial_time = time.time()
```

Obtenim el número de workers i la mida de l’arxiu per més tard preparar les particions del text a analitzar.

```
while index < num_workers:
    ko = True
    offset = 1
    final_bytes = int(tamany_particionat * (index + 1))
    if index != (num_workers - 1):
        while ko:
            text = cos_backend.get_object('sanchoericbucket', sys.argv[1], extra_get_args =
            {'Range': 'bytes=' + str(final_bytes) + '-' + str(final_bytes)})
            if not re.search(regex, str(text)):
                final_bytes = final_bytes + 1
                offset = offset + 1
            else:
                ko = False
    else:
        final_bytes = int(mida)

    text = cos_backend.get_object('sanchoericbucket', sys.argv[1], extra_get_args =
    {'Range': 'bytes=' + str(inici_bytes) + '-' + str(final_bytes)})
    decoded_text = text.decode("latin1")
    params = {'config_file':config_file, 'text':decoded_text, 'index':index}
    invocador.invoke('word_count', params)
    index = index + 1
    inici_bytes = int((tamany_particionat * index) + offset)

tots_acabats = False
```

Per a cada partició hem de dividir el text en les diferents parts i obtenim els bytes corresponents del fitxer penjat a l'ibm cloud. Per repartir aquests bytes hem decidit que l'últim byte de cada partició ha de correspondre a un espai en blanc, així doncs les paraules no queden dividides (tasca opcional 1). Un cop fet això preparem els arguments d'entrada per al word_count i invoquem la funció.

```
while not tots_acabats:
    llista_objectes = cos_backend.list_objects('sanchoericbucket', 'word_count')
    if len(llista_objectes) == num_workers:
        tots_acabats = True
```

En la versió simple per assegurar-nos que totes les particions han acabat comprovem que el nombre d'arxius amb el prefix word_count és igual al nombre de particions.

```
params = {'config_file':config_file}
invocador.invoke('reducer', params)
tots_acabats = False

while not tots_acabats:
    llista_objectes = cos_backend.list_objects('sanchoericbucket', 'reducer')
    if len(llista_objectes) == 1:
        tots_acabats = True
```

Un cop el word_count ha acabat per a totes les particions invoquem la funció reducer i ens esperem que obtinguem la solució.

```
    invocador.invoke('count_words', params)
    tots_acabats = False

    while not tots_acabats:
        llista_objectes = cos_backend.list_objects('sanchoericbucket', 'count_word')
        if len(llista_objectes) == 1:
            tots_acabats = True
```

Fem el mateix per al count_words.

```
      final_time = time.time()
      total_time = final_time - initial_time
      print(total_time)
      objecte_count_words = cos_backend.get_object('sanchoericbucket', 'count_words.txt')
      objecte_decoded_count_words = objecte_count_words.decode('utf-8')
      objecte_reducer = cos_backend.get_object('sanchoericbucket', 'reducer.txt')
      objecte_decoded_reducer = objecte_reducer.decode('utf-8')
      f1 = open('resultat_count_words.txt', 'w+')
      f1.write(objecte_decoded_count_words)
      f2 = open('resultat_reducer.txt', 'w+')
      f2.write(objecte_decoded_reducer)
      f1.close()
      f2.close()
  else:
      print("Necessites introduir dos parametres")
```

Per últim capturem el temps que ha trigat a obtenir el resultat i obtenim els fitxers que són solució i hi guardem dins la solució.

1.2.3. word_count

```
def main (args):
    config = args.get("config_file")
    text = args.get("text")
    index = args.get("index")
    cos_backend = COS_backend(config)
    #carreguem el text que volem passar per parametre al word_count
    params_word_count = {'words':text}
    resultat_word_count = word_count(params_word_count)
    cos_backend.put_object('sanchoericbucket', 'word_count_' + str(index) + '.txt', str(resultat_word_count))
    result = {'resultat':'be'}
    return result
```

Preparem els arguments d'entrada que necessitarem per tal d'executar la funció word_count. Necessitarem les credencials per inicialitzar el cos_backend, el text i l'índex per saber quin número de partició s'ha d'executar i per tant quin número s'ha de posar al fitxer resultat.

```python
def word_count(args):
    content = args.get('words')
    words = ""
    punctuation2 = "['-]"
    regex2 = r"(\s)"

    for value in content:
        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(' ')
    dictionary = {}
    for word in words:
        if not word == "":
            if word in dictionary:
                dictionary[word] = dictionary.get(word) + 1
            else:
                dictionary[word] = 1

    return dictionary
```

Utilitzem el mateix codi per la funció que hem utilitzat a la versió seqüencial.

1.2.4. reducer

```python
def main (args):
    config = args.get("config_file")
    cos_backend = COS_backend(config)
    llista_txt = cos_backend.list_objects('sanchoericbucket', 'word_count')
    llista_final = []
    for i in llista_txt:
        nom = i['Key']
        objecte = cos_backend.get_object('sanchoericbucket', nom)
        cadena = objecte.decode('utf-8')
        dicc_entrada = literal_eval(cadena)
        llista_final.append(dicc_entrada)

    diccionari_entrada = {'all':llista_final}
    resultat_reducer = reducer(diccionari_entrada)
    cos_backend.put_object('sanchoericbucket', 'reducer.txt', str(resultat_reducer))
    result = {'resultat':'be'}

    return result
```

Preparem els arguments d'entrada que necessitarem per tal d'executar la funció reducer. Necessitarem les credencials per inicialitzar el cos_backend, i la llista de diccionari resultant de les funcions word_count executades.

Unim en una única llista el resultat dels diferents diccionaris per tal d'unificar tots els resultats i així introduir-ho com a paràmetre d'entrada a la funció reducer.

```
def reducer(args):
    content = args.get('all')
    dictionary = {}
    for words in content:
        for word in words:
            if word in dictionary:
                dictionary[word] = dictionary.get(word) + words.get(word)
            else:
                dictionary[word] = words.get(word)

    return dictionary
```

La funció reducer captura totes les paraules contingudes en el mapa que passem per paràmetre i les unifica en un diccionari, aquest serà la solució final del count_words.

1.2.5. count_words

```
def main (args):
    config = args.get("config_file")
    cos_backend = COS_backend(config)
    final_reducer = cos_backend.get_object('sanchoericbucket', 'reducer.txt')
    final_decode = final_reducer.decode('utf-8')
    diccionari_reducer = literal_eval(final_decode)
    diccionari_entrada = {'all':diccionari_reducer}
    resultat = count_words(diccionari_entrada)
    cos_backend.put_object('sanchoericbucket', 'count_words.txt', str(resultat))
    result = {'resultat':'be'}

    return result
```

Preparem els arguments d'entrada que necessitarem per tal d'executar la funció count_words. Necessitarem les credencials per inicialitzar el cos_backend, i la solució de la funció reducer.

Utilitzarem aquesta última com argument d'entrada de la funció count_words i per últim posarem el resultat de la funció al cloud.

```python
def count_words(args):
    content = args.get('all')
    num_words = 0
    words = content.keys()

    for word in words:
        num_words = num_words + content[word]

    return {"Number of words": num_words}
```

La funció count_words utilitza la solució del reducer en forma de string i fa un diccionari amb una única clau on guarda el nombre de paraules totals que hi ha en el text.

1.3. Versió amb cues (Tasca opcional 2)

Les variacions que necessitem per fer aquesta tasca són:

1.3.1. ibm_cf_connector.py

```
queue_sd:
    amqp : amqp://
```

Hem d'afegir l'URL de la cua per tal que ens funcioni.

1.3.2. orchestrator.py

```python
config_file = res['ibm_cos']
config_file_cloud = res['ibm_cf']
amqp_url = res['queue_sd']
invocador = CloudFunctions(config_file_cloud)
cos_backend = COS_backend(config_file)
```

Capturem les credencials del rabbit mq.

```python
def callback(ch, method, properties, body):
    message = body.decode('latin1')
    if message == "stop":
        channel.stop_consuming()
```

Aquesta funció serveix per fer l'espera mentre el word_count, reducer i count_words no hagin finalitzat.

```
        params = {'config_file':config_file, 'amqp':amqp_url, 'num_workers':num_workers}
        invocador.invoke('reducer', params)
        url = amqp_url.get("amqp")
        params = pika.URLParameters(url)
        connection = pika.BlockingConnection(params)
        channel = connection.channel()
        channel.queue_declare(queue='message_queue')
        channel.basic_consume(callback, queue='message_queue', no_ack=True)
        channel.start_consuming()

        final_time = time.time()
        total_time = final_time - initial_time
        print(total_time)
        objecte_count_words = cos_backend.get_object('sanchoericbucket', 'count_words_amqp.txt')
        objecte_decoded_count_words = objecte_count_words.decode('utf-8')
        objecte_reducer = cos_backend.get_object('sanchoericbucket', 'reducer_amqp.txt')
        objecte_decoded_reducer = objecte_reducer.decode('utf-8')
        f1 = open('resultat_count_words_amqp.txt', 'w+')
        f1.write(objecte_decoded_count_words)
        f2 = open('resultat_reducer_amqp.txt', 'w+')
        f2.write(objecte_decoded_reducer)
        f1.close()
        f2.close()
    else:
        print("Necessites introduir dos parametres")
```

Establim una connexió amb el pika, declarem una cua que permetrà la comunicació entre reducer i orchestrator. Es comença a consumir els missatges de la cua i no et desbloqueges fins que el reducer no t'envia el senyal.

1.3.3. reducer

```
def main(args):
    global amqp_url
    global connection
    global channel
    global num_workers
    global cos_backend
    global dictionaries
    dictionaries = []
    config = args.get("config_file")
    amqp = args.get("amqp")
    amqp_url = amqp.get('amqp')
    num_workers = args.get("num_workers")
    cos_backend = COS_backend(config)
    params = pika.URLParameters(amqp_url)
    connection = pika.BlockingConnection(params)
    channel = connection.channel()
    channel.queue_declare(queue = 'word_count_queue')
    channel.queue_declare(queue = 'message_queue')
```

Genero les variables globals per tal que la funció on_message pugui obtenir els valors d'aquestes variables.

```
    def on_message(channel, method_frame, header_frame, body):
        channel.basic_ack(delivery_tag = method_frame.delivery_tag)
        finish = 0
        num_files = cos_backend.list_objects('sanchoericbucket','word_count_')
        if len(num_files) == num_workers:
            finish = 1
        else:
            finish = 0

        if finish == 1:
            id_file = 0
            while id_file < num_workers:
                file_dicc = num_files[id_file]
                name_file = file_dicc['Key']
                dictionary = cos_backend.get_object('sanchoericbucket', name_file)
                dictionary = dictionary.decode('utf-8')
                dictionary = literal_eval(dictionary)
                dictionaries.append(dictionary)
                id_file = id_file + 1

            params = {'all':dictionaries}
            final_dicc = reducer(params)
            cos_backend.put_object('sanchoericbucket','reducer_amqp.txt', str(final_dicc))
            params = {'all':final_dicc}
            num_words = count_words(params)
            cos_backend.put_object('sanchoericbucket', 'count_words_amqp.txt', str(num_words))
            channel.stop_consuming()

    channel.basic_consume(on_message, queue = 'word_count_queue')
    channel.start_consuming()
    channel.basic_publish(exchange = '', routing_key = 'message_queue', body = 'stop')
    connection.close()
    result = {'resultat':'be'}

    return result
```

Consisteix en el mateix codi que utilitzàvem en la versió simple, però en aquest cas en comptes de fer un bucle fins que el nombre d'objectes era igual al nombre de particions, esperem que el nombre d'objectes sigui igual al nombre de particions i enviem un missatge per desbloquejar l'orchestrator.

```
def main (args):
    config = args.get("config_file")
    text = args.get("text")
    index = args.get("index")
    amqp = args.get("amqp")
    amqp_url = amqp.get('amqp')
    cos_backend = COS_backend(config)
    params_word_count = {'words':text}
    resultat_word_count = word_count(params_word_count)
    cos_backend.put_object('sanchoericbucket', 'word_count_' + str(index) + '.txt', str(resultat_word_count))
    params = pika.URLParameters(amqp_url)
    connection = pika.BlockingConnection(params)
    channel = connection.channel()
    channel.queue_declare(queue='word_count_queue')
    message='word_count_'+str(index)+'.txt'
    channel.basic_publish(exchange='', routing_key='word_count_queue', body=message)
    connection.close()
    result = {'resultat':'be'}
    return result
```

Enviem un missatge al reducer dient-li el nom del fitxer que ha finalitzat.

2. Validació

2.1. Versió paral·lelitzada

Primer de tot comprovem que el nombre total de paraules correspon al que indica un programari auxiliar, per exemple l'Atom (editor de text). Ens hem adonat que l'Atom separa tots els caràcters especials en diferents paraules i nosaltres en el cas del guió (-) i l'apòstrof (') no ho hem considerat oportú.

fitxer/programa	count_word	Atom
pg10.txt	855352	857227
pg2000.txt	384473	410993
big.txt	889081	902493

A continuació agafem les tres primeres paraules del diccionari i comprovem que apareguin el mateix nombre de vegades que en l'Atom.

- Primera paraula: The

fitxer/programa	word_count	Atom
pg10.txt	1916	1916
pg2000.txt	12	12
big.txt	6129 + 15 + 323 + 1 + 1 = 6469	6266

- Segona paraula: Project

fitxer/programa	word_count	Atom
pg10.txt	84	84
pg2000.txt	85	85
big.txt	199 + 1 = 200	200

- Tercera paraula: Gutenberg

fitxer/programa	word_count	Atom
pg10.txt	19 + 55 + 1 = 84	84
pg2000.txt	28 + 1 + 55 + 1 = 85	85
big.txt	81 + 111 + 2 + 1 = 195	195

Aquestes diferències en el nombre de paraules és a causa del tractament de caràcters especials, en el nostre cas hem suposat que qualsevol paraula amb guió i/o apòstrof no se separa en dues, per tant en l'exemple de Gutenberg en el text pg10.txt tenim:

'Gutenberg' : 19 'Gutenberg-tm' : 55 'Gutenberg-tm's' : 1

En total les 84 que diu l'Atom.

2.2. Versió seqüencial

Primer de tot comprovem que el nombre total de paraules correspon al que indica un programari auxiliar, per exemple l'Atom (editor de text).

fitxer/programa	count_word	Atom
pg10.txt	855352	857227
pg2000.txt	384473	410993
big.txt	889081	902493

A continuació agafem les tres primeres paraules del diccionari i comprovem que apareguin el mateix nombre de vegades que en l'Atom.

- Primera paraula: The

fitxer/programa	word_count	Atom
pg10.txt	1916	1916
pg2000.txt	12	12
big.txt	6129 + 15 + 323 + 1 + 1 = 6469	6266

- Segona paraula: Project

fitxer/programa	word_count	Atom
pg10.txt	84	84
pg2000.txt	85	85
big.txt	199 + 1 = 200	200

- Tercera paraula: Gutenberg

fitxer/programa	word_count	Atom
pg10.txt	19 + 55 + 1 = 84	84
pg2000.txt	28 + 1 + 55 + 1 = 85	85
big.txt	81 + 111 + 2 + 1 = 195	195

2.3. Versió amb cues

Primer de tot comprovem que el nombre total de paraules correspon al que indica un programari auxiliar, per exemple l'Atom (editor de text).

fitxer/programa	count_word	Atom
pg10.txt	855352	857227
pg2000.txt	384473	410993
big.txt	889081	902493

A continuació agafem les tres primeres paraules del diccionari i comprovem que apareguin el mateix nombre de vegades que en l'Atom.

- Primera paraula: The

fitxer/programa	word_count	Atom
pg10.txt	1916	1916
pg2000.txt	12	12
big.txt	6129 + 15 + 323 + 1 + 1 = 6469	6266

- Segona paraula: Project

fitxer/programa	word_count	Atom
pg10.txt	84	84
pg2000.txt	85	85
big.txt	199 + 1 = 200	200

- Tercera paraula: Gutenberg

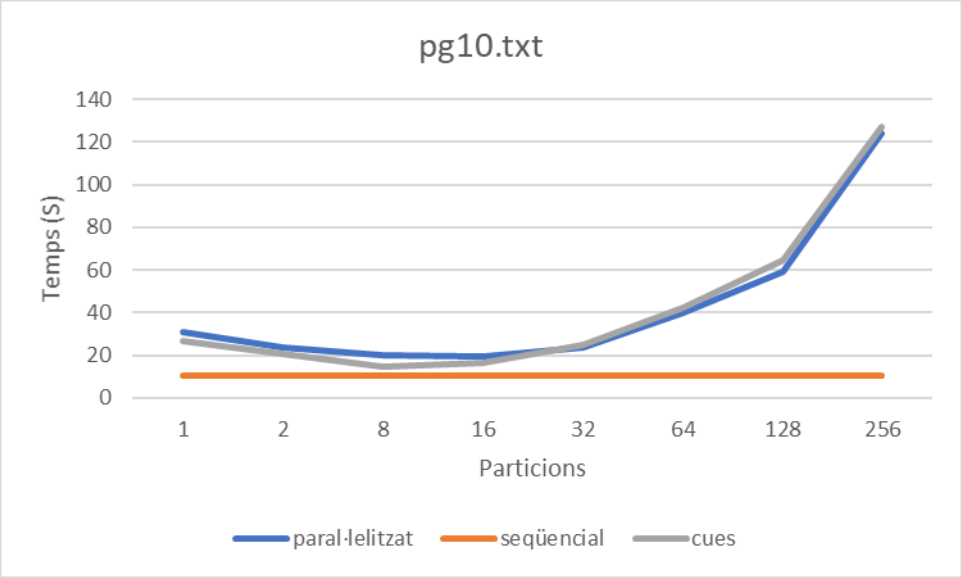
fitxer/programa	word_count	Atom
pg10.txt	19 + 55 + 1 = 84	84
pg2000.txt	28 + 1 + 55 + 1 = 85	85
big.txt	81 + 111 + 2 + 1 = 195	195

Podem veure que en les tres implementacions obtenim el mateix resultat per tant podem concloure que les implementacions són correctes segons el nostre criteri de contar les paraules.

3. Speedups

3.1. Fitxer pg10.txt (4.5M)

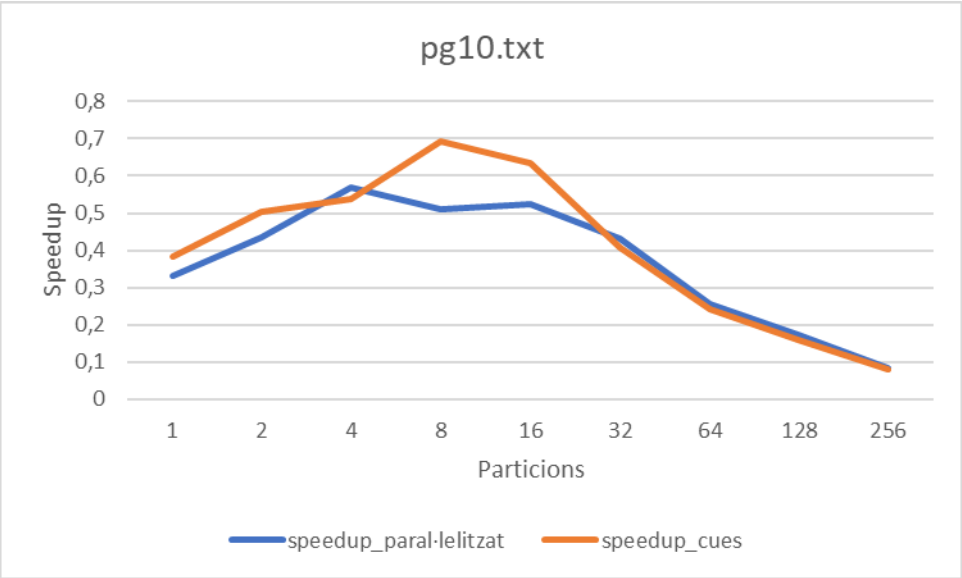
particions/tipus	1	2	4	8	16	32	64	128	256
paral·lelitzat	30,92	23,49	18	20,02	19,54	23,67	40,15	59,22	123,89
seqüencial	10,23	10,23	10,23	10,23	10,23	10,23	10,23	10,23	10,23
cues	26,67	20,33	19,05	14,74	16,12	25,05	42,26	64,57	127
speedup_paral·lelitzat	0,331	0,436	0,568	0,511	0,524	0,432	0,255	0,173	0,083
speedup_cues	0,384	0,503	0,537	0,694	0,635	0,408	0,242	0,158	0,081



Primerament és fàcil veure que el codi que s'executa de manera seqüencial és el més ràpid, això pot ser degut a la mida del fitxer, ja que no és molt gran per suposar un problema en l'execució de la CPU. Per tant, si establim una connexió amb el cloud, serà més lent.

D'altra banda els dos altres codis que utilizen el cloud, tenen un rendiment similar entre elles, amb una lleugera millora de la implementació amb cues. Les particions milloren el temps d'execució a partir de 2 particions, arribant a un rendiment màxim entre 8 i 16 particions, però a mesura que anem augmentant més el nombre de particions disminuïm molt l'eficiència.

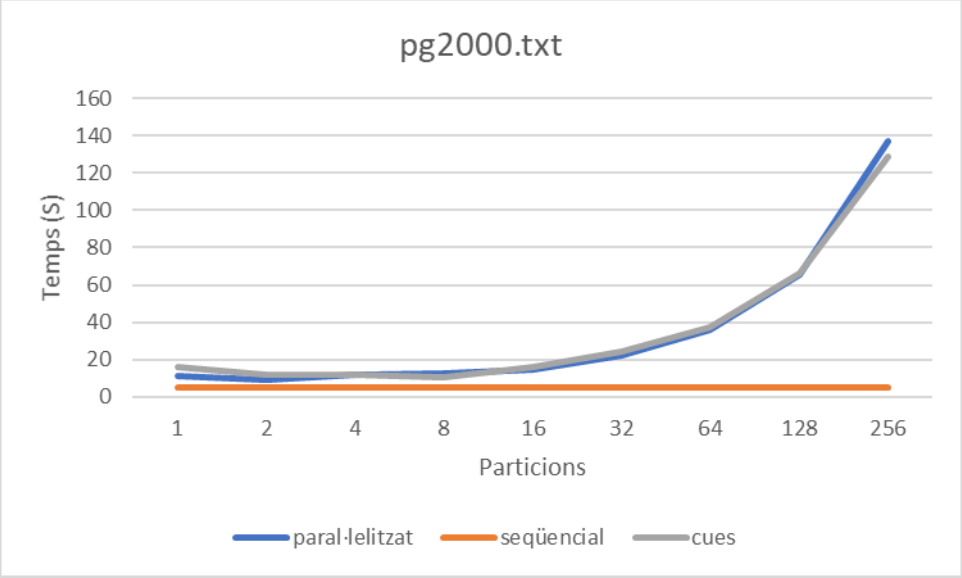
Això es pot explicar veient la mida del fitxer que es tracta, ja que com s'ha comentat anteriorment no és suficientment gran perquè requereixi una gran partició de treballs.



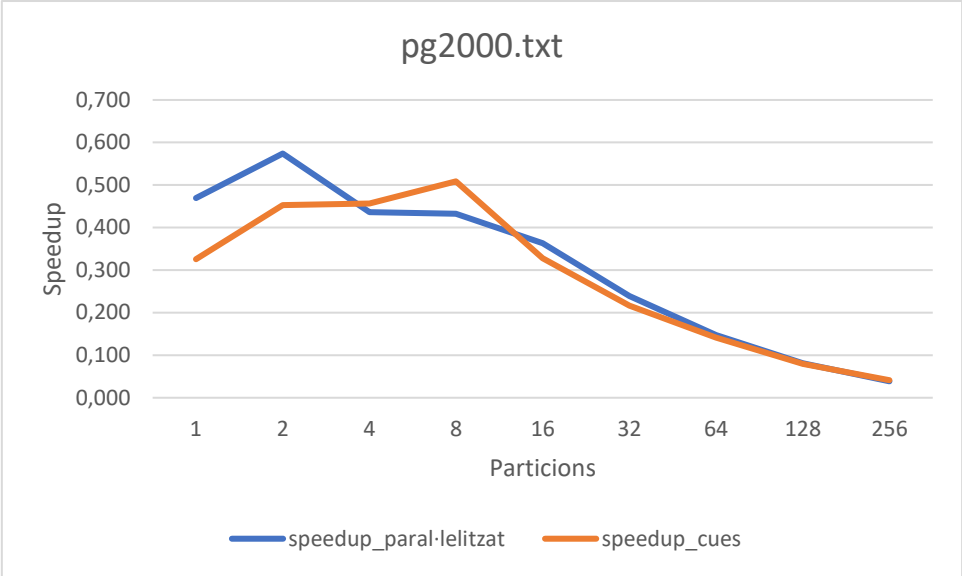
Amb aquesta gràfica que relaciona el temps de la versió seqüencial amb les versions del cloud, es pot veure millor que el màxim rendiment es produeix entre 8 i 16 particions. A més cal observar que en utilitzar una implementació de cues i no demanar tants recursos al cloud millora el rendiment de l'execució i el consum de CPU local.

3.2. Fitxer pg2000.txt (2.2M)

particions/tipus	1	2	4	8	16	32	64	128	256
paral·lelitzat	11,32	9,25	12,18	12,28	14,63	22,24	36,12	65,27	137,41
seqüencial	5,31	5,31	5,31	5,31	5,31	5,31	5,31	5,31	5,31
cues	16,31	11,72	11,63	10,44	16,18	24,52	37,62	66,5	128,66
speedup_paral·lelitzat	0,469	0,574	0,436	0,432	0,363	0,239	0,147	0,081	0,039
speedup_cues	0,326	0,453	0,457	0,509	0,328	0,217	0,141	0,080	0,041



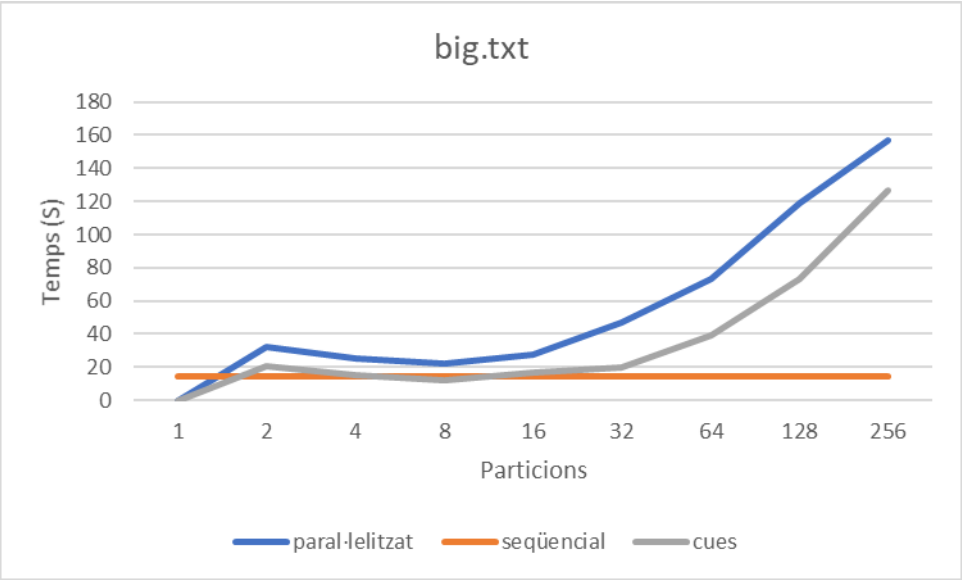
En aquest cas el fitxer és més petit que l'anterior i el resultat obtingut és similar.



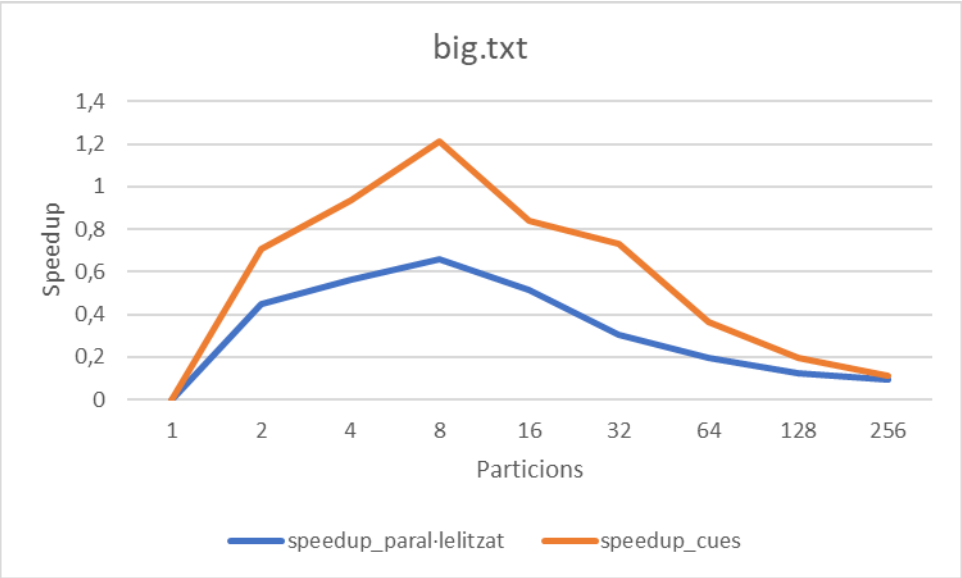
Pel que fa al rendiment respecte a la implementació seqüencial, en aquest cas podem veure que la implementació paral·lela obté un rendiment màxim amb 2 particions i la implementació amb cues obté un rendiment màxim amb 8 particions, tot i que és inferior a la que obté la paral·lela. Podem observar que el nombre de particions ideals ha disminuït una mica respecte a l'anterior, ja que la mida del fitxer és inferior.

3.3. Fitxer big.txt (6.5M)

particions/tipus	1	2	4	8	16	32	64	128	256
paral·lelitzat	0	32,17	25,61	21,91	27,99	47,18	73,58	118,93	157,19
seqüencial	14,41	14,41	14,41	14,41	14,41	14,41	14,41	14,41	14,41
cues	0	20,44	15,34	11,9	17,11	19,72	39,26	73,66	126,64
speedup_paral·lelitzat	0	0,448	0,563	0,658	0,515	0,305	0,196	0,121	0,092
speedup_cues	0	0,705	0,939	1,211	0,842	0,731	0,367	0,196	0,114



Finalment en aquesta anàlisi s'utilitza un fitxer més gran que en els dos casos anteriors, per tant com és d'esperar hi ha un punt que la implementació més eficient (la implementació amb cues) supera la velocitat de la implementació seqüencial, aquest fet succeeix amb 8 particions, un nombre de particions adequat per la mida del fitxer. La resta de comportaments són similars a les gràfiques anteriors.



En el cas del rendiment respecte a la implementació seqüencial, la implementació amb cues supera en tots els casos a la implementació paral·lela. Aquest fet confirma que en augmentar la mida del fitxer val més la pena utilitzar una implementació a priori més complexa però que al final, amb grans quantitats de dades millora el rendiment, ja que optimitza els recursos del cloud i aprofita els recursos locals de la CPU.