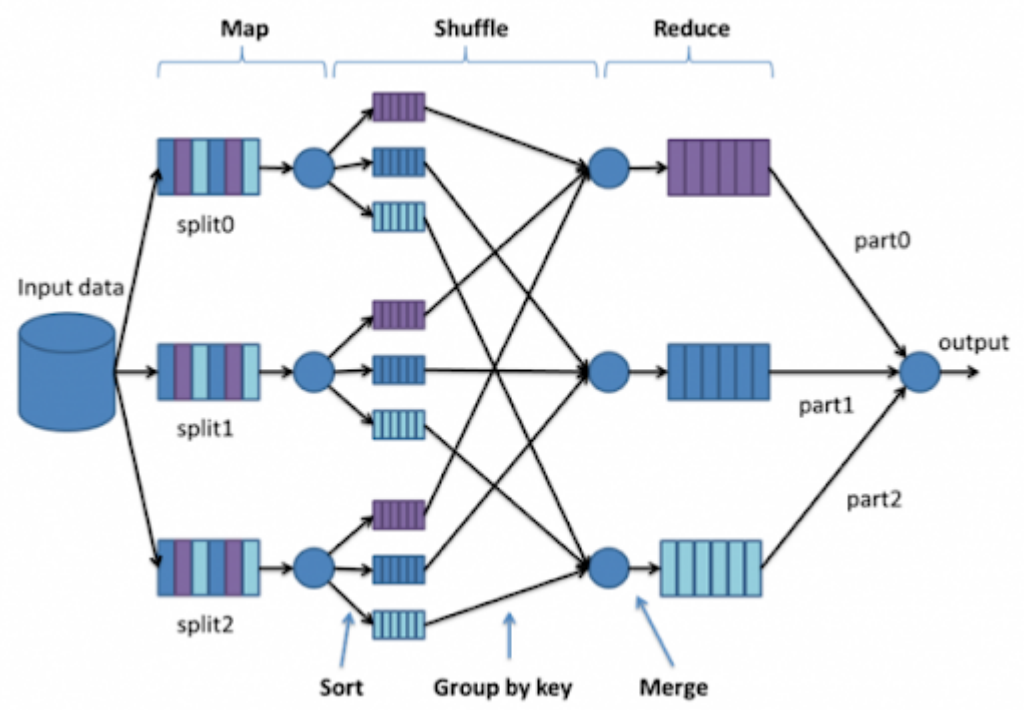
Arquitectura i Tecnologies del Software

Práctica 1 Big Data

Parte 1. Implementación algoritmo Map-Reduce: Aplicación para contar palabras



Daniel Calvo Ramos 1494116
Angel Garcia Calleja 1490917

ÍNDICE

ANÁLISIS Y DISEÑO **3**

Implementación 3
Imports 3
comprobarExtension(sys.argv[numFichero+1]) 3
leerFichero(filename) 3
Splitting(lineas) 3
Mapping(palabras) 4
Shuffling(mapa, diccionario) 4
Reducing(tupla) 5
Flujo de ejecución 5

Diagrama de clases 7
Diagrama de secuencia 7

MANUAL DE USUARIO **8**

PRUEBAS DE TEST REALIZADAS **9**

Test de gestión de errores inputs 9
Extensión del fichero de texto de tipo pdf 9
Nombre del fichero de texto erróneo 9
No indicar ningún fichero (ni nada) por parámetro 10
Nombre del fichero y extensión incorrecto 10
Múltiples ficheros ambos incorrectos 10
Múltiples ficheros uno de ellos incorrecto 10

Test de filtración de datos 11
Test de funcionamiento 11
Test de rendimiento / memoria 12

ANÁLISIS Y DISEÑO

En este apartado se explica el análisis del programa línea por línea así como se explica que se hace en cada momento y por que. Posteriormente se muestran varios diagramas UML para ilustrar de forma más visual las clases y las secuencias del programa.

Implementación

La explicación del código se realizará por el orden de ejecución de las líneas del programa. Así daremos un contexto previo antes de explicar cada función.

Primero, explicaremos el motivo por el cual hemos importado ciertas librerías, seguidamente que realizan las funciones creadas y , y finalmente, el flujo de ejecución del programa o main.

Imports

- sys: para poder obtener los argumentos de la línea de comandas.
- re: para poder filtrar caracteres a través de regex.
- time: para cronometrar el tiempo de la ejecución completa del programa así como de sus fases por separado.
- multiprocessing: para poder utilizar la función cpu_count() la cual nos devuelve el número de cores de nuestra cpu. Esto lo utilizaremos para asignar cuantos procesos en paralelo podremos ejecutar.
- Pool from multiprocessing: utilizado para poder paralelizar la ejecución del código.

comprobarExtension(sys.argv[numFichero+1])

Controlamos el hecho de que el fichero tenga extensión .txt, si tiene otra extensión nos saltamos ese fichero. Para comprobar la extensión utilizamos la función endswith('.txt'). Seguidamente, si la extensión es de tipo txt la función devuelve un 1. En caso contrario devuelve -1.

leerFichero(filename)

Función donde cada proceso leerá una línea del txt. Todo esto será volcado en la variable lineas, que acabará siendo una lista con todas las líneas leídas por los procesos.

Splitting(lineas)

Esta función recibe una línea del fichero leído. Por cada línea recibida, primeramente se convierten todos los caracteres a minúscula (ya que el tener mayúsculas nos puede crear inconsistencias), y posteriormente mediante la función

sub de la librería importada re, sustituimos todos los caracteres que no cumplan con la expresión regular: `[^\w\s'-]` por un espacio en blanco o por 'nada'. Esta expresión regular quiere decir que se substituye todo lo que no sea una letra, un dígito o una barra baja (\w), cualquier espacio, enter o tabulación (\s), apóstrofos o guiones.
Puesto que las barra baja o los dígitos no nos interesan, volvemos a realizar otra vez la función sub para sustituir todos los dígitos o barras baja por 'nada'. Para ello utilizamos la siguiente expresión regular:`[0-9_]`.

Una vez hemos filtrado toda la línea, separamos la misma por palabras, para poder realizar la siguiente fase mapping. Esto lo hacemos fácilmente con la función split().
Finalmente la función devuelve una lista de palabras.

Mapping(palabras)

En esta función recorremos la lista de palabras de una línea recogidas de la función anterior. Para cada palabra creamos una tupla con la palabra y un 1 tal que (palabra,1) y la metemos dentro de una lista con las demás palabras de la línea.

El resultado de esta función es una lista grande con sublistas de todas las líneas, las cuales dentro tienen todas las tuplas de palabras de esa línea, llamada mapa.

Shuffling(mapa, diccionario)

Esta función no conseguimos hacerla de manera paralelizada con map de Pool, por tanto está hecha de manera secuencial.
Primero creamos un diccionario vacío donde se volcará el resultado de la función.
Esta función recibe el resultado de la anterior, mapa, y el diccionario vacío.
En esta función simplemente recorremos la variable mapa y actualizamos el diccionario según si la palabra de la tupla está ya o no en el diccionario. Si no está, creamos una nueva entrada donde la llave será la palabra y el valor será una lista con un uno tal que [1]. Si la palabra ya se encuentra en el diccionario, añadimos un uno a esa lista tal que [1,1].

Finalmente acabamos teniendo un diccionario con entradas de todas las palabras del texto y como una lista de tantos unos como veces aparezca en el texto.

Cabe mencionar que intentamos hacer esta función de manera distribuida (hemos dejado el código en un comentario grande).
Lo pensamos de manera que primero teníamos que meter un diccionario compartido para todos los procesos (Preparar) al final de cada línea del mapa, de manera que teníamos [[(palabra1,1),(palabra2,1),d], [(palabra3,1),(palabra4,1),d]], y el Shuffling era igual que como se ha explicado arriba. Esto solucionaba el problema de que cuando haces un map solo puedes pasar 1 solo argumento a la función, pero de esta manera se estaba insertando el mismo diccionario tantas veces como líneas tenía el fichero. Para ficheros pequeños funcionaba, pero ya se podía ver que la

ejecución se ralentizaba mucho, y para ficheros más grandes simplemente no acababa y acababa petando.

Reducing(tupla)

La función Reducing recibe una tupla del estilo: ('hola',[1,1]). Puesto que las tuplas son inmutables, convertimos la tupla a lista para poder modificarla.
Posteriormente igualamos la segunda posición de la lista a la longitud de dicha lista, ya que queremos convertir la lista de 1's a un único valor.
Finalmente volvemos a convertir la lista en tupla y devolvemos la misma.
Es decir, convertimos la tupla a lista, modificamos la segunda posición, la volvemos a convertir a tupla y la devolvemos por return.

Flujo de ejecución

La primera línea que nos encontramos es time.perf_counter(). Esto es simplemente para tener el tiempo de inicio de la ejecución del programa. Lo utilizaremos para observar el tiempo total de ejecución, útil para la parte de test de rendimiento.

Acto seguido, establecemos el número de procesos que habrá durante la ejecución del programa. Para ello se puede hacer de dos maneras: fijando manualmente un valor concreto o, estableciendo como número de procesos el número de cores del computador mediante el cual se está ejecutando. Nosotros hemos creído que la mejor opción es la de fijar el número de procesos al máximo permitido, ya que de esta forma se va a utilizar todo el potencial disponible. La opción de fijar manualmente el número de procesos, nos ha sido útil para la realización de las gráficas de rendimiento.
En caso de querer fijar manualmente el número de procesos hay que comentar la línea 84 y descomentar la 85.

El siguiente paso es recoger el número de ficheros que se han recibido en la llamada. A raíz de este número, se entrará o no en un bucle para realizar el MapReduce de todos los ficheros recibidos.

Una vez dentro del bucle de cada fichero, realizamos la comprobación de la extensión llamando a comprobarExtension() y le pasamos el nombre del fichero.

Lo siguiente es utilizar Pool(nProcesos) para crear un objeto tipo Pool para distribuir las tareas entre el número de procesadores definido anteriormente.

Antes de repartir la faena, abrimos el fichero dentro de un mecanismo de gestión de errores try except que posteriormente cerraremos con el close().En caso de entrar en el except, la variable control = -2 (esto nos servirá posteriormente para evitar ejecutar todo el flujo de ejecución en caso de que el fichero no se pueda abrir correctamente, para así evitar posibles errores consecuencia de una lectura de fichero errónea). Dentro del try llamamos a la función map para indicarle que la función leerFichero() tiene que hacerse de manera paralelizada.

En la siguiente parte decidimos meter las 2 siguientes funciones (Splitting y Mapping) en una función llamada Map, a la cual llamaremos de manera paralelizada pasándole la variable lineas. Esta función es la que se encarga de juntar ambas funciones. El resultado queda en la variable mapa.

El siguiente paso es realizar, como ya se ha comentado anteriormente, de manera secuencial, la función Shuffling(). Esta función recibe la variable mapa y un diccionario vacío para devolvernos el diccionario con todas las palabras del texto.

Antes de llamar a Reducing(), realizamos un ajuste. Creamos la variable listaItems, donde obtendremos en una misma lista todas las tuplas obtenidas en Shuffling() con diccionario.items(). Lo hicimos de esta manera porque si no un proceso solamente “veía” la primera parte de la tupla, es decir la palabra, y no veía la lista que tenía que ser modificada.

El último paso de MapReduce consiste en llamar a la función Reducing() con map, donde le pasaremos la variable listaItems. Como resultado de ello obtendremos la lista final de tuplas que enseñaremos por pantalla, listaDeTuplas.

Finalmente, como bien indica la documentación de Pool, llamamos a la función close(), ya que es la última vez que trabajaremos con la instancia de Pool, los procesos “worker” terminarán cuando todo el trabajo asignado se haya completado. Además llamando a la función join() esperamos a que todos los procesos terminen, añadiendo un punto de sincronización para evitarnos problemas.

Como final del bucle de un fichero mostramos por pantalla el resultado en el formato indicado por el profesorado.

Como detalle, en caso de querer ver el tiempo que tarda la ejecución de todo el programa, descomentar el último print.

Diagrama de clases

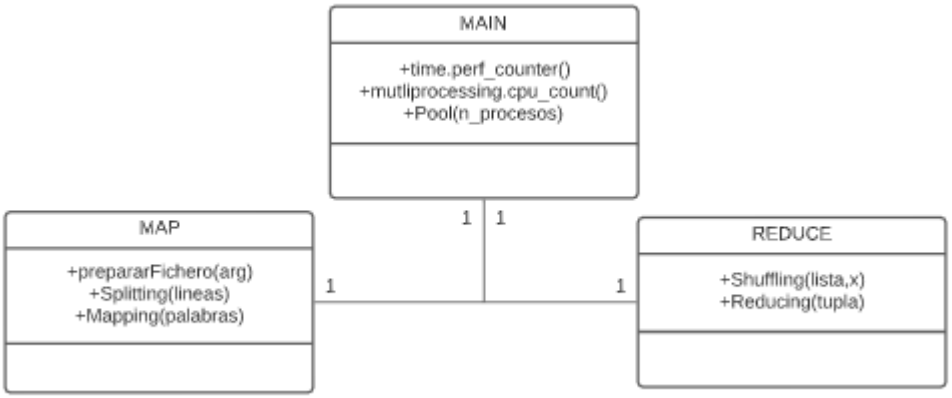
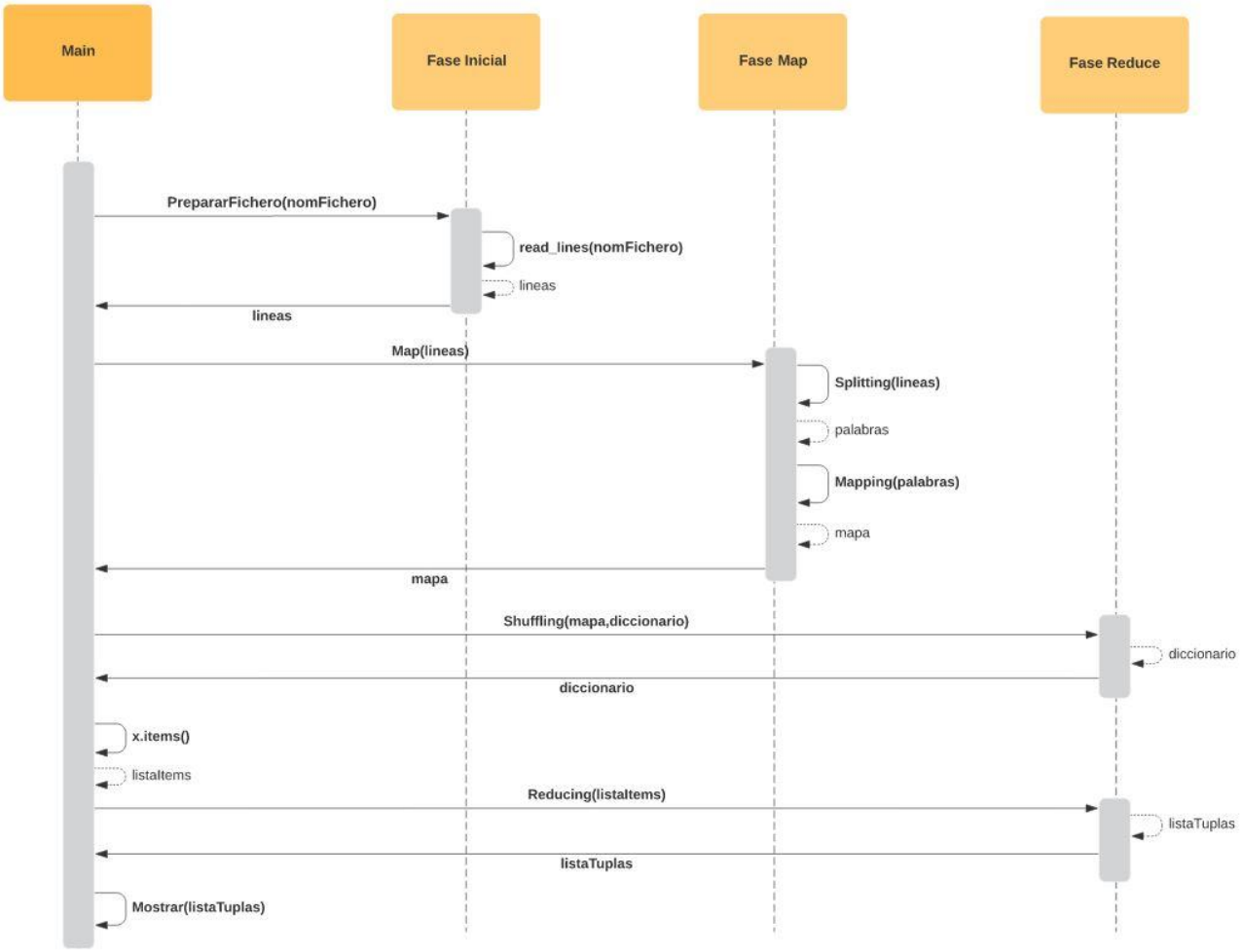


Diagrama de secuencia



MANUAL DE USUARIO

Para compilar y ejecutar el programa es muy sencillo.
Como el programa está desarrollado en python, no hace falta realizar compilación previa.
Se debe de acceder por terminal a la carpeta donde se encuentran los archivos txt y el archivo de python TextCounter.py. Una vez dentro de ella ejecutar la siguiente comanda en el mismo terminal (supongamos que quiero realizar el conteo de palabras sobre 2 ficheros llamados: cuento1.txt y cuento2.txt):
python3 TextCounter.py cuento1.txt cuento2.txt

```
daniel@debian:~/Escritorio$ python3 TextCounter.py text1.txt


text1.txt :
roc : 1
l'esmolada : 1
d'esma : 1
cingles : 1
font : 1
pels : 1
nines : 1
la : 1
ribera : 1
li : 1
l'esquellot : 1
cua : 1
massa : 1
companyes : 1
damunt : 1
amb : 1
```

Aquí se muestra un ejemplo en Debian10 de cómo se debería realizar la llamada desde terminal. En este caso en el escritorio tenemos tanto el archivo TextCounter.py como el fichero text1.txt .

IMPORTANTE: Este código está realizado con la versión 3.8 de Python. Puesto que el código ha sido desarrollado con python, no es necesario la creación de ningún ejecutable para ser utilizado, por lo que en el zip de esta práctica no hay ninguna carpeta bin.

PRUEBAS DE TEST REALIZADAS

En este apartado se van a desglosar los diferentes tipos de test que se han realizado y se va a explicar cada uno de ellos.

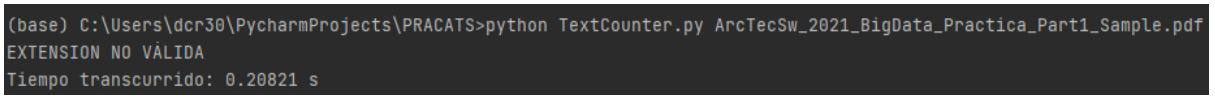
Test de gestión de errores inputs

El objetivo de este tipo de test es intentar introducir valores erróneos mediante los inputs que puedan llegar a corromper la ejecución del programa.

Puesto que el programa realizable no es de gran complejidad, únicamente tenemos un lugar en el que se introducen datos por teclado, que es en el momento de la ejecución por terminal. Como se mencionó anteriormente, se debe de ejecutar una comanda del estilo: python <nombre_del_fichero.py> <nombre_del_textfile.txt>. Por lo tanto hemos realizado los siguientes tests:

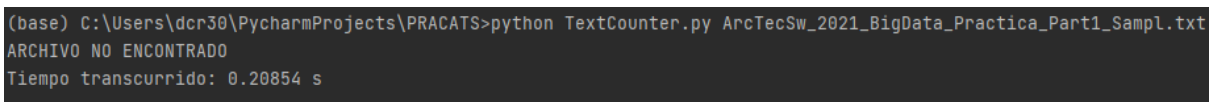
1. **Extensión del fichero de texto de tipo pdf**

   Puesto que nuestro código únicamente funciona con archivos de texto de extensión txt, hay que tener en cuenta dicho aspecto. El output obtenido fue el siguiente:



2. **Nombre del fichero de texto erróneo**

   Esta prueba consiste en introducir un nombre del archivo de texto que no exista en el directorio del programa. Esto es un error bastante común que puede suceder simplemente al escribir en nombre de forma incorrecta por despiste. En nuestro test particular lo que hemos hecho ha sido ‘comernos’ la letra ‘e’ de la última palabra del txt (Sample). El output obtenido fue el siguiente:



3. **No indicar ningún fichero (ni nada) por parámetro**

   El objetivo de este test es ver que el programa puede manejar el hecho de que no le introduzcan ningún fichero de texto (ni de ningún otro tipo) por parámetro. Para realizar este test simplemente no le pasamos ningún fichero de texto como parámetro. El output obtenido fue el siguiente:

```
(base) C:\Users\dcr30\PycharmProjects\PRACATS>python TextCounter.py
NO SE HA INTRODUCIDO NINGÚN PARÁMETRO
Tiempo transcurrido: 0.00010 s
```

4. **Nombre del fichero y extensión incorrecto**

   Este test es una combinación de los test 1 y 2 mencionados anteriormente. En este caso se desea comprobar si la combinación de varios errores afecta a la funcionalidad o simplemente nos muestra el primer error encontrado. El output obtenido fue el siguiente:

```
(base) C:\Users\dcr30\PycharmProjects\PRACATS>python TextCounter.py ArcTecSw_2021_BigData_Practica_Part1_Sampl.pdf
EXTENSION NO VÁLIDA
Tiempo transcurrido: 0.20999 s
```

5. **Múltiples ficheros ambos incorrectos**

   En este test se quiere comprobar si al pasarle dos ficheros erróneos al programa este funciona y muestra los mensajes de error correspondientes. En el siguiente ejemplo le pasamos dos veces el mismo fichero ambos escritos de forma errónea. El output obtenido es el siguiente:

```
(pythonProject) C:\Users\Angel\Escritorio\pythonProject>python TextCounter.py dato¡s.txt datos.tt
ARCHIVO NO ENCONTRADO
EXTENSION NO VÁLIDA
Tiempo transcurrido completo: 0.14305 s
```

6. **Múltiples ficheros uno de ellos incorrecto**

   En este test se quiere comprobar si el hecho de que un archivo se pase de forma errónea, no impide que el siguiente fichero, escrito de forma correcta, se lea correctamente. Para realizar el test le pasamos primeramente un fichero datos.txt escrito de manera errónea, y posteriormente escrito de manera correcta. El output obtenido fue el siguiente:

```
(base) C:\Users\dcr30\PycharmProjects\PRACATS>python TextCounter.py text2.tx text2.txt
EXTENSION NO VÁLIDA


text2.txt :
supervivientes : 6
' : 156
ya : 52
tiene : 26
```

Como se puede apreciar en el resultado anterior, efectivamente primero se muestra un mensaje de error correspondiente al primer fichero, y a continuación, se procesa correctamente el segundo fichero.

Test de filtración de datos

Este tipo de test consta de llenar un fichero txt con todo tipo de caracteres alfanuméricos y ver si el programa elimina todos aquellos caracteres requeridos y si mantiene todos aquellos que sean caracteres alfabéticos, apóstrofos y guiones. El texto introducido fue el siguiente:
<< *a' b5 3d&4-l 7 **æ!, "e"ø? å/. #h0la]* >>
Y la salida obtenida fue la siguiente:

```
(base) C:\Users\dcr30\PycharmProjects\PRACATS>python TextCounter.py ArcTecSw_
[("a'", 1), ('b', 1), ('d-l', 1), ('æ', 1), ('eø', 1), ('å', 1), ('hla', 1)]
```

Como podemos ver, el filtraje de datos se ha realizado correctamente.

Test de funcionamiento

Este test consiste en comprobar si el programa completo hace lo que tiene que hacer. Para realizar este test se ha utilizado el fichero txt proporcionado por el profesorado y se ha comparado su salida con la salida del programa. No contentos con esto, cogimos un texto aleatorio de internet y lo 'enganchamos' en un fichero txt y lo pasamos por parámetro al programa. Paralelamente y para comprobar que el output era correcto, pusimos ese mismo código en alguna página de internet que contabilizara palabras y así comparamos ambas salidas para ver si coincidían.

Test de paralelización

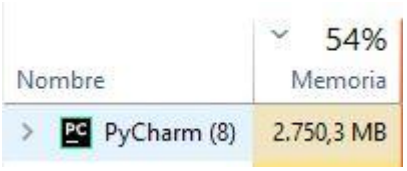
En este test se quiere comprobar si realmente la ejecución del programa se hace de forma distribuida. Para realizar el test se ha utilizado la variable reservada current_process de la librería multiprocessing. Posteriormente se ha puesto un print en todas las funciones que se realizan de forma distribuida.
El output obtenido en una de ellas es el siguiente:

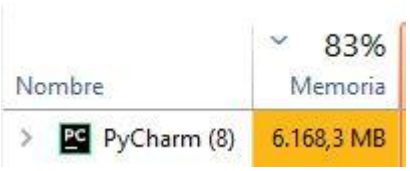
```
<function current_process at 0x0000021A41033430>
<function current_process at 0x000002C739AB3430>
<function current_process at 0x000002C739AB3430>
<function current_process at 0x00000158C0B53430>
<function current_process at 0x00000158C0B53430>
<function current_process at 0x000001DC1A053430>
```

Como se puede apreciar en la imagen de la izquierda, vemos que hay 4 procesos creados en diferentes posiciones de memoria, lo que indica que paraleliza correctamente.

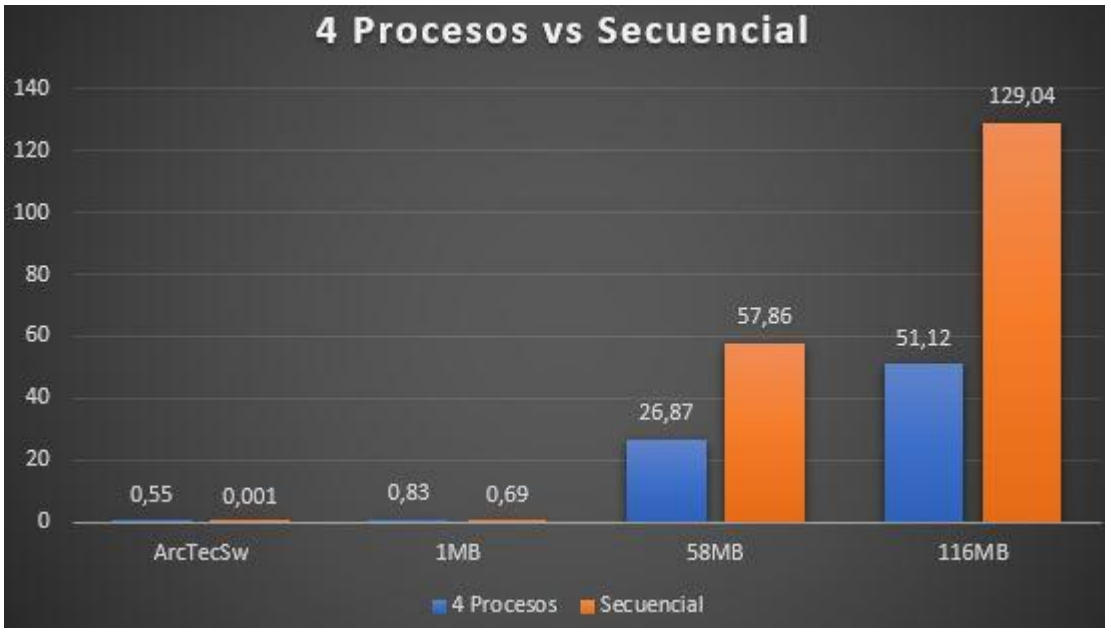
Test de rendimiento / memoria

Para comprobar que el programa realmente es útil y está optimizado y no hay ningún problema relacionado con la gestión de memoria, se le pasa al programa un fichero txt de un tamaño cercano a 1GB para comprobar si toda la ejecución se realiza de manera correcta y ver si todo funciona correctamente con un fichero de un mayor tamaño.
Desafortunadamente, al pasarle un archivo de 500MB, nuestros ordenadores se quedaron congelados. Tras analizar el problema de forma exhaustiva, vimos que mientras se realiza la ejecución del programa, la memoria del ordenador aumentaba hasta que el programa se detenía. Esto se debe a que no guardamos los datos entre fases a disco. Por lo que SÍ hay problemas relacionados con la gestión de memoria.

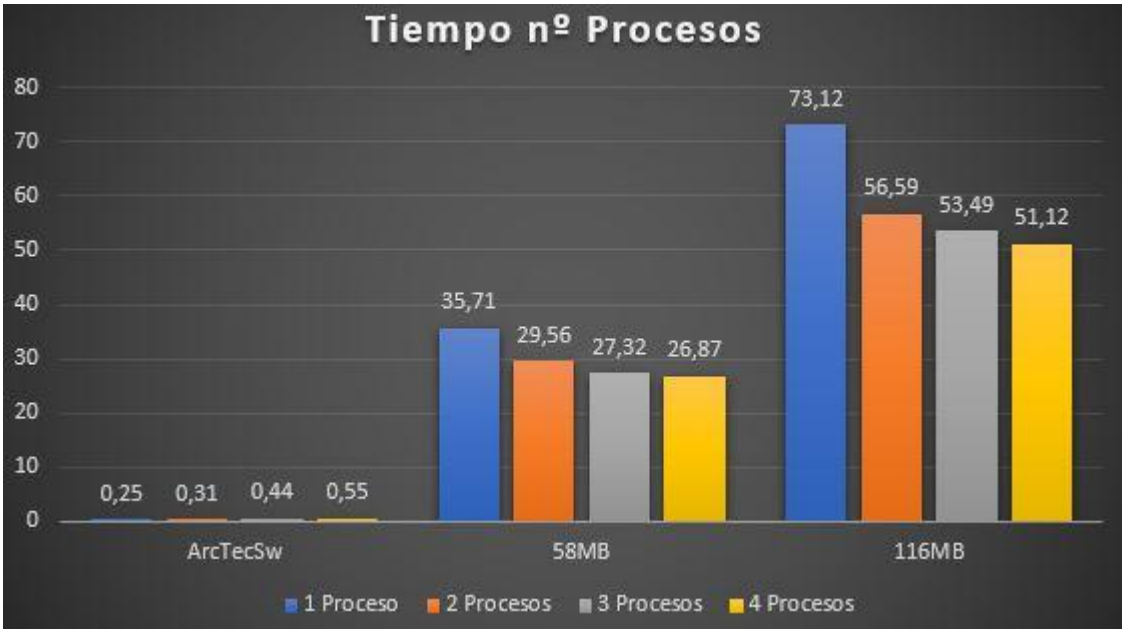




En cuanto a rendimiento, realizamos distintas pruebas cambiando tanto el número de procesos, como la forma de hacerlo y el archivo a leer.
En la siguiente gráfica se muestra la diferencia en cuanto a tiempo dependiendo de si realizamos la ejecución en secuencial o si la realizamos de manera distribuida con 4 procesos.
Podemos observar como en archivo a leer pequeños, la forma secuencial es más rápida, pero a medida que aumentamos el tamaño del archivo, el tiempo de la forma secuencial se eleva mucho más que la manera distribuida. Podemos decir que la tendencia de la forma secuencial es mucho más pronunciada que la manera distribuida.



En la gráfica siguiente realizamos el experimento de comprobar cuánto se veía afectado el tiempo dependiendo del número de procesos que ejecutamos. Realizamos la ejecución del fichero proporcionado por el profesorado, un archivo de 58MB y un archivo de 116MB con 1, 2, 3 y 4 procesos.
Podemos observar cómo, al igual que antes, en el caso del archivo pequeño el tiempo aumenta a medida que aumentamos el número de procesos. Pero a medida que aumentamos el tamaño del fichero las ejecuciones con más procesos nos dan mejores resultados.



Como conclusiones tras la realización de esta última parte, decir que para archivos de tamaño reducido, el hecho de tener muchos procesos nos lastra, ya que perdemos más tiempo creando, asignando y repartiendo la faena entre procesos que en leer y contar las palabras. Una analogía muy clara para explicar este resultado sería como decir que tienes que recorrer 10 km en coche (secuencial) o en avión (distribuido). Por todos es sabido que el avión es mucho más rápido pero al ser un trayecto muy corto no se llega a ‘explotar’ la ventaja de ir en avión, por lo que acaba siendo más lento que ir en coche. Por lo que para ejecuciones con archivos de poco tamaño tener menos procesos es más rápido.

Por contra, al ejecutar el programa con archivos de gran tamaño, vemos mucha diferencia entre hacerlo secuencial o con 1 proceso que repartiendo la faena aunque sea solo con 2 procesos. Esto lo que nos indica que es en un principio, el cuello de botella del programa era claramente la secuencialidad del mismo. De ahí que al aumentar simplemente 1 proceso (de 1 a 2) la mejoría en el tiempo sea de 1.29x. Otro hecho que reafirma nuestra teoría es que como se puede apreciar en la segunda gráfica, la reducción de tiempo de ejecución en función del aumento del número de procesos es prácticamente mínima. Esta teoría se ve más clara contra

más grande es el tamaño del fichero a procesar, ya que a mayor cantidad de Bytes a procesar más acentuado y notorio se observa el cuello de botella.

En cuanto a la comparación entre la ejecución de forma secuencial o de forma distribuida,como se puede apreciar en la primera gráfica, a medida que aumentamos el tamaño del fichero a procesar, mayor es la diferencia de tiempo entre las dos ejecuciones. Tomando como ejemplo la ejecución con el fichero de 58MB, la mejora de la versión distribuida con respecto a la secuencial es de 2.15x. Al aumentar el tamaño del fichero al doble, la diferencia de speedup entre ambas versiones aumenta hasta los 2.52x de mejoría de la versión distribuida respecto a la versión secuencial.

Cabe decir que se notaría más aún en el tiempo si hubiéramos realizado toda la ejecución de manera distribuida. En estas ejecuciones realizamos la fase de Shuffling() secuencialmente, como ya hemos mencionado antes.