Universitat Autònoma de Barcelona

Grup 12 - Pràctica 2
Jose Antonio Ramos Andrades - 1565479
Víctor Sancho Aguilera - 1529721

Introducció

Al present document es detalla el procés de disseny de la segona pràctica d'arquitectura i tecnologies del software, en aquest document es detallen les estructures utilitzades i els mètodes usats per a complir amb les especificacions de l'enunciat.

L'objectiu d'aquesta pràctica consisteix en implementar el codi del paradigma map and reduce de manera que es faci una bona gestió de la memòria i dels recursos de còmput usats durant aquesta.
Concretament, hem fet una aproximació del mapreduce per a calcular la freqüència d'aparició de cada lletra per paraula en fitxers de text de petit i de gran tamany.

En el nostre cas la pràctica la hem realitzat en Java, utilitzant la versió de sdk 17.

Anàlisi del problema

Per a solucionar aquest problema hem hagut de pensar primer com fer-ho de manera seqüencial i seguidament com poder-ho fer utilitzant multithreading per a intentar minimitzar el temps de resposta del programa.

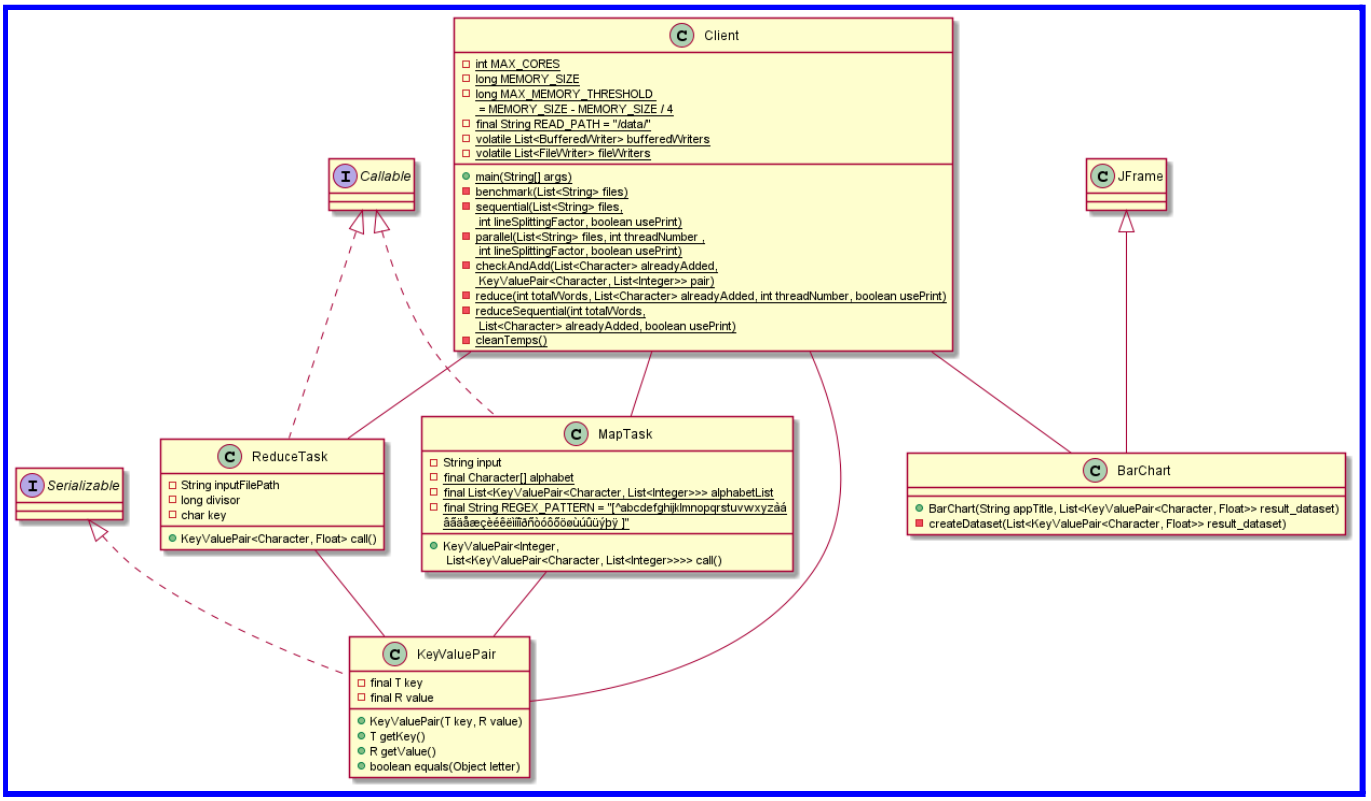
Així doncs, hi ha 2 aproximacions, en seqüencial i en paral·lel, i aquestes dues fan servir la mateixa lògica de funcionament. Concretament, les dues tenen la possibilitat de modificar el factor de separació del fitxer d'entrada (line splitting factor) per determinar cada quantes línies es fa una tasca per a executar-la.

Contextualitzat el problema i analitzades les fases que haurem de fer les exposem aqui:

1. Split dels fitxers per a processar-los de N en N linies.
2. Mapeig d'aquestes N linies en parelles de valors.
3. Ordenació i fusió d'aquestes línies en grups per a poder processar-les.
4. Reducció dels resultats de la ordenació per a mostrar el resultat.

Estructures utilitzades

Diagrama de classes del programa



Com es pot observar en la imatge superior la nostra aplicació consta de 5 classes, en aquest apartat explicarem cada una de les classes, el seu funcionament i els aclariments que calgui donar.

Client

Aquesta és la classe d'entrada de l'aplicació i on s'orquestra tot el procés necessari per a poder completar amb èxit la tasca en la qual consisteix la pràctica.
Es defineixen de manera estàtica el número de threads que són possibles utilitzar en funció del sistema on s'executa el programa (MAX_CORES).
Es defineixen, també de manera estàtica, la memòria màxima que té disponible el sistema (concretament la màquina virtual de java) i el llindar definit per nosaltres tal que el màxim que hauria d'usar-se és 3 quartes parts del màxim disponible.
També tenim el path on es guarden els fitxers a ser processats (/data/) per tal de fer més mantenible el codi en cas que aquest directori canvies.

Els mètodes que té el programa es divideixen segons si es tria l'execució seqüencial o la paral·lela.
Primerament, el mètode "sequential" fa l'execució clàssica en un loop convencional. Aquest mètode s'encarrega de fer la fase d'split i map i, un cop acaba, s'encarrega de fer la crida al

mètode d'ordenació i fusió "shuffle" on és re-ordenen els diferents parells de manera que queden agrupats per poder procedir a la fase de reducció "reduceSequential", mètode on es fa el càlcul i es procedeix a mostrar per pantalla els resultats.
El mètode "parallel" té el mateix objectiu que l'anterior amb la peculiaritat que aquest basa el seu funcionament de les fases de map i reduce en una aproximació paral·lela, fent ús d'una piscina d'execució per a fer l'execució de les tasques de manera paral·lela en diferents threads.
El mètode "checkAndAdd" s'encarrega de gestionar la part de shuffle que comprova si una lletra ja té un fitxer associat per a escribir-hi una nova línia amb aquesta lletra, si no té un arxiu creat el crea i guarda els buffers d'escriptura necessaris per a maximitzar l'eficiència d'escriptura. A la classe de reducció s'explica quin mètode hem fet servir per maximitzar el càlcul.
Per últim, el mètode "cleanTemps" s'encarrega de netejar el contingut de la carpeta temporal i buidar els arxius que derivin de l'execució de tot el programa i que no siguin necessaris.

KeyValuePair

Aquesta classe creada per nosaltres busca suplir la carència a Java de poder fer servir llistes o tuples dinàmiques, com ja sabem al definir una array a Java tot el seu contingut ha de ser del mateix tipus, per aquest motiu hem creat aquesta classe en comptes d'utilitzar alguna que s'hi assembli.
La creació de classes genèriques a Java és trivial, per tant, l'hem pogut fer servir per a diferents estructures.
El funcionament de la classe és molt senzill, té 2 tipus dinàmics, T i R, aquests dos es defineixen quan es crea una instància de la classe i poden ser els que es vulgui, nosaltres hem utilitzat aquesta classe amb les següents permutacions:

- KeyValuePair<Character, List<Integer>>
- KeyValuePair<Character, List<Float>>
- KeyValuePair<Character, Float>>
- KeyValuePair<Integer, KeyValuePair<Character, List<Integer>>>
- KeyValuePair<Integer, List<KeyValuePair<Character, List<Integer>>>>

Tot i que al codi final no hi surtin hem cregut oportú dir-les totes ja que han format part del desenvolupament en primeres etapes.

MapTask

Aquesta classe és la que més importància té en tot el procés, ja que és l'encarregada de fer la feina de "mapejar" tots els caràcters del fitxer de text en el seu respectiu KeyValuePair on els cada lletra tindrà la següent forma (exemple):

- T = Caràcter = “a”
- R = Valor = [1]

En aquesta classe s’ha fet ús d’una estratègia per tal d’estalviar memòria molt interessant, el que hem fet és declarar una array estàtica de caràcters on per cada caràcter vàlid als alfabets basats en llengües romàniques/llatines:

```
private static final Character[] alphabet = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm'
        , 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', 'à', 'á', 'â', 'ã', 'ä', 'å', 'æ'
        , 'ç', 'è', 'é', 'ê', 'ë', 'ì', 'í', 'î', 'ï', 'ð', 'ñ', 'ò', 'ó', 'ô', 'õ', 'ö', 'ø', 'ù', 'ú', 'û'
        , 'ü', 'ý', 'þ', 'ÿ'};

//This object is used to avoid the creation of multiple objects of the same type;
2 usages
private static final List<KeyValuePair<Character, List<Integer>>> alphabetList = new ArrayList<>();

static {
    for (Character c : alphabet) {
        KeyValuePair<Character, List<Integer>> pair = new KeyValuePair<>(c, new ArrayList<>());
        pair.getValue().add(1);
        alphabetList.add(pair);
    }
}
```

Aquesta captura mostra el procés de creació d'un diccionari de KeyValuePairs un per cada lletra que es troba a la variable alphabet. Amb això aconseguim tenir carregat estàticament els objectes per tal de no haver de fer un "new" cada cop que trobem un tipus de lletra conegut. Això no vol dir que si es troba una lletra que no estigui a aquest "diccionari" no s'afegirà, sinó que si en troba algun el crearà amb un new, però és menyspreable el cost en memòria d'aquesta casuística.

Entrant més en detall en el que fa la classe podem dir que implementa la interfície callable per tal de retornar un objecte KeyValuePair amb el total de paraules i una llista de tots els caràcters amb el segon valor inicialitzat amb una llista amb un element (1).
Entrant més en matèria podem dir que es fa un reemplaçament de tots aquells caràcters que no siguin els que surten a la variable alphabet.
L'interessant d'aquesta classe és que es pot fer servir tan seqüencial com en paral·lel, ja que el mètode run és públic i dependrà de si s'executa en un thread o no el que determini el mode d'execució.

ReduceTask

Aquesta classe rep per paràmetre el divisor que s'ha de fer i un path a un fitxer, que serà el que contingui les n línies corresponents a la lletra amb la qual està relacionat el fitxer, per a fer el càlcul i aprofitant que durant el procés hem emmagatzemat les dades en fitxers diferents podem llegir només el nombre de línies donat que ho hem emmagatzemat així, i podem fer ús d'aquesta estratègia per estalviar fase de còmput.

Aquesta classe també implementa la interfície callable i retorna la lletra amb el seu percentatge d'aparició.

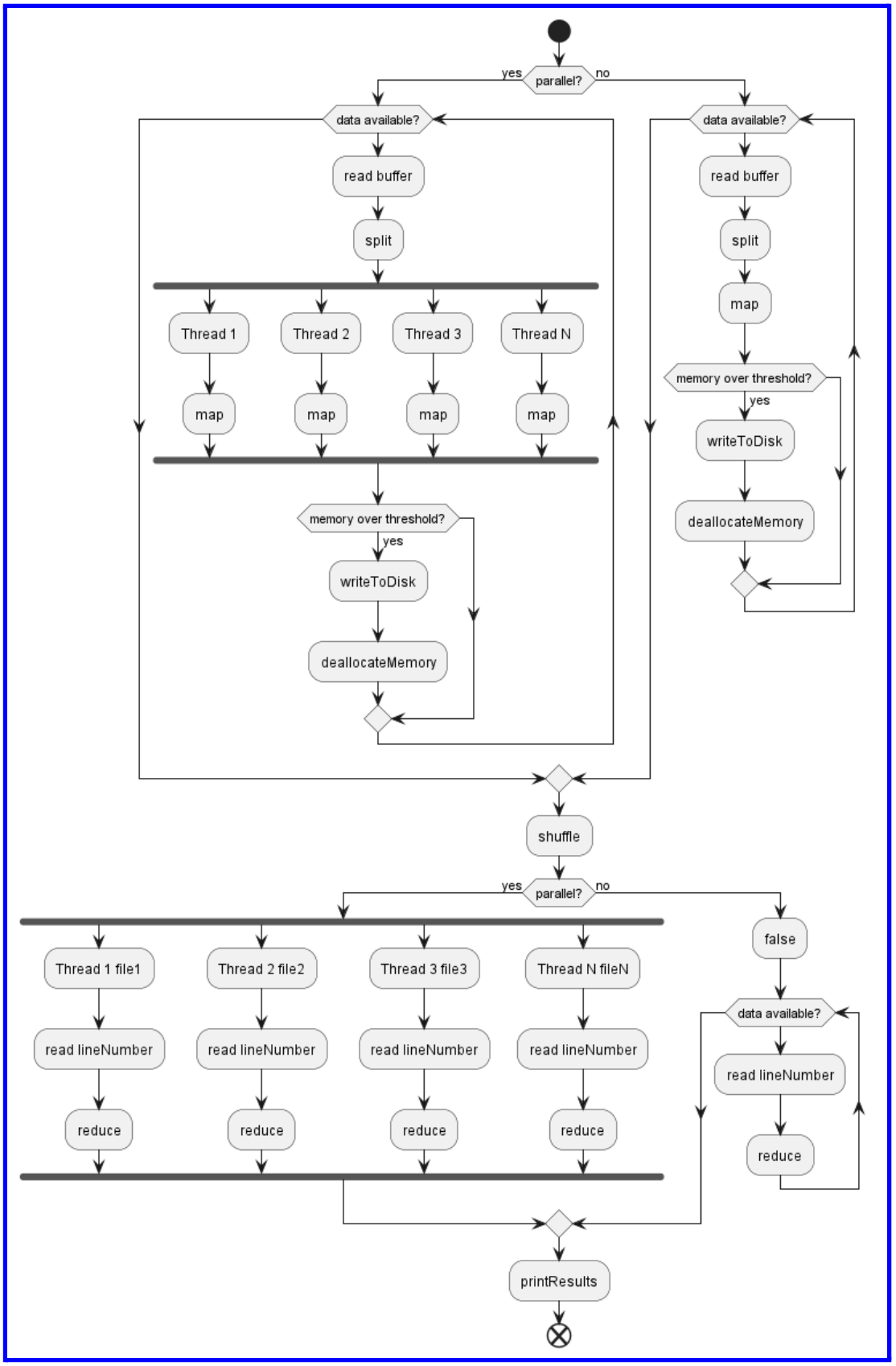
BarChart

Aquesta classe és l'encarregada de generar la gràfica de barres per a l'histograma del programa. La classe rep com a paràmetre al constructor el nom de l'histograma i una llista de la nostra classe KeyValuePair que és el resultat de la fase de reduce de l'execució del programa, posteriorment aquestes dades són processades per a poder ser representades en forma d'histograma.

Aquesta classe es crida dins de la classe Client al mètode "createBarChart" on s'inicialitza y es crida a la representació per pantalla. El mètode "createBarChart" el cridem a la funció "histogram" per a poder representar els resultats de l'execució en paral·lel de cadascun dels fitxers donats o de tots en comú segons el paràmetre introduït al programa.

Diagrama d’activitats

En el següent diagrama d’activitats es mostra el flux que segueix el programa segons si s’executa o no en paral·lel.



Funcionament seqüencial

En el funcionament seqüencial es processa cada fitxer de manera que primer es va llegint del buffer de lectura, seguidament es fa l'extracció de n línies i es tracten aquestes, es fa el mapeig d'aquestes línies i en el cas que durant el mapeig el programa es quedi sense memòria es guarda a fitxer els resultats parcials obtinguts.
Un cop processat tots els fitxers d'igual manera el que queda en memòria es guarda a disc i un cop fet això es passa a la fase de shuffle, on es fa la fusió de totes aquelles lletres en un mateix fitxer per a poder fer el recompte més endavant.
Per últim, es fa la reducció fitxer per fitxer per extreure el percentatge d'aparició de cada lletra.

Funcionament paral·lel

En paral·lel a la pràctica es fa el mateix però, a l'hora de fer la segmentació del fitxer no es para aquí, sinó que continua fent fragmentació de les dades i va creant tasques que es fiquen a una piscina d'execució on posteriorment s'executaran aquestes tasques.
Un cop acabada la fase de mapeig (que també guarda a disc tots els resultats parcials) fa la mateixa fase de shuffle que el funcionament seqüèncial. És a l'hora de fer la reducció que tornem a fer ús de la piscina d'execució per a fer aquesta fase i paral·lelitzar el pas final on es fa el càlcul final.

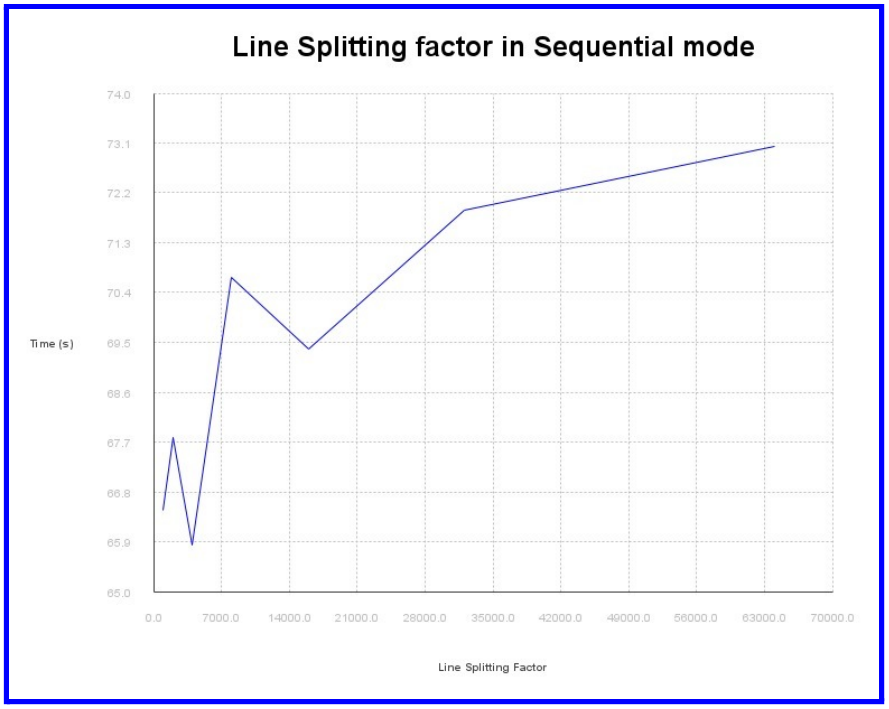
Aproximació map reduce intermitja

Vam arribar a desenvolupar una versió intermitja que anava reduint la quantitat de keyvalues que es guardaven en memòria, el que feia era estalviar memòria a base d'augmentar fase de còmput.
En el cas d'executar amb fitxers molt grans perdria efectivitat, en principi l'aproximació en paral·lel que hem fet nosaltres no perd efectivitat amb més GB.
Els resultats van ser que per a un arxiu d'1.2GB triga uns 43 segons en processar davant dels 5 minuts que ens triga amb l'aproximació que es demana a l'enunciat.
La versió del codi que funciona amb aquesta aproximació es pot trobar a la carpeta computationalVersion que hi ha al projecte.

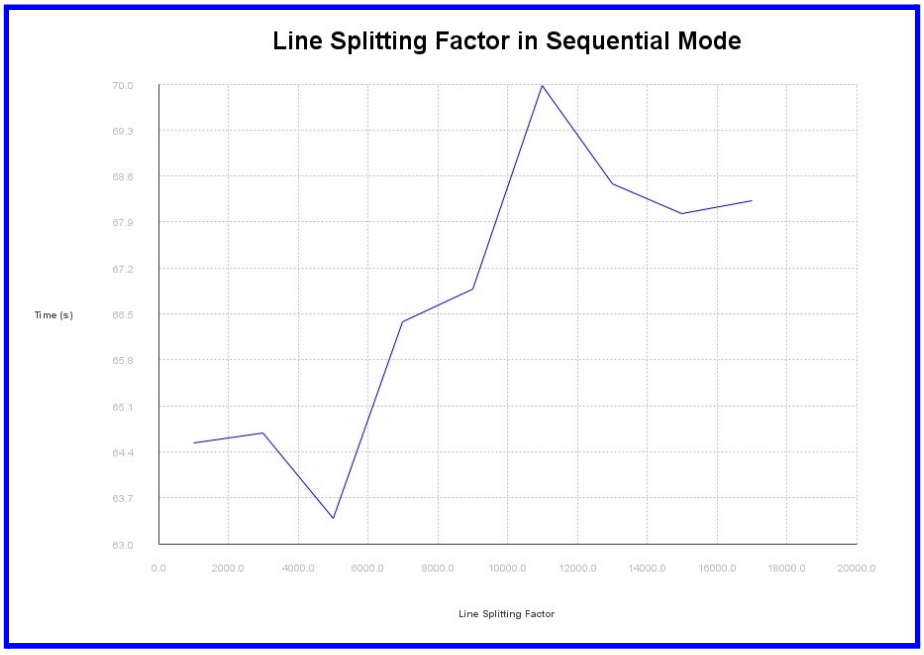
Buscant els millors paràmetres

Trobant el millor factor d'split per seqüencial

Al començar a fer gràfiques de rendiment del programa vam fer una simulació doblant el factor de line splitting per trobar quin era més adient, primerament vam fer que anés de 1000 a 1000000 multiplicant cada cop per 2, però el resultat ens va aclarir que no havíem de fer servir factors tant grans per fer split, com podem veure a la següent gràfica:

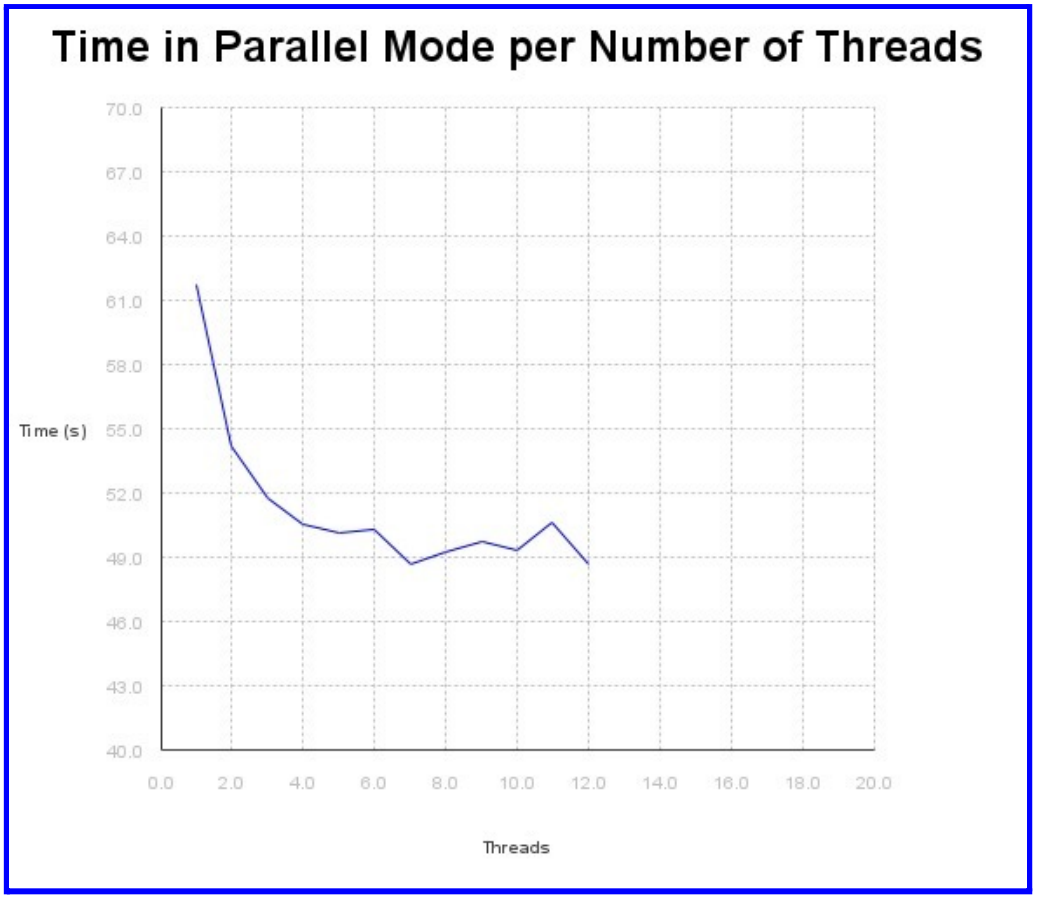


Per tant vam baixar el factor per comprovar des de 0 a 20000, el que ens va resultar en aquesta gràfica de rendiment, trobant que el més òptim en seqüencial és fer servir 5000 línies per fer l'split de l'arxiu:



Trobant el millor número de threads per paral·lel

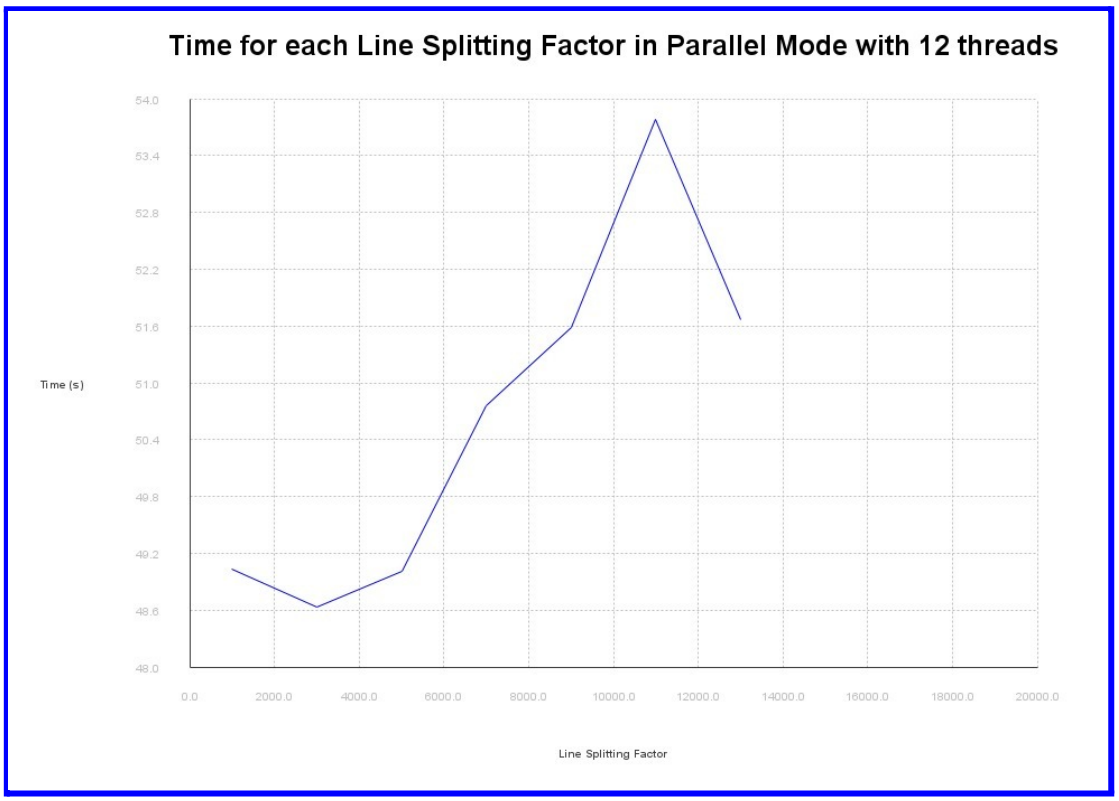
Es pot comprovar a la següent gràfica que els resultats van millorant a mesura que introduïm més threads, però hi ha un estancament de rendiment a partir de 6 threads, el que fa pensar que afegir-ne més no varia el resultat, caldrà veure amb processadors amb més threads quin seria el resultat.



La conclusió d'aquesta prova és que segons sembla es pot utilitzar el màxim de threads del procesador sense por a empitjorar el rendiment, ja que la tendència és mantenir el rendiment a mesura que es passa dels 6 threads.
També podem concloure que millora significativament el rendiment tenint una forta pendent descendent des del thread 1 al thread 4.

Trobant el millor line splitting en paral·lel

Un cop trobat que el número de threads no és significatiu sempre que siguin més de 6, vam fixar aquests al màxim del processador on vam fer les proves i vam iniciar el test per veure el factor d'split òptim, la gràfica resultant de les proves és la següent:



Com podem veure a la gràfica el factor d'split més òptim en paral·lel és de 3000 línies.

Comparació dels dos modes

Un cop obtinguts els valors òptims de cada mode d'execució ja podem començar a fer un benchmark entre ambdós modes per veure i comparar resultats. Com es pot veure a la gràfica següent, per una banda, en blau tenim el temps d'execució en seqüencial amb un Line Splitting Factor de 5000 en diferents fitxers de diferents mides. Per altra, tenim en vermell el temps d'execució en paral·lel amb 12 threads i un Line Splitting Factor de 3000 en els mateixos fitxers que el mode seqüencial.

Com a conclusió d'aquesta prova podem dir que si bé al principi el rendiment entre ambdós modes és quasi el mateix, a mesura que el fitxer augmenta de mida la diferència en el rendiment es torna cada cop més significativa declarant guanyador al mode paral·lel per a fitxers de gran volum.

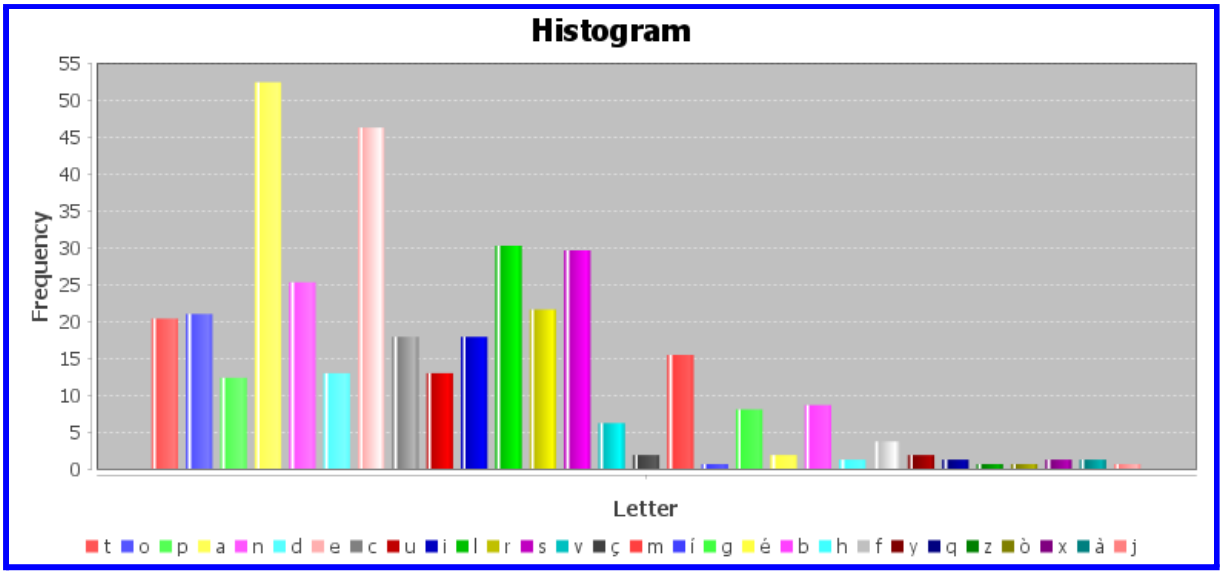
És precisament aquest l'objectiu de la pràctica, demostrar que amb la implementació del paradigma ***Map-Reduce*** en la nostra aplicació podem reduir el temps d'execució per a grans volums de dades.

Histogrames dels resultats

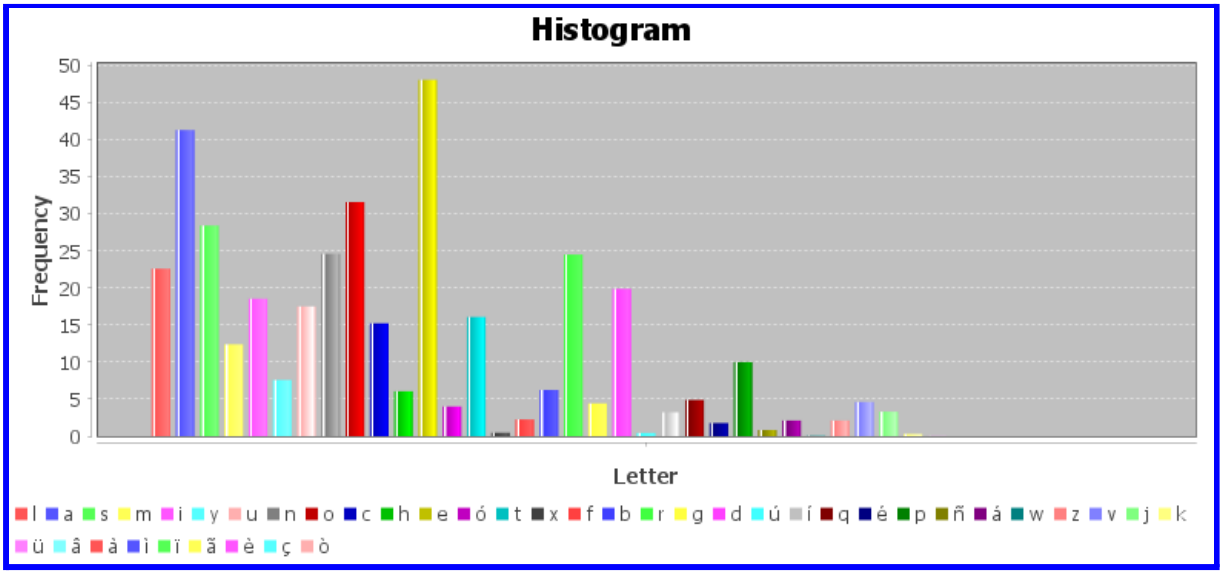
Els fitxers utilitzats per fer els histogrames següents han estat el de prova que tenim al moodle i un modificat per nosaltres. Tots els histogrames han estat generats usant l'aproximació en paral·lel.

Histograma fitxer per fitxer

Aquí podem veure el resultat d'executar el programa amb el paràmetre de l'histograma de la primera opció, és a dir un histograma per arxiu on hi han representades la freqüència de cada lletra del fitxer.



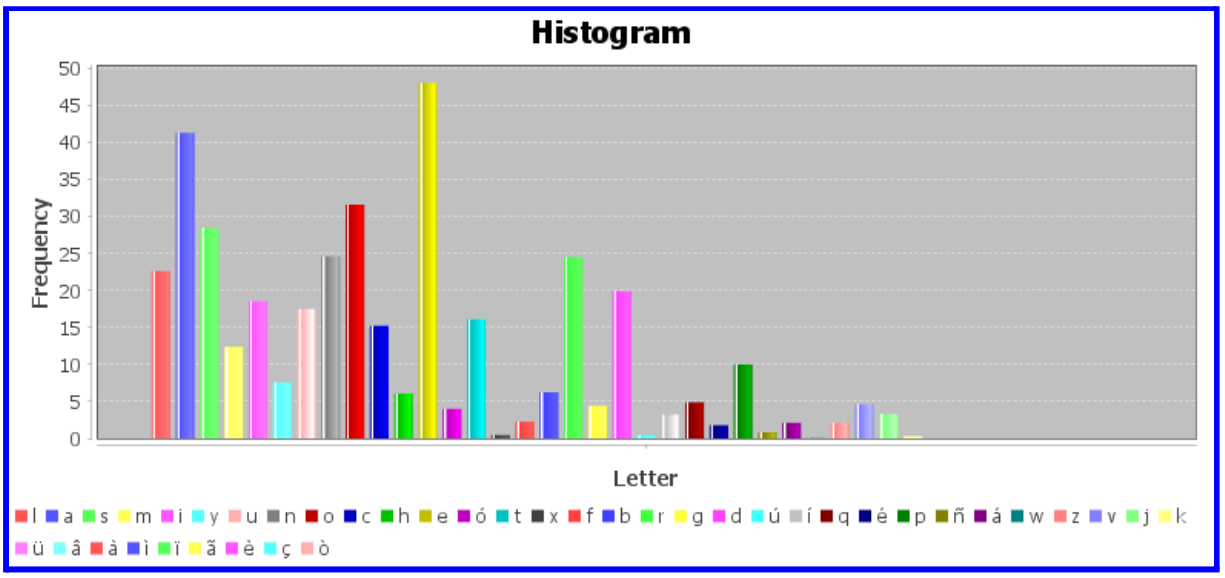
Histograma del fitxer de prova del moodle



Histograma del fitxer generat per nosaltres

Histograma de tots els fitxers

Per altra banda, aquí podem veure el resultat de l'execució del programa amb el paràmetre de l'histograma, però amb l'altre opció, és a dir un histograma comú de tots els arxius que passen per paràmetre, en aquest cas els mateixos que en l'apartat anterior.



Histograma comú dels dos fitxers de l'apartat anterior

Explicació de funcionament del programa

Per a l'execució del programa tenim diferents modes de funcionament.

Executar benchmark:

- -b File1.txt File2.txt … FileN.txt
- --benchamark File1.txt File2.txt … FileN.txt

Executar en paral·lel:

- -p -sf <Numero per definir el factor d'split> File1.txt File2.txt … FileN.txt
- --parallel -sf <Numero per definir el factor d'split> File1.txt File2.txt … FileN.txt

Executar en sequencial:

- -s -sf <Numero per definir el factor d'split> File1.txt File2.txt … FileN.txt
- --sequential -f <Numero per definir el factor d'split> File1.txt File2.txt … FileN.txt

Executar histograma per tots els arxius:

- -his True -sf <Numero per definir el factor d'split> File1.txt File2.txt … FileN.txt

Executar histograma arxiu per arxiu:

- -his False -sf <Numero per definir el factor d'split> File1.txt File2.txt … FileN.txt

Veure ajuda:

- -h
- --help

Execució de l'entorn de validació

Per a poder compilar el programa haurem de situar-nos a l'arrel del projecte, on es troba el fitxer dockerfile i el fitxer d'script que hem d'utilitzar. Un cop allà proseguim a executar les comandes següents:

docker build -t mapreduce .
docker run -it -v "path on es tinguin els arxius":/data mapreduce ./TextCounter File1.txt File2.txt File3.txt