Sistemes Concurrents i Paral·lels

Pràctica 1

Joel Aumedes i Joel Farré

Anàlisi del disseny

Per a poder obtenir una versió més eficient i ràpida del problema Manfut, s'ha pensat que faria falta alguna cosa més que simplement repartir els equips a calcular entre els *threads*. Per això, es va decidir que la millor manera seria reescriure el problema, de manera que per a trobar tots els equips es generés un arbre recursiu.

La idea original era implementar també programació dinàmica, però es va descartar, ja que les condicions del problema fan que sigui gairebé impossible que es repeteixi una configuració, i això causaria un sobrecost molt gran al haver de mantenir una matriu enorme de possibles configuracions.

Disseny sequencial

La idea principal de la implementació és la següent:

```
Començant per l'últim jugador:
      Tinc 2 opcions; Agafar-lo o no agafar-lo
      Si l'agafo:
            CRIDA RECURSIVA: Repeteixo la funció amb el jugador a l'equip per calcular el millor equip amb aquest jugador
      Si no l'agafo:
            CRIDA RECURSIVA: Repeteixo la funció amb el jugador fora de l'equip per calcular el millor equip sense aquest jugador
      Un cop tinc els 2 equips:
      Si el millor equip és agafant-lo:
            Retorno el millor equip amb el jugador
      Si el millor equip és sense agafar-lo:
            Retorno el millor equip sense el jugador
```

D'aquesta manera, pot semblar que solament es calcula sobre un jugador, però, degut a les crides recursives, es veu que es necessita calcular tots els equips inferiors per a trobar el superior, per exemple, es necessita saber totes les combinacions de porters per a poder decidir sobre un defensa.

Aplicació de la concurrència

En aquest cas, s'utilitza la bisecció recursiva per a repartir les tasques. Tot i això, no es pot generar *threads* a cada bisecció, ja que hi ha un límit de *threads* i també perquè causaria un sobrecost immens. Aleshores el que es fa és biseccionar amb un thread solament quan es té un nombre inferior de *threads* actius als permesos.

Per biseccionar s'utilitza el següent disseny:

```
Començant per l'últim jugador:
      Tinc 2 opcions; Agafar-lo o no agafar-lo
      Si l'agafo:
            Si tinc espai per generar un thread:
                  pthread_create(CRIDA RECURSIVA: Repeteixo la funció amb el jugador a l'equip per calcular el millor equip amb aquest jugador)
            Si no tinc espai:
                  CRIDA RECURSIVA: Repeteixo la funció amb el jugador a l'equip per calcular el millor equip amb aquest jugador
      Si no l'agafo:
            CRIDA RECURSIVA: Repeteixo la funció amb el jugador fora de l'equip per calcular el millor equip sense aquest jugador
      Si he creat un thread:
            pthread_join()
      Un cop tinc els 2 equips:
      Si el millor equip és agafant-lo:
            Retorno el millor equip amb el jugador
      Si el millor equip és sense agafar-lo:
            Retorno el millor equip sense el jugador
```

Es realitza la bisecció recursiva, però si no hi ha espai per a un *thread* nou, no es creem. Es pot veure de la següent manera: Al necessitar calcular si és millor agafar un jugador o no, el *thread* principal, si hi ha espai, crea un altre *thread* que calcularà afegint el jugador, i el *thread* principal calcularà sense afegir-lo. Si no hi ha espai el càlcul el farà tot el *thread* principal.

Anàlisi del rendiment

Per a dur a terme l'anàlisi del rendiment per a cada fitxer, s'ha agafat com a valors del pressupost 1000 i com a nombre de *threads* 2. Un cop seleccionats aquest valors, es mostren els temps d'execució a la taula següent. Per a que les condicions siguin les mateixes, s'ha agafat com a referència de temps el valor obtingut en la primera execució.

Fitxers	**Temps**			
	C		Java	
	Secuencial_C	Concurrent_C	Secuencial_Java	Concurrent_Java
mercat15j	0,198''	0,004''	0,202''	0,410''
mercat25j	4,119''	0,018''	0,8''	0,552''
mercat50j	3' 36,661''	0,595''	45,321''	3,399''
mercat60j	24' 14,709''	5,202''	5' 12,662''	18,543''
mercat75j	2h 45' 52,911''	34,422''	3h 12' 34,865''	2' 13,874''
mercat100j	23h 13' 32,031''	4' 58,954''	22h 28' 54,274''	16' 42,990''

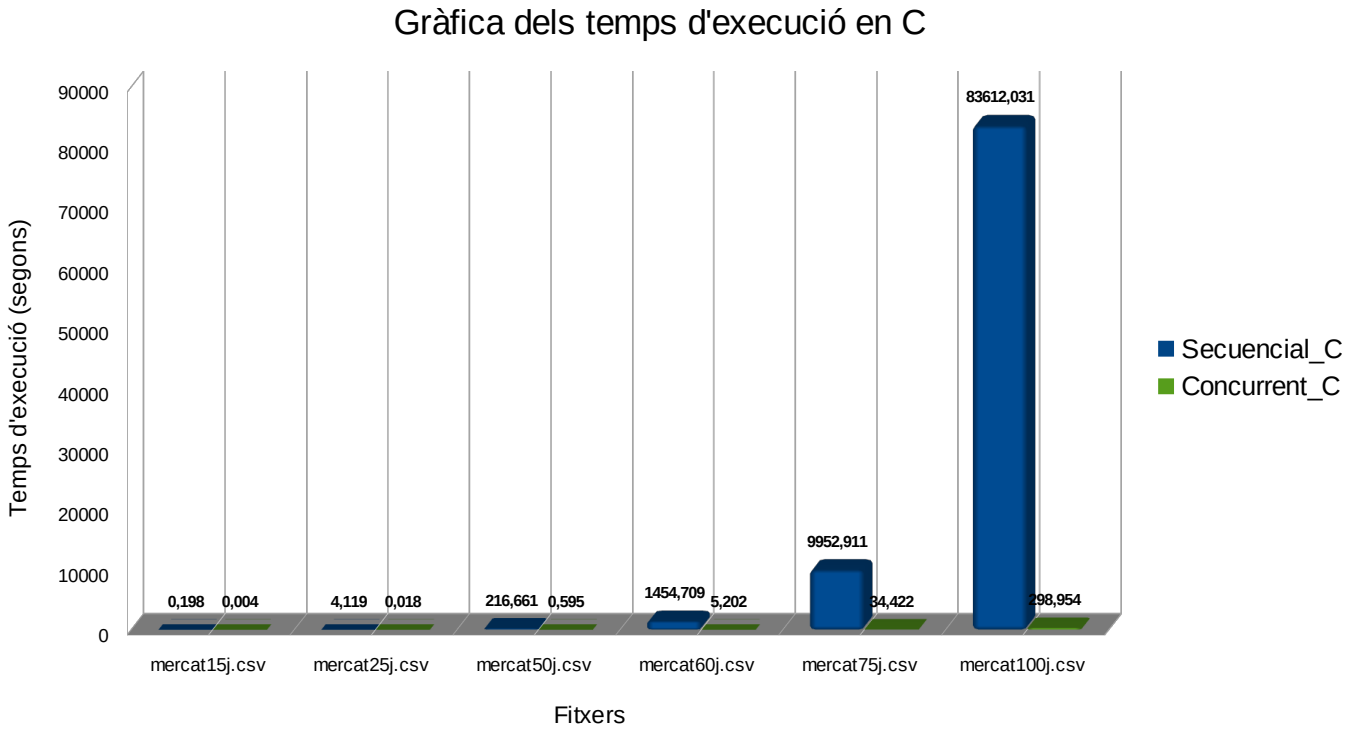
Després d'executar, per a cada fitxer, tan els programes seqüencials com els concurrents, es pot observar que al principi la millora de temps es molt poca per no dir quasi inexistent. Per al fitxer *mercat15j.csv* la millora es quasi inapreciable en C. En Java es pot observar que l'execució de forma concurrent tarda una mica més que la seqüencial, això es degut al sobrecost de la recursivitat.

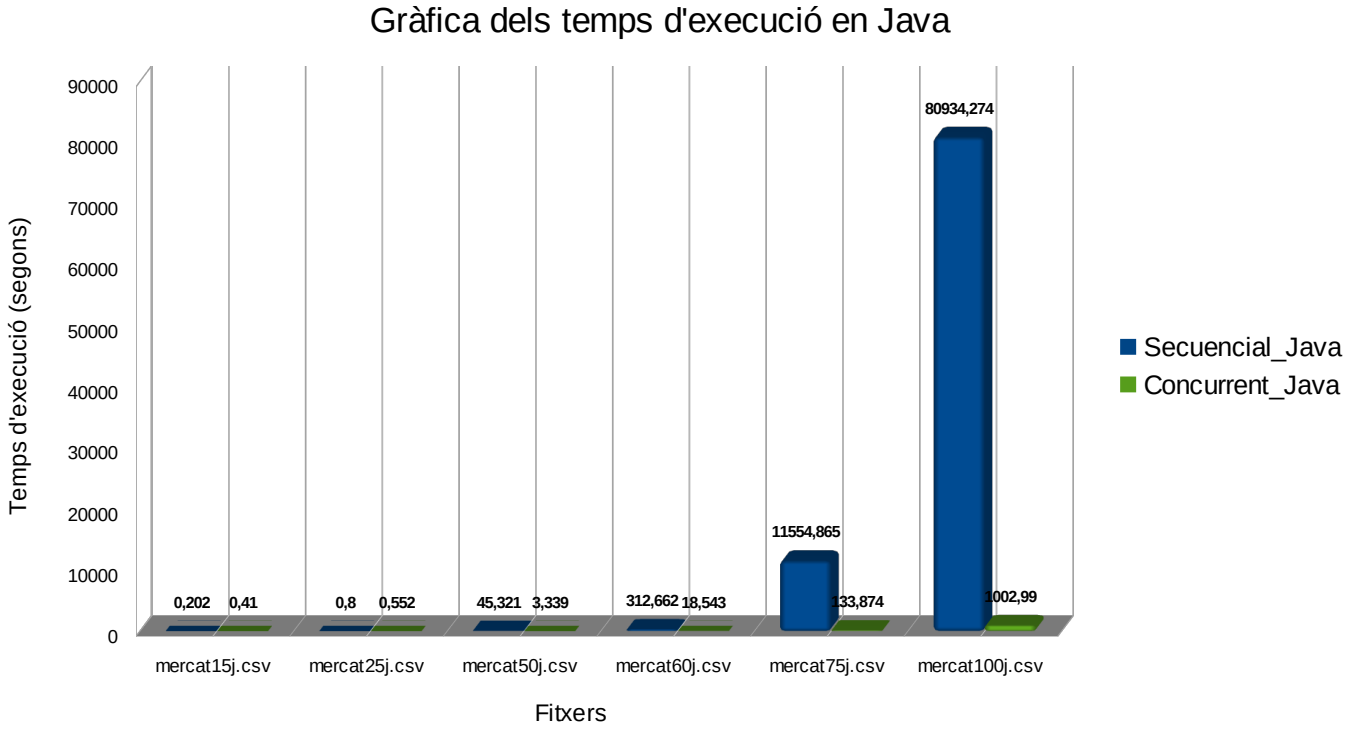
A partir del fitxer *mercat25j.csv* ja es pot veure un petit increment en la millora del temps de forma concurrent.

Al fitxer *mercat50j.csv* es nota més la diferència essent aquesta de quasi 3 minuts i mig entre l'execució seqüencial i concurrent en la versió de C i de 5 minuts en la versió de Java.

Per als fitxers *mercat60j.csv*, *mercat75j.csv* i *mercat100j.csv* la diferència de temps es abismal, ja que la diferència entre l'execució seqüencial i concurrent pot arribar a ser de minuts i fins i tot d'hores.

Com a resultat d'aquest anàlisis podem destacar que l'aplicació de *threads* als programes fa que aquests tinguin un major rendiment i eficiència.


Gràfica dels temps d'execució en C
Temps d'execució (segons)
90000
80000
70000
60000
50000
40000
30000
20000
10000
0
0,198
0,004
4,119
0,018
216,661
0,595
1454,709
5,202
9952,911
34,422
83612,031
298,954
mercat15j.csv
mercat25j.csv
mercat50j.csv
mercat60j.csv
mercat75j.csv
mercat100j.csv
Fitxers
Secuencial_C
Concurrent_C


Gràfica dels temps d'execució en Java
Temps d'execució (segons)
90000
80000
70000
60000
50000
40000
30000
20000
10000
0
0,202
0,41
0,8
0,552
45,321
3,339
312,662
18,543
11554,865
133,874
80934,274
1002,99
mercat15j.csv
mercat25j.csv
mercat50j.csv
mercat60j.csv
mercat75j.csv
mercat100j.csv
Fitxers
Secuencial_Java
Concurrent_Java

Errors durant el desenvolupament

Pel que fa als errors que hem tingut durant el desenvolupament han sigut pocs. El principal problema era que a l'hora de calcular l'equip òptim resulta que per algun motiu no es sumaven be els valors i els costos dels jugadors i el resultat final que es mostrava per pantalla no era el mateix que si es sumaven els valors i els costos dels jugadors manualment. El error era que enlloc de copiar els equips, el que fèiem era passar una referència i per tant això provocava dependència de dades. Per a solucionar això s'ha creat una funció que copia els equips enlloc de passar-los per referència.

Un altre petit error era que quan volíem accedir a algunes posicions per a imprimir un jugador resulta que ens donava un error i deia que aquella posició tenia com a valor *null*. El error es trobava en la funció *addPlayer()* de la classe *JugadorsEquip*, i resulta que faltava afegir la comanda *break;* a l'hora d'afegir un jugador ja que al tenir un bucle, s'afegia el mateix jugador a totes les posicions de l'*array* enlloc d'afegir-lo a la primera posició que trobava buida de l'*array* i llavors sortir de la funció.

Conclusions

Ha estat interessant implementar aquest problema d'optimització i observar les conseqüències que sorgeixen al implementar diferents mètodes de disseny. Ens ha impressionat l'efecte que causa el implementar *threads* als programes ja que la seva millora en temps pot arribar a ser d'hores o de dies.

Consideracions

S'ha utilitzat variables compartides per a la sincronització per assegurar una execució determinista, però degut a que no tenen tanta precisió com un mutex o un bloc syncronized, és possible que es segueixin donant errors, però molt més rarament.

Per a cancel·lar els fils en cas d'error, s'ha assegurat de que els fils són daemons, de manera que en cas d'error sol fa falta realitzar un *exit* del fil principal, que es realitza un cop detectat l'error.