Universitat de Lleida
Escola Politècnica Superior

10/01/2021

Pràctica 2

Grau en Enginyeria Informàtica

Joel Farré Cortés (78103400T)
Joel Aumedes Serrano (48051307Y)
ESCOLA POLITÈCNICA SUPERIOR – UNIVERSITAT DE LLEIDA

Errors de la primera pràctica

Per a poder implementar la pràctica 2, primer vam arreglar els erros de la pràctica 1 de la següent manera:

Si l'usuari passa 0 *threads* per paràmetre, s'avisa a l'usuari i s'estableix per defecte 1 *thread.*

L'execució a vegades era no determinista perquè hi havia un error al passar el resultat d'un *thread* al *thread* pare, de manera que el *thread* pare es pensava que no era possible tenir aquell jugador quan si que ho era. Pel mateix motiu, en algunes execucions es mostrava un equip de solament sis jugadors.

També vam alliberar els espais de memòria necessaris.

Respecte a la suma del valor dels jugadors que era errònia, vam revisar moltes execucions calculant el cost a mà i en cap donava errònia. Creiem que hi va haver una confusió a la correcció de la pràctica 1 degut a dos equips amb jugadors diferents, costos diferents però el mateix valor.

En la nostra implementació seguim emprant fils *daemon* ja que serveixen per a oferir un servei als *threads* usuari. Els *threads daemon* quan no queden fils usuari vius a la JVM automàticament es cancel·len i s'eliminen del sistema. No es queden vius tal i com se'ns va dir a la correcció de la pràctica 1.

Sincronització

- Evitar les condicions de carrera

Respecte a la pràctica 1 hem pogut evitar les condicions de carrera de forma més eficient utilitzant *mutexs* i blocs *syncronized.* Degut al disseny de l'algoritme emprat és probable que les estadístiques no donin sempre el mateix ja que es pot donar el cas de que l'accés al processador no sigui sempre el mateix.

- Thread de missatges

Hem utilitzat variables de condició per a implementar el *thread* de missatges de forma eficient. La idea principal es que cada *thread* que vol escriure un missatge adquireix un *mutex* o un bloc *syncronized* per escriure el seu missatge a una cua de missatges. Un cop escrit el missatge i abans d'alliberar el *mutex* el *thread* notifica al *thread* de missatges. El *thread* de missatges ha estat esperant en un *wait()* amb la variable de condició. Un cop despert, comprova si la condició es compleix o no. Aquesta condició és si tenim 100 missatges a la cua. Si no disposem de 100 missatges, el *thread* de missatges se'n torna a esperar. Si en canvi, disposem d'aquests 100, els imprimeix per pantalla tots i els elimina per a deixar espai per a la resta. La implementació és molt semblant entre C i Java.

- Espera del resultat final

Aquí es realitza de forma diferent entre C i Java. A les dues implementacions utilitzem un mètode de sincronització i variables compartides per al resultat però també fem servir la funció *join()* per a assegurar-nos de que el *thread* ha finalitzat correctament. La diferència entre els dos llenguatges es troba en el mètode de sincronització implementat. En C utilitzem una barrera i en Java un semàfor.

En C inicialitzem una barrera per a esperar a dos *threads*. Aquesta barrera es planta en el *thread* que espera el resultat quan el necessita i en el *thread* que el calcula quan aquest resultat ja està disponible per a l'altre *thread*. Un cop sortit de la barrera, el *thread* que ha calculat simplement acaba i l'altre *thread* fa el *join*.

En Java utilitzem un semàfor. Aquest semàfor l'inicialitzem a zero *permits*, llavors quan el *thread* que necessita el resultat, el necessita, intenta adquirir un *permit*, l'altre *thread* alliberarà aquest *permit* quan el resultat ja estigui disponible a la variable compartida.

- Estadístiques

Per a l'emmagatzematge i tractament de les estadístiques en C s'ha utilitzat un *struct* i en Java una *class*. Degut a l'arquitectura del programa, solament contem les estadístiques per a cada *slot* de del *thread* i no per a cada *thread* ja que es creen i s'eliminen amb molta freqüència. Les estadístiques es comptabilitzen de la següent manera:

- Nombre de combinacions vàlides: Es sumarà una unitat cada cop que el *thread* tingui un equip complet.

- Nombre de combinacions no vàlides: Es sumarà una unitat quan s'intenti afegir un jugador en un equip però que no sigui possible ja que o l'equip estigui complet o no es disposi del suficient pressupost. També es sumarà quan s'hagi comprovat tots els jugadors i l'equip no estigui complet.

- Nombre de combinacions avaluades: És la suma de les combinacions vàlides i no vàlides.

- Cost mig de les combinacions vàlides: És la divisió del cost total de les combinacions vàlides entre el nombre de combinacions vàlides.

- Puntuació mitja de les combinacions vàlides: És la divisió de la puntuació total de les combinacions vàlides entre el nombre de combinacions vàlides.

- Millor i pitjor combinacions: Combinació amb tots els jugadors amb millor i pitjor puntuació.

Quan un *thread* modifica les seves pròpies estadístiques, no utilitza cap mètode de sincronització ja que sabem que l'únic *thread* que les modificarà serà ell mateix. Per l'altra banda, per als *threads* globals, es regula l'accés amb l'ús de semàfors.

- Progrés

Per al progrés es té en compte el nombre de combinacions avaluades de cada *thread*. L'arquitectura del programa podria causar que, en el cas de que quan un *thread* arriba a M combinacions s'esperi a que arribin els altres, ens trobéssim en una situació de *deadlock* ja que un *thread* parat no generaria més *threads* i això faria que no arribés a les M combinacions. Aleshores, el que fem es imprimir les estadístiques globals i les parcials cada cop que tots els *threads* hagin completat una etapa, però quan un *thread* ha completat l'etapa, no s'espera a que els altres la completin.

Anàlisi del rendiment

Per a fer l'anàlisi de rendiment per a cada diferent mercat els valors emprats han sigut els següents: 4 *threads*, un pressupost de 200. I les característiques del processador de l'ordinador que s'ha fet servir son les següents: Processador d'1 *socket*, 2 cores per cada *socket* i 2 *threads* per *core*.

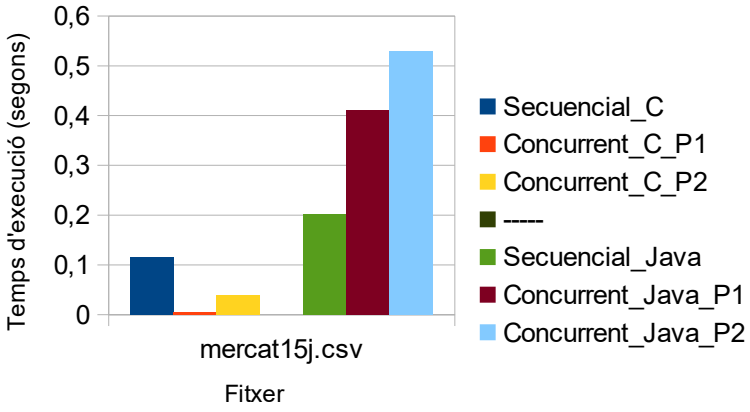
Fitxers	**Temps**					
	Seqüencial C	Seqüencial Java	Concurrent C_P1	Concurrent Java_P1	Concurrent C_P2	Concurrent Java_P2
mercat15j	0,116s	0,202s	0,004s	0,410s	0,039s	0,53s
mercat25j	3,979s	0,8s	0,018s	0,552s	0,072s	1,232s
mercat50j	3m 36,661s	45,321s	0,595s	3,399s	1,973s	4,004s
mercat60j	24m 14,709s	5m 12,662s	5,202s	18,543s	17,867s	45,023s
mercat75j	2h 45m 52,911s	3h 12m 34,865s	34,422s	2m 13,874s	1m 29,043s	4m 22,078s
mercat100j	23h 13m 32,031s	22h 28m 54,274s	4m 58,954s	16m 42,990s	10m 39,431s	27m 15,234s

Després d'observar els diferents temps d'execució per als programes tan en C com en Java podem treure les següents conclusions: Per al fitxer *mercat15j.csv* la millora de C i Java entre la forma seqüencial i la concurrent es inapreciable ja que únicament la diferència es de dècimes de segon. Pel que fa la diferència entre la pràctica 1 i la 2, l'increment de temps també es d'unes dècimes i centèsimes de segon.

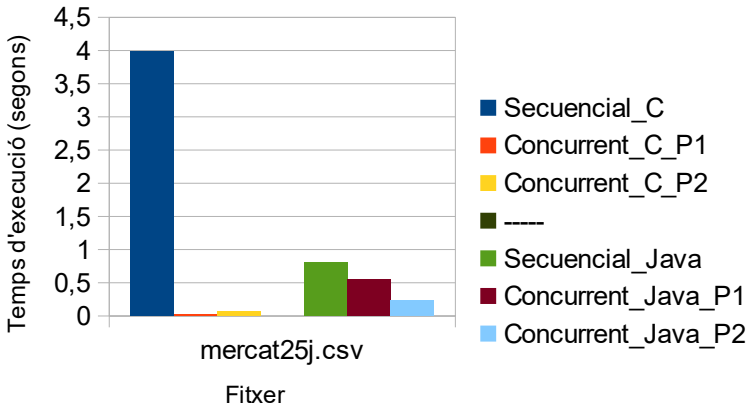
Per al *mercat25j.csv* ja podem començar a observar una millora d'alguns segons en el temps d'execució respecte a la versió seqüencial en la part de C i un increment però lleuger de temps pel que fa al temps de la pràctica 1.

En relació al *mercat50j.csv* i el *mercat60j.csv* la millora de temps ja s'aprecia bastant, arribant a ser de minuts en totes dues versions i l'increment de temps respecte a la pràctica 1 només es de segons.

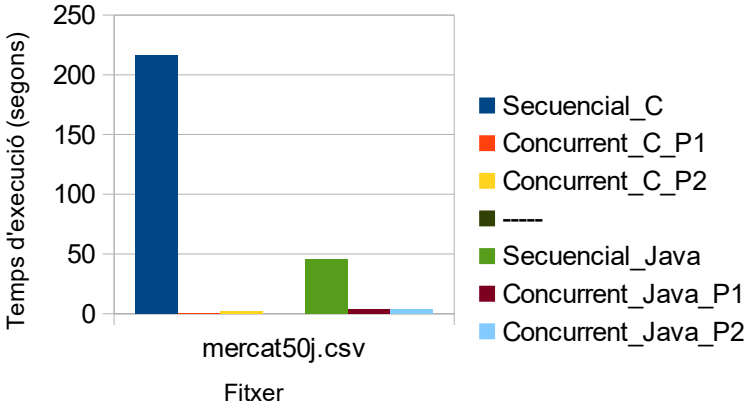
Finalment pel que fa als fitxers *mercat75j.csv* i *mercat100j.csv* la diferència es molt notòria ja que la versió concurrent és algunes hores més ràpida i l'increment entre la pràctica 1 i la 2 és d'alguns minuts únicament.


0,6
0,5
0,4
0,3
0,2
0,1
0
Temps d'execució (segons)
mercat15j.csv
Fitxer
Secuencial_C
Concurrent_C_P1
Concurrent_C_P2

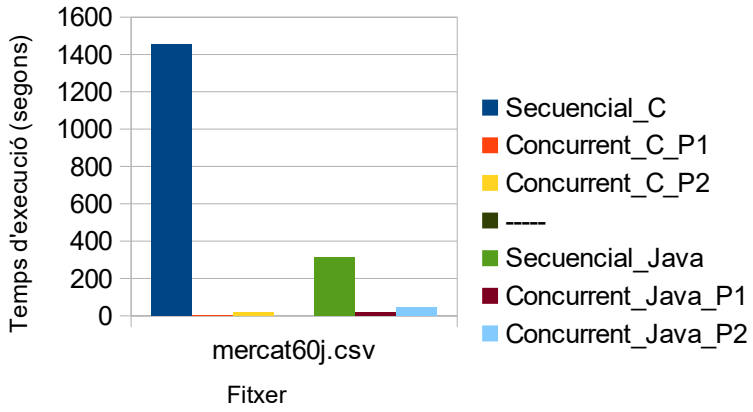
Secuencial_Java
Concurrent_Java_P1
Concurrent_Java_P2

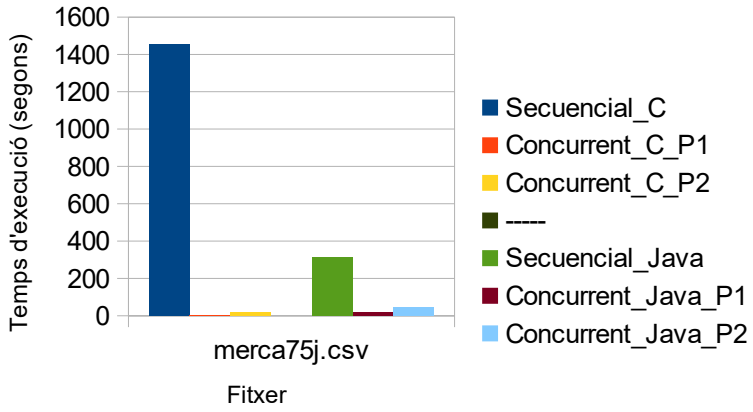

4,5
4
3,5
3
2,5
2
1,5
1
0,5
0
Temps d'execució (segons)
mercat25j.csv
Fitxer
Secuencial_C
Concurrent_C_P1
Concurrent_C_P2

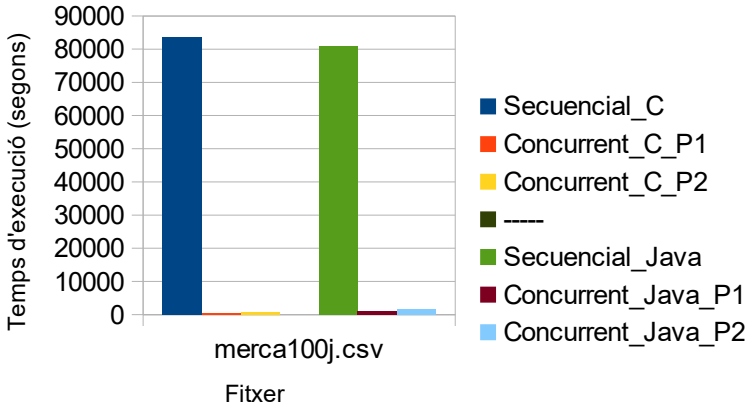
Secuencial_Java
Concurrent_Java_P1
Concurrent_Java_P2


250
200
150
100
50
0
Temps d'execució (segons)
mercat50j.csv
Fitxer
Secuencial_C
Concurrent_C_P1
Concurrent_C_P2

Secuencial_Java
Concurrent_Java_P1
Concurrent_Java_P2







Conclusions

Com a conclusions podem destacar la gran ajuda que aporta la sincronització a través de semàfors, *mutexs*, i de barreres per a així fer que el programa sigui determinista i també poder evitar les condicions de carrera.