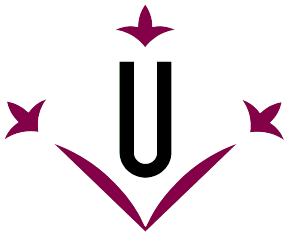

Universitat de Lleida
Escola Politècnica Superior

Sistemes Concurrents i Paral·lels

Pràctica 2: *Threads i sincronització*

Sergi Puigpinós Palau
Jordi Rafael Lazo Florensa

10 de gener de 2021

Índex

Índex de figures

1 Versió en JAVA

Per la implementació en Java, hem introduït dues noves classes: la *Statistics* i la *threadMessenger*. La primera s'encarrega de totes les dades i operacions relacionades amb les estadístiques, llavors cada *thread* tindrà una instància d'aquesta classe per emmagatzemar i calcular les seves pròpies estadístiques i després tindran una instància d'aquesta que serà compartida la qual s'encarregarà de les estadístiques globals. La segona, serà la classe que executarà el *thread* responsable d'imprimir missatges per pantalla i la qual tindrà una llista de *strings* on cada *thread* deixarà els missatges per imprimir. Així doncs, tant les classes *Market* com *threadEvaluator* han sigut modificades per adaptar els nous canvis proposats per aquesta segona pràctica.

Evaluadors

L'evaluador comença l'execució semblant a la darrera pràctica, però aquest cop quan arriba a la part de veure si ha trobat un millor equip o no, aquesta part, s'executarà en una funció que conté *synchronized* per assegurar que només un *thread* pugui executar-la. Això és pel fet que estem tractant amb una part crítica del programa, ja que es fa lectura i escriptura d'una variable compartida (en aquesta pràctica es comparà amb el millor equip de tots, en comptes del millor de cada *thread*). Després de fer la comprovació, passa a calcular les seves estadístiques parcials i torna a començar el bucle.

Quan els *thread* avaluadors han calculat M combinacions, passen a executar la funció *statisticsSummary*, aquesta funció s'executa tant per les estadístiques parcials com per les finals i és responsable de sincronitzar tots els *threads*, calcular les estadístiques globals i enviar una "senyal"al pare en el cas de les estadístiques finals. El funcionament d'aquesta és el següent: primer s'utilitza un *lock*, una condició i un comptador combinat amb un bucle *while*, per esperar que tots els *thread* arribin. Un cop l'últim arribi, aquest no entrarà dins del bucle sinó que entrarà a la zona critica la qual consistirà a imprimir les seves estadístiques parcials, afegir la seva informació a les estadístiques globals, despertar un *thread* dels que estaven esperant al bucle i anar-se a esperar, utilitzant una condició, a què l'últim *thread* acabi d'executar aquesta secció crítica. Això amb tots els *threads*, excepte l'últim que farà el mateix, però a més, s'encarregarà d'imprimir les estadístiques globals, reiniciar el comptador i els valors de la instància de l'estadística global, i quan hagi acabat, despertarà a tots els altres *threads* i continuaran amb la seva execució. En cas de tractar-se de les estadístiques finals, s'incrementa el comptador (perquè puguin sortir del bucle *while* del final del programa, el qual és explicat al següent paràgraf) i es desperta al pare amb una condició (senyal) perquè sàpiga que els *threads* avaluadors ja han acabat.

En l'etapa final d'un avaluador, quan ja ha acabat de calcular totes les seves combinacions, incrementa un comptador que portarà compte de quants *threads* han acabat i s'anirà dins d'un bucle on estarà executant la funció *statisticsSummary* fins que acabin tots els *threads*. Això es fa perquè com el repartiment de feina no pot ser exacte, hi ha alguns que acabaran abans que altres, i com que per executar les estadístiques és necessari tots els *threads*, els que acabin es queden esperant en aquella funció en el cas que s'hagi d'imprimir més estadístiques parcials o que l'últim acabi i imprimeixin les estadístiques finals.

Messenger

Quan algun dels *threads* avaluadors ha d'imprimir un missatge, utilitza la funció *addMessageToQueue*. Aquesta funció fa ús d'un semàfor per limitar la quantitat de missatges dins la llista i un *lock* per controlar que solament un *thread* estigui accedint en aquesta. Cada vegada que s'afegeix un element a la llista, es desperta al *thread* responsable d'imprimir i aquest comprova si la llista està plena o no. Quan aquesta ho està, el *threads* responsable imprimeix tots els

missatges de la llista i seguidament allibera tantes instàncies del semàfor com espais lliures hi han definits a la llista.

Sincronització final

Aquesta vegada, per esperar que els avaluadors calculin la solució final, el thread principal s'espera mitjançant una condició. Llavors quan els threads avaluadors ja han acabat, es desperta al principal i seguidament aquest executarà la funció per tancar el thread messenger, la qual consistirà a despertar-lo i amb un bolea li dirà que es tanqui, llavors aquest imprimirà el que hi hagui a la llista i sortirà del bucle i notificarà al thread principal, mitjançant una condició, que ja ha acabat. Llavors quan el principal es desperta, continua amb la seva execució normal.

1.1 Temps d'execució amb 15j

Aplicació no sincronitzada amb 15j
- *Seqüencial* té un temps d'execució de: **0.155 s**.
- 2 *threads* té un temps d'execució de: **0.134 s**.
- 3 *threads* té un temps d'execució de: **0.127 s**.
- 4 *threads* té un temps d'execució de: **0.127 s**.

Aplicació sincronitzada amb 15j
- *Seqüencial* té un temps d'execució de: **0.230 s**.
- 2 *threads* té un temps d'execució de: **0.199 s**.
- 3 *threads* té un temps d'execució de: **0.200 s**.
- 4 *threads* té un temps d'execució de: **0.200 s**.

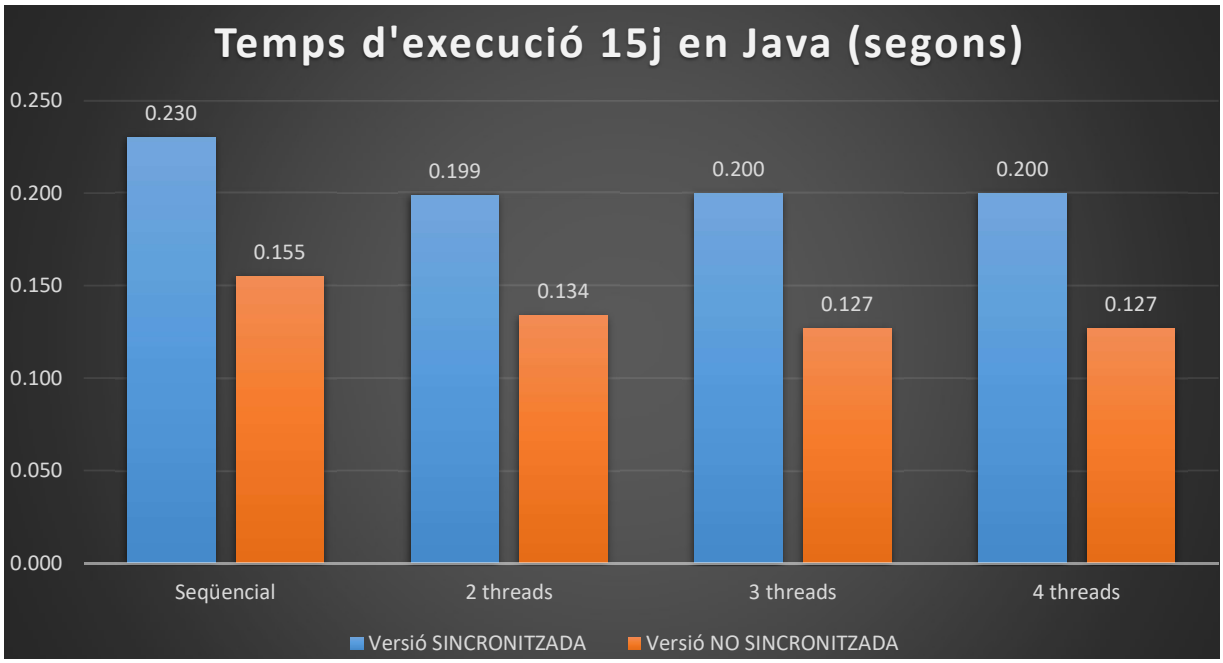

Figura 1: Gràfica del temps d'execució de la versió seqüencial, concurrent sincronitzada i concurrent no sincronitzada en Java amb 15j

	2 *threads*	**3 *threads***	**4 *threads***
Seqüencial	13.48%	13.04%	13.04%
2 *threads*	-	-0.50%	-0.50%
3 *threads*	-	-	0.00%

Taula 1: Percentatges de reducció de temps entre la versió concurrent sincronitzada i la versió seqüencial en Java amb 15j

	VC - No sincronitzada		
VC - Sincronitzada	2 *threads*	3 *threads*	4 *threads*
2 *threads*	32.66%	36.18%	36.18%
3 *threads*	33.00%	36.50%	36.50%
4 *threads*	33.00%	36.50%	36.50%

Taula 2: Percentatges de reducció de temps entre la versió concurrent sincronitzada i la versió concurrent no sincronitzada en Java amb 15j

En la Figura 1 es pot observar com el temps d'execució de la versió seqüencial a la versió concurrent sincronitzada amb només 2 *threads* s'ha reduït de 0.230 s fins a 0.199 s , reduint el temps d'execució en un 13.48%. Amb la utilització de 3 i 4 *threads* els temps s'ha reduït un 13.04%. Si es compara els temps obtinguts entre els *threads* sincronitzats es pot observar que els percentatges son negatius o iguals a 0. Això és degut al fet que el fet d'incrementar el nombre de *threads* en aquest tipus de dades tan petites, acaba perjudicant el temps d'execució del programa perquè es perd més temps en la creació,unió i sincronització dels *threads* que en l'execució de les tasques.

Pel que fa a la Taula 2, aquesta mostra les millores en percentatges de les versions sincronitzades versus les no sincronitzades. Així doncs, es pot observar com hi ha una millora de entre el 32% i 36% entre les diferents configuracions de *threads*. Això es degut a que en sincronització si s'esta executant algun fil, s'ha d'esperar a que acabi abans de començar a fer una altra tasca. Aquesta sincronització inexorablement fa augmentar el temps d'execució.

1.2 Temps d'execució amb 25j

Aplicació no sincronitzada amb 25j
- *Seqüencial* té un temps d'execució de: **1.886 s**.
- 2 *threads* té un temps d'execució de: **1.340 s**.
- 3 *threads* té un temps d'execució de: **0.822 s**.
- 4 *threads* té un temps d'execució de: **0.796 s**.

Aplicació sincronitzada amb 25j
- *Seqüencial* té un temps d'execució de: **3.456 s**.
- 2 *threads* té un temps d'execució de: **2.480 s**.
- 3 *threads* té un temps d'execució de: **2.224 s**.
- 4 *threads* té un temps d'execució de: **1.562 s**.

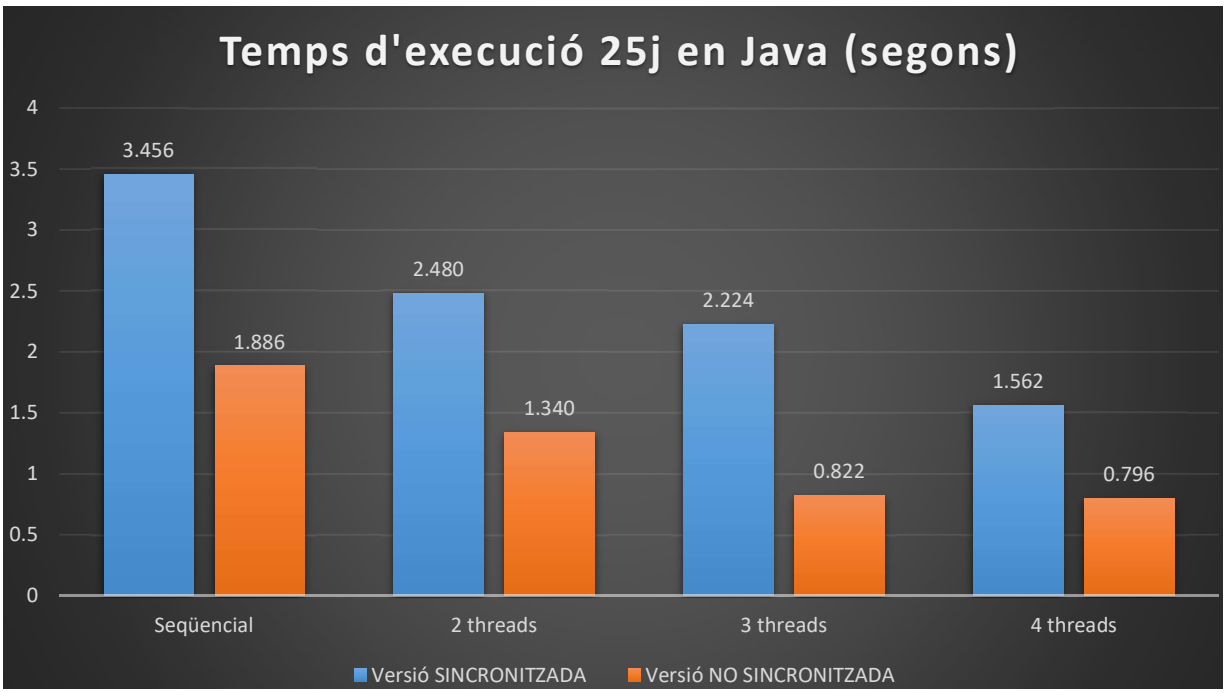

Figura 2: Gràfica del temps d'execució de la versió seqüencial, concurrent sincronitzada i concurrent no sincronitzada en Java amb 25j

	2 *threads*	**3 *threads***	**4 *threads***
Seqüencial	28.24%	35.65%	54.80%
2 *threads*	-	10.32%	37.20%
3 *threads*	-	-	29.77%

Taula 3: Percentatges de reducció de temps entre la versió concurrent sincronitzada i la versió seqüencial en Java amb 25j

	VC - No sincronitzada		
VC - Sincronitzada	2 *threads*	3 *threads*	4 *threads*
2 *threads*	45.97%	66.85%	67.90%
3 *threads*	39.75%	63.04%	64.21%
4 *threads*	14.21%	47.38%	49.04%

Taula 4: Percentatges de reducció de temps entre la versió concurrent sincronitzada i la versió concurrent no sincronitzada en Java amb 25j

Pel que fa als temps d'execució amb el mercat de 25j, aquests segueixen la mateixa característica que els obtinguts a Figura 1 amb la petita diferencia que, al tractar-se d'una quantitat de dades més gran, la diferencia entre utilitzar 3 i 4 *threads* es major i més visible. Així doncs, el millor temps obtingut es quan s'executa el programa amb 4 *threads* reduint el temps amb la versió seqüencial fins un 54.80%. Per tant, a major quantitat de fils, millor temps s'execució s'obtindrà (tenint en compte la configuració del processador).

Pel que fa a la Taula 4, es pot observar com hi ha una millora de entre el 14% i 67% entre les diferents configuracions de *threads* de la versió sincronitzada versus la no sincronitzada. A diferencia de la Taula 2, ara la reducció de temps es encara més notable ja que en sincronització si s'esta executant algun fil, s'ha d'esperar a que acabi abans de començar a fer una altra

tasca. Aquesta sincronització inexorablement fa augmentar el temps d'execució. La major diferencia es pot observar quan es compara la versió sincronitzada amb 2 *threads* amb la versió no sincronitzada de 4 *threads* perquè la millora de rendiment es del 67.90%.

1.3 Temps d'execució amb 50j

Aplicació no sincronitzada amb 50j
- *Seqüencial* té un temps d'execució de: **2.02 min**.
- 2 *threads* té un temps d'execució de: **1.04 min**.
- 3 *threads* té un temps d'execució de: **0.45 min**.
- 4 *threads* té un temps d'execució de: **0.36 min**.

Aplicació sincronitzada amb 50j
- *Seqüencial* té un temps d'execució de: **3.11 min**.
- 2 *threads* té un temps d'execució de: **2.08 min**.
- 3 *threads* té un temps d'execució de: **1.53 min**.
- 4 *threads* té un temps d'execució de: **1.40 min**.

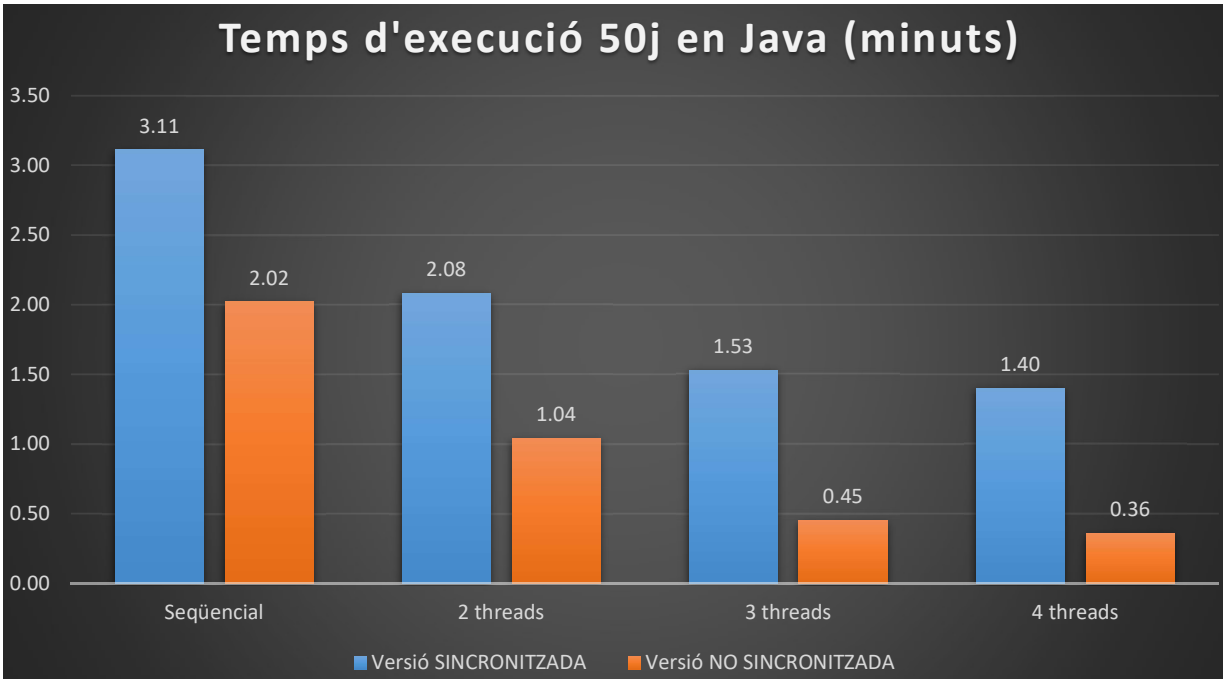

Figura 3: Gràfica del temps d'execució de la versió seqüencial, concurrent sincronitzada i concurrent no sincronitzada en Java amb 50j

	2 *threads*	**3 *threads***	**4 *threads***
Seqüencial	33.10%	50.80%	54.98%
2 *threads*	-	26.44%	32.69%
3 *threads*	-	-	8.50%

Taula 5: Percentatges de reducció de temps entre la versió concurrent sincronitzada i la versió seqüencial en Java amb 50j

	VC - No sincronitzada		
VC - Sincronitzada	2 *threads*	3 *threads*	4 *threads*
2 *threads*	50.00%	78.37%	82.69%
3 *threads*	32.03%	70.59%	76.47%
4 *threads*	25.71%	67.86%	74.29%

Taula 6: Percentatges de reducció de temps entre la versió concurrent sincronitzada i la versió concurrent no sincronitzada en Java amb 50j

Pel que fa als temps d'execució amb el mercat de 50j, aquests segueixen la mateixa característica que els obtinguts a Figura 2. Així doncs, el millor temps obtingut es quan s'executa el programa amb 4 *threads* reduint el temps amb la versió seqüencial fins un 54.98%. Per tant, a major quantitat de fils, millor temps s'execució s'obtindrà (tenint en compte la configuració del processador).

Pel que fa a la Taula 6, es pot observar com hi ha una millora de entre el 25% i 83% entre les diferents configuracions de *threads* de la versió sincronitzada versus la no sincronitzada. A diferencia de la Taula 4, ara la reducció de temps es encara més notable ja que en sincronització si s'esta executant algun fil, s'ha d'esperar a que acabi abans de començar a fer una altra tasca. Aquesta sincronització inexorablement fa augmentar el temps d'execució. La major diferencia es pot observar quan es compara la versió sincronitzada amb 2 *threads* amb la versió no sincronitzada de 4 *threads* perquè la millora de rendiment es del 82.69%. Finalment també es pot afirmar que ha major quantitat de dades a processar, la diferencia entre la versió sincronitzada i no sincronitzada serà més gran.

2 Versió en C

Pel que fa a la versió en C no es diferencia molt de la versió en Java, moltes de les diferències estan basades en la sintaxi i algunes coses que pel llenguatge s'ha de fer d'una manera diferent, però en termes d'implementació, aquesta és la mateixa que en Java. Alguns dels canvis que s'han fet, és en el cas de la funció *statisticsSummary*, la qual en la versió de Java utilitzàvem comptadors i condicions per esperar que els *threads* arribessin a un cert punt, però en canvi amb C com que disposem de barreres que fan la mateixa funció, ens podem estalviar alguns comptadors i condicions.

També com no disposem de *synchronized* en C, utilitzem un *mutex* on és necessari. I en canvi d'una llista utilitzem un *array* juntament amb un índex per emmagatzemar missatges, i finalment un *buffer* per cada *thread* perquè puguin passar el missatge al *array* sense tindre problemes.

Finalment, ja que en C no s'allibera memòria automàticament, hem de destruir tots els *locks*, condicions, barreres, etc, quan s'han acabat d'utilitzar.

2.1 Temps d'execució amb 15j

Aplicació no sincronitzada amb 15j
- *Seqüencial* té un temps d'execució de: **0.056 s**.
- 2 *threads* té un temps d'execució de: **0.046 s**.
- 3 *threads* té un temps d'execució de: **0.039 s**.
- 4 *threads* té un temps d'execució de: **0.037 s**.

Aplicació sincronitzada amb 15j $\begin{cases} \textit{Seqüencial} \text{ té un temps d'execució de: } \mathbf{0.090\ s.} \\ 2\ \textit{threads} \text{ té un temps d'execució de: } \mathbf{0.073\ s.} \\ 3\ \textit{threads} \text{ té un temps d'execució de: } \mathbf{0.060\ s.} \\ 4\ \textit{threads} \text{ té un temps d'execució de: } \mathbf{0.054\ s.} \end{cases}$

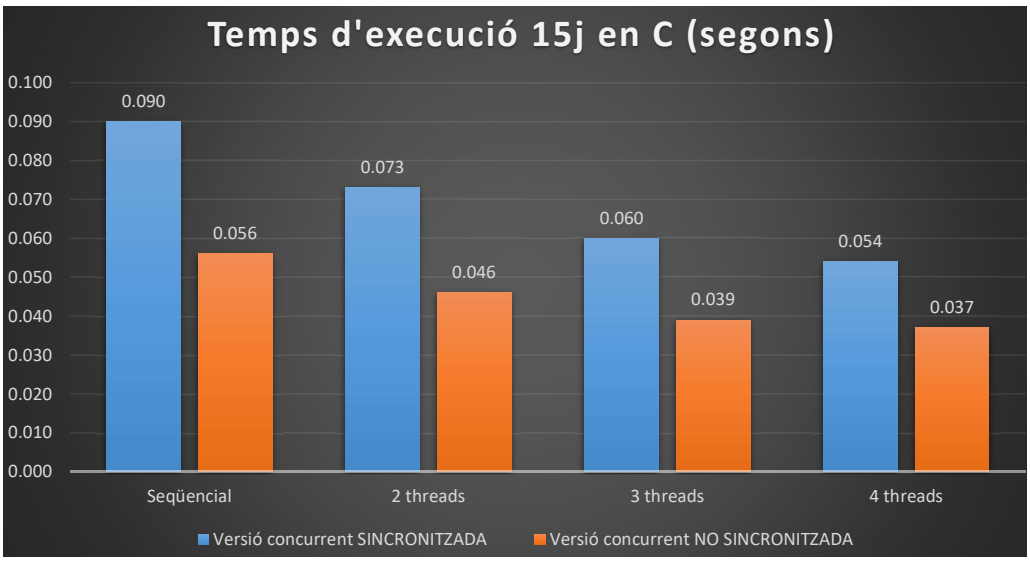

Figura 4: Gràfica del temps d'execució de la versió seqüencial, concurrent sincronitzada i concurrent no sincronitzada en C amb 15j

	2 *threads*	**3 *threads***	**4 *threads***
Seqüencial	18.89%	33.33%	40.00%
2 *threads*	-	17.81%	26.30%
3 *threads*	-	-	10.00%

Taula 7: Percentatges de reducció de temps entre la versió concurrent sincronitzada i la versió seqüencial en C amb 15j

	VC - No sincronitzada		
VC - Sincronitzada	2 *threads*	3 *threads*	4 *threads*
2 *threads*	36.99%	46.58%	49.32%
3 *threads*	23.33%	35.00%	38.33%
4 *threads*	14.81%	27.78%	31.48%

Taula 8: Percentatges de reducció de temps entre la versió concurrent sincronitzada i la versió concurrent no sincronitzada amb 15j

En la Figura 4 es pot observar com el temps d'execució de la versió seqüencial a la versió concurrent sincronitzada amb només 2 *threads* s'ha reduït de 0.090 s fins a 0.056 s , reduint el temps d'execució en un 18.89%. Amb la utilització de 3 i 4 *threads* els temps s'ha reduït un 33.33% i un 40% respectivament. Si es compara els temps obtinguts entre els *threads* sincronitzats es pot observar que els percentatges van del 10% al 26%. Encara que sembli molta diferencia, el fet de tractar temps d'execució tan ínfims (mil·lisegons) la realitat es que son quasi imperceptibles i cada vegada que s'executen aquest temps poden variar.

Pel que fa a la Taula 8, aquesta mostra les millores en percentatges de les versions sincronitzades versus les no sincronitzades. Així doncs, es pot observar com hi ha una millora de entre el 14% i 49% entre les diferents configuracions de *threads*. Això es degut a que en sincronització si s'esta executant algun fil, s'ha d'esperar a que acabi abans de començar a fer una altra tasca. Aquesta sincronització inexorablement fa augmentar el temps d'execució.

2.2 Temps d'execució amb 25j

Aplicació no sincronitzada amb 25j
- *Seqüencial* té un temps d'execució de: **1.803 s**.
- 2 *threads* té un temps d'execució de: **1.456 s**.
- 3 *threads* té un temps d'execució de: **1.263 s**.
- 4 *threads* té un temps d'execució de: **1.121 s**.

Aplicació sincronitzada amb 25j
- *Seqüencial* té un temps d'execució de: **2.228 s**.
- 2 *threads* té un temps d'execució de: **2.009 s**.
- 3 *threads* té un temps d'execució de: **1.873 s**.
- 4 *threads* té un temps d'execució de: **1.796 s**.

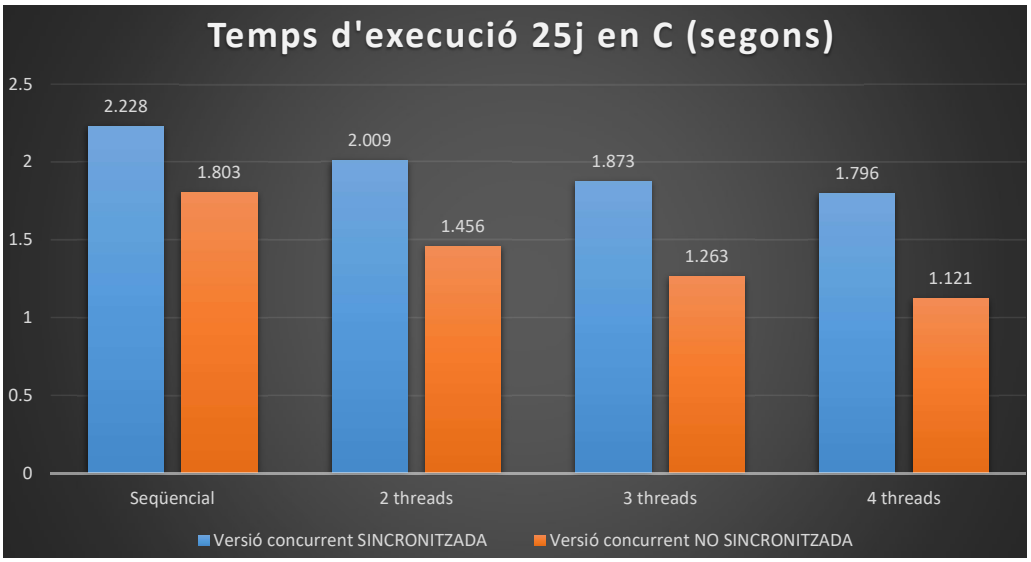

Figura 5: Gràfica del temps d'execució de la versió seqüencial, concurrent sincronitzada i concurrent no sincronitzada en C amb 25j

	2 threads	***3 threads***	***4 threads***
Seqüencial	9.83%	15.93%	19.39%
2 *threads*	-	6.77%	10.60%
4 *threads*	-	-	4.11%

Taula 9: Percentatges de reducció de temps entre la versió concurrent sincronitzada i la versió seqüencial en C amb 25j

	VC - No sincronitzada		
VC - Sincronitzada	2 *threads*	3 *threads*	4 *threads*
2 *threads*	27.53%	37.13%	44.20%
3 *threads*	22.26%	32.57%	40.15%
4 *threads*	18.93%	29.68%	37.58%

Taula 10: Percentatges de reducció de temps entre la versió concurrent sincronitzada i la versió concurrent no sincronitzada en C amb 25j

Pel que fa als temps d'execució amb el mercat de 25j, els resultats obtinguts segueixen les mateixes característiques als obtinguts amb la versió de JAVA. Així doncs, el millor temps obtingut es quan s'executa el programa amb 4 *threads* reduint el temps amb la versió seqüencial fins un 19.39%. Per tant, a major quantitat de fils, millor temps s'execució s'obtindrà (tenint en compte la configuració del processador).

Pel que fa a la Taula 10, es pot observar com hi ha una millora de entre el 18% i 44% entre les diferents configuracions de *threads* de la versió sincronitzada versus la no sincronitzada. Aquesta diferencia es molt semblant a la obtinguda a Taula 8. La major diferencia es pot observar quan es compara la versió sincronitzada amb 2 *threads* amb la versió no sincronitzada de 4 *threads* perquè la millora de rendiment es del 44.20%.

2.3 Temps d'execució amb 50j

Aplicació no sincronitzada amb 50j
- *Seqüencial* té un temps d'execució de: **2.03 min**.
- 2 *threads* té un temps d'execució de: **1.36 min**.
- 3 *threads* té un temps d'execució de: **1.26 min**.
- 4 *threads* té un temps d'execució de: **1.18 min**.

Aplicació sincronitzada amb 50j
- *Seqüencial* té un temps d'execució de: **2.28 min**.
- 2 *threads* té un temps d'execució de: **2.13 min**.
- 3 *threads* té un temps d'execució de: **2.11 min**.
- 4 *threads* té un temps d'execució de: **2.10 min**.

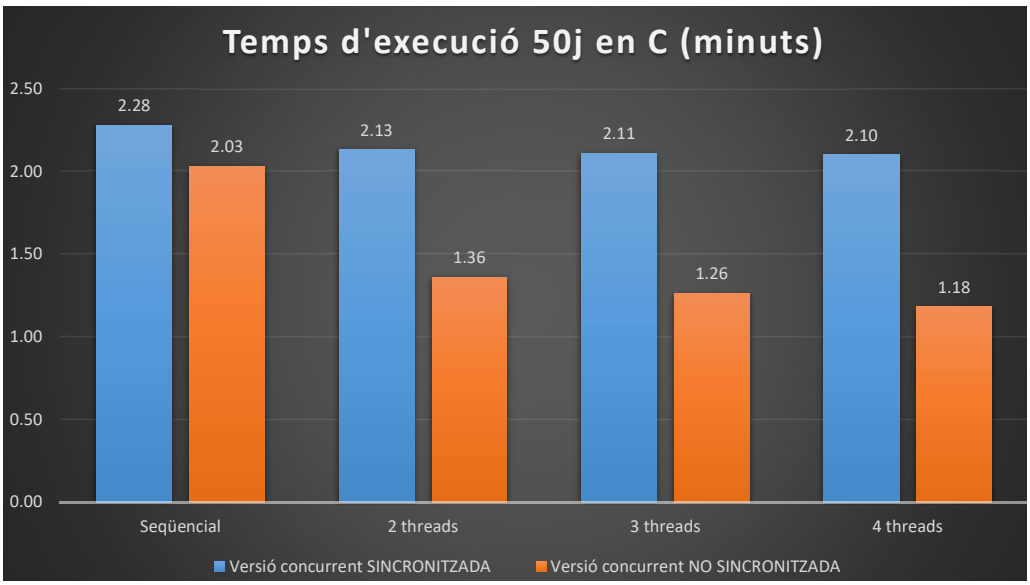

Figura 6: Gràfica del temps d'execució de la versió seqüencial, concurrent sincronitzada i concurrent no sincronitzada en C amb 50j

	2 *threads*	**3 *threads***	**4 *threads***
Seqüencial	6.58%	7.46%	7.89%
2 *threads*	-	0.94%	1.41%
3 *threads*	-	-	0.47%

Taula 11: Percentatges de reducció de temps entre la versió concurrent sincronitzada i la versió seqüencial en C amb 50j

	VC - No sincronitzada		
VC - Sincronitzada	2 *threads*	3 *threads*	4 *threads*
2 *threads*	36.15%	40.85%	44.60%
3 *threads*	35.55%	40.28%	44.08%
4 *threads*	35.24%	40.00%	43.81%

Taula 12: Percentatges de reducció de temps entre la versió concurrent sincronitzada i la versió concurrent no sincronitzada en C amb 50j

Pel que fa als temps d'execució amb el mercat de 50j, aquests segueixen la mateixa característica que els obtinguts amb el mercat de 25j. Així doncs, el millor temps obtingut es quan s'executa el programa amb 4 *threads* reduint el temps amb la versió seqüencial fins un 7.89%.

Pel que fa a la Taula 12, es pot observar com hi ha una millora de entre el 35% i 44% entre les diferents configuracions de *threads* de la versió sincronitzada versus la no sincronitzada. La major diferencia es pot observar quan es compara la versió sincronitzada amb 2 *threads* amb la versió no sincronitzada de 4 *threads* perquè la millora de rendiment es del 44.60%.

3 Característiques del hardware

Les diferents versiones del programa, tant en Java com en C, han estat executades en:

Sistema Operatiu: Ubuntu 20.04.1 LTS

Màquina virtual: Virtual Box 6.1

Processador: Intel® Core™ i5-8250U Processor

Amb la següent configuració:

```
root@lazo:~# lscpu
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   39 bits physical, 48 bits virtual
CPU(s):                          2
On-line CPU(s) list:             0,1
Thread(s) per core:              1
Core(s) per socket:              2
Socket(s):                       1
NUMA node(s):                    1
Vendor ID:                       GenuineIntel
CPU family:                      6
Model:                           142
Model name:                      Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
Stepping:                        10
CPU MHz:                         1800.000
BogoMIPS:                        3600.00
Hypervisor vendor:               KVM
Virtualization type:             full
L1d cache:                       64 KiB
L1i cache:                       64 KiB
L2 cache:                        512 KiB
L3 cache:                        12 MiB
NUMA node0 CPU(s):               0,1
Vulnerability Itlb multihit:     KVM: Vulnerable
Vulnerability L1tf:              Mitigation; PTE Inversion
Vulnerability Mds:               Mitigation; Clear CPU buffers; SMT Host state u
                                 nknown
Vulnerability Meltdown:          Mitigation; PTI
Vulnerability Spec store bypass: Vulnerable
Vulnerability Spectre v1:        Mitigation; usercopy/swapgs barriers and __user
                                  pointer sanitization
Vulnerability Spectre v2:        Mitigation; Full generic retpoline, STIBP disab
                                 led, RSB filling
Vulnerability Srbds:             Unknown: Dependent on hypervisor status
Vulnerability Tsx async abort:   Not affected
Flags:                           fpu vme de pse tsc msr pae mce cx8 apic sep mtr
```

Figura 7: Característiques del hardware on s'han executat tots el programes

4 Problemes

Un dels problemes més grans que hem tingut en la realització de la pràctica ha sigut el tractament de *strings* en C. En Java no ens havia resultat gran problema, ja que allí és més fàcil, però en C en intentar mostrar la millor i pitjor combinació en les estadístiques, no paraven de sortir errors de tota mena. Sobretot perquè per mostrar la combinació, havíem de cridar una funció externa, i aquesta inicialment retornava la cadena de text en forma de punter i això anava directe al *string* de l'estadística, però això causava molts errors i no funcionava i a base de modificacions vam arribar al que tenim ara. Però va ser molt temps malgastat per culpa de C.

Una altra situació va ser que a vegades saltava un error de *stack smashing* el qual era provocat perquè alguns dels *arrays* eren massa petits per emmagatzemar tota la combinació de jugadors. Llavors vam haver d'ajustar mides i fer múltiples execucions per veure que no tornava a sorgir.

En les primeres versions del programa per imprimir els últims missatges, teníem una funció que forçava al *thread* responsable d'imprimir missatges a imprimir tots el que tingués, i s'executava des dels *threads* avaluadors. Però això provocava que els avaluadors juntament amb el principal anessin més ràpidament que el *messenger* i l'execució del programa acabava abans que aquest pogués imprimir tots els missatges.

Un altre error era que a vegades quan es calculava les estadístiques globals, algun dels *threads* no havia trobat cap combinació vàlida i per tant causava errors en el càlcul de les mitges (ja que provocava divisions entre 0) i en l'hora d'imprimir la millor i pitjor combinació en les estadístiques de cada *thread*. També en l'hora de calcular les estadístiques globals no tenia en compte una cosa que sí que havia tingut en compte en les estadístiques de cada *thread* i per tant feia que les globals no donessin. I a vegades en C, per culpa de no inicialitzar el *numComb* a 0, donava nombres negatius enormes.

La resta d'errors eren bastant breus, com per exemple que en Java s'ha de reservar el *lock* per utilitzar el *signal* de les condicions, o coses per aquest estil.

5 Conclusions i anàlisis dels temps

Podem extreure com a conclusió que la nova versió del programa té un comportament similar al de la pràctica anterior respecte als temps d'execució, i això no és d'extranyar perquè és la modificació de la pràctica prèvia, així que no es tornarà a comentar el que ja vam comentar a l'anterior document. Però generalitzant, els dos es beneficien de la utilització de threads i aquesta vegada en fer la comparació utilitzant menys de 4 threads, donen temps més raonables, ja que la màquina on s'executa ho suporta.

Si ens fixem en els temps entre la versió sincronitzada i la que no, veiem que la sincronitzada té un major temps, això és degut al fet que aquesta no només té més feina en calcular i imprimir estadístiques, sinó que cada M combinacions ha de fer una sincronització entre tots els threads que estiguin funcionant, provocant així un retard significatiu en el temps. Això també voldrà dir que depenent de la M, el temps d'execució variarà, en el nostre cas hem utilitzat una M de 25000 per fer les estadístiques.