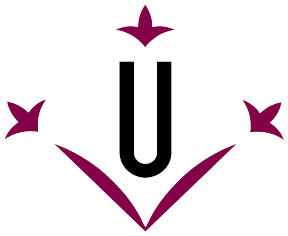

Universitat de Lleida
Escola Politècnica Superior

Sistemes Concurrents i Paral·lels

Pràctica 1: *Threads*

Sergi Puigpinós Palau
Jordi Rafael Lazo Florensa

29 de novembre de 2020

Índex

Índex de figures

1 Disseny concurrent del problema

1.1 Descomposició del problema

Per la resolució d'aquest problema hem utilitzat la descomposició de dades de domini, ja que permet dividir el domini del nostre problema entre diferents tasques, per tant, cada tasca és responsable de calcular la resposta per a la seva partició del domini.

1.2 Definició de la tasca

La tasca per a cada *thread* consisteix en: donat un pressupost, seleccionar la millor plantilla de jugadors disponibles en funció del cost i puntuació, dins del domini establert per a cada *thread*.

1.3 Granularitat

Granularitat fina: cada una de les combinacions possibles de jugadors.
Granularitat gruixuda: agrupació de les combinacions possibles de jugadors.

En el nostre cas hem escollit la granularitat gruixuda, ja que hem observat que la quantitat de combinacions de jugadors possibles era excessivament gran com per utilitzar granularitat fina.

1.4 Càrrega de les tasques

Les tasques d'aquest problema són homogènies en el sentit que en l'assignació de treball per a cada *thread* és equitatiu, però com que algunes combinacions tenen equips invàlids, provocarà que alguns *threads* hagin de fer menys treball que altres.

1.5 Grau màxim concurrencia

El grau màxim de concurrència és el nombre total de combinacions possibles de jugadors que puguin haure en l'arxiu csv que s'especifiqui.

1.6 Dependència de dades

No existeix dependències de dades, ja que cada combinació és independent de les altres. L'únic cas on hi hauria alguna mena de dependència de dades, és quan el thread principal ha d'esperar que tots els threads li retornin els seus millors equips per a poder comparar-los i traure la millor combinació d'equip d'entre tots ells.

1.7 Sincronització de les tasques

No existeix cap sincronització entre les tasques excepte en el cas del programa en C el qual és necessari la sincronització entre els diferents *threads* per imprimir per pantalla els resultats que van obtenint.

1.8 Patró del problema

SPMD ja que tenim tasques homogènies i cada *thread* realitza la mateixa feina en un rang especificat.

1.9 Algoritme de balanceig de càrrega

El balanceig de càrrega és estàtic perquè assignem les tasques a l'hora de la creació dels *threads*.

1.10 Modalitat d'assignació de balanceig de càrrega

Hem escollit l'assignació proporcional perquè distribueix proporcionalment les tasques entre els diferents *threads*.

1.11 Tècnica d'assignació de tasques

Hem decidit realitzar-ho en blocs per poder fer una millor distribució de les tasques i tindre més control sobre la repartició dels dominis.

2 Versió en JAVA

Per a la versió concurrent en JAVA hem creat una classe que extendeix de *thread* la qual será executada per cada un dels *threads*.

Dins d'aquesta classe, la funció run, executarà el bucle encarregat de calcular totes les combinacions possibles, el qual formava part de la funció de calcular l'equip òptim.

A l'hora de l'inicialització dels *threads*, s'especificà que executin aquesta classe (per defecte s'executà la funció *run* d'aquesta) i també el domini del treball pel qual han d'executar la tasca. D'aquesta manera cada un dels *threads* calcularà l'equip òptim dins del seu domini i enviará aquest equip cap al *thread* principal el qual compararà tots els equips que vaiguin retornant els *threads* i es quedarà amb el millor d'entre tots ells.

Un altre parametre que se li ha de passar als *threads*, és la referencia a la classe *market* que està sent utilitzada. D'aquesta manera es poden executar certes funcions de la classe que es necessiten per l'execució de la tasca.

Per l'assignació del domini, quan iterem pel bucle de creació de *threads*, anirem dividint la càrrega de treball que ens queda entre la quantitat de *threads* que hi ha restant. D'aquesta manera s'obtindrà una distribució més proporcional de carrega entre els processos.

A més a més, per observar la disminució del temps d'execució de la versió concurrent s'ha comprovat amb 2, 4 i 8 *threads*.

2.1 Temps d'execució versió seqüencial

L'aplicació seqüencial amb 15j té un temps d'execució de: **262 ms**.

L'aplicació seqüencial amb 25j té un temps d'execució de: **1921 ms**.

L'aplicació seqüencial amb 50j té un temps d'execució de: **2.08 min**.

2.2 Temps d'execució versió concurrent 15j

L'aplicació concurrent amb 15j
- 2 *threads* té un temps d'execució de: **158 ms**.
- 4 *threads* té un temps d'execució de: **141 ms**.
- 8 *threads* té un temps d'execució de: **152 ms**.

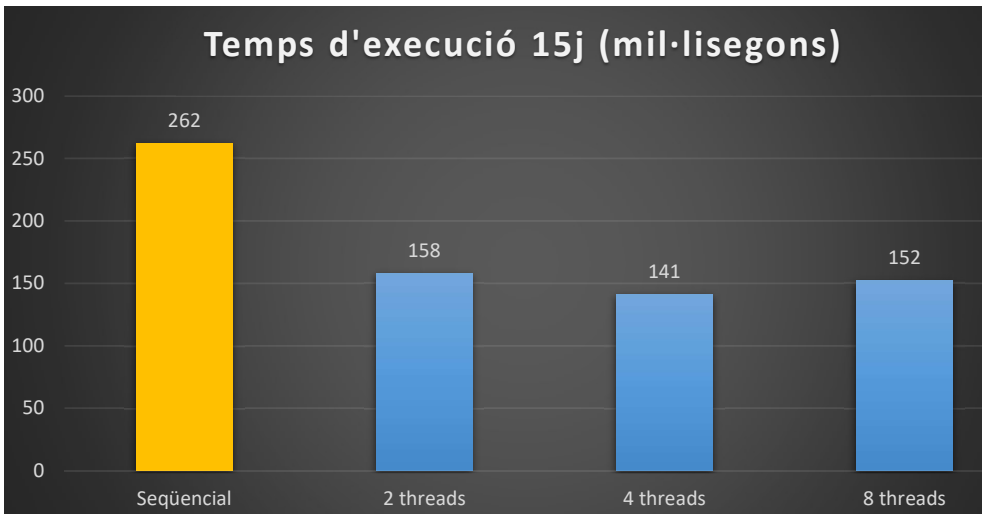

Figura 1: Gràfica del temps d'execució de la versió concurrent en Java amb 15j

	2 *threads*	**4 *threads***	**8 *threads***
Seqüencial	39.69%	46.18%	41.98%
2 *threads*	-	10.76%	3.80%
4 *threads*	-	-	-7.80%

Taula 1: Percentatges de reducció de temps amb 15j

En la Figura 1 es pot observar com el temps d'execució de la versió seqüencial a la versió concurrent amb només 2 *threads* s'ha reduït de 262 ms fins a 158 ms, reduint el temps d'execució en un 39.69%. Amb la utilització de 4 *threads* els temps s'ha reduït un 46.18%, en canvi amb la utilització de 8 s'ha reduït tan sols un 41.98%, un 4.2% menys que amb 4 *threads* i, fins i tot, donant una millora negativa del -7.80% si es compara el temps de l'execució de 4 amb 8. 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 i unió dels *threads* que en l'execució de les tasques.

2.3 Temps d'execució versió concurrent 25j

L'aplicació concurrent amb 25j
- 2 *threads* té un temps d'execució de: **1378 ms**.
- 4 *threads* té un temps d'execució de: **959 ms**.
- 8 *threads* té un temps d'execució de: **996 ms**

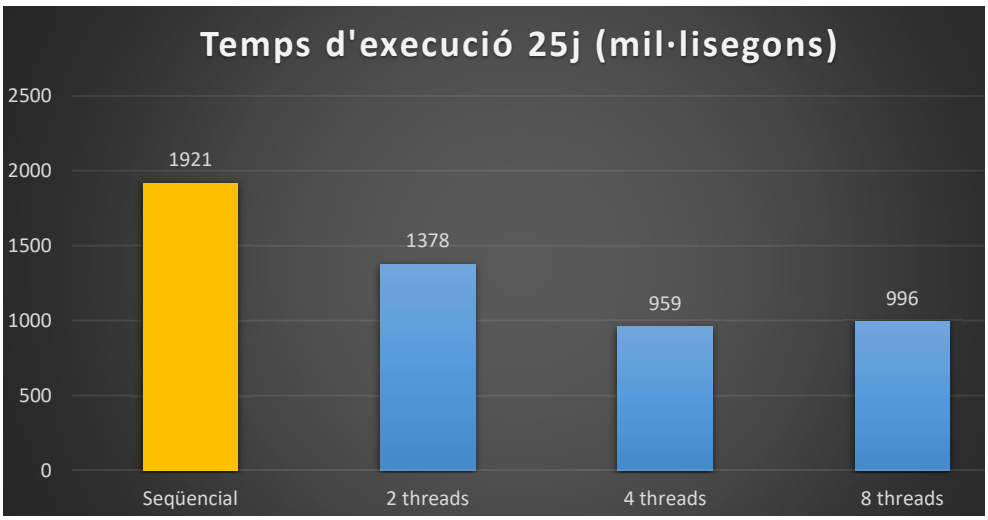

Figura 2: Gràfica del temps d'execució de la versió concurrent en Java amb 25j

	2 *threads*	**4 *threads***	**8 *threads***
Seqüencial	28.27%	50.08%	48.15%
2 *threads*	-	30.41%	27.72%
4 *threads*	-	-	-3.86%

Taula 2: Percentatges de reducció de temps amb 25j

Pel que fa el temps d'execució amb el mercat amb 25j, els temps obtinguts segueixen la mateixa característica que en la Figura 1. El millor temps obtingut és quan s'executa amb 4 *threads* reduint fins un 50.08% si es compara amb la versió seqüencial.

També es pot observar que la diferencia del temps d'execució entre 4 i 8 *threads* encara es negativa -3.86%, però és menys si ho comparem amb la del mercat amb 15j perquè s'ha incrementat el nombre de combinacions.

2.4 Temps d'execució versió concurrent 50j

L'aplicació concurrent amb 50j
- 2 *threads* té un temps d'execució de: **1,09 min**.
- 4 *threads* té un temps d'execució de: **0,39 min**.
- 8 *threads* té un temps d'execució de: **0,34 min**.

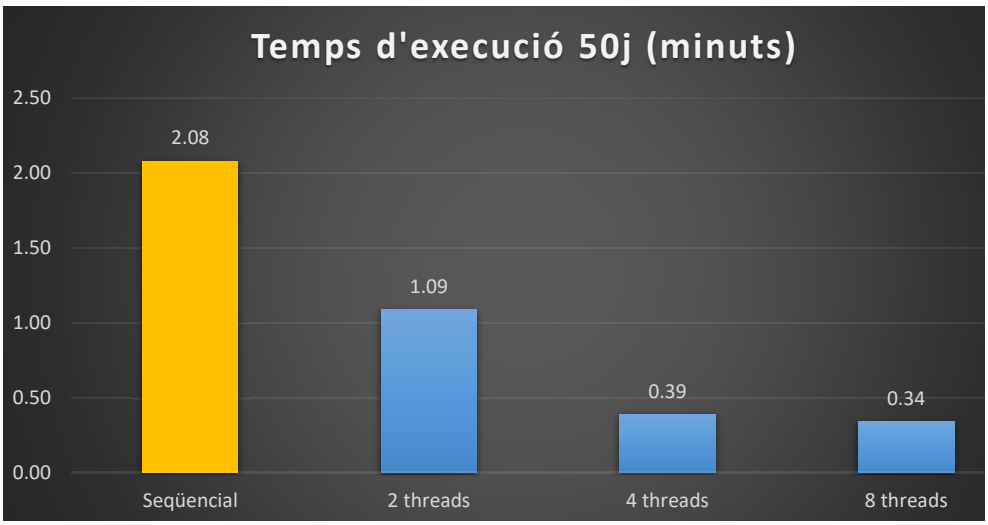

Figura 3: Gràfica del temps d'execució de la versió concurrent en Java amb 50j

	2 *threads*	**4 *threads***	**8 *threads***
Seqüencial	47.60%	81.25%	83.65%
2 *threads*	-	64.22%	68.81%
4 *threads*	-	-	12.82%

Taula 3: Percentatges de reducció de temps amb 50j

En la Figura 3 es pot observar com el temps d'execució de la versió seqüencial a la versió concurrent amb només 2 *threads* s'ha reduït de 2,08 min fins a 1,09 min, reduint el temps d'execució en un 47,60%. Amb la utilització de 4 *threads* els temps s'ha reduït un 81.25% i amb 8 fins a un 83.65% sent aquesta la màxima obtinguda. Per tant, la reducció del temps d'execució de 2 a 4 *threads* és del 64.22% i sent la reducció de 2 a 8 del 68.81%.

Així doncs, a diferència dels resultats obtinguts amb els mercats de 15 i 25 jugadors, en aquest cas es pot observar que la diferencia del temps d'execució entre 4 i 8 *threads* és positiva i s'incrementa un 12.82% millorant el rendiment sempre que la quantitat de dades sigui suficientment gran.

Finalment, es pot observar com a mesura que s'incrementen el número de *threads* s'acaba complint la lley Amdahl, ja que la millora de rendiment que es pot obtenir d'un programa concurrent està limitat per la fracció de codi seqüencial d'aquest.

3 Versió en C

Pel que fa a la versió en C no es diferencia molt de la versió en java, moltes de les diferencies estan basades en la sintaxis i algunes coses que pel llenguatge s'ha de fer d'una manera diferent. Com per exemple en el cas de passar els parametres als threads hem de fer ús d'una estructura, també s'ha de fer ús constant de punters i mallocs... Però en termes d'implementació, aquesta es la mateixa que en java.

3.1 Temps d'execució versió seqüencial

L'aplicació seqüencial amb 15j té un temps d'execució de: **130 ms**.

L'aplicació seqüencial amb 25j té un temps d'execució de: **3189 ms**.

L'aplicació seqüencial amb 50j té un temps d'execució de: **3.30 min**.

3.2 Temps d'execució versió concurrent 15j

L'aplicació concurrent amb 15j
- 2 *threads* té un temps d'execució de: **120 ms**.
- 4 *threads* té un temps d'execució de: **120 ms**.
- 8 *threads* té un temps d'execució de: **120 ms**.

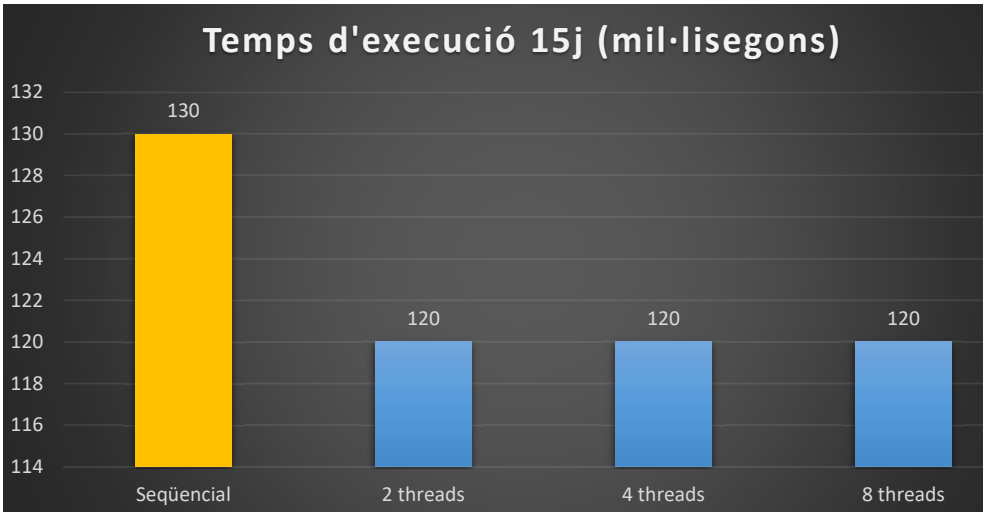

Figura 4: Gràfica del temps d'execució de la versió concurrent en C amb 15j

	2 *threads*	**4 *threads***	**8 *threads***
Seqüencial	7.69%	7.69%	7.69%
2 *threads*	-	0.00%	0.00%
4 *threads*	-	-	0.00%

Taula 4: Percentatges de reducció de temps amb 15j

En la Figura 4 es pot observar com el temps d'execució de la versió seqüencial a la versió concurrent amb 2, 4 i 8 *threads* s'ha reduït de 130 ms fins a 120 ms, reduint el temps d'execució en tan sols un 7.69%. Sent inexistent la millora de temps d'execució a partir de la utilització de més de 2 *threads* a causa de la ràpida compilació de C, mentre que en JAVA la millora era existent en aquest cas no.

3.3 Temps d'execució versió concurrent 25j

L'aplicació concurrent amb 25j
- 2 *threads* té un temps d'execució de: **3018 ms**.
- 4 *threads* té un temps d'execució de: **2848 ms**.
- 8 *threads* té un temps d'execució de: **2854 ms**.

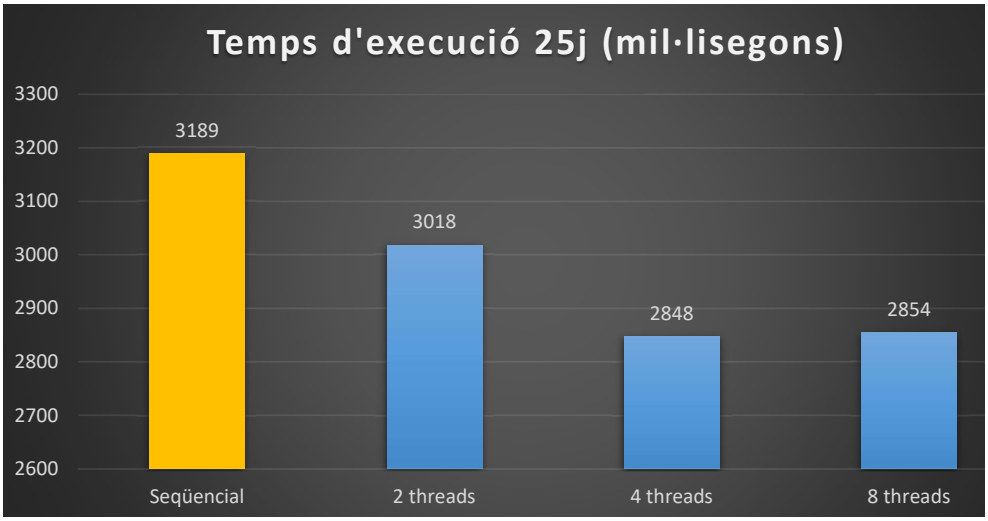

Figura 5: Gràfica del temps d'execució de la versió concurrent en C amb 25j

	2 *threads*	**4 *threads***	**8 *threads***
Seqüencial	5.36%	10.69%	10.50%
2 *threads*	-	5.63%	5.43%
4 *threads*	-	-	-0.21%

Taula 5: Percentatges de reducció de temps amb 25j

Pel que fa al temps d'execució de 25 jugadors els resultats obtinguts són molt semblants als obtinguts amb la versió de JAVA. La màxima reducció de temps es troba quan s'executa amb 4 *threads* amb una millora del 10.69% i l'increment de *threads* no millora aquest registre, fins i tot l'empitjora si es compara de 4 a 8 amb una reducció del -0.21%. 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 i unió dels *threads* que en l'execució de les tasques.

3.4 Temps d'execució versió concurrent 50j

L'aplicació concurrent amb 50j
- 2 *threads* té un temps d'execució de: **3.04 min**.
- 4 *threads* té un temps d'execució de: **2.54 min**.
- 8 *threads* té un temps d'execució de: **2.33 min**.

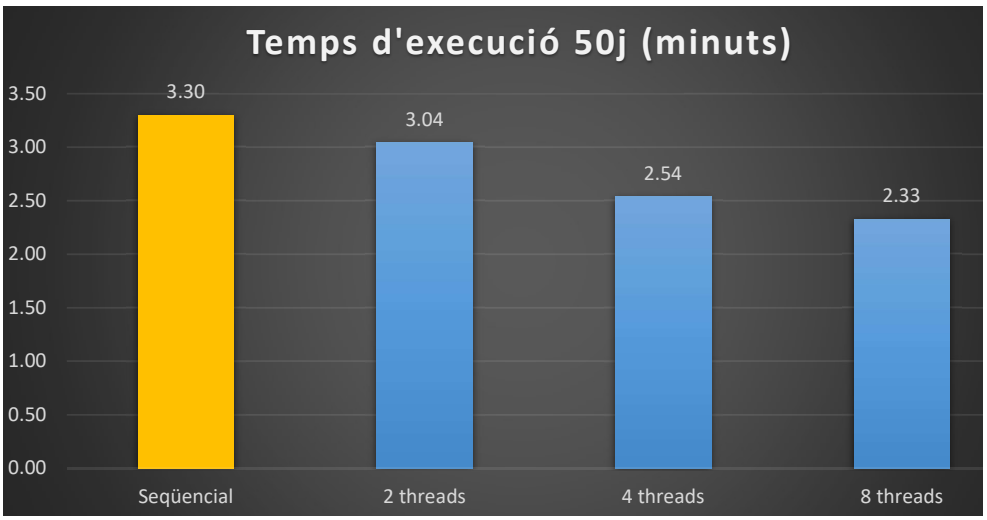

Figura 6: Gràfica del temps d'execució de la versió concurrent en C amb 50j

	2 *threads*	**4 *threads***	**8 *threads***
Seqüencial	7.88%	23.03%	29.39%
2 *threads*	-	16.45%	23.36%
4 *threads*	-	-	8.27%

Taula 6: Percentatges de reducció de temps amb 50j

Finalment, al igual que en la versió de JAVA, es pot observar com el temps d'execució de la versió seqüencial a la versió concurrent amb només 2 *threads* s'ha reduït un 7.88%, amb 4 un 23.03% i amb 8 fins un 29.39%.

Així doncs, a diferència dels resultats obtinguts amb els mercats de 15 i 25 jugadors, en aquest cas es pot observar que la diferencia del temps d'execució entre 4 i 8 *threads* és positiva i s'incrementa un 8.27% millorant el rendiment sempre que la quantitat de dades sigui suficientment gran.

Finalment, es pot observar com a mesura que s'incrementen el número de *threads* s'acaba complint la lley Amdahl, ja que la millora de rendiment que es pot obtenir d'un programa concurrent està limitat per la fracció de codi seqüencial d'aquest.

4 Problemes

Un dels problemes que hem tingut en la realització d'aquesta pràctica, ha sigut en fer el programa en C el qual comportava tindre bastants coneixements sobre punters i referències per a poder realitzar-la sense problemes.

En Java, inicialment no sabíem com fer perquè els *threads* poguessin executar algunes de les funcions de la instància de la classe *market* que eren necessàries per a la realització de les tasques. Finalment se'ns va acudir passar als *threads* la referència a la instància.

Finalment no hem pogut incorporar els temps i les gràfiques dels arxius csv de 60, 75, 100 ja que per culpa del nostre hardware no hem pogut executar-los (tardaven massa temps).

5 Conclusions i anàlisis dels temps

Podem extreure com a conclusió que la utilització dels *threads* en les operacions que realitzen una gran quantitat de càlculs pot ajudar significativament a reduir la tasca i al mateix temps a reduir el temps d'execució permetin l'optimització de programa.

A més a més, milloren la utilització de la CPU, els programes són més receptius i ofereixen una distribució més justa dels recursos de la CPU.

En els casos en què la versió seqüencial té poques dades a processar, i aquesta s'executà en qüestió de mil·lisegons, no es pot apreciar un gran rendiment, ja que la diferencia és mínima. A més a més, el fet d'incrementar el nombre de *threads* en aquest tipus de dades, acaba perjudicant el temps d'execució del programa perquè es perd més temps en la creació i unió dels *threads* que en l'execució de les tasques.

Creiem que els temps del programa en C no són tan bons a causa de la utilització de màquines virtuals. Pensem d'aquesta manera perquè la distribució de la feina i el programa en si mateix, és igual que el programa en Java (el qual dóna temps correctes).

Un altre factor que ha influenciat en els resultats és que depenent de la CPU en la qual s'executava el programa, donava temps diferents, ja que alguns processadors són millors que altres i depenent de la marca i tipus pot provocar mala gestió dels *threads* dels programes concurrents i fer que aquests no es beneficiïn tant. Finalment, que el programa sigui indeterminista en el temps d'execució (tant en la versió concurrent com seqüencial) no ha ajudat en la determinació del temps.