Rendiment d'arquitectures multifil

Models de programació

Francesc Guim
Ivan Rodero

PID_00215407

Universitat Oberta de Catalunya

Índex

Introducció

En el mòdul "Arquitectures multifil" s'han presentat diferents arquitectures de processadors que implementen més d'un fil d'execució. Així, doncs, hi ha arquitectures que faciliten l'accés a un nombre reduït de fils, com les *shared multithreading*, i n'hi ha que donen accés a desenes de fils d'execució, com les arquitectures multinucli.

En general, una arquitectura és més complexa com més fils facilita. Les arquitectures multinucli són força escalables i poden donar accés a un nombre elevat de fils. Ara bé, com s'ha vist, per tal de dur-ho a terme cal dissenyar sistemes que són més complexos. Un cas que exemplifica aquest aspecte és dissenyar una xarxa d'interconnexió que permeti donar l'amplada de banda que els diferents nuclis necessiten quan es vol accedir a una memòria cau compartida de darrer nivell.

Com més complexa és l'arquitectura, més factors cal tenir en compte a l'hora de programar-la. Si tenim una aplicació que té una zona paral·lela que en un processador seqüencial es pot executar en temps x, cal esperar que en una màquina multifil amb m fils disponibles aquesta aplicació potencialment es pugui executar en un temps de x/m. No obstant això, hi ha certs factors inherents al tipus d'arquitectura sobre la qual s'està executant i al model de programació emprat que poden limitar aquest increment de rendiment. Per exemple, com s'estudia més endavant, l'accés a les dades que es troben compartides per fils que s'estan executant en diferents nuclis.

Per tal d'obtenir el màxim rendiment de les arquitectures sobre les quals s'executen les aplicacions paral·leles és necessari considerar les característiques d'aquestes arquitectures. Per tant, cal considerar la jerarquia de memòria del sistema, el tipus d'interconnexió sobre el qual s'envien dades, l'amplada de banda de memòria, etc. És a dir, si se'n vol extreure el màxim rendiment caldrà redissenyar o adaptar els algorismes a les característiques del maquinari que hi ha per sota.

Si bé és cert que cal adaptar les aplicacions en funció de les arquitectures en què es volen executar, també ho és que hi ha utilitats que permeten no haver de tenir en compte algunes de les complexitats que presenten aquestes arquitectures. La majoria apareixen en forma de models de programació i biblioteques que poden ser emprats per les aplicacions.

Durant aquest mòdul, en primer lloc, es presenten els factors més importants que poden limitar l'accés al paral·lelisme que dóna una arquitectura lligats al model de programació. En segon lloc, es discuteixen els factors relacionats

amb les característiques de l'arquitectura sobre la qual s'executen. I finalment, s'analitzen alguns models de programació orientats a maximitzar l'ús dels recursos d'arquitectures de processadors multifil.

Objectius

Els principals objectius d'aquest mòdul són:

1. Estudiar quins són els factors lligats al model de programació que cal tenir en compte a l'hora de desenvolupar aplicacions multifil.

2. Estudiar quins són els factors lligats a l'arquitectura d'un processador multifil que cal considerar en el desenvolupament d'aplicacions multifil.

3. Estudiar quines són les característiques del model de programació del POSIX Threads i com cal desenvolupar aplicacions paral·leles emprant la seva interfície.

4. Estudiar quines són les característiques dels models de programació que Intel facilita per tal de desenvolupar aplicacions paral·leles per a les seves arquitectures multifil.

5. Aprofundir en conceptes avançats de la programació d'arquitectures multifil necessaris per a desenvolupar aplicacions que necessitin un rendiment alt.

1. Factors importants per a la llei d'Amdahl en arquitectures multifil

La llei d'Amdahl estableix que una aplicació dividida en una part inherentment seqüencial (és a dir, només pot ser executada per un fil) i una part paral·lela *P*, potencialment podrà tenir una millora de rendiment de *S* (en anglès, *speedup*) només augmentant el nombre de fils de l'aplicació en la part paral·lela.

$$T' = T \text{ x } 1/(1 - P + P/S)$$

Ara bé, per a arribar al màxim teòric és necessari considerar les restriccions inherents al model de programació i restriccions inherents a l'arquitectura sobre la qual s'està executant l'aplicació.

El primer conjunt de restriccions fa referència als límits vinculats a l'algorisme paral·lel considerat, i també a les tècniques de programació emprades. Un cas en què apareixen aquestes restriccions és quan s'ordena un vector, ja que hi ha limitacions causades per l'eficiència de l'algorisme, com per exemple *radix sort*, i per la seva implementació, com per exemple la manera d'accedir a les variables compartides, etc.

El segon conjunt fa referència a límits lligats a les característiques del processador sobre el qual s'està executant l'aplicació. Factors com ara la jerarquia de memòria cau, el tipus de memòria cau o el tipus de xarxa d'interconnexió dels nuclis poden limitar aquest rendiment. Per exemple, pot causar moltes ineficiències que una variable es comparteixi entre dos fils que no es troben dins el mateix nucli.

En els dos propers subapartats es presenten alguns dels factors més importants d'aquests dos blocs esmentats en el darrers paràgrafs. Del primer conjunt no s'estudien algorismes paral·lels (Gibbons i Rytte, 1988), atès que no és l'objectiu d'aquest mòdul, sinó que s'estudien els mecanismes que fan servir aquests algorismes per tal d'implementar tasques paral·leles i tots els factors que cal considerar. Del segon bloc es discuteixen les característiques més rellevants de l'arquitectura que cal considerar en el desenvolupament d'aquest tipus d'aplicacions.

És important remarcar que els factors que s'estudien a continuació són una part dels molts que s'han identificat durant les darreres dècades. A causa de la importància d'aquest àmbit, s'ha fet molta recerca centrada a com millorar el rendiment d'aquestes arquitectures i com millorar el disseny de les aplicacions que s'executen (com ara les aplicacions de càlcul numèric o les aplicacions de càlcul del genoma humà). Per tal d'aprofundir més en els problemes i estudis

duts a terme tant en l'àmbit acadèmic com empresarial, és molt recomanable estendre la lectura d'aquest mòdul didàctic amb les referències bibliogràfiques facilitades.

1.1. Factors vinculats al model de programació

En altres mòduls didàctics s'ha introduït el concepte de *programació paral·lela*. També s'han descrit les característiques més importants que cal considerar en el desenvolupament d'algorismes paral·lels. En aquest subapartat es presenten els reptes més importants en aquest tipus d'implementació i la seva relació en arquitectures de processadors multifil.

1.1.1. Definició i creació de les tasques paral·leles

En el disseny d'algorismes paral·lels es consideren dos tipus de paral·lelisme: amb relació a les dades i amb relació a la funció. El primer defineix quines parts de l'algorisme s'executen de manera concurrent i el segon, la manera com les dades es processen de manera paral·lela.

En la creació del paral·lelisme amb relació a la funció és important considerar que les tasques que treballin amb les mateixes funcions i dades tinguin localitat al nucli on s'executaran. D'aquesta manera, els fluxos del mateix nucli comparteixen les entrades corresponents a la memòria cau de dades i també les seves instruccions.

El paral·lelisme amb relació a les dades també ha de considerar la localitat comentada en el nivell de funció, però a més a més ha de tenir en compte la mida de les memòries cau de què disposa. És a dir, el flux d'instruccions ha de treballar amb les dades de manera local a la L1 tant com es pugui i la resta intentar-les mantenir a nivells de memòria cau tan a prop com sigui possible (L2 o L3). D'altra banda, també cal evitar efectes ping-pong entre els diferents nuclis del processador. Això pot succeir quan fils de diferents nuclis accedeixin als mateixos blocs d'adreces de manera paral·lela.

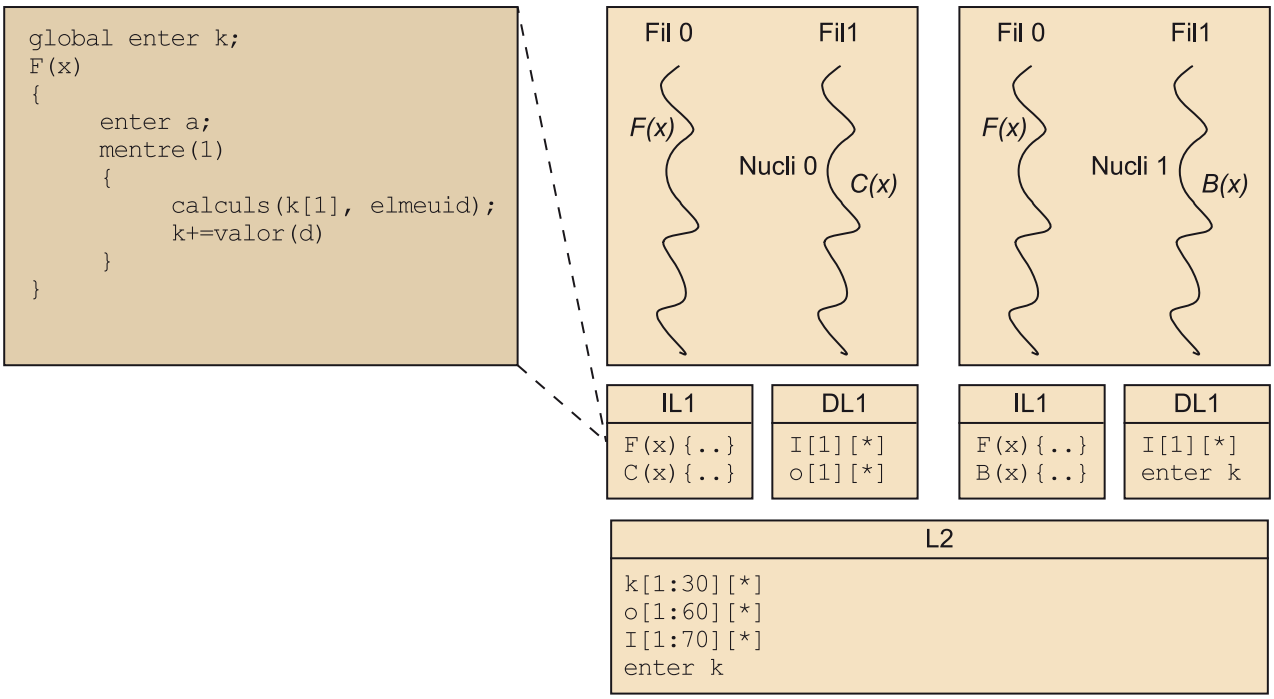
La figura 1 mostra un exemple d'escenari que s'ha d'evitar en la creació de fils, tant en l'assignació de tasques com en les dades. En aquest escenari, els dos fils número 0 d'ambdós nuclis estan executant la mateixa funció $F(x)$. Aquesta funció conté un bucle intern que fa alguns càlculs sobre el vector k i el resultat l'afegeix a la variable global k. Durant l'execució d'aquests fils s'observa que:

- Els dos nuclis cada vegada que vulguin accedir a la variable global k hauran d'invalidar la L1 de dades de l'altre nucli. Com ja s'ha tractat en el mòdul "Arquitectures multifil", depenent del protocol de coherència pot ser extremament costós. Per tant, aquest aspecte pot baixar el rendiment de l'aplicació.

- Els dos fils accedeixen potencialment a les dades del vector *l*. Així, doncs, les L1 de dades d'ambdós nuclis tindran emmagatzemades les mateixes dades (assumint que la línia es troba en estat compartit). En aquest cas, seria més eficient que els dos fils s'executessin en el mateix nucli per tal de compartir les dades de la L1 de dades (és a dir, més localitat). Això permetria usar més eficientment l'espai total del processador.

- De manera similar, la L1 d'instruccions d'ambdós nuclis tindrà una còpia de les instruccions del mateix codi que executen els dos fils (*F*). Igual que en el punt anterior, aquest aspecte provoca una utilització ineficient dels recursos.

Figura 1. Definició i creació de fils



L'exemple anterior mostra una situació força senzilla de solucionar. Ara bé, la definició, la creació de tasques i l'assignació de dades no és una tasca senzilla. Com s'ha esmentat, cal considerar la jerarquia de memòria, la manera com els processos s'assignen als nuclis, el tipus de coherència que el processador facilita, etc.

Lectures recomanades

Per tal d'aprofundir en aquest àmbit és recomanable la lectura de:

X. Martorell; J. Corbalán; M. González; J. Labarta; N. Navarro; E. Ayguadé (1999). "Thread Fork/Join Techniques for Multi-level Parallelism Exploitation in NUMA Multiprocessors". *13th International Conference on Supercomputing*.

B. Chapman; L. Huang; E. Biscondi; E. Stotzer; A. G. Shrivastava (2008). "Implementing OpenMP on a High Performance Embedded Multicore MPSoC". *IPDPS*.

Tot i la complexitat d'aquesta tasca, hi ha molts recursos que ajuden a definir aquest paral·lelisme. Per exemple:

- Aplicacions que permeten la paral·lelització automàtica d'aplicacions seqüencials. Molts compiladors inclouen opcions per tal de generar codi paral·lel de manera automàtica. Ara bé, és fàcil trobar-se en situacions en les quals el codi generat no és òptim o no té en compte alguns dels aspectes introduïts.

- Aplicacions que donen assistència en la paral·lelització dels codis seqüencials. Per exemple, ParaWise (ParaWise, 2011) és un entorn que guia l'usuari en la paral·lelització de codi Fortran. En qualsevol cas, el resultat pot ser similar a la paral·lelització automàtica.

- Finalment, també hi ha biblioteques que proporcionen interfícies per a la implementació d'algorismes paral·lels. Aquestes interfícies acostumen a ser la solució més eficaç per a treure el màxim rendiment de les aplicacions. Faciliten l'accés a funcionalitats que permeten definir el paral·lelisme amb relació a les dades i funcions, afinitats de fils a nuclis, afinitats de dades a la jerarquia de memòria, etc. Algunes d'aquestes biblioteques són: OpenMP, Cilk (Intel, Intel Cilk Plus, 2011), TBB (Reinders, 2007), etc.

Lectura complementària

ParaWise (2011). *ParaWise: the Computer Aided Parallelization Toolkit*. Recuperat el 27 de desembre del 2011: www.parallelsp.com/parawise.htm

1.1.2. Assignació de tasques a fils

Una tasca és una unitat de concurrència d'una aplicació, és a dir, inclou un conjunt d'instruccions que poden ser executades de manera concurrent (paral·lela o no) a les instruccions d'altres tasques. Un fil és una abstracció del sistema operatiu que permet l'execució paral·lela de diferents fluxos d'execució.

Una aplicació pot estar formada des d'un nombre relativament petit de tasques fins a milers. Exemples clars d'això són els servidors de videojocs o els servidors de pàgines web: poden tenir milers de tasques atenent les peticions dels usuaris.

No obstant això, el sistema operatiu no pot donar accés a un nombre tan elevat de fils d'execució per la limitació dels recursos. Per una banda, la gestió d'un nombre tan elevat és altament costós (molts canvis de contextos, gestió de moltes interrupcions, etc.). Per l'altra, tot i que potencialment el sistema operatiu pugui donar accés a un miler de fils, el processador sobre el qual s'executen les aplicacions donarà accés a pocs fils. Per tant, caldrà anar fent canvis de context entre tots els fils disponibles en el sistema i els fils que el maquinari faciliti. Aquest és el motiu pel qual el nombre de fils del sistema serà configurat coherentment amb el nombre de fils del processador.

L'aplicació, per tal de treure el màxim rendiment del processador, haurà de definir una assignació eficient i adequada de les tasques que vol executar als diferents fils de què disposa. Alguns dels algorismes d'assignació de tasques a fils són:

- Patró màster/esclau (*master/slave*): un fil s'encarrega de distribuir les tasques que resten per executar als fils esclaus que no estiguin executant res. El nombre de fils esclaus serà variable.

- Patró *pipeline* o cadena: en què cadascun dels fils fa una operació específica en les dades que s'estan processant, alhora que facilita el resultat al següent fil de la cadena.

- Patró *task pool*: en què hi ha una cua de tasques pendents de ser executades i cadascun dels fils disponibles agafa una d'aquestes tasques pendents quan acaba de processar l'actual.

Aquesta assignació es podrà basar en molts criteris diferents. No obstant això, els aspectes introduïts al llarg d'aquest mòdul didàctic s'haurien de considerar en arquitectures multifil heterogènies. Depenent de com les tasques s'assignin als fils i com els fils estiguin assignats als nuclis, el rendiment de les aplicacions varia molt.

Lectura recomanada

Un treball de recerca molt interessant en aquest aspecte és el presentat per Philbin i altres a l'article següent, on els autors presenten tècniques d'assignació i gestió de fils per tal de mantenir la localitat a les diferents memòries cau:

J. Philbin; J. Edler; O. J. Anshus; C. Douglas; K. Li (1996). "Thread scheduling for cache locality". *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*.

Exemple d'assignació de tasques

Un exemple d'aquesta situació es presenta en la figura 2. Una aplicació multifil s'executa sobre una arquitectura composta per dos processadors, cadascun dels quals amb accés a memòria i tots dos connectats per un bus. Els fils que s'estan executant al nucli 2 i al nucli 3 accediran a les dades i a la informació que el màster els facilita.

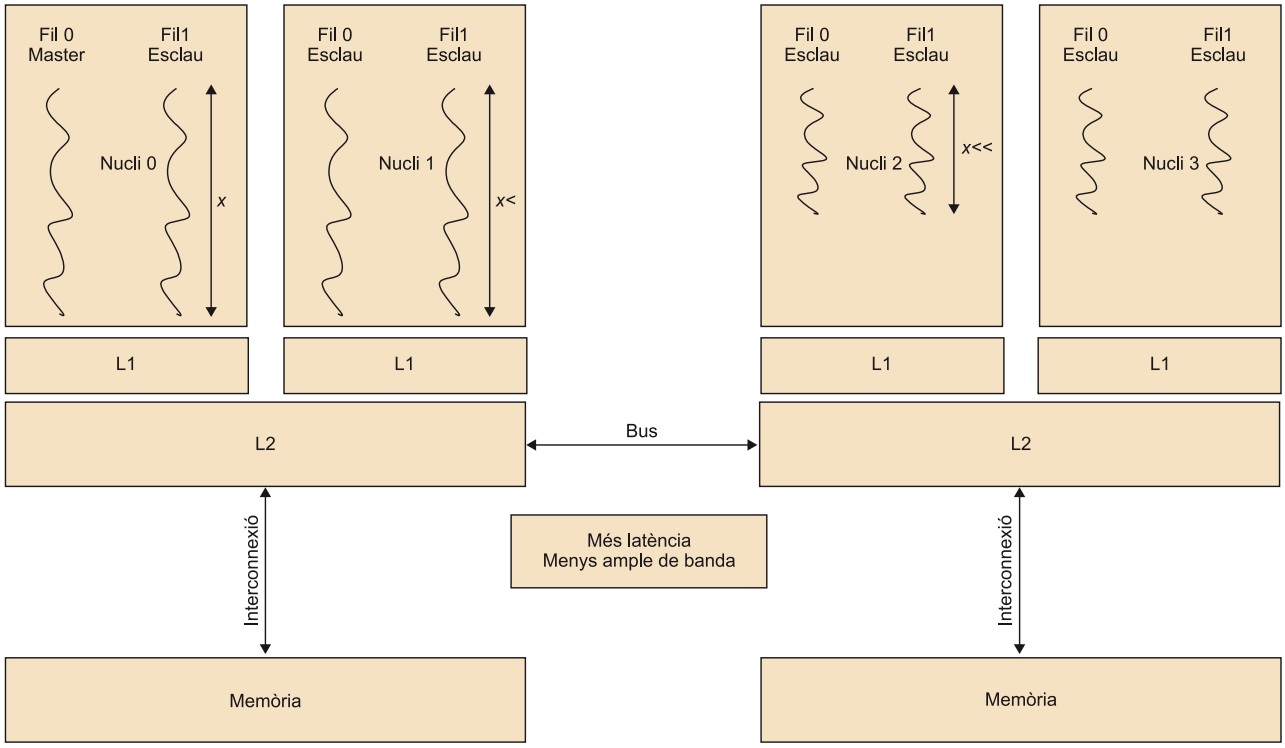
Respecte d'una xarxa d'interconnexió local el bus acostuma a tenir una latència més elevada i menys amplada de banda. Per aquest motiu els fils que s'executen al nucli 2 i al nucli 3 tindran un cert desbalanceig respecte dels que s'executen al nucli 0 i al nucli 1. Aquest factors cal considerar-los a l'hora de decidir com s'han d'assignar les tasques i la feina a cadascun dels fils.

En l'exemple considerat el rendiment del processador i de l'aplicació serà força inferior al que potencialment es podria assolir. En l'instant de temps T el fil esclau del nucli 1 haurà acabat de fer la feina X. Els fils 0 i 1 del nucli 1 tardaran

un cert temps (T') més a finalitzar la seva tasca. I els fils dels nuclis 2 i 3 tardaran un temps (T'') força superior a T fins acabar. Per tant, els nuclis 0 i 1 estaran sense utilitzar durant $T - T''$ i $T' - T''$, respectivament.

No sols la utilització del processador serà més baixa, sinó que aquest desbalanceig farà finalitzar l'aplicació més tard. En aquest cas, a l'hora de dissenyar el repartiment de tasques cal tenir en compte en quina arquitectura s'executarà l'aplicació, i també quins són els elements que potencialment poden causar desbalancejos i quins fils podran dur a terme més feina per unitat de temps.

Figura 2. Patró màster/esclau en un multinucli amb dos processadors



1.1.3. Definició i implementació de mecanismes de sincronització

Sovint les aplicacions multifil usen mecanismes per a sincronitzar les tasques que els diferents fils estan duent a terme. Un exemple es troba en la figura presentada en el subapartat anterior (figura 2), en què el fil màster s'esperarà que els esclaus acabin mitjançant funcions d'espera.

En general, s'acostumen a fer servir tres tipus de mecanismes diferents de sincronització:

- L'ús de variables per a controlar l'accés a determinades parts de l'aplicació o a determinades dades de l'aplicació (com un comptador global). Exemples d'aquest tipus de variables són els semàfors o les d'exclusió mútua.

- L'ús de barreres per tal de controlar la sincronització entre els diferents fils d'execució. Aquestes ens permetran assegurar que tots els fils no passen d'un determinat punt fins que tots no hi han arribat.

- L'ús de mecanismes de creació i espera de fils. Com en l'exemple anterior, el fil màster esperarà la finalització dels diferents fils esclaus mitjançant crides a funcions d'espera.

Des del punt de vista d'arquitectures multifil/nucli, el segon i el tercer punt són menys intrusius al rendiment de l'aplicació (Villa, Palermo i Silvano, 2008). Com es veurà a continuació, les barreres són mecanismes que s'empren en només certes parts de l'aplicació i que es poden implementar de manera eficient dins d'un multifil/nucli. En canvi, un ús excessiu de variables de control pot provocar un descens significatiu del rendiment de les aplicacions.

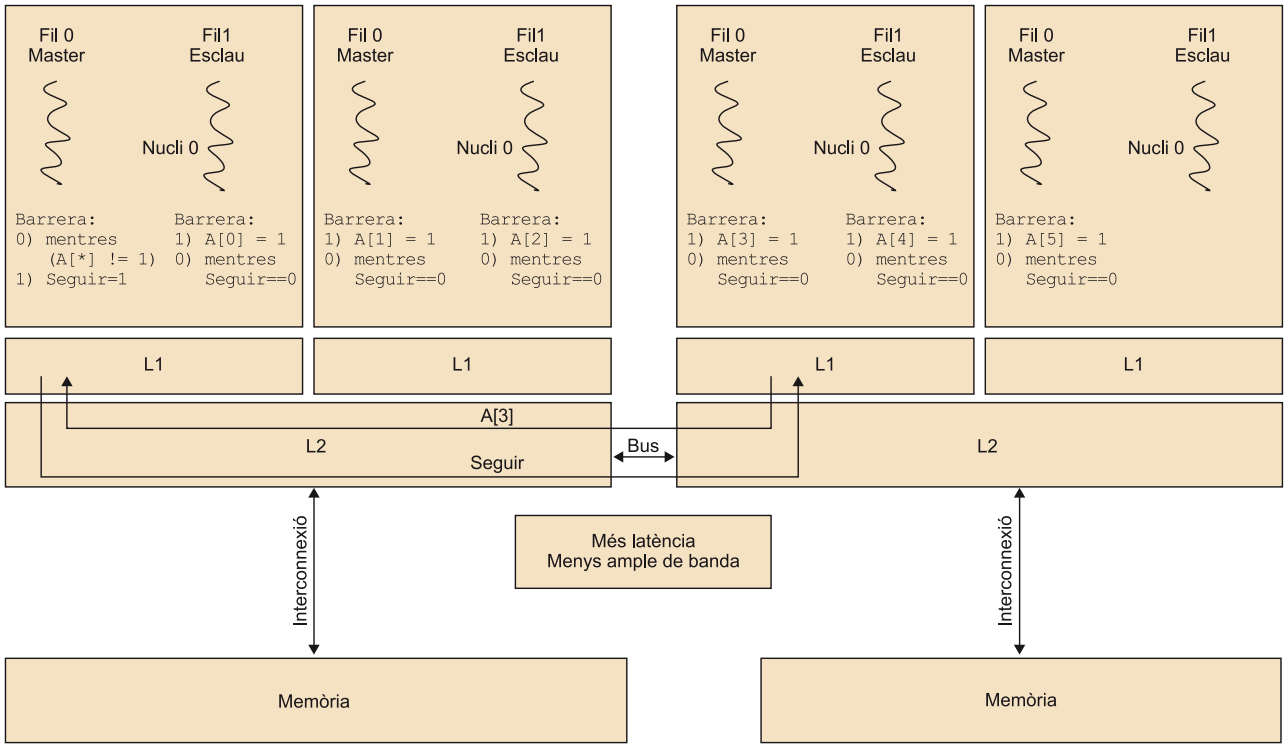
Lectura complementària

O. Villa; G. Palermo; C. Silvano (2008). "Efficiency and scalability of barrier synchronization on NoC based many-core architectures". *CASES '08 Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*.

Barreres en arquitectures multinucli

La figura 3 presenta un exemple de possible implementació de barrera i com es comportaria en un multinucli. En aquest cas, un fil s'encarrega de controlar el nombre de nuclis que han arribat a la barrera. Cal fer notar que el nucli 0 per tal d'accedir al vector *A* tindrà totes aquestes dades a la L1, en estat compartit o exclusiu.

Figura 3. Funcionament d'una barrera en un multinucli



Cada vegada que un fil *i* arribi a la barrera, voldrà modificar la posició corresponent del vector. Per tant, invalidarà la línia corresponent (la que conté *A[i]*) de tots els nuclis i la modificarà. Quan el nucli 0 torni a llegir l'adreça de *A[i]*

n'haurà de demanar el valor al nucli *i*. En funció del tipus de protocol de coherència que implementi el processador, el nucli 0 invalidarà la línia del nucli *i* o bé la mourà a estat compartit.

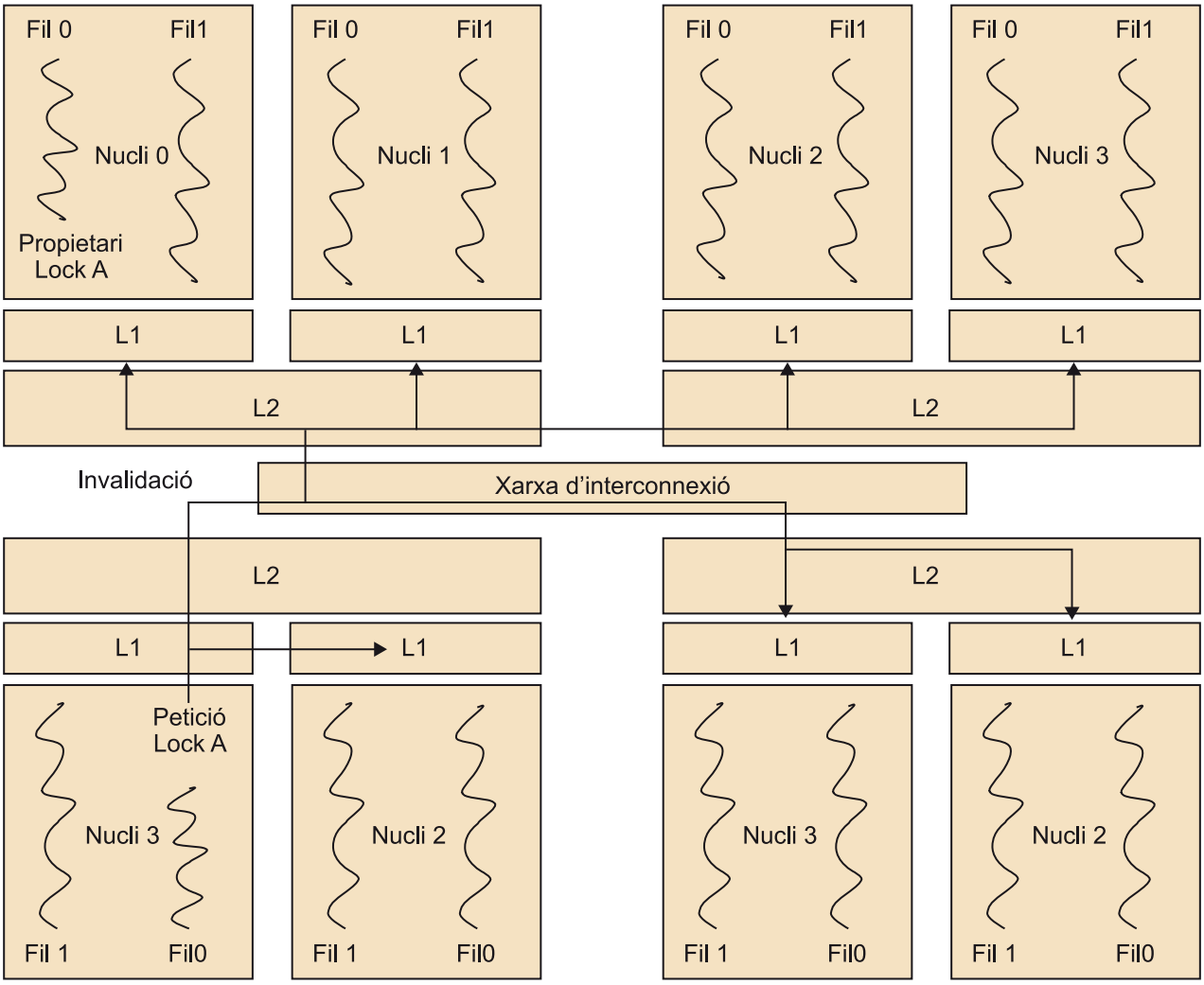
Com es pot deduir d'aquest exemple, el rendiment d'una barrera podrà ser més o menys eficient en funció de la seva implementació i de l'arquitectura sobre la qual s'està executant. Així, una implementació en què en lloc d'un vector hi ha una variable global que compta els que han acabat seria més ineficient, ja que els diferents nuclis haurien de competir per agafar la línia, invalidar els altres nuclis i escriure'n el valor nou.

Mecanismes d'exclusió mútua en arquitectures multinucli

Com s'ha introduït prèviament, aquests mecanismes s'empren per tal de poder accedir de manera exclusiva a certes parts del codi o a certes variables de l'aplicació. Aquests mecanismes són necessaris per a mantenir l'accés coordinat als recursos compartits i evitar condicions de carreres, interbloquejos (*deadlocks*) o situacions similars. Alguns d'aquests tipus de recursos són: semàfors, *mutex-locks* o *read-writer locks*.

En arquitectures d'un sol nucli l'accés a aquests tipus d'estructures pot tenir un impacte relativament inferior. Amb molta probabilitat tots els fils estaran compartint els accessos a les mateixes línies de la L1 que les guardaran. No obstant això, en un sistema multinucli l'accés concurrent de diferents fils a aquestes estructures pot comportar problemes d'escalabilitat i de rendiments greus. De manera similar al que s'ha mostrat amb les barreres, l'accés a les línies de memòria que contenen les variables emprades per a gestionar l'exclusió mútua implicaran invalidacions i moviments de línies entre els nuclis del processador.

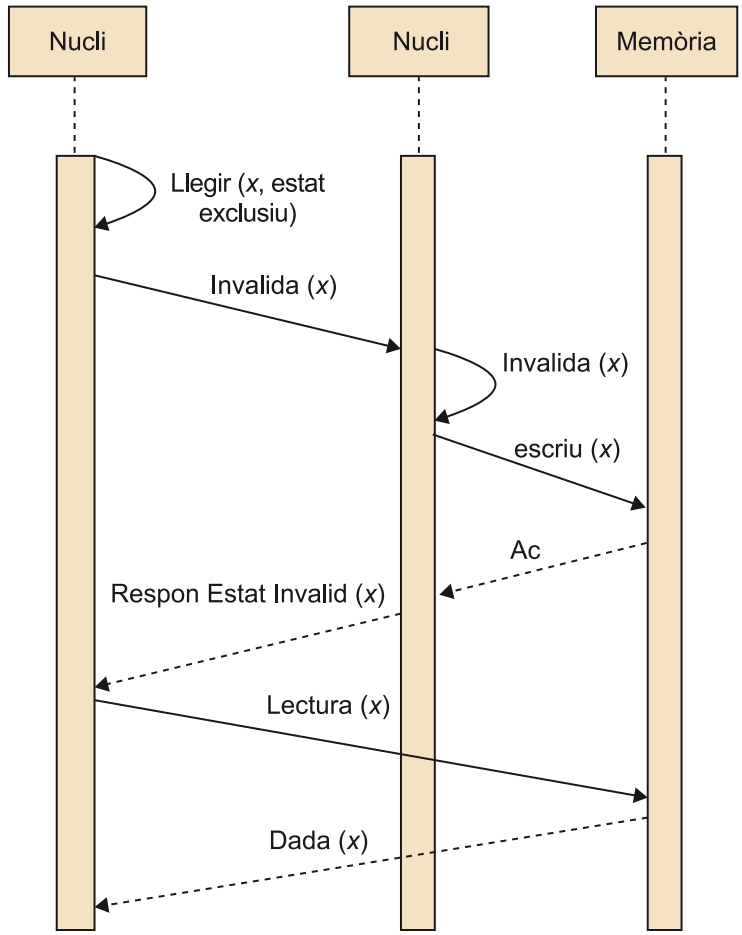
Figura 4. Adquisició d'una variable d'exclusió mútua



La figura 4 mostra un escenari en què l'ús freqüent d'accés a zones d'exclusió mútua entre els diferents fils pot reduir substancialment el rendiment de l'aplicació. En aquest exemple s'assumeix un protocol de coherència entre els diferents nuclis de tipus *snoop*, per tant, cada vegada que un dels fils d'un nucli vol agafar la propietat de la variable d'exclusió mútua (*lock*) ha d'invalidar tota la resta de nuclis. Cal fer notar que la línia de la memòria cau que conté la variable en qüestió es trobarà en estat modificat cada cop que el nucli l'actualitzi. Cada vegada que un fil agafa el *lock,* modifica la variable per tal de marcar-la com a pròpia. En cas que el fil només l'estigui consultant no caldria invalidar els altres nuclis.

Es pot assumir que la variable es reenvia del nucli que la té (el nucli 0) al nucli que la demana (el nucli 3) en estat modificat. Ara bé, depenent de quin tipus de protocol de coherència implementés el processador, la línia s'escriuria primer a memòria i, llavors, el nucli 3 la podria llegir. En aquest cas, el rendiment seria extremament baix: per cada lectura del *lock x,* s'invalidarien tots els nuclis, així el que la tingués en estat modificat l'escriuria a memòria i finalment el nucli que l'estigués demanant la llegiria de memòria (figura 5).

Figura 5. Lectura del *lock x* en estat exclusiu



Com s'ha pogut veure, és recomanable un ús moderat d'aquest tipus de mecanismes en arquitectures amb molts nuclis i també en arquitectures heterogènies. Així, doncs, a l'hora de decidir quin tipus de mecanismes de sincronització es fan servir cal considerar l'arquitectura sobre la qual s'executa l'aplicació (jerarquia de memòria, protocols de coherència i interconnexions entre nuclis) i com s'implementen tots aquests mecanismes.

Lectura recomanada

Per tal d'aprofundir en la creació de mecanismes d'exclusió mútua escalables en arquitectures multinucli es recomana la lectura de l'article:

M. Chynoweth; M. Lee (2009). *Implementing Scalable Atomic Locks for Multi-Core*. Recuperat el 28 de desembre del 2011: http://software.intel.com/en-us/articles/implementing-scalable-atomic-locks-for-multi-core-intel-em64t-and-ia32-architectures/

1.1.4. Gestió d'accés concurrent a dades

Al subapartat anterior s'han estudiat les implicacions d'usar diferents tècniques d'exclusió mútua en arquitectures multiprocessador. Aquest tipus de tècniques són emprades per tal d'assegurar que l'accés a les dades entre diferents fils és controlat. Si no s'usen aquestes tècniques de manera adequada les aplicacions poden acabar tenint carreres d'accés o *race accesses*.

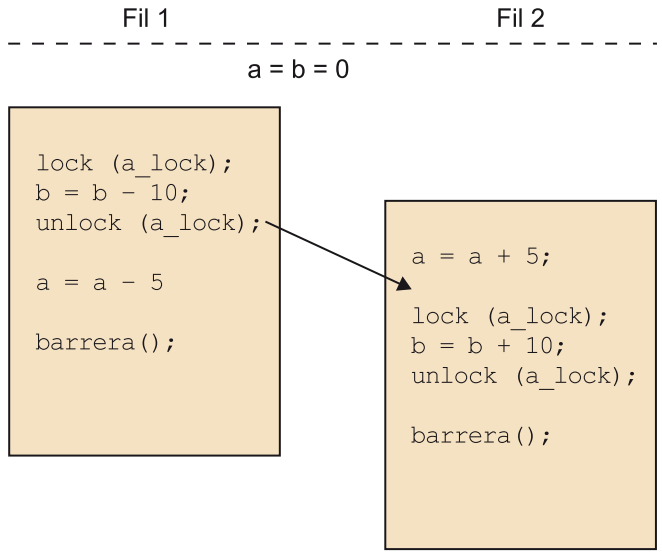
En general, es poden distingir dos tipus de carreres d'accés que cal evitar quan es desenvolupa un codi paral·lel:

- Carreres d'accés a dades: aquestes succeeixen quan diferents fils estan accedint de manera paral·lela a les mateixes variables sense cap tipus de protecció. Per tal que succeeixi una carrera d'aquest tipus, un dels accessos ha de ser fet en forma d'escriptura. La figura 6 mostra un codi que potencialment pot tenir un accés a carrera de dades. Suposant que els dos fils s'estan executant al mateix nucli, quan ambdós fils arriben a la barrera, quin valor té ***a***? Atès que l'accés a aquesta variable no està protegit i s'hi accedeix tant en mode lectura com escriptura (afegint-hi 5 i –5), aquesta variable pot tenir els valors següents: 0, 5 i –5.

- Carreres d'accés generals: aquest tipus de carreres succeeixen quan dos fils diferents han de seguir un ordre específic en l'execució de diferents parts del seu codi però no tenim estructures que forcin aquest ordre. La taula mostra un exemple d'aquest tipus de carreres. Ambdós fils usen una estructura guardada en memòria compartida en què el primer fil posa un treball i el segon, el processa. Com es pot observar, si no s'hi afegeix cap tipus de control o variable de control, és possible que el fil 2 finalitzi la seva execució sense processar el treball ***a***.

Cal esmentar que una carrera d'accés a dades és un tipus específic de carrera d'accés general. Ambdós tipus poden ser evitats usant els mecanismes d'accés introduïts en el subapartat anterior (barreres, variables d'exclusió mútua, etc.). Sempre que es dissenyi una aplicació multifil cal considerar que els accessos en mode escriptura i lectura a parts de memòries compartides han d'estar protegits; si els diferents fils assumeixen un ordre específic en l'execució de diferents parts del codi cal forçar-ho via mecanismes de sincronització i espera.

Figura 6. Carrera d'accés a dades



El primer dels dos exemples presentats (figura 6) es pot evitar afegint una variable d'exclusió mútua que controli l'accés a la variable ***a***. D'aquesta manera, independentment de qui accedeixi primer a modificar el valor de la variable, aquest valor un cop arribat a la barrera serà 0. El segon dels dos exemple podria ser evitat emprant mecanismes d'espera entre fils, és a dir, com es pot veure en la taula 2, el segon fil hauria d'esperar que el primer fil notifiqués que li ha facilitat el treball.

Taula 1. Exemple de carrera d'accés general

Fil 1	Fil 2

`Treball a = nou_treball();`
`Configurar_Treball(a);`
`EncuaTreball(a,Cua);`
`PostProces();`
	`Treball b = AgafaTreball(Cua);`
	`si(b != INVALID)`
	`ProcessaTreball(b);`
	`Acaba();`

Taula 2. Evita la carrera d'accés mitjançant sincronització

Fil 1	Fil 2

`Treball a = nou_treball();`
`Configurar_Treball(a);`
`EncuaTreball(a,Cua);`
`NotificaTreballDisponible();`
`PostProces();`
	`EsperaTreball();`
	`Treball b = AgafaTreball(Cua);`
	`si(b != INVALID)`
	`ProcessaTreball(b);`
	`Acaba();`

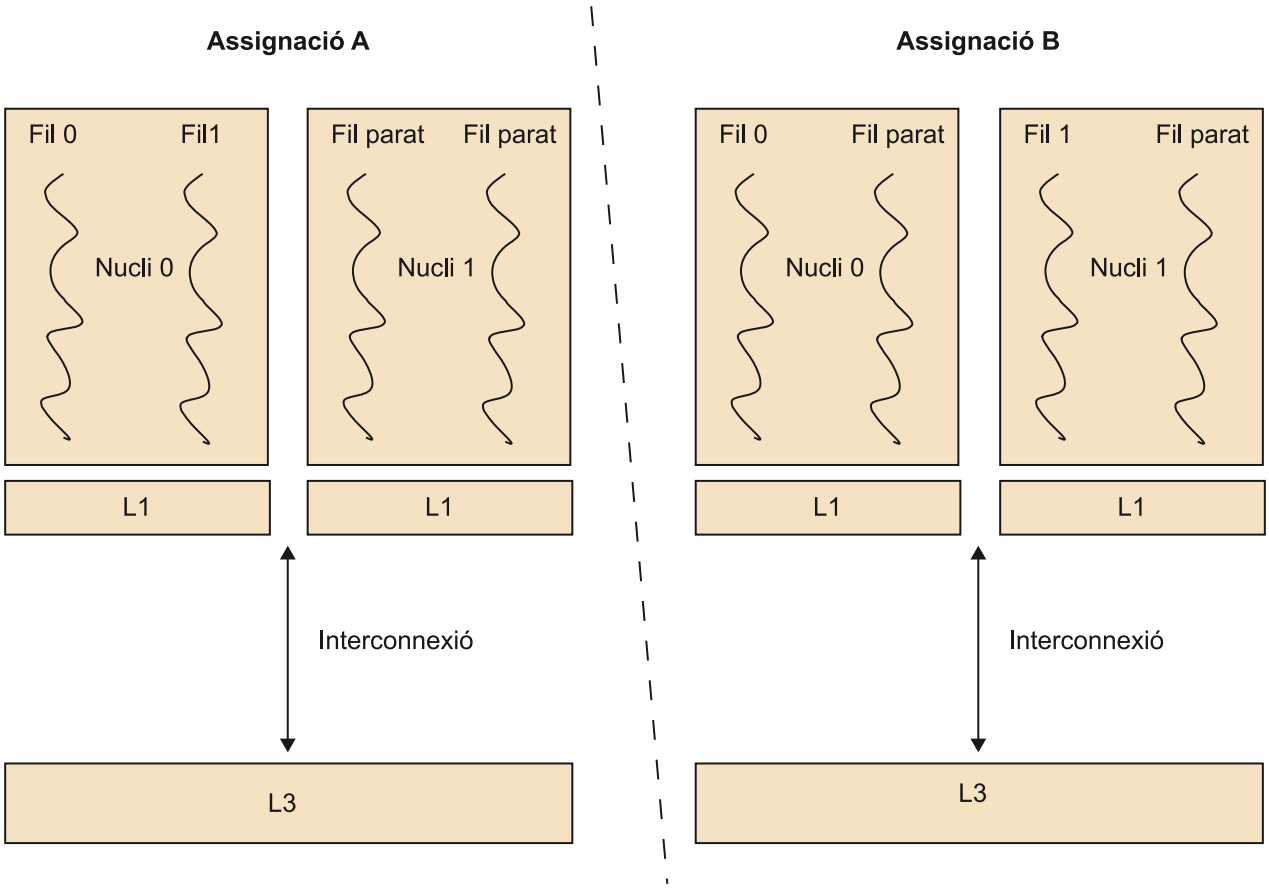
Condicions de carrera en arquitectures multinucli

En el subapartat anterior s'han introduït diverses situacions en què l'ús de memòria compartida entre diferents fils és inadequat. El primer, les carreres d'accés a dades, esdevé quan dos fils diferents estan llegint i escrivint de manera descontrolada a una zona de memòria. S'ha mostrat que el valor d'una variable pot ser funcionalment incorrecte quan ambdós fils finalitzen els seus fluxos d'instruccions (taula 1). Ara bé, aquesta cadena d'esdeveniments succeeix d'aquesta manera independentment de l'arquitectura i de l'assignació de fils sobre els quals s'executa l'aplicació?

En aquest subapartat es vol mostrar que el mateix codi es pot comportar de manera diferent dins d'un mateix processador en funció de com s'assignin els fils als nuclis, ja que depenent d'aquest aspecte la condició de carrera analitzada en el subapartat anterior succeirà o no.

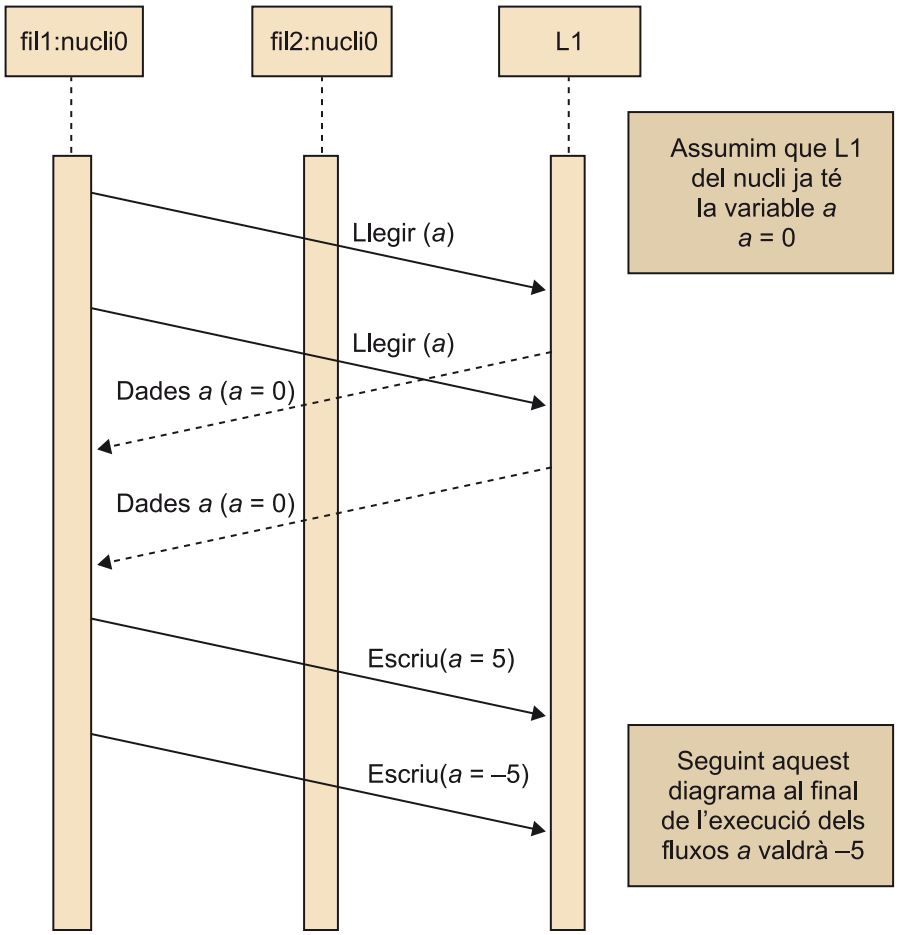
Suposem els dos escenaris mostrats en la figura 7.

Figura 7. Dues assignacions de fils diferents executant l'exemple anterior



En el primer dels dos escenaris els dos fils s'estan executant en el mateix nucli. Tal com es mostra en la figura 8, els dos fils accedeixen al contingut de la variable ***a*** de la mateixa memòria cau de nivell dos. Primer el fil 1 en llegeix el valor.

Figura 8. Accés compartit a *a* en un mateix nucli



A continuació, tot i que el fil 1 ja està modificant la variable, el segon fil en llegeix el valor original. Finalment, ambdós fils escriuran els valors a memòria. No obstant això, el segon fil sobreescriurà el valor actual (5) pel valor resultant de l'operació aritmètica que el fil haurà aplicat (–5). Aquest flux d'esdeveniments, com s'ha vist, és funcionalment incorrecte, és a dir, el resultat de l'execució dels diferents fils no és l'esperat per l'aplicació en qüestió.

Assumim ara el segon dels escenaris, en què cadascun dels fils s'executa en un nucli diferent. Com s'ha vist en mòdul "Arquitectures multifil", en les arquitectures de processadors amb memòria coherent si dos nuclis diferents estan accedint en un mateix moment a una mateixa línia de memòria el protocol de coherència assegurarà que només un nucli pugui modificar el valor d'aquesta línia. Per tant, en un instant de temps tan sols un nucli podrà tenir la línia en estat exclusiu per tal de modificar-la.

En aquest escenari nou, i gràcies al protocol de coherència, la carrera d'accés mostrada en el cas anterior no succeirà. Com es pot veure en la figura 9, el protocol de coherència protegirà l'accés en mode exclusiu a la variable ***a***. Quan el primer fil vol llegir el valor de la variable en mode exclusiu per a modificar-lo, n'invalida totes les còpies dels nuclis del sistema (en aquest cas només un).

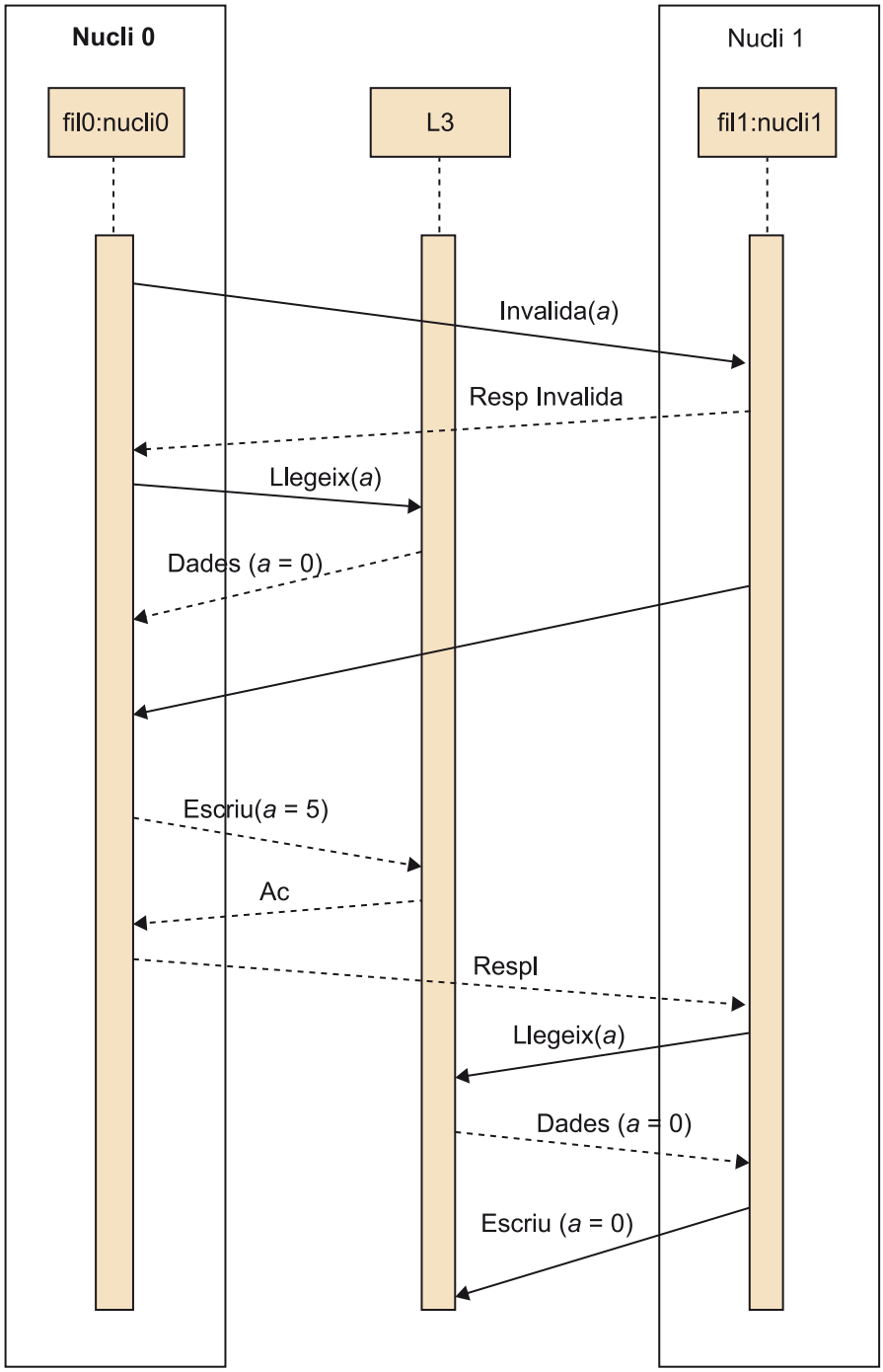
Un cop el nucli 0 notifica que té la línia en estat invàlid, en demana el valor a la memòria cau de tercer nivell. Per tal de simplificar l'exemple suposem que modifica el valor de ***a*** tan aviat com la rep.

A continuació, el fil 0 que s'està executant en el nucli 1, per tal d'agafar la línia en mode exclusiu invalida la línia dels altres nuclis del sistema (nucli 0). Quan el nucli 0 en rep la petició d'invalidació, aquest en valida l'estat. En estar en estat modificat n'escriu el valor a la memòria cau de tercer nivell. Un cop en rep l'*acknowlegement*[1], respon al nucli 1 que la línia està en estat invàlid. Arribat a aquest punt, el nucli 1 en demanarà el contingut a la L3, rebrà la línia, la modificarà i l'escriurà de tornada amb el valor correcte.

(1)D'ara endavant abreujarem *acknowlegement* amb *ack*.

Per una banda, és important remarcar que si bé en aquest cas no es dóna la carrera d'accés, la implementació paral·lela continua tenint un problema de sincronització. Depenent de quin tipus d'assignació s'apliqui als fils de l'aplicació es trobarà l'error ja estudiat.

Figura 9. Accés compartit a *a* en nuclis diferents



Per l'altra, cal tenir present que depenent de quin tipus de protocol de coherència implementi el processador, el comportament de l'aplicació podria variar. Per aquest motiu, emprar les tècniques de sincronització per tal de fer la seva execució determinista i no lligada a factors arquitectònics o d'assignació és realment rellevant.

1.1.5. Altres factors que cal considerar

Durant aquest subapartat s'han presentat diferents factors que cal considerar a l'hora de dissenyar, desenvolupar i executar aplicacions paral·leles en arquitectures multifil, i també les característiques principals i les implicacions que aquestes aplicacions tenen sobre les arquitectures multifil.

Com s'ha esmentat, el disseny d'aplicacions paral·leles és un camp en què s'ha fet molta recerca (tant acadèmica com industrial). És, per tant, aconsellable aprofundir en algunes de les referències facilitades. Altres factors que no s'han esmentat però que també són importants són:

Lectures recomanades

Per tal d'aprofundir en aquest aspecte es recomana la lectura de:

B.-C. Kim; S.-W. H.-K. Jun (2009). "Visualizing Potential Deadlocks in Multithreaded Programs". *10th International Conference on Parallel Computing Technologies*.

S. Prasad (1996). *Multithreading Programming Techniques*. Nova York: McGraw-Hill, Inc.

- Els interbloquejos. Aquests succeeixen quan dos fils es bloquegen esperant un recurs que té l'altre. Ambdós fils restaran bloquejats per sempre, per tant bloquejant l'aplicació (Kim i Jun, 2009).

- Composició dels fils paral·lels. És a dir, la manera com s'organitzen els fils d'execució. Aquesta organització depèn del model de programació (com OpenMP o MPI) i de com es programa l'aplicació (per exemple, depèn de si els fils estan gestionats per l'aplicació amb POSIX Threads).

- Escalabilitat del disseny. Factors com ara el nombre de fils, baixa concurrència en el disseny o bé massa contenció en accessos als mecanismes de sincronització poden reduir substancialment el rendiment de l'aplicació (Prasad, 1996).

En el subapartat següent es presenten alguns factors que poden impactar en el rendiment de les aplicacions paral·leles que no es troben directament lligats al model de programació. Com es veurà a continuació alguns d'aquests factors apareixen depenent de les característiques del processador multifil sobre el qual s'executa l'aplicació.

1.2. Factors lligats a l'arquitectura

Durant els subapartats següents s'estudien alguns dels factors més importants que cal tenir en compte quan es desenvolupen aplicacions paral·leles per a arquitectures multifil.

Per una banda, com es veu a continuació, la compartició de recursos entre diferents fils pot comportar situacions de conflictes que degraden molt el rendiment de les aplicacions. Un cas de compartició és el de les mateixes entrades d'una memòria cau.

Per l'altra, el disseny de les arquitectures multifil implica certes restriccions que cal considerar implementant les aplicacions. Per exemple, tal com s'ha vist en el mòdul "Arquitectures multifil", un processador multinucli té un sistema de jerarquia de memòria en què els nivells inferiors (per exemple: la L3) es

comparteixen entre diferents fils i els superiors es troben separats pel nucli (per exemple: L1). Els errors als nivells superiors són força menys costosos que als nivells inferiors. No obstant això, la mida d'aquestes memòries és força inferior, per tant, cal adaptar les aplicacions perquè tinguin el màxim de local possible als nivells superiors.

Aquest subapartat està centrat en aspectes lligats als protocols de coherència i gestió de memòria, tot i que hi ha molts altres factors que cal considerar si es vol treure el màxim rendiment de l'algorisme que s'està dissenyant. Per exemple: optimitzacions del compilador (Lo, Eggers, Levy, Parekh i Tullsen, 1997), característiques de les memòries cau (Hily i Seznec, 1998) o el joc d'instruccions del processador (Kumar, Farkas, Jouppi, Ranganathan i Tullsen, 2003). Es recomana a l'estudiant aprofundir en les diferents referències facilitades.

Lectures recomanades

J. L. Lo; S. J. Eggers; H. M. Levy; S. S. Parekh; D. M. Tullsen (1997). "Tuning Compiler Optimizations for Simultaneous Multithreading". *International Symposium on Microarchitecture* (pp. 114-124).

S. Hily i A. Seznec (1998). "Standard Memory Hierarchy Does Not Fit Simultaneous Multithreading". *Workshop on Multithreaded Execution, Architecture, and Compilation.*

R. Kumar; K. I. Farkas; N. P. Jouppi; P. Ranganathan; D. M. Tullsen (2003). *Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction* (pp. 81-92).

1.2.1. Falsa compartició o *false sharing*

En moltes situacions es voldrà que diferents fils de l'aplicació comparteixin zones de memòria concretes. Tal com s'ha vist anteriorment, sovint es donen situacions en què diferents fils comparteixen comptadors o estructures en què es desen dades resultants de càlculs fets per cadascun d'ells, en què s'usen variables d'exclusió mútua per tal de protegir aquestes zones.

Les dades que els diferents fils estan usant en un instant de temps concret es troben guardades en les diferents memòries cau de la jerarquia (des de la memòria cau de darrer nivell, fins a la memòria cau de primer nivell del nucli que l'està fent servir). Per tant, en els casos en què dos fils estan compartint una dada també compartiran les mateixes entrades de les diferents memòries cau que guarden aquesta dada. Per exemple, en la figura 8 introduïda anteriorment ambdós fils accedeixen a la mateixa entrada de la L1 que guarda la variable *a*.

Des del punt de vista de rendiment, el que es busca és que els accessos dels diferents fils encertin alguna de les memòries cau (com més propera al nucli millor), ja que l'accés a memòria és molt costós. D'això se'n diu mantenir la localitat en els accessos (Grunwald, Zorn i Henderson, 1993).

Ara bé, es poden donar situacions en què les mateixes entrades de la memòria cau siguin compartides per dades diferents. El càlcul de quina entrada d'una memòria cau es fa servir es defineix en funció de l'associativitat i el tipus d'assignació de la memòria (Handy, 1998). Per exemple, una memòria 2-associativa (Seznec, 1993) dividirà la memòria cau en conjunts de dues entrades. Primer es calcularà a quin dels conjunts pertany l'adreça i després s'escollirà quina de les dues entrades conjunt es fa servir.

Es pot donar la situació en què dos fils estiguin accedint a dos blocs de memòria diferents, que es mapen al mateix conjunt d'una memòria cau. En aquest cas, els accessos d'un fil al seu bloc de memòria invalidaran les dades de l'altre fil guardades a les mateixes entrades de la memòria. Tot i que les adreces coincidiran en les mateixes entrades de la memòria, les dades i les adreces seran diferents. Per tant, per a cada accés s'invaliden les dades de l'altre fil i es demana la dada al següent nivell de la jerarquia memòria (per exemple: la L3 o memòria principal). Com es pot deduir, aquest aspecte implica una reducció important del rendiment de l'aplicació. Això s'anomena *falsa compartició* o *false sharing*.

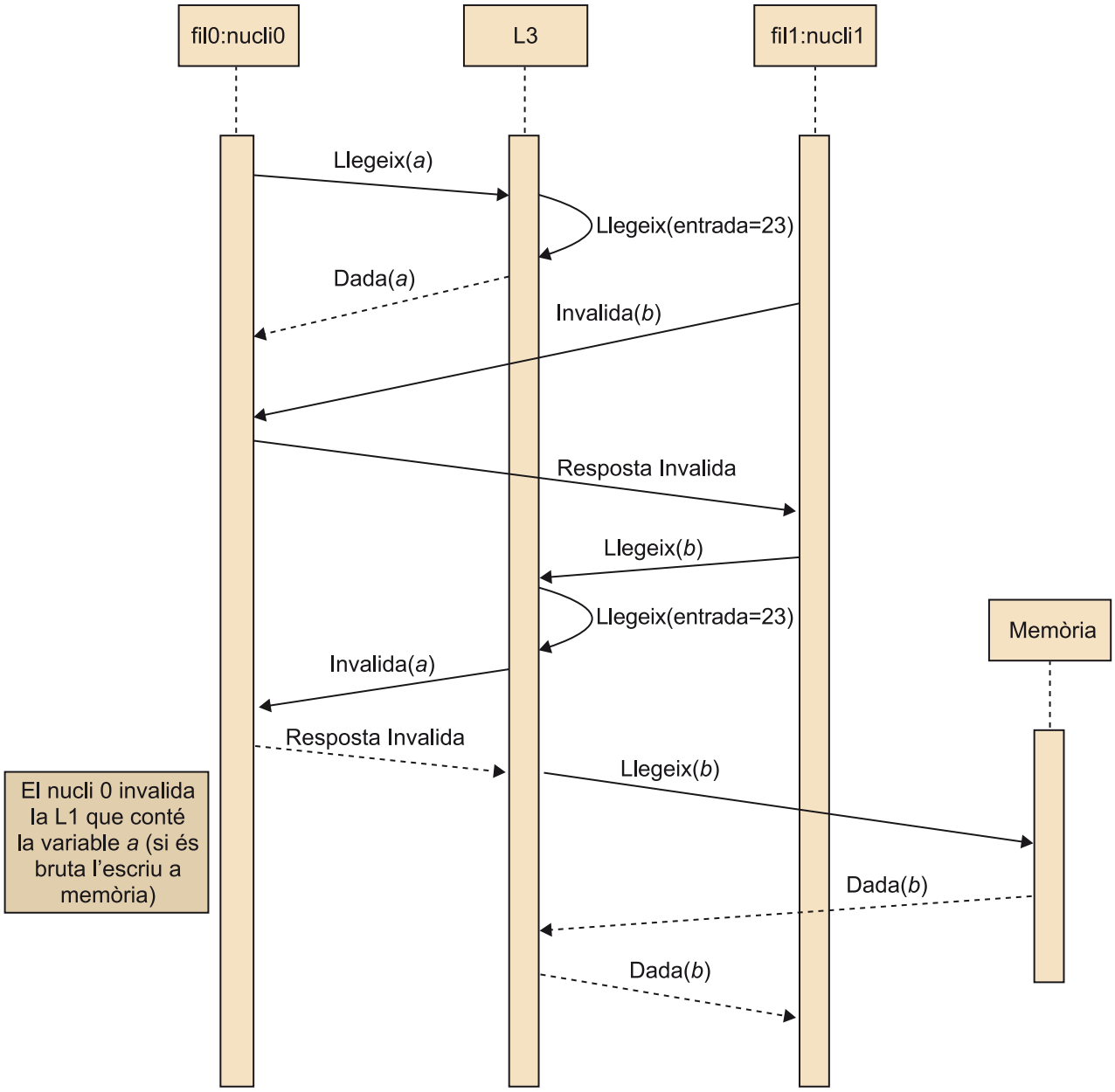
Lectures complementàries

D. Grunwald; B. Zorn; R. Henderson (1993). "Improving the cache locality of memory allocation". *ACM SIGPLAN 1993, conference on Programming language design and implementation*.

J. Handy (1998). *The cache memory book*. Londres: Academic Press Limited.

A. Seznec (1993). "A case for two-way skewed-associative caches". *20th Annual International Symposium on Computer Architecture*.

Figura 10. Falsa compartició a la L3



La figura 10 mostra un exemple del que podria passar en una arquitectura multinucli en què dos fils de diferents nuclis estan experimentant falsa compartició al mateix *set* de la L3. Per a cada accés d'un dels fils s'invalida una línia de l'altre fil, que es troba guardada en el mateix conjunt. Com s'observa, per a cada accés es generen un nombre important d'accions: invalidació d'adreça a l'altre nucli, lectura a la L3, es victimitza la dada de l'altre fil de l'altre nucli i s'escriu a memòria si és necessari, es llegeix la dada a memòria i s'envia al nucli que l'ha demanada.

En cas que ambdós fils no tinguessin conflicte en els mateixos conjunts de la L3, amb alta probabilitat encertarien a L3 i s'estalviarien la resta de transaccions que s'esdevenen per culpa de la falsa compartició.

La falsa compartició es pot detectar quan una aplicació mostra un rendiment molt baix i el nombre d'invalidacions d'una memòria cau concreta i misos és molt més elevat de l'esperat. En aquest cas, és molt probable que dos fils estiguin competint pels mateixos recursos de la memòria cau. Hi ha eines com el VTune d'Intel (Intel, Boost Performance Optimization and Multicore Scalability, 2011), que permeten detectar aquest tipus de problemes.

Lectura complementària

Intel (2011). *Boost Performance Optimization and Multicore Scalability*. Recuperat el 3 de gener del 2012: http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/

Evitar la falsa compartició és relativament més senzill del que pot semblar. Un cop s'han detectat quines són les estructures que probablement estan causant aquest efecte, cal afegir-hi un desplaçament per tal que caiguin en posicions de memòria diferent per a cadascun dels fils. D'aquesta manera, quan se'n faci el còmput per a saber a quin conjunt de la memòria cau se li assigna la variable, aquest *set* serà diferent.

1.2.2. Penalitzacions per a errors a la L1 i tècniques de *prefetch*

Moltes aplicacions paral·leles tenen una alta localitat a les memòries cau del nucli sobre les quals s'estan executant. És a dir, la majoria d'accessos que fan a memòria encerten la memòria cau de primer nivell.

Ara bé, els casos en què les peticions a memòria no encerten la memòria cau del nucli han de fer un procés molt més llarg fins que la dada està a disposició del fil: petició a la L3, error a la L3, petició a memòria, etc. Evidentment, com més alt és el percentatge d'errors, més penalitzada es troba l'aplicació. La causa és que un accés a memòria que falla a la L1 té una latència molt més elevada que una que encerta (un o dos ordres de magnituds més elevats, depenent de l'arquitectura i protocol de coherència).

Una de les tècniques que s'usa per a mirar d'amagar la latència més llarga de les peticions que fallaran a la memòria cau s'anomena *prefecthing*. Aquesta tècnica consisteix a demanar la dada a memòria molt més aviat del que l'aplicació la necessita. D'aquesta manera, quan realment la necessita aquesta ja es trobarà a la memòria local del nucli (L1). Per tant, encertarà, la latència de la petició a memòria serà molt més baixa i l'aplicació no es bloquejarà.

Hi ha dos tipus de tècniques de *prefetching* usades en les arquitectures multinucli: *hardware prefetching* i *software prefetching*.

Hardware prefetching

Les primeres d'aquestes tècniques s'implementen en les peces dels nuclis que s'anomenen *hardware prefetchers*. Aquests intenten predir quines seran les properes adreces que l'aplicació demanarà i les demanen de manera proactiva abans que l'aplicació ho faci.

Aquest component es basa en tècniques de predicció que no requereixin una lògica gaire complexa, per exemple cadenes de Markov (Joseph i Grunwald, 1997). El problema principal d'aquest tipus de tècniques és que són agnòstiques respecte del que l'aplicació està executant. Per tant, tot i que per a algunes aplicacions pot funcionar força bé, per a altres les prediccions poden ser errònies i fer que el rendiment de l'aplicació empitjori.

Lectura complementària

D. Joseph; D. Grunwald (1997). "Prefetching using Markov predictors". *24th Annual International Symposium on Computer Architecture*.

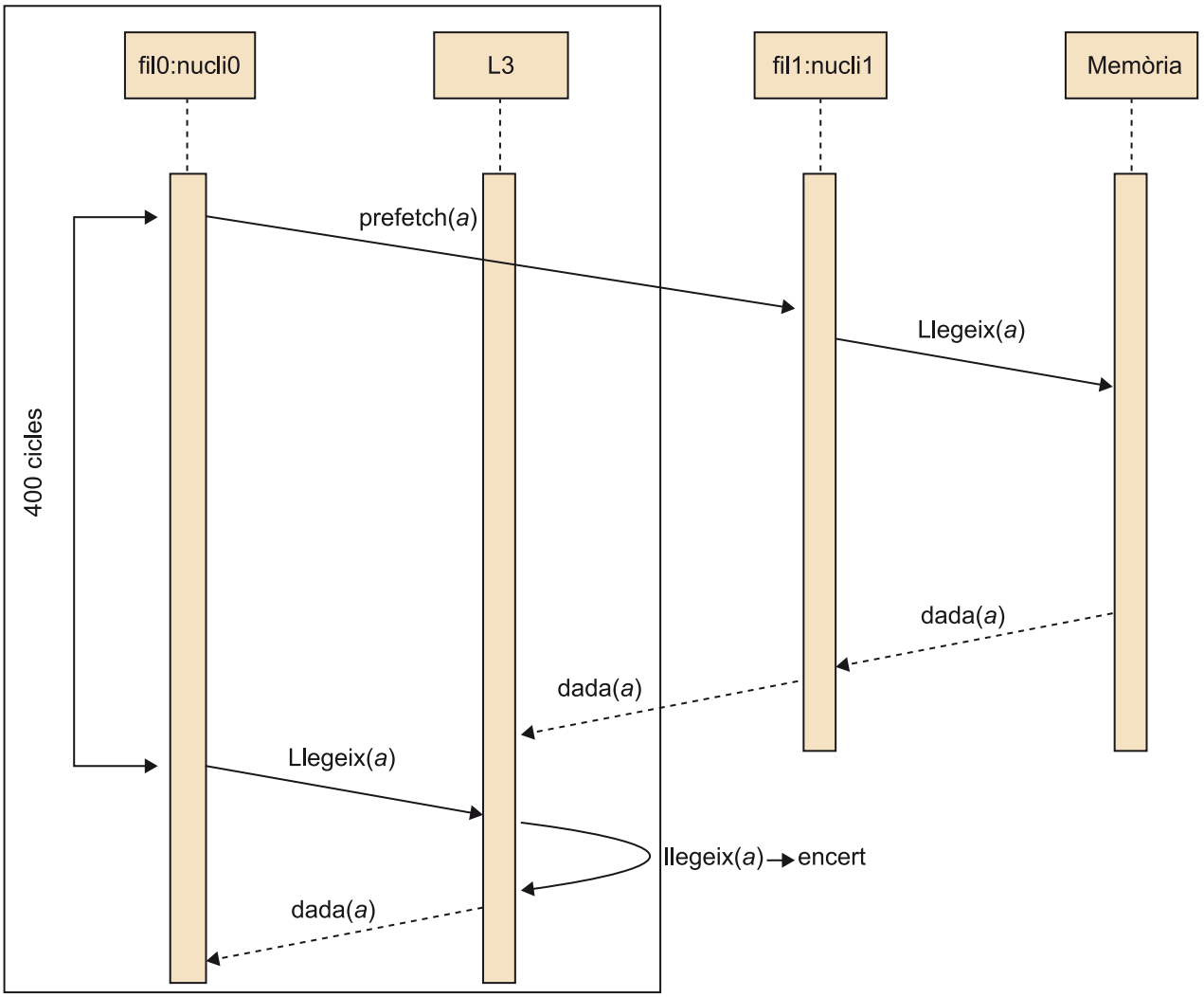
Software prefetching

La majoria de sistemes multinucli faciliten instruccions per tal que les aplicacions puguin demanar de manera explícita les dades a memòria, usant la instrucció de *prefetch*. Per exemple, en l'arquitectura Intel es fa servir la instrucció *vprefetch*. L'aplicació és responsable de demanar les dades a memòria de manera anticipada usant aquesta instrucció.

L'avantatge d'aquest mecanisme és que l'aplicació sap exactament quan necessitarà les dades. Per tant, basant-se en la latència d'un accés a memòria, haurà de demanar les dades amb el nombre de cicles suficients per tal que quan la necessiti la tingui. L'ús d'aquest tipus de *prefetch* és el que acostuma a permetre obtenir el rendiment més elevat de l'aplicació.

El desavantatge principal és que el codi es troba lligat a una arquitectura concreta. És a dir, la majoria d'arquitectures multinucli tindran latències diferents. Per tant, cada vegada que es vulgui executar aquesta aplicació en una nova plataforma caldrà calcular les distàncies de *prefetch* adequades al nou sistema.

Figura 11. Exemple de *prefetch*



La figura 11 presenta un exemple de com un *prefetch* funcionaria en un sistema format per un multinucli amb una memòria cau de tercer nivell. En aquest cas, l'aplicació necessita la dada ***a*** al cicle *X*, assumint que si la latència d'un accés a memòria que falla a la L3 és de 400 cicles, haurà de llançar el *prefetch* *X* – 400 cicles abans.

Cal tenir en compte, però, les consideracions següents.

- Els *prefetch* acostumen a poder ser eliminats del *pipeline* del processador si no hi ha prou recursos per a dur-lo a terme (per exemple, si la cua que guarda les dades que tornen de la L3 està plena). Per tant, si el nombre de *prefetch* que llança una aplicació és massa elevat, aquests poden ser eliminats del sistema.

- La latència de les peticions a memòria poden variar depenent del camí que segueixin. Per exemple, un encert a la L3 farà que una dada estigui disponible a la mateixa memòria L3 molt abans del previst, i en cas que hi hagués una víctima interna aquesta seria molt més tardana. En emprar

aquest tipus de peticions cal tenir en compte l'ús de tota la jerarquia de memòria, i també les característiques de l'aplicació que s'usa.

- Els *prefetch* poden tenir impactes de rendiment tant positius com negatius en l'aplicació desenvolupada. Cal estudiar quins requeriments té l'aplicació desenvolupada i com es comporten en l'arquitectura que està executant l'aplicació.

Lectura recomanada

L'ús de tècniques de *prefetch* en arquitectures multifil és un recurs freqüent per tal de treure rendiment en aplicacions paral·leles. Per aquest motiu, es recomana la lectura de l'article:

Intel (2007). *Optimizing Software for Multi-core Processors*. Portland: Intel Corporation - White Paper.

1.2.3. Impacte del tipus de memòria cau

Cada vegada que un nucli accedeix a una línia de memòria per primera vegada, l'ha de demanar al següent nivell de memòria. En els exemples estudiats els errors a la L2 es demanaran a la L3 i els errors a la L3 es demanaran a memòria. Cadascun d'aquests errors generarà un conjunt de víctimes en les diferents memòries cau: és necessari alliberar una entrada per la línia que s'està demanant.

Per tal de treure rendiment de les aplicacions paral·leles que es desenvolupen és important mantenir al màxim la localitat en els accessos a les memòries cau. És a dir, maximitzar el reús (percentatge d'encert) a les diferents memòries cau (L1, L2, L3, etc.). Per aquest motiu, és important considerar les característiques de la jerarquia de memòria cau: inclusiva / no inclusiva / exclusiva, mides, etc.

A continuació, es discuteixen els diferents punts esmentats des del punt de vista de l'aplicació.

Inclusivitat

En cas que dues memòries (L*x* i L*x* – 1) siguin inclusives voldrà dir que qualsevol adreça @X que es troba a la L*x* – 1 serà sempre a la L*x*. Per exemple, si la L1 és inclusiva amb la memòria L2, aquesta inclourà la L1 i altres línies. En aquest cas, caldrà considerar que:

1) Quan una línia de la *Lx* – 1 es victimitza, al final de la transacció aquesta estarà disponible al següent nivell de memòria L*x*.

2) Quan una línia de la L*x* es victimitza, al final de la transacció aquesta línia ja no estarà disponible al nivell superior L*x* – 1. Per exemple, en el cas de victimitzar la línia @X a L3, aquesta s'invalidarà també a les memòria cau L2. I si la L1 és inclusiva amb la L2, la primera també invalidarà la línia en qüestió.

3) Quan un nucli llegeix una adreça @X que no es troba en la memòria L*x* – 1, al final de la transacció aquesta també es trobarà inclosa a la L*x*. Per exemple, en cas que tinguem una L1, L2 i L3 inclusives, la @X s'escriurà a totes les memòries. Cal remarcar que cadascuna d'aquestes entrades usades potencialment haurà generat una víctima. És a dir, una adreça @Y que usa el *way* i el *set* en què s'ha desat @X. La selecció d'aquesta posició dependrà de la política de gestió de cada memòria cau, i també de la seva mida.

Els punts 2 i 3 poden causar un impacte força important en el rendiment de l'aplicació. Per tant, és important dissenyar les aplicacions per tal que el nombre de víctimes generades en nivells superiors sigui tan petit com sigui possible i maximitzar el percentatge d'encerts de les diferents jerarquies de memòria cau més properes al nucli (per exemple: L1).

Exclusivitat i no-inclusivitat

En cas que dues memòries cau siguin exclusives implicarà que si la memòria cau L*x* té una línia @X, la memòria cau L*x* – 1 no la tindrà. No s'acostumen a tenir arquitectures en què tots els nivells siguin exclusius. Habitualment les jerarquies que tenen memòries cau exclusives acostumen a ser híbrides.

Un exemple de processador amb memòria exclusiva és l'AMD Athlon (AMD, 2011) i l'Intel Nehalem (Intel, Nehalem Processor, 2011). El primer té una L1 exclusiva amb la L2. El segon té una L2 i L1 no inclusives i la L3 és inclusiva de la L2 i la L1.

En els casos en què una memòria cau L*x* és exclusiva amb L*x* – 1 caldrà considerar que:

- Quan s'accedeix a una línia @Y a la memòria L*x* – 1, aquesta es mourà de la memòria L*x*.

- Quan una línia es victimitza de la memòria L*x* – 1, aquesta es mourà a la memòria cau L*x*. En aquests casos, pel fet que la memòria és exclusiva, caldrà trobar una entrada en la memòria L*x*. Per tant, caldrà victimitzar la línia que es trobi desada en el *way* i en el *set* seleccionats.

- Quan una línia es victimitza de la memòria L*x* no caldrà victimitzar la memòria cau L*x* – 1.

Hi haurà certes situacions en què caldrà conèixer en detall quin tipus de jerarquia de memòria i protocol de coherència implementa el processador. Per exemple, si es considera una arquitectura multinucli en què la L1 és exclusiva amb la L2, en els casos en què els diferents fils estiguin compartint l'accés a un conjunt elevat d'adreces el rendiment de l'aplicació es pot veure deduït.

En aquesta situació, cada accés a una dada @X en un nucli podria implicar la victimització d'aquesta mateixa adreça en un altre nucli i demanar l'adreça a memòria.

Algunes memòries cau exclusives permeten que en certes situacions algunes dades es trobin en dues memòries que són exclusives entre elles per defecte. Per exemple, en les línies de memòria que es troben compartides per diferents nuclis.

Mida de la memòria

Com s'ha discutit, el rendiment de l'aplicació dependrà en gran manera de la localitat dels accessos dels diferents fils en les memòries cau. En situacions en què els fils demanen diverses vegades les mateixes línies a memòria per mala praxi de programació, el rendiment de l'aplicació caurà substancialment. Això succeirà en aquells casos en què un nucli demana una adreça @X, aquesta línia es victimitza i es torna a demanar més tard.

Un exemple senzill és en accedir a una matriu d'enters de 8 bytes per files quan aquesta es troba emmagatzemada per columnes. En aquest cas, cada 8 enters consecutius d'una mateix columna es trobarien mapats a la mateixa línia. Ara bé, els elements i i $i + 1$ d'una fila estarien desats en línies diferents. Per tant, en el cas de recórrer la matriu per columnes el nombre d'errors a la L1 serà molt més elevat.

Per aquest motiu, és important usar tècniques d'accés a les dades que intentin mantenir localitat a les diferents memòries cau. De manera similar a altres factors ja introduïts anteriorment, s'ha fet molta recerca en aquest àmbit.

Lectures recomanades

Una referència en tècniques de partició en blocs per la multiplicació de matrius és:

K. Kourtis; G. Goumas; N. Koziris (2008). "Improving the Performance of Multithreaded Sparse Matrix-Vector Multiplication Using Index and Value Compression". *37th International Conference on Parallel Processing*.

En tècniques de compressió en el procés de grafs seria:

R. Jin; **T.-S. Chung** (2010). "Node Compression Techniques Based on Cache-Sensitive B+-Tree". *9th International Conference on Computer and Information Science (ICIS)* (pp. 133-138).

No obstant això, en molts casos les aplicacions paral·leles dissenyades no segueixen cap dels patrons analitzats en altres estudis acadèmics (p. ex.: tècniques de partició de matrius). Per a aquests problemes hi ha aplicacions disponibles que permeten analitzar com es comporten les aplicacions paral·leles i veure de quina manera es poden millorar. Exemples d'aquestes aplicacions són: *VTune* o *cachegrind* (Valgrind, 2011).

1.2.4. Arquitectures multinucli i multiprocessador

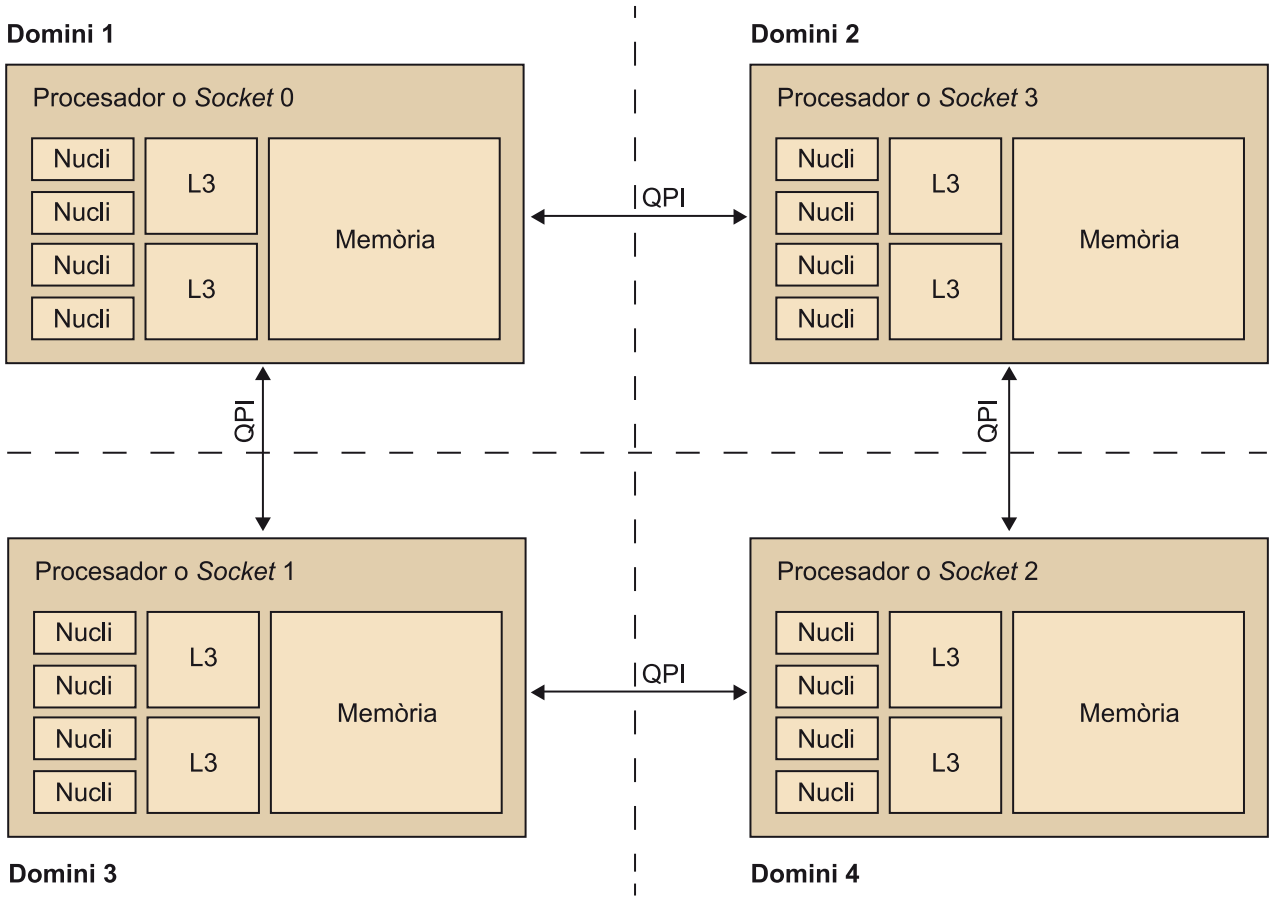
En algunes situacions les arquitectures en què s'executen les aplicacions paral·leles no solament donen accés a múltiples fils i múltiples nuclis, sinó que donen accés a múltiples processadors. En aquestes arquitectures una mateixa placa conté un conjunt de processadors connectats amb una connexió d'alta velocitat. Un exemple d'aquest tipus de connexió és l'Intel Quickpath Interconnect (Intel, Intel® Quickpath Interconnect Maximizes Multi-Core Performance, 2012), també conegut com a QPI.

En aquestes arquitectures es poden assignar els diferents fils de la nostra aplicació a cadascun dels fils disponibles en cadascun dels nuclis dels diferents processadors connectats a la placa. Tots els diferents fils acostumen a compartir un espai de memòria coherent, tal com s'ha estudiat en les seccions anteriors. Des del punt de vista de l'aplicació, aquesta té accés a *N* nuclis diferents i a un espai de memòria comú.

No obstant això, tot i que aquest espai de memòria és compartit, l'accés d'un fil a determinades zones de memòria pot tenir latències diferents. Cada processador disposa d'una jerarquia de memòria pròpia (des de la L1 fins a la memòria principal) i aquest gestiona un rang d'adreces de memòria concret. Així, si un processador disposa d'una memòria principal de 2 GB, aquest processador gestionarà l'accés de l'espai d'adreces assignat a aquests 2 GB (per exemple, de 0x a FFFFFFFF).

Cada vegada que un fil accedeixi a una adreça que es troba assignada a un espai que gestiona un altre processador, haurà d'enviar la petició de lectura d'aquesta adreça al processador que la gestiona per mitjà de la xarxa d'interconnexió. Aquests accessos seran molt més costosos atès que hauran d'enviar la petició per la xarxa, arribar a l'altre processador i accedir-hi (seguint el protocol de coherència que segueixi l'arquitectura).

Figura 12. Arquitectura multifil



Aquest model de programació seguirà el paradigma del que s'anomena *non-uniform memory access* (NUMA), en què un accés de memòria pot tenir diferents tipus de latència depenent d'on estigui assignat.

Quan es programi per a aquest tipus d'arquitectura caldrà considerar tots els factors que s'han anat explicant però en una escala superior. Així, doncs, en l'accés a variables d'exclusió mútua s'haurà de considerar que per cada vegada que s'agafi el *lock* s'haurà d'invalidar tota la resta de nuclis del sistema. Els que estiguin fora del processador aniran per la xarxa d'interconnexió i tardaran més a retornar la resposta. Val a dir que el comportament dependrà del protocol de coherència i de la jerarquia de memòria del sistema.

Lectures recomanades

Dins l'àmbit de programació multifil per a aquest tipus d'arquitectures es poden trobar moltes referències interessants, algunes de les quals són:

G. R. Andrews (1999). *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley.

M. Herlihy; N. Shavit (2008). *The Art of Multiprocessor Programming*. Morgan Kaufmann.

D. E. Culler; J. Pal Singh (1999). *Parallel computer architecture: a hardware/software approach*. Morgan Kaufmann.

2. Entorns per a la creació i gestió de fils

A l'apartat anterior s'han vist alguns dels factors més importants que cal tenir en compte a l'hora de desenvolupar aplicacions multifil. En aquest apartat s'introdueixen tres models de programació que han estat pensats per a explotar aquest paral·lelisme: els POSIX Threads, l'Intel Cilk i els Intel Thread Building Blocks.

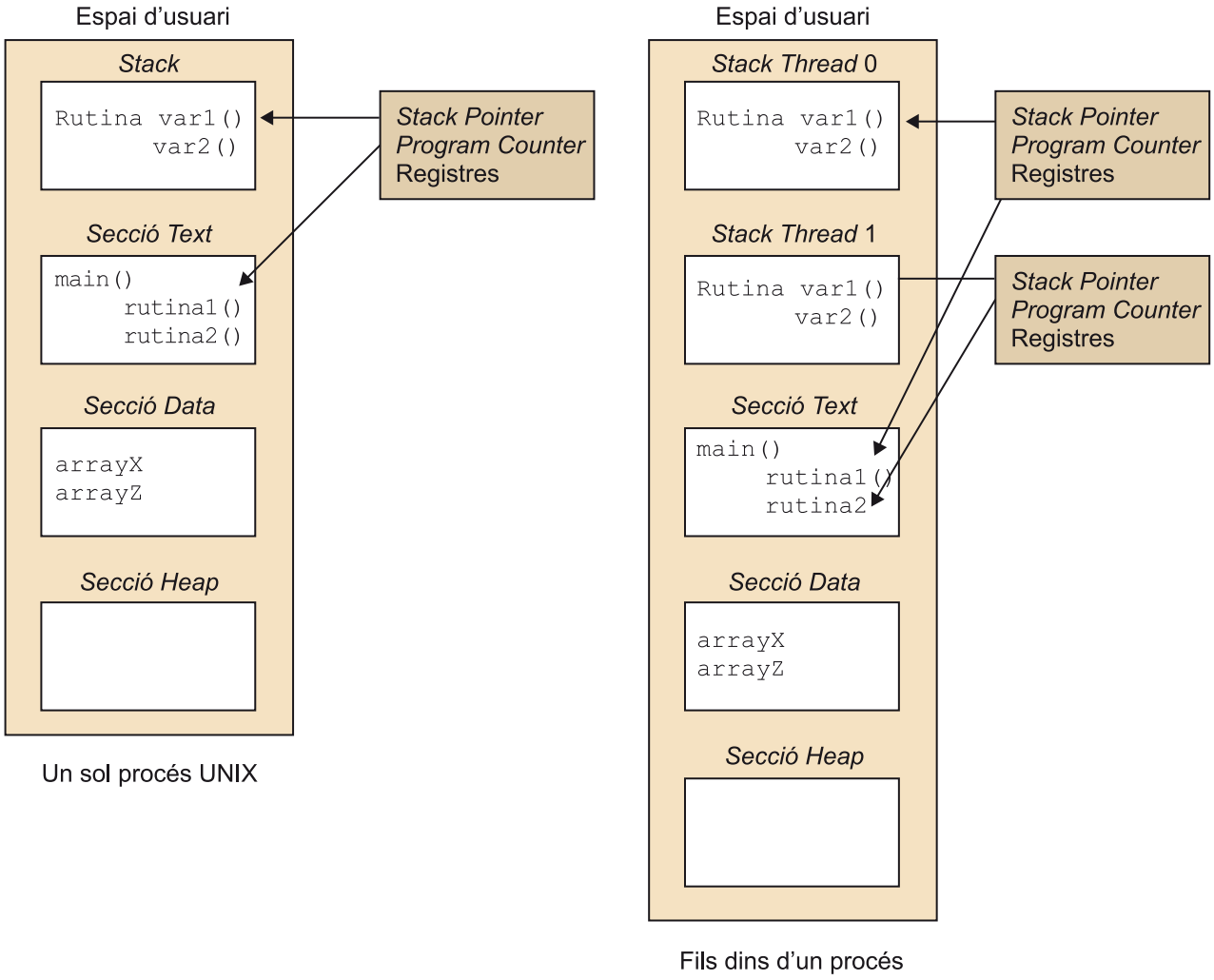
2.1. POSIX Threads

Els POSIX Threads són uns dels models de programació orientats a aplicacions multifil que més s'han usat durant les darreres dècades. Inicialment, la majoria de fabricants d'entorns que donaven suport a aplicacions multifil facilitaven els seus propis entorns i biblioteques de desenvolupament.

No obstant això, per tal de fer les aplicacions més portables i compatibles, es va definir un estàndard de programació atès que va esdevenir necessari. El 1995 IEEE va presentar l'estàndard POSIX 1003.1c (IEEE, 2011). Precisament definia una sèrie de característiques i interfícies per a la programació d'aplicacions paral·leles.

Es van desenvolupar diferents implementacions d'aquest estàndard per a moltes de les plataformes existents. Totes segueixen les mateixes interfícies i s'anomenen POSIX *Threads* o *pthreads*.

Figura 13. Model de fils POSIX



La figura 13 presenta el model d'aplicació que defineixen els *pthreads*. A la dreta de la figura s'observa la visió del procés UNIX tradicional, en què hi ha un sol flux d'execució. Aquest té:

a) Un context amb:

- Un *stack pointer* que apunta la pila de dades del procés.
- Un *program counter* que apunta la part de la secció de text de la memòria del procés que conté les instruccions que s'han d'executar.

b) Un conjunt de registres amb els valors que el procés ha anat modificant durant la seva execució.

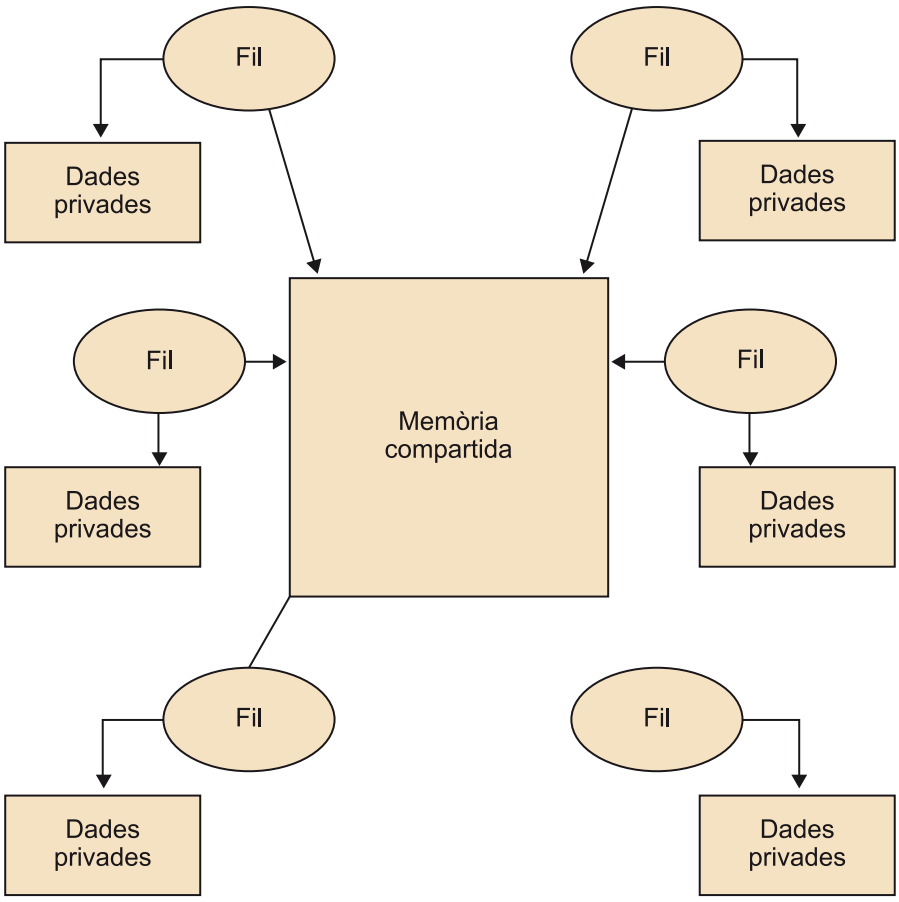
c) Una secció de *text* que conté les diferents instruccions que defineixen cadascuna de les rutines o funcions que el procés potencialment pot executar.

d) Una secció de dades en què es guarden les dades de l'aplicació.

e) Un *heap* en què es guarda la memòria dinàmica que el procés va demanant durant la seva execució.

A l'esquerra d'aquesta mateixa figura es pot observar l'extensió d'aquest model unifil al model multifil que defineixen els *pthreads*. En aquesta extensió el procés continua estant definit per les tres seccions (*data*, *text* i *heap*). Però en lloc de tenir un sol *stack* i un sol context, en té tants com fils s'han definit. Per tant, cada fil té el seu propi context i pila d'execució.

Figura 14. Model de memòria



Tal com mostra la figura 14, cada fil té una part de memòria que només és visible per a ell i té accés a una part de memòria que és visible per a tots els fils. Tal com s'ha remarcat anteriorment, quan un fil vol accedir a zones de memòria que poden ser modificades, cal usar mecanismes d'exclusió mútua per a evitar carreres d'accés.

Durant els subapartats següents es presenten les interfícies més importants d'aquest model de programació. Aquestes es troben dividides en quatre grans blocs:

- Gestió de fils. Estan orientades a la creació, gestió i destrucció de fils.

- Gestió de zones d'exclusió mútua. Aquestes faciliten l'accés a determinades zones del codi o variables en què es vol que tan sols un fil execució estigui treballant a la vegada.

- Comunicació entre fils. Orientada a enviar senyals entre els diferents fils.

- Mecanismes de sincronització. Orientats a la gestió de *locks* i barreres.

2.1.1. Creació i destrucció de fils

El codi 2.1 mostra les diferents interfícies relacionades amb la creació i la destrucció de fils. La primera és la que permet crear un fil d'execució nou. El més important és que permet definir un conjunt d'atributs que definiran com es comportarà el fil, la rutina que el fil ha d'executar i els arguments que la rutina rebrà.

Un cop el fil es creï, aquest executarà la rutina que s'ha especificat i un cop acabi invocarà la rutina de sortida (la segona del codi). Si algun fil vol finalitzar l'execució d'un altre, ho pot fer mitjançant la tercera funció. No obstant això, habitualment un fil acabarà la seva pròpia execució.

```
pthread_create (fil,atributs,rutina,arguments)
pthread_exit (estat)
pthread_cancel (fil)
pthread_attr_init (atributs)
pthread_attr_destroy (atributs)
```

Codi 2.1. Interfícies per a la creació de fils

Les dues darreres rutines permeten destruir o configurar els atributs d'un fil d'execució. De la mateixa manera que el *pthread_create*, aquestes proporcionaran com a paràmetre els atributs que es volen configurar o destruir. Els atributs es troben compartits entre els fils que s'han creat amb ells. Els paràmetres principals que cal especificar es descriuen a continuació:

- Quina mida es vol que tingui la pila.
- Un punter a l'adreça de la pila que es vol que utilitzi.
- Quina mida màxima es vol autoritzar que el fil utilitzi de la pila per tal d'evitar que s'escriguin a zones de memòria incontroladament.
- Quin tipus de política de planificació es vol dur a terme entre els diferents fils que s'han creat usant el mateix conjunt d'atributs.

El codi 2.2 mostra un exemple de com es poden crear diferents fils d'execució. Com es pot observar, en aquest cas els diferents fils són creats sense cap conjunt d'atributs. En aquest cas, els fils compartiran els atributs per defecte.

```
#include <pthread.h>
#include <stdio.h>
#define NUM_FILS 5

void *DiguesHola(void *fildid)
{
  long tid;
  tid = (long)fildid;
  printf("Hola! Soc el fil #%ld!\n", tid);
  pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
  pthread_t fils[NUM_FILS];
  int rc;
  long t;
  for(t=0; t<NUM_FILS; t++){
    printf("Soc la rutine principal i le meu identificador
            es %ld\n", t);
    rc = pthread_create(&fils[t], NULL, DiguesHola, (void *)t);
    if (rc){
      printf("ERROR: El codi es %d\n", rc);
      exit(-1);
    }
  }
  pthread_exit(NULL);
}
```

Codi 2.2. Creació i destrucció de fils

2.1.2. Espera entre fils

En l'exemple anterior el fil principal ha creat un conjunt de fils i ha finalitzat la seva execució un cop tots s'havien creat. Probablement, l'aplicació haurà acabat quan el darrer dels fils creats hagi acabat. No obstant això, caldria contrastar-ho per a comprovar-ho. D'altra banda, amb molta probabilitat, el comportament variaria entre execució i execució.

Per aquest motiu hi ha mecanismes que permeten que els fils s'esperin entre ells. D'aquesta manera, si un fil crea *N* fils diferents, aquest es podrà esperar que acabin per tal de continuar fent la feina. Aquest és el procés que es diu de *join* i s'aplica sovint en situacions en què s'està aplicant el model de programació esclau i màster. En aquest model, un fil fa de màster distribuint la feina entre un conjunt d'esclaus. I aquest s'esperarà que tots els esclaus finalitzin.

En el cas del *pthreads*, el fil màster esperarà activament que els fils acabin, usant la primera de les funcions presentades en el codi 2.3. Com es pot observar, aquesta funció té dos paràmetres: el primer permet especificar quin fil vol esperar; el segon contindrà l'estat del fil un cop hagi acabat la crida al *pthread_join* (i per tant el fil que espera hagi finalitzat la seva execució).

```
pthread_join (filid,estat)
 pthread_detach (filid)
 pthread_attr_setdetachstate (attr,detachestat)
 pthread_attr_getdetachstate (attr,detachestat)
```

Codi 2.3. Espera entre fils

No obstant això, en altre situacions no es voldrà que un fil esperi els altres fils. En aquestes situacions, quan es creen els fils es podrà especificar en els atributs que es creïn en estat *detached*. Aquest estat dirà al sistema que el fil que ha creat els diferents fils d'execució no els esperarà.

En els casos en què això no s'especifiqui en la creació dels fils, es podrà reprogramar usant les tres darreres funcions presentades en el codi 2.3. Per exemple, això es podria donar en casos en què es crea un fil i s'especifica que és *joinable* (és a dir, que el pare el pot esperar), però depenent del flux de l'aplicació el pare no caldrà que esperi.

Un exemple d'ús d'aquest tipus de crides es mostra en el codi 2.4. En aquest exemple es volen mostrar dues coses:

- Que un fil màster crea dos fils que inicialitzen un vector, els espera i finalitza usant les crides introduïdes en aquest subapartat.
- Que es poden passar paràmetres entre el fil màster i els fils esclaus.

En aquest exemple, el fil màster instancia un vector de 100k posicions i crea dos fils que inicialitzaran el seu contingut a zero. Cada fil rep com a paràmetre un punter al vector que es vol inicialitzar i la quantitat d'elements que haurà d'inicialitzar. Cal remarcar que els dos paràmetres que els fils necessiten es proporcionen a partir d'un *struct*. Això es deu al fet que la creació de *fils* només permet passar un sol punter al paràmetre que es facilitarà al fil creat.

```
typedef struct {
  int *ar;
  long n;
  } vector_nou;

void *
inicialitza(void *arg)
{
  long i;
  for (i = 0; i < ((vector_nou *)arg)->n; i++)
    ((vector_nou *)arg)->ar[i] = 0;
}

int main(void)
{
  int ar[1000000];
  pthread_t th1, th2;
  vector_nou v_nou_1, v_nou_2;

  v_nou_1.ar = &ar[0];
  v_nou_1.n = 500000;
  (void) pthread_create(&th1, NULL, inicialitza, &v_nou_1);

  v_nou_2.ar = &ar[500000];
  v_nou_2.n = 500000;
  (void) pthread_create(&th2, NULL, inicialitza, &v_nou_2);

  (void) pthread_join(th1, NULL);
  (void) pthread_join(th2, NULL);
  return 0;
}
```

Codi 2.4. Exemple d'espera

2.1.3. Ús de memòria compartida i sincronització

Un dels aspectes més importants dels *pthreads* és l'ús de memòria compartida i la sincronització entre els diferents fils d'execució.

Per una banda, hi ha moltes situacions que requereixen accedir de manera controlada a determinades zones de memòria. Per exemple, en cas que dos fils estiguin sumant diferents files d'una matriu en una variable compartida. En aquest cas, caldrà accedir-hi de manera controlada o potencialment un fil podrà sobreescriure el valor que l'altre ha escrit perdent una part de la suma.

Per una altra banda, ens trobem en situacions en què caldrà que els diferents fils estiguin sincronitzats a l'hora d'executar diferents parts de l'aplicació. Per exemple, sovint es donen situacions en què cal fer l'algorisme paral·lel en dues etapes i els fils no poden començar la segona etapa fins que tots no hagin acabat la primera. En aquests casos caldrà que els diferents fils s'esperin de manera coordinada a acabar la primera etapa abans de començar la segona.

Com s'ha dit anteriorment, aquest tipus d'accessos es controlen a través de mecanismes d'exclusió mútua. Aquestes interfícies permeten sincronitzar l'accés a zones de memòria compartida i sincronitzar també l'execució de fils.

En particular, les interfícies que s'usen en aquest entorn per a implementar els mecanismes descrits són els *mutex*. Els *mutex* són variables que funcionen com un testimoni. És a dir, quan un fil vol accedir a una acció que es controla amb un *mutex* o fer-la haurà d'agafar el testimoni d'aquest *mutex*. Quan aquest hagi acabat de fer la tasca en qüestió el fil deixarà el testimoni d'aquest *mutex* per tal que altres fils el puguin agafar quan el necessitin.

Les primeres funcions del codi 2.5 permeten inicialitzar, modificar i destruir un *mutex*. Aquestes seran usades pel fil principal. Les tres últimes funcions seran emprades pels diferents fils per tal d'agafar l'accés o alliberar el testimoni del *mutex* en qüestió.

```
pthread_mutex_init (mutex,atributs)
pthread_mutexattr_init (atributs)
pthread_mutexattr_destroy (atributs)
pthread_mutex_destroy (mutex)

pthread_mutex_lock (mutex)
pthread_mutex_trylock (mutex)
pthread_mutex_unlock (mutex)
```

Codi 2.5. Interfícies per a la creació de *mutex*

El codi 2.6 mostra un exemple de com es podrien usar els *mutex* per tal de protegir un comptador compartit entre dos nuclis diferents. Com es pot observar, es declaren dues variables globals: el comptador i el *mutex*. El fil principal crea dos fils que accedeixen de manera concurrent al comptador. No obstant això, abans de modificar-lo, agafen el testimoni del *mutex*. En aquest cas, no s'ha usat cap de les crides d'inicialització, perquè s'usa el tipus de *mutex* per defecte (el més habitual). En aquests casos, només cal declarar-lo i agafar/alliberar testimoni amb les darreres dues crides mostrades. Per tal de veure els diferents comportaments que un *mutex* pot tenir es recomana accedir a les referències ja citades.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *suma();
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int contador = 0;

main()
{
  int rc1, rc2;
  pthread_t primer_fil, segon_fil;

  if( (rc1=pthread_create( &primer_fil, NULL, &suma, NULL)) )
  {
  printf("No s'ha pogut crear: %d\n", rc1);
  }

  if( (rc2=pthread_create( &segon_fil, NULL, &suma, NULL)) )
  {
  printf("No s'ha pogut crear: %d\n", rc2);
  }

  pthread_join( primer_fil, NULL);
  pthread_join( segon_fil, NULL);

  printf("Suma total: %d\n",contador);
  exit(0);
}

void *suma()
{
  pthread_mutex_lock( &mutex1 );
  contador++;
  printf("contador value: %d\n",contador);
  pthread_mutex_unlock( &mutex1 );
}
```

Codi 2.6. Exemple d'ús de *mutex*

2.1.4. Senyals entre fils

El codi 2.7 mostra una de les interfícies interessants per al desenvolupament d'aplicacions paral·leles que els *pthreads* faciliten. Aquestes estan orientades a la comunicació entre diferents fils d'execució. Sovint no sols es vol controlar l'accés a la memòria compartida, sinó també ser capaç d'enviar senyals entre fils. Per exemple, per a avisar que una feina ja ha estat processada o bé que s'espera algun tipus més d'informació per a poder continuar treballant.

El primer bloc de rutines permet que dos fils puguin estar sincronitzats de manera genèrica. És a dir, emprant la primera rutina un fil es pot bloquejar fins a rebre un senyal d'un altre fil. Un altre fil, usant la segona funció podrà desbloquejar el fil en qüestió usant la segona rutina. La tercera pot ser usada per a enviar un senyal a tota la resta de fils de l'aplicació. Com es pot observar, una senyal no té cap tipus d'informació extra. Simplement, un fil pot saber que l'han "despertat" però no per què.

```
pthread_cond_init (condicio,atributs)
pthread_cond_destroy (condicio)
pthread_condattr_init (atributs)
pthread_condattr_destroy (atributs)

pthread_cond_wait (condicio,mutex)
pthread_cond_signal (condicio)
pthread_cond_broadcast (condicio)
```

Codi 2.7. Interfícies per a la creació de *mutex*

El segon bloc de rutines permet filar més prim. Quan s'envia un senyal a un fil es pot fer especificant un atribut associat a aquest senyal. En terminologia *pthread*, aquestes funcions permeten l'ús de variables de condició. L'ús d'aquests tipus de funcions permet que els fils es bloquegin esperant que succeeixi una condició concreta. En la part d'activitats es proposa una activitat relacionada amb els senyals per tal d'aprofundir en el seu funcionament.

Lectura recomanada

Es poden trobar molts exemples de com es poden emprar aquestes rutines a:

Kernel.org (2010). *Linux Programmer's Manual*. Recuperat el 3 de gener del 2012: http://www.kernel.org/doc/man-pages/online/pages/man7/pthreads.7.html

2.1.5. Altres funcionalitats

En aquest subapartat s'han presentat les interfícies més importants que els *pthreads* faciliten per a definir paral·lelisme i per a crear mecanismes de sincronització entre fils. No obstant això, els *pthreads* faciliten altres recursos que permeten fer accions més complexes. Per a estendre el coneixement d'aquest entorn és recomanable aprofundir en l'especificació dels *pthreads* o accedir a altres recursos com els que es presenten a continuació.

2.2. Model de programació Intel per a aplicacions paral·leles

Intel facilita un conjunt d'entorns de desenvolupament orientats a treure rendiment d'arquitectures multifil i arquitectures amb maquinari d'altes prestacions. Entre aquests entorns destaquen, per una banda, un conjunt d'aplicacions orientades a analitzar el rendiment de les aplicacions que es desenvolupen (com el ja esmentat *VTune*) i, per l'altra, un conjunt de models de programació orientats a desenvolupar aplicacions paral·leles per a aquest tipus d'entorns.

Aquest subapartat se centra a estudiar el segon dels dos punts: models de programació per a arquitectures multifil. Intel facilita un família extensa de compiladors, llenguatges de programació i models de programació orientats a treure rendiment de les arquitectures multifil que la mateixa empresa llança al mercat. Un exemple n'és el processador multinucli Sandy Bridge (Intel, Intel Sandy Bridge - Intel Software Network, 2012) del qual ja n'hem parlat.

Dins d'aquesta família de productes destaquen tres models de programació: Intel Cilk (Intel, Intel Cilk Plus, 2011); Intel Thread Building Blocks (Intel, Intel® Threading Building Blocks Tutorial, 2007), també anomenat Intel TBB, i Intel Array Building Blocks, també anomenat Intel ArBB (Intel, Intel® Array Building Blocks 1.0 Release Notes, 2011).

Ambdós models de programació estan orientats a desenvolupar aplicacions paral·leles, és a dir, faciliten mecanismes per a explotar el paral·lelisme que ofereixen les arquitectures. El primer, el Cilk, va ser dissenyat originàriament per l'Institut Tecnològic de Massachussets (MIT). Aquest ofereix una interfície més senzilla que els TBB. L'objectiu és facilitar un entorn més simple al desenvolupador perquè creï les aplicacions i, com es veurà més endavant, amb una semàntica similar al codi serial. A més a més, Cilk també facilita primitives per tal d'explotar el paral·lelisme amb relació a les dades que algunes arquitectures ofereixen, conegudes com a *unitats vectorials* o *single instruction multiple data*.

Els Intel TBB, de la mateixa manera que el Cilk, són també un model de programació orientat a desenvolupar aplicacions paral·leles. No obstant això, a diferència de l'anterior, faciliten una interfície molt més extensa i a la vegada més complexa. Faciliten accés a:

- Una biblioteca d'algorismes paral·lels, i també a estructures de dades per a aquests algorismes.
- Recursos que faciliten la gestió i planificació de tasques paral·leles.
- Recursos que faciliten la reserva i gestió de memòria de manera escalable. Tal com s'ha presentat en aquest apartat, aquest és un factor determinant en el rendiment d'aquestes arquitectures.
- Primitives orientades a la sincronització dels fils d'execució.

A diferència del Cilk, els TBB no faciliten recursos orientats a l'explotació d'arquitectures vectorials. No obstant això, les aplicacions que usen TBB també poden emprar ArBB.

L'ArBB, el darrer dels models esmentats, és un model de programació orientat a explotar el rendiment d'arquitectures vectorials. De manera similar al Cilk, facilita interfícies per a poder explotar el paral·lelisme amb relació a les dades dins les aplicacions. No obstant això, aquesta model és força més sofisticat que l'anterior.

A continuació, s'introduiran dos entorns orientats a explotar el paral·lelisme de les arquitectures multifil.

2.2.1. Intel Cilk

Com s'ha esmentat prèviament, el Cilk va ser dissenyat per tal d'oferir un model senzill de programació que permetés paral·lelitzar de manera senzilla aplicacions seqüencials, o bé implementar aplicacions paral·leles emprant una interfície senzilla. Durant aquest subapartat s'introduiran alguns dels conceptes més fonamentals d'aquest entorn.

La interfície que Cilk facilita per tal de definir paral·lelisme està formada per tres paraules reservades principals: *cilk_spawn, cilk_sync* i *cilk for*. A continuació se'n presentarà l'ús, i també es presentaran altres recursos que Cilk facilita per al desenvolupament d'aquest tipus d'aplicacions.

cilk_spawn i *cilk_sync*

Per tal d'explicar aquestes dues crides es presenta la paral·lelització d'un algorisme seqüencial extensament conegut: el còmput del nombre de Fibonacci. El codi 2.8 conté la implementació que considerarem per a aquest exemple. El còmput d'aquest nombre es fa de manera recursiva. És a dir, es crida la mateixa funció dues vegades: un cop per a $n - 1$ i l'altre a per $n - 2$. El còmput de Fibonacci de $n - 1$ i $n - 2$ es pot fer de manera paral·lela, ja que no tenen cap tipus de dependència.

```
#include <stdio.h>
#include <stdlib.h>

int fibonnaci(int n)
{
  if (n < 2) return n;

  int fiba = fibonnaci (n-1);
  int fibb = fibonnaci (n-2);
  return fiba + fibb;
}

int main(intargc, char *argv[])
{
  int n = atoi(argv[1]);
  int result = fib(n);
  printf("El Fibonacci de %d es %d.\n", n, resultat);
  return 0;
}
```

Codi 2.8. Implementació seqüencial del còmput del nombre de Fibonacci

Per tal de paral·lelitzar la funció *fibonnaci* s'empraran les crides *cilk_spawn* i *cilk_sync*. La primera de les dues funcions crea una execució paral·lela de la funció que se li passa per paràmetre. La segona crea una barrera de control, per la qual cosa l'execució del programa (el punter d'instrucció) no podrà continuar fins que tots els fils creats amb *cilk_spawn* hagin finalitzat.

```
int fibonnaci(int n)
{
  if (n < 2) return n;

  int fiba = cilk_spawn fibonnaci(n-1);
  int fibb = cilk_spawn fibonnaci(n-2);
  cilk_sync;
  return fiba + fibb;
}
```

Codi 2.9. Implementació paral·lela del nombre de Fibonacci

El codi 2.9 mostra les modificacions que cal fer sobre el codi presentat anteriorment per tal d'adaptar-lo a Cilk. Com es pot observar, només cal notificar que es vol executar de manera paral·lela l'execució de *fibonnaci* de $n - 1$ i de $n - 2$, i que, a més a més, l'execució del fil que està executant la funció *fibonnaci* actual no pot continuar fins que ambdues crides paral·leles no retornin.

Com es pot observar, l'ús d'ambdues interfícies és extremament senzill. Només cal modificar el codi seqüencial existent afegint l'*spawn* davant de les funcions que es volen executar amb un fil paral·lel i afegint la sincronització on sigui adequat.

Des del punt de vista de l'arquitectura del sistema, cada processador o nucli té una cua amb el conjunt de tasques que s'han creat amb la crida a *cilk_spawn*. Quan un processador o nucli es queda sense tasques per processar, agafa tasques de les cues d'altres processadors o nuclis que encara tenen feina pendent. Val a dir que quan la quantitat de paral·lelisme és elevada aquestes situacions rarament ocorren. No obstant això, cal mirar d'evitar-les, atès que robar tasques d'un altre nucli és costós.

cilk_for

El codi 2.10 mostra un exemple de com es podria paral·lelitzar un bucle *for* usant les crides introduïdes en el subapartat anterior. Per a cada iteració es crea una nova tasca amb la crida *cilk_spawn* i després del bucle l'execució del fil principal es bloqueja fins que tots els fils que s'han creat hagin acabat la seva execució.

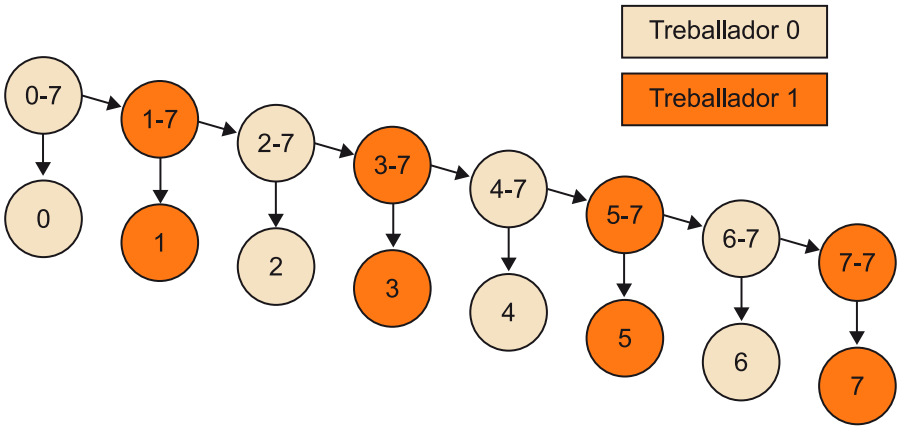
```
Implementació seqüencial:

for(int i = 0; i < T; i++)
{
  calcula(i);
}
Implementació paral·lela:

for(int i = 0; i < T; i++)
{
  cilk_spawn calcula(i);
}
cilk_sync;
```

Codi 2.10. Implementació paral·lela d'un bucle

La figura 15 mostra com s'executarien les diferents tasques que el codi anterior generaria. Per simplicitat, en aquest exemple s'assumeix una arquitectura en què només hi ha dos fils treballadors en execució i en què el temps d'execució de la funció *calcula* és relativament petit.

Com es pot observar el resultat de fer *spawn* a cada iteració acaba resultant en una implementació seqüencial molt similar a l'actual. En els casos en què el cos de la funció que cal executar fos més elevat, aquest efecte seria menys important.

Figura 15. Execució del bucle usant *cilk_spaw*



Per a aquestes situacions s'ha dissenyat la interfície *cilk_for*. Aquesta permet aplicar els algorismes de dividir i vèncer per tal d'executar de manera més eficient el bucle anterior.

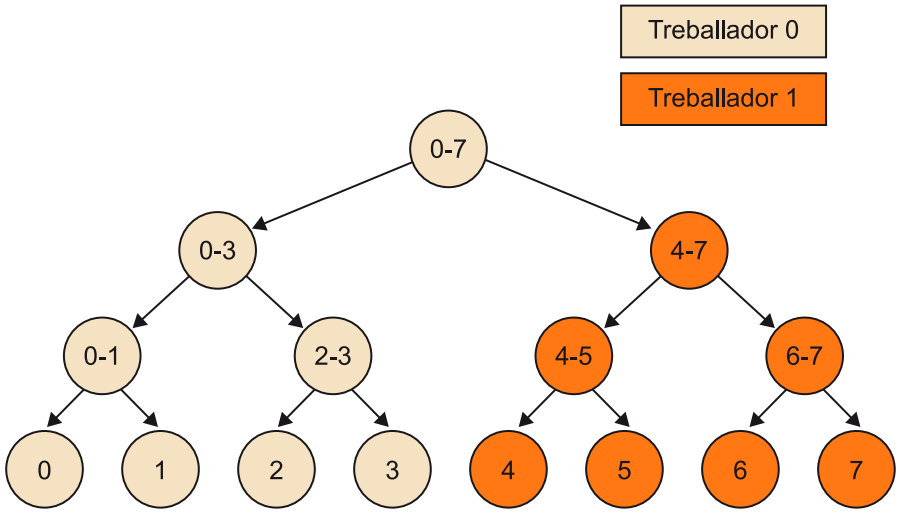
```
cick_for (int i = 0; i < T; i++)
{
  calcula(i);
}
```

Codi 2.11. Implementació paral·lela usant el *cilk_for*

El codi 2.11 mostra que el bucle anterior s'hauria de modificar per tal d'executar les iteracions en la forma ja esmentada de dividir i vèncer. Com es pot observar només cal emprar la directiva *cilk_for* en lloc del *for* tradicional. En aquest cas, el compilador ja processarà que es vol usar la versió paral·lela del *for* per a executar el que hi ha dins el cos de la iteració.

El resultat d'executar el cos del *cilk_for* es presenta en la figura 16. Com es pot observar el nivell de paral·lelisme augmenta de manera substancial respecte de la implementació amb el *cilk_spawn*.

Figura 16. Execució del bucle usant *cilk_for*



Dins d'un *cilk_for* es poden usar els anomenats *reducer hyperobjects*. Conceptualment, són força similars a la funció *gather* d'MPI (introduïda anteriorment). Una variable pot ser definida com a *reducer* sobre una operació associativa (per exemple: suma, multiplicació, concatenació, etc.). Cadascuna de les tasques generades modificarà aquesta variable sense considerar si altres tasques l'estan modificant o no.

No obstant això, d'aquesta variable hi haurà *n* vistes lògiques diferents, una per tasca executada. El *runtime* de Cilk serà l'encarregat de crear la vista final un cop hagin finalitzat totes les tasques. Quan una sola vista resta viva, el valor volgut es pot consultar de manera segura.

El codi 2.12 mostra un exemple d'ús d'un *reducer* sobre l'operació associativa de la suma. En aquest cas, la tasca creada amb l'*spawn* incrementarà la seva vista de la variable i de manera concurrent el fil principal també ho farà. Un cop acabi la tasca creada i el fil principal en demani el valor mitjançant el *get_value*, el *runtime* de Cilk en retornarà la suma.

```
#include <cilk/cilk.h>
#include <cilk/reducer_opadd.h>

cilk::reducer_opadd<int> suma;

void mesun()
{ sum += 1; }

int main()
{
  sum += 1;
  cilk_spawn mesun();
  sum += 1;
  cilk_sync;
  return suma.get_value();
}
```

Codi 2.12. Exemple d'ús d'un *reducer*

Restriccions del *cilk_for* i del *cilk_spawn*

Com hem vist, les interfícies de Cilk són senzilles d'emprar. Ara bé, la seva simplicitat implica en molts casos certes limitacions. Cal tenir en compte que:

- Les diferents iteracions d'un *cilk_for* han de ser independents entre elles.

- El codi de la tasca resultant del *cilk_spawn* ha de ser independent del codi que hi ha entre la crida al mateix *cilk_spawn* i el *cilk_sync*. Per exemple, el codi presentat en el codi 2.13 pot desencadenar en una carrera d'accés en la modificació de la variable *x*.

- En situacions en què es vulguin modificar concurrentment variables de memòria caldrà emprar *reducer*s o variables de sincronització per tal d'evitar condicions de carrera.

```
void incrementa(int& i)
{
  ++i;
}

int main ()
{
  int x = 0;
  cilk_spawn incrementa(x);
  int y = x -1;
  return y;
}
```

Codi 2.13. Implementació amb una carrera d'accés potencial

Cilk facilita l'entorn d'avaluació d'aplicacions paral·leles desenvolupades amb aquest model de programació anomenat *Cilkscreen*. Aquest permet avaluar el rendiment de les aplicacions desenvolupades. Una de les utilitats més interessants que permet és l'estudi d'escalabilitat. D'aquesta manera, facilita un conjunt de funcionalitats que ajuden a entendre l'escalabilitat de l'aplicació que s'està executant.

D'altra banda, també facilita un *race detector*. Aquesta funcionalitat permet avaluar si l'aplicació desenvolupada té algun tipus de condició de carrera. Tot i no explorar totes les combinacions possibles, és un bon inici per a detectar errors de programació que poden causar carreres d'accés a dades.

2.2.2. Intel Thread Building Blocks

Aquest entorn de desenvolupament és força més complex i extens que el presentat anteriorment. De manera similar a l'anterior facilita un conjunt d'interfícies que permeten declarar tasques paral·leles. No obstant això, a més a més, afegeix funcionalitats força més potents per tal de millorar el rendiment de les aplicacions paral·leles: definició de grups de tasques, primitives de sincronització entre fils, etc.

Una de les complexitats afegides dels Intel TBB respecte de Cilk és que estan basats en la biblioteca estàndard de plantilles, més coneguda com a *standard template library* (STL). Aquí no farem una introducció a la STL. No obstant això, se'n poden trobar moltes introduccions, per exemple (Hewlett-Packard, 1994).

Els Intel TBB es troben dividits en sis components diferents:

1) El component d'algorismes paral·lels genèrics. Proporciona interfícies similars al *cilk_for* i *cilk_spawn* estudiades en el subapartat anterior.

2) El component de gestió de tasques o *task scheduler*. Proporciona accés a recursos que permeten la gestió de les tasques que el component anterior permet crear.

3) El component de primitives de sincronització. Proporciona accés a un conjunt de primitives que poden ser emprades per a sincronitzar diferents fils d'execució (per exemple: per a accedir a variables compartides, punts de sincronització, etc.). Són força similars a les que faciliten els POSIX Threads.

4) El component de fils o *threads*. Proporciona l'abstracció del concepte de fil dins d'aquest entorn. Com es veurà a continuació, aquesta es facilita mitjançant el tipus *tbb_threads*.

5) El component de contenidors paral·lels. Proporciona accés a un conjunt d'estructures de dades que han estat dissenyades per tal d'accedir-hi de manera paral·lela.

6) El component per a la gestió de memòria. Proporciona accés a un conjunt d'interfícies orientades a fer una gestió eficient i escalable de la memòria que l'aplicació usa. Tal com s'ha remarcat anteriorment, aquest és un dels factors més crítics a l'hora de treure'n un bon rendiment.

A continuació es fa una introducció als components més importants i més diferents dels Intel TBB. Alguns, com ara les primitives de sincronització, no es descriuran atès que són força similars a altres sistemes (com ara els POSIX Threads).

Algorismes paral·lels

Els TBB faciliten accés a interfícies que permeten la definició de paral·lelisme dins de les aplicacions. Aquestes interfícies es poden dividir en dos grups:

- Un bloc d'algorismes més senzills: *parallel_for*, *parallel_reduce*, *parallel_scan*.

- Un bloc d'algorismes més avançats que permet definir paral·lelisme de manera més complexa: *parallel_while*, *parallel_do*, *parallel_pipeline*, *parallel_sort*.

Ara es presenta l'ús dels algorismes *parallel_for* i *parallel_reduce*. Val a dir que la metodologia per a emprar els algorismes més avançats és la mateixa però amb una semàntica més complexa. Per tal d'aprofundir més en l'ús de les diferents interfícies *parallel* es pot accedir a (Intel, 2007).

Lectura recomanada

Intel (2007). *Intel® Threading Building Blocks Tutorial*. Intel.

1) L'ús del *parallel_for*

De manera similar al *cilk_for* les crides *parallel* permeten la definició de zones d'execució paral·lela. La més senzilla de totes és la *parallel_for*. Aquesta es comporta de manera molt similar a la ja introduïda *cilk_for*.

```
void ImplementacioSequancial( float a[], size_t n )
{
  for( size_t i=0; i<n; ++i )
  Foo(a[i]);
}
```

Codi 2.14. Implementació seqüencial

El codi 2.14 mostra un codi seqüencial que vol ser modificat per tal de ser paral·lel. En el cas de Cilk només hauria estat necessari afegir-hi la paraula reservada *cilk_for* en lloc del tradicional *for*.

De manera similar a Cilk, la plantilla *tbb::parallel_for* divideix l'espai de les diferents iteracions, en l'exemple de 0 a n – 1, en k blocs diferents i executa cadascun d'ells en un fil diferent. Cada fil executarà el que en terminologia TBB s'anomena un *functor.* Aquest és una classe que implementa el codi que s'aplicarà sobre un bloc en qüestió. En el cas anterior, caldrà dissenyar un *functor* que recorri el rang del vector que li toca processar de manera que per cada posició cridi la funció *Foo.*

El codi 2.15 mostra la redefinició del bucle anterior en un nou *functor*. Com es pot observar s'ha creat una classe *ExecutaFoo* que defineix l'operador d'execució "*( )*" al qual es facilita un objecte de tipus bloc. Aquesta funció executarà el bucle anterior dins el rang que el *runtime* dels TBB li faciliti. Com es pot observar, al constructor de la classe se li facilita el punter al vector *a*, sobre el qual s'aplicarà la funció *Fool.*

La classe *ExecutaFoo* defineix com es processarà el bloc del vector que el sistema o *runtime* li ha assignat. Cal remarcar que aquesta codi tan sols executa el processament d'un bloc i no de tot el vector sencer. Un cop s'ha definit l'execució paral·lela d'aquest dins del *functor* ja es pot crear el codi que l'usarà.

```
#include "tbb/blocked_range.h"

class ExecutaFoo {
  float *const my_a;

public:
  void operator()( const blocked_range<size_t>& r ) const
  {
    float *a = my_a;
  for( size_t i=r.begin(); i!=r.end(); ++i )
    Foo(a[i]);
  }

ExecutaFoo( float a[] ) :
    my_a(a)
  {};
}
```

Codi 2.15. Paral·lelització del recorregut del vector

El codi 2.16 mostra com es faria la crida a la classe *ExecutaFoo* usant la interfície del *parallel_for*. Aquesta funció rep com a paràmetre la definició del rang i una instància de la classe *ExecutaFoo*. El *runtime* dels TBB crearà *n/Granularitat* fils i a cadascun li proporcionarà accés als diferents blocs de dades que li toqui computar.

```
#include "tbb/parallel_for.h"

void ExecuccioParallelaFoo( float a[], size_t n ) {
  parallel_for(blocked_range<size_t>(0,n,Granularitat),
    ExecutaFoo(a) );
}
```

Codi 2.16. Crida a *ExecutaFoo* amb el *parallel_for*

La definició de la *granularitat* cal fer-la en funció de la mida del rang global que es vulgui processar. En general, es recomana que el nombre d'iteracions que executarà cada bloc es trobi entre 10.000 i 100.000. Per a valors inferiors o superiors el rendiment de l'aplicació pot ser pitjor. Per una banda, una granularitat massa gruixuda pot implicar poc nivell de paral·lelisme. Per l'altra, una granularitat massa elevada pot implicar massa sobrecost a l'hora de gestionar tants fils d'execució.

Una alternativa és emprar l'*auto_partitioner*. En cas que al *parallel_for* no se li faciliti la granularitat, el *runtime* usarà un conjunt d'heurístiques pròpies per tal de calcular la mida de bloc que donarà més rendiment i més balanceig dins l'aplicació. És recomanable usar-lo per a comprovar el rendiment que dóna. No obstant això, el més adient és fer un estudi variant el nivell de granularitat per tal d'analitzar el que doni el rendiment més gran.

2) L'ús del *parallel_reduce*

De manera similar als *reducers* de Cilk, els TBB faciliten el *parallel_reduce*. Els diferents fils treballen de manera concurrent amb diferents vistes d'una variable que el *runtime* unificarà en una sola vista al final de l'execució paral·lela.

El codi presentat en el codi 2.17 suma *n* vegades el resultat d'executar la funció *Foo* a cadascuna de les posicions del vector *a*. Assumint que cadascuna de les iteracions són independents, el codi es podria paral·lelitzar seguint una metodologia similar a l'emprada en el cas anterior. No obstant això, en aquest cas s'usarà el *parallel_reduce* per tal de fer la reducció de les diferents sumes.

```
float SumaFoo( float a[], size_t n ) {
  float suma = 0;
  for( size_t i =0; i!=n; ++i )
    suma += Foo(a[i]);
  return suma;
}
```

Codi 2.17. Crida a *ExecutaFoo* amb el *parallel_for*

El codi 2.18 mostra com caldria modificar el codi anterior per tal que el *parallel_reduce* el pugui utilitzar. De manera similar al *parallel_for,* caldrà definir un nou *functor* per a aquesta interfície. A diferència de la classe *ExecutaFoo* feta en el cas anterior, es definirà la classe *SumaFoo* amb dos nous mètodes importants:

- El primer és un nou constructor que rep com a paràmetre una referència a una altre objecte de tipus *SumaFoo* i un objecte *split*. El segon paràmetre no es fa servir per res, no obstant això cal definir-lo per tal que el compilador distingeixi aquest constructor del constructor per còpia de la mateixa classe.

- El segon és un mètode necessari per tal que el *map_reduce* funcioni. Com es veurà a continuació, aquest es fa servir quan es crea una nova subtasca que ajuda a processar la feina que la tasca actual està duent a terme.

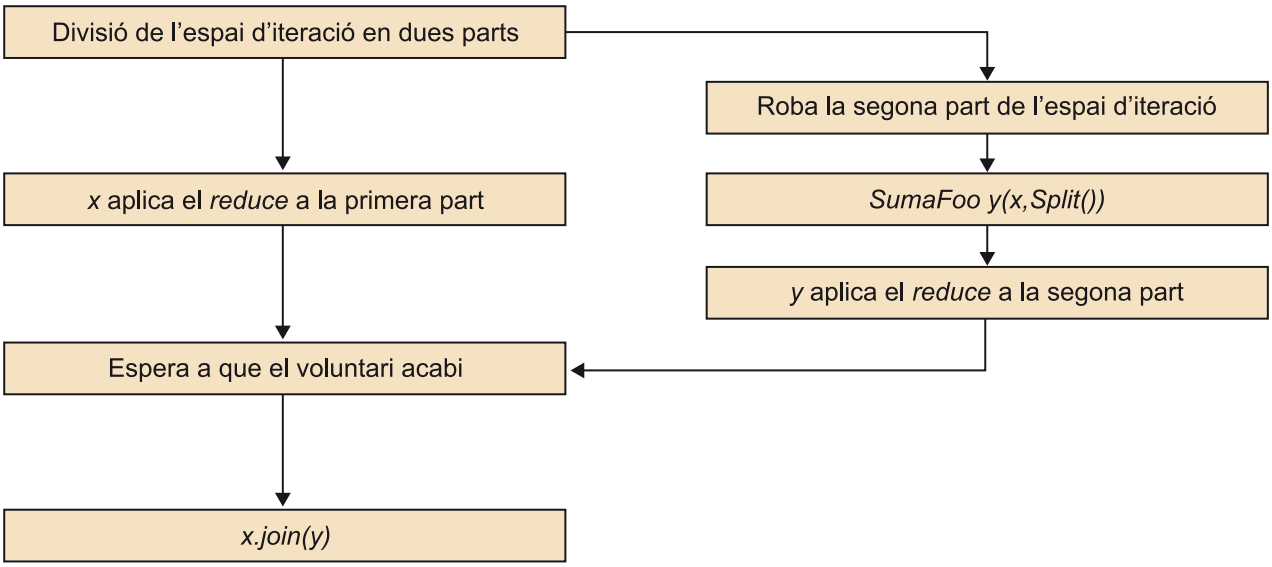
Quan un nou fil treballador està disponible, el gestor de tasques pot crear una nova subtasca. Aquesta farà el còmput d'una part del rang que la tasca actual està processant. El *runtime* dels TBB serà el responsable de decidir quina part es delega. La creació d'aquesta subtasca es farà emprant aquest nou constructor. La tasca principal s'esperarà que la nova tasca acabi i mitjançant el mètode *join* en farà la suma global.

```
class SumaFoo {
  float* my_a;
public:
  float suma;
  void operator()( const blocked_range<size_t>& r ) {
    float *a = my_a;
    for( size_t i=r.begin(); i!=r.end(); ++i )
    suma += Foo(a[i]);
  }

  SumFoo( SumFoo& x, split ) : my_a(x.my_a), suma(0) {}
  void join( const SumFoo& y ) {suma+=y.suma;}

  SumFoo(float a[] ) :
  my_a(a), suma(0)
  {}
};
```

Codi 2.18. Paral·lelització de la suma

La figura 17 descriu la seqüència d'esdeveniments temporals que succeiran dins el *runtime* dels TBB quan el gestor de tasques decideixi crear una subtasca i dividir la feina que l'actual està processant en dues parts.

Figura 17. Graf de la seqüència de *split* i *join*



De la mateixa manera que en l'exemple del *parallel_for*, un cop s'ha definit la classe que implementarà la feina que cal dur a terme a partir d'un rang determinat, cal implementar la crida al *parallel_reduce*. Aquesta es presenta en el codi 2.19.

```
#include "tbb/parallel_reduce.h"

float SumaParallelaFoo( const float a[], size_t n ) {
  SumaFoo sf(a);
  parallel_reduce(blocked_range<size_t>(0,n,Granularitat), sf );
  return sf.suma;
}
```

Codi 2.19. Crida a *SumaFoo* amb el *parallel_reduce*

Tal com s'ha pogut observar, els *functors* són la base dels TBB. La resta d'interfícies paral·leles d'aquest entorn de programació es troben també basades en aquests tipus d'objectes.

Contenidors de dades

Molts dels codis desenvolupats en C++ usen estructures de la STL per a desar dades i compartir-les entre les diferents parts de l'aplicació. Exemples d'aquestes estructures són els *maps*, *vectors* o *dqueue*. Són extremament potents atès que estan definides amb plantilles genèriques i tenen una eficiència molt elevada.

Un dels problemes d'aquestes estructures és que no acostumen a donar suport per a l'accés concurrent. Per tant, en moltes situacions, si dos fils intenten accedir de manera concurrent a una estructura (per exemple: un *map<int,int>*) amb alta probabilitat aquesta es corromprà. Aquest problema es pot solucionar usant mecanismes d'exclusió mútua que protegeixin l'accés simultani a aquestes estructures. No obstant això, aquesta solució redueix la concurrència de l'aplicació, i per tant el seu rendiment.

Els Intel TBB faciliten un conjunt de contenidors que segueixen la mateixa filosofia que els de la STL però als quals a més a més es pot accedir concurrentment. Aquests contenidors permeten dos tipus d'accessos diferents:

- Accessos amb exclusió mútua de gra fi. Aquest tipus d'accés es farà quan diferents fils vulguin accedir de manera simultània a zones del contenidor que volen modificar. Els fils podran demanar accés exclusiu a aquestes zones del contenidor. A la resta de zones del contenidor es podrà accedir sense cap tipus de restricció.

- Accessos sense cap tipus d'exclusió. En els casos en què per la construcció de l'aplicació el codi pugui assegurar que no hi haurà conflictes al contenidor, els fils hi podran accedir de manera concurrent sense haver de fer cap tipus d'acció.

Els TBB defineixen quatre tipus diferents de contenidors: dos de tipus cua (*concurrent_queue* i *concurrent_bounded_queue*); un contenidor de tipus vector (*concurrent_vector*), i un contenidor de tipus *hash* (*concurrent_hash_map*). A tall d'exemple, a continuació es presentarà l'ús de les cues i dels vectors.

1) L'ús de les cues

Les cues tenen la mateixa semàntica i el mateix funcionament que les de la STL. És a dir, proporcionen un mètode per a posar una dada a la cua, un mètode per a agafar el primer mètode de la cua si n'hi ha i un altre mètode per a consultar-ne la mida. La diferència principal rau en la consulta del primer element de la cua.

Tal com es pot observar en el codi 2.20, la STL facilita dos mètodes diferents. El primer retorna el primer valor de la cua i el segon permet eliminar-lo. En el cas dels TBB això no es pot fer d'aquesta manera, atès que entre les dues crides un altre fil podria haver modificat la cua.

Per aquest motiu, facilita el mètode *try_pop*. Aquest mètode retorna cert si la cua tenia alguna valor fals. En cas que la cua no estigui buida, modificarà la variable passada per referència amb el valor del darrer element de la cua i l'eliminarà d'aquesta.

```
STL
void pop();
T& front();

TBB concurrent_queue
  bool try_pop(T& x);
  void push(T& x);

TBB concurrent_bounded_queue
  void pop(T& x);
  void push(T&x) ;
  bool try_push(T&x);
```

Codi 2.20. Consulta d'un element de la cua

Els TBB faciliten la variant *concurrent_bounded_queue*. Aquesta afegeix operacions bloquejants i la possibilitat de configurar un límit d'espai. En aquestes cues la funció de *push* es bloqueja si no hi ha espai a la cua i la funció *pop* es bloqueja fins que hi hagi algun element a la cua. D'altra banda, faciliten la funció *try_push,* que es comporta similar a la funció *push* però no és bloquejant. En cas que no es pugui posar l'element a la cua retornarà fals.

2) L'ús de *hash_maps*

Un dels contenidors més interessants de la STL són els *maps*. Aquests permeten fer insercions i cerques associatives. El codi 2.21 mostra un exemple de codi de la STL, en què s'usa una estructura *map* per a desar la correspondència entre DNI i noms. Com es pot observar s'usen *iteradors* per a cercar dins l'estructura i s'hi afegeixen / se n'eliminen elements usant el mètodes *insert/delete*.

```
#include <iostream>
#include <map>
using namespace std;

int main ()
{
  map<uint32,string>::iterator iterador;
  map<uint32,string> map_dni_nom;

  map_dni_nom.insert(pair<uint32,string>(43433243,"Jordi"));

  iterador = map_dni_nom.find(43433243);

  if(iterador != map_dni_nom.end())
  {
    cout << " El dni és " << iterador->second << endl;
    map_dni_nom.erase(iterador);
}
  else
    cout << " Error hi hauria de ser " << endl;
}
```

Codi 2.21. Exemple d'ús d'un *map* de la STL

Com s'ha esmentat anteriorment, aquestes estructures no donen suport a l'accés concurrent. Per aquest motiu, els TBB implementen els *concurrent_hash_map*. Aquests *maps* permeten l'accés concurrent al seu contingut mitjançant dues classes:

- La classe *accessor* que permet agafar el dret en exclusiva de modificar una entrada determinada.
- La classe *const_accessor* que permet agafar el dret a lectura compartida d'una entrada.

A diferència dels *maps* de la STL, quan es vol cercar o es vol inserir un element dins d'aquesta estructura cal usar les classes anteriors. El codi 2.22 mostra dos exemples de com s'haurien d'usar aquestes dues classes a l'hora d'accedir a un *map* i modificar-lo.

```
Inserció d'un map

  StringTable accessor a;
  for( string* p=range.begin(); p!=range.end(); ++p ) {
    table.insert( a, *p );
    a->second += 1;
    a.release();
  }

Lectura d'un map
  StringTable const_accessor a;
  for( string* p=range.begin(); p!=range.end(); ++p ) {
    table.find ( a, *p );
    a.release();
  }
```

Codi 2.22. Inserció i modificació d'un *map*

Gestió de memòria

Ja hem parlat de la importància d'una bona gestió de la memòria que l'aplicació utilitza. S'han presentat situacions, com, per exemple, la falsa compartició, en què accessos a determinades línies en poden reduir dràsticament el rendiment.

La biblioteca STL proporciona la *std::allocator* que és el responsable de reservar i gestionar la memòria que les aplicacions demanen. Els Intel TBB faciliten dos tipus nous d'*allocators* orientats a mirar d'adreçar les situacions següents:

a) Problemes d'escalabilitat. Originalment els *allocators* van ser dissenyats per a ser emprats en aplicacions seqüencials. En situacions amb diferents fils d'execució, es poden donar situacions en què aquests estan competint per obtenir recursos als quals només pot accedir un fil en un mateix instant de temps. Evidentment això implica problemes d'escalabilitat greus. Per evitar aquesta situació podem emprar l'*scalable_allocator<T>*.

b) Falsa compartició. Aquest problema, ja discutit anteriormen, succeeix quan dues variables diferents a les quals accedeixen fils diferents es troben mapades en les mateixes línies de la memòria cau. Els Intel TBB proporcionen el *cache_aligned_allocator<T>*. Aquest assegura que dos objectes instanciats que usen aquest gestor no es guardaran en la mateixa línia de memòria. El que els TBB no asseguren és que dos objectes, l'un creat amb l'*allocator* per defecte i l'altre creat amb el *cache_aligned_allocator,* no puguin experimentar falsa compartició.

El codi 2.23 mostra un exemple de com es podrien usar els dos *allocators* que acabem d'introduir. El primer instancia un objecte de la classe tipus *Classe* usant l'*scalable_allocator*. El segon declara un vector de la STL i especifica que vol que la memòria que aquest utilitzi es reservi usant el *cache_aligned_allocator.*

```
Clase * s = scalable_allocator<Classe>().allocate( sizeof(Classe) );

std::vector<int,cache_aligned_allocator<int> >;
```

Codi 2.23. Exemple d'ús dels dos *allocators* dels TBB

3. Factors determinants en el rendiment en arquitectures modernes

En el darrer apartat s'han presentat diferents arquitectures de processadors que implementen més d'un fil d'execució, com també les arquitectures vectorials. Així doncs, hi ha arquitectures que faciliten dos fils d'execució, com les *shared multithreading*, i n'hi ha que donen accés a desenes de fils d'execució, com les arquitectures multinucli.

En general, una arquitectura és més complexa com més fils facilita. Les arquitectures multinucli són força escalables i poden donar accés a un nombre elevat de fils. Ara bé, com ja s'ha vist, per tal de dur-ho a terme cal dissenyar sistemes que són més complexos. D'altra banda, la complexitat d'aquesta també es veu incrementada si la formen components MIMD i SIMD. En aquest cas, el model de programació ha de tenir mecanismes per a poder explotar, per exemple, el joc d'instruccions vectorials.

Com més complexa és l'arquitectura, més factors cal tenir en compte a l'hora de programar-la. Si tenim una aplicació que té una zona paral·lela que en un processador seqüencial es pot executar en temps x, és d'esperar que en una màquina multifil amb m fils disponibles, aquesta aplicació potencialment ho faci en un temps de x/m. No obstant això, hi ha certs factors inherents al tipus d'arquitectura sobre la qual s'executa i al model de programació emprat que poden limitar aquest increment de rendiment; per exemple, l'accés a les dades compartides per fils que s'estan executant en diferents nuclis.

Per tal d'obtenir el màxim rendiment de les arquitectures sobre les quals s'executen les aplicacions paral·leles, s'han de considerar les característiques d'aquestes arquitectures. Per tant, cal considerar la jerarquia de memòria del sistema, el tipus d'interconnexió sobre el qual s'envien dades, l'amplada de banda de memòria, etc. És a dir, si es vol extreure el màxim rendiment cal redissenyar o adaptar els algorismes a les característiques del maquinari que hi ha per sota.

Si bé és cert que cal adaptar les aplicacions segons les arquitectures en què es volen executar, també és cert que hi ha utilitats que permeten no haver de tenir en compte algunes de les complexitats d'aquestes arquitectures. La majoria apareixen en forma de models de programació i biblioteques que poden ser emprades per les aplicacions.

Aquest apartat presenta els factors més importants que poden limitar l'accés al paral·lelisme que dóna una arquitectura lligats al model de programació.

3.1. Factors importants per a la llei d'Amdahal en arquitectures multifil

La llei d'Amdahal estableix que una aplicació dividida en una part inherentment seqüencial (és a dir, només pot ser executada per un fil) i una part paral·lela P, potencialment pot tenir una millora de rendiment de S (en anglès *speedup*) augmentant el paral·lelisme de l'aplicació a *P*.

$$T' = T * \frac{1}{\left(1 - P + \frac{P}{S}\right)}$$

Ara bé, per a arribar al màxim teòric cal considerar les restriccions inherents al model de programació i restriccions inherents a l'arquitectura sobre la qual s'està executant l'aplicació.

El primer conjunt de restriccions fa referència als límits vinculats a l'algorisme paral·lel considerat, com també a les tècniques de programació emprades. Un cas en què apareixen aquestes restriccions és quan s'ordena un vector, ja que hi ha limitacions causades per l'eficiència de l'algorisme (per exemple, Radix Sort) i per a implementar-lo (per exemple, la manera d'accedir a les variables compartides, etc.).

El segon conjunt fa referència a límits lligats a les característiques del processador sobre el qual s'està executant l'aplicació. Factors com ara la jerarquia de memòria cau, el tipus de memòria cau o el tipus de xarxa d'interconnexió dels nuclis poden limitar aquest rendiment.

Exemple

Pot causar moltes ineficiències que una variable sigui compartida entre dos fils que no són dins el mateix nucli.

Els propers dos subapartats presenten alguns dels factors més importants d'aquests dos blocs que acabem d'esmentar. Del primer conjunt no s'estudien algorismes paral·lels (Gibbons i Rytte, 1988), ja que no és l'objectiu de la unitat, sinó els mecanismes que fan servir aquests algorismes per tal d'implementar tasques paral·leles i tots els factors que cal considerar. Del segon bloc es discuteixen les característiques més rellevants de l'arquitectura que cal considerar en el desenvolupament d'aquest tipus d'aplicacions.

És important remarcar que els factors que s'estudien a continuació són una part dels molts que s'han identificat durant les darreres dècades. A causa de la importància d'aquest àmbit, s'ha fet molta recerca centrada a millorar el rendiment d'aquestes arquitectures i el disseny de les aplicacions que s'hi executen (com les aplicacions de càlcul numèric o les de càlcul del genoma humà).

Lectures recomanades

Per tal d'aprofundir més en les problemàtiques i estudis fets tant en l'àmbit acadèmic com empresarial, és molt recomanable estendre la lectura d'aquesta unitat didàctica amb les referències bibliogràfiques facilitades.

3.2. Factors vinculats al model de programació

3.2.1. Definició i creació de les tasques paral·leles

En el disseny d'algorismes paral·lels es consideren dos tipus de paral·lelisme: a nivell de dades i a nivell de funció. El primer defineix quines parts de l'algorisme s'executen de manera concurrent, i el segon, com es processen les dades de manera paral·lela.

1) En la creació del **paral·lelisme a nivell de funció** és important considerar que les tasques que treballin amb les mateixes funcions i dades tinguin localitat al nucli en què s'executaran. D'aquesta manera els fluxos del mateix nucli comparteixen les entrades corresponents a la memòria cau de dades, com també les seves instruccions.

2) El **paral·lelisme a nivell de dades** també ha de considerar la localitat esmentada, però a més a més ha de tenir en compte la mida de les memòries cau de què disposa. És a dir, el flux d'instruccions ha de treballar amb les dades de manera local a la L1 tant com sigui possible i la resta intentar mantenir-la a nivells de memòria cau tan propers com sigui possible (L2 o L3). D'altra banda, també cal evitar efectes ping-pong entre els diferents nuclis del processador. Això pot succeir quan fils de diferents nuclis accedeixin als mateixos blocs d'adreces de manera paral·lela.

La figura següent mostra un exemple d'escenari que s'ha d'evitar en la creació de fils, tant en l'assignació de tasques com en les dades. En aquest escenari, els dos fils número 0 de tots dos nuclis estan executant la mateixa funció $F(x)$. Aquesta funció conté un bucle intern que fa alguns càlculs sobre el vector k i el resultat l'afegeix a la variable global k. Durant l'execució d'aquests fils s'observa el següent:

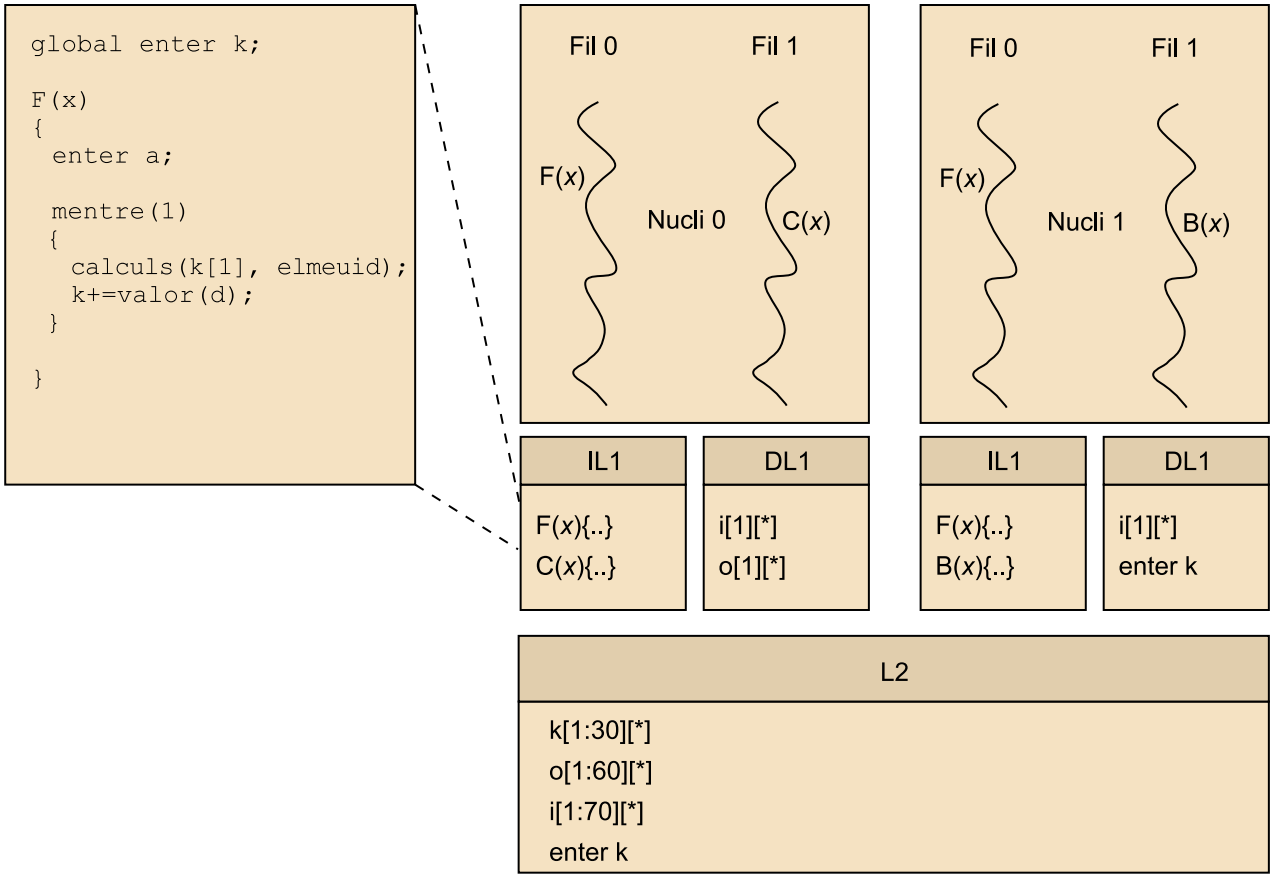
- Els dos nuclis cada vegada que vulguin accedir a la variable global k han d'invalidar la L1 de dades de l'altre nucli. Com ja s'ha vist, depenent del protocol de coherència, pot ser extremadament costós. Per tant, aquest aspecte pot fer baixar el rendiment de l'aplicació.

- Els dos fils accedeixen potencialment a les dades del vector l. Així doncs, les L1 de dades de tots dos nuclis tenen emmagatzemades les mateixes dades (assumint que la línia es troba en estat compartit). En aquest cas seria més eficient que els dos fils s'executessin en el mateix nucli per tal de compartir les dades de la L1 de dades (és a dir, més localitat). Això permetria usar més eficientment l'espai total del processador.

- De manera similar, la L1 d'instruccions de tots dos casos nuclis tenen una còpia de les instruccions del mateix codi que executen els dos fils (F). Semblantment al punt anterior, aquest aspecte provoca una utilització inefici-

ent dels recursos, ja que les mateixes dades es troben replicades en totes dues memòries cau. Cal remarcar que, en aquest cas, els dos nuclis no invaliden les línies compartides, ja que en generar les entrades de memòria d'instruccions, es troben en estat compartit i aquestes no s'acostumen a modificar.

Figura 18. Definició i creació de fils



L'exemple anterior mostra una situació força senzilla de solucionar. Ara bé, la definició, la creació de tasques i l'assignació de dades no és una tasca fàcil. Com ja s'ha esmentat, cal considerar la jerarquia de memòria, la manera en què els processos s'assignen als nuclis, el tipus de coherència que el processador facilita, etc.

Tot i la complexitat d'aquesta tasca hi ha molts recursos que ajuden a definir aquest paral·lelisme, com els següents:

- Aplicacions que permeten la paral·lelització automàtica d'aplicacions seqüencials. Molts compiladors inclouen opcions per tal de generar codi paral·lel automàticament. Ara bé, és fàcil trobar-se en situacions en què el codi generat no és òptim o no té en compte alguns dels aspectes introduïts.

- Aplicacions que donen assistència en la paral·lelització dels codis seqüencials. Per exemple, ParaWise (ParaWise, 2011) és un entorn que guia

Lectures recomanades

Per tal d'aprofundir en aquest àmbit és recomanable llegir:

X. Martorell; J. Corbalán; M. González; J. Labarta; N. Navarro; E. Ayguadé (1999). "Thread Fork/Join Techniques for Multi-level Parallelism Exploitation in NUMA Multiprocessors". *13th International Conference on Supercomputing.*

B. Chapman; L. Huang; E. Biscondi; E. Stotzer; A. G. Shrivastava (2008). "Implementing OpenMP on a High Performance Embedded Multicore MPSoC". *IPDPS.*

l'usuari en la paral·lelització de codi Fortran. En qualsevol cas el resultat pot ser similar a la paral·lelització automàtica.

- Finalment, també hi ha biblioteques que proporcionen interfícies per a implementar algorismes paral·lels. Aquestes interfícies acostumen a ser la solució més eficaç per a treure el màxim rendiment de les aplicacions: faciliten l'accés a funcionalitats que permeten definir el paral·lelisme a nivell de dades i funcions, afinitats de fils a nuclis, afinitats de dades a la jerarquia de memòria, etc. Algunes d'aquestes biblioteques són OpenMP, Cilck (Intel, Intel Cilk Plus, 2011), TBB (Reinders, 2007), etc.

3.2.2. Mapatge de tasques a fils

Una tasca és una unitat de concurrència d'una aplicació, és a dir, inclou un conjunt d'instruccions que poden ser executades de manera concurrent (paral·lela o no) a les instruccions d'altres tasques. Un fil és una abstracció del sistema operatiu que permet executar paral·lelament diferents fluxos d'execució. Una aplicació pot estar formada des d'un nombre relativament petit de tasques fins a milers.

Tasques

Exemples clars són els servidors de videojocs o els servidors de pàgines web: poden tenir milers de tasques atenent les peticions dels usuaris.

No obstant això, el sistema operatiu no pot donar accés a un nombre tan elevat de fils d'execució per la limitació dels recursos. D'una banda, la gestió d'un nombre tan elevat és altament costós (molts canvis de contextos, gestió de moltes interrupcions, etc.). D'altra banda, tot i que potencialment el sistema operatiu pugui donar accés a un miler de fils, el processador sobre el qual s'executen les aplicacions dóna accés a pocs fils físics[2]. Per tant, cal anar fent canvis de context entre tots els fils disponibles a nivell de sistema i els fils que el maquinari faciliti. Aquest és el motiu pel qual el nombre de fils del sistema ha de ser configurat coherentment amb el nombre de fils del processador.

(2)En anglès anomenats *hardware threads*.

L'aplicació, per tal de treure el màxim rendiment del processador, ha de definir un mapatge eficient i adequat de les tasques que vol executar als diferents fils de què disposa. Alguns dels algorismes de mapatge de tasques a fils més emprats són els següents:

- Patró *master/slave*: un fil s'encarrega de distribuir les tasques que resten per executar als fils esclaus que no estiguin executant res. El nombre de fils esclaus és variable.

- Patró *pipeline* o cadena: en què cadascun dels fils fa una operació específica en les dades que s'estan processant, alhora que facilita el resultat al fil següent de la cadena.

- Patró *task pool*: en què hi ha una cua de tasques pendents a ser executades i cadascun dels fils disponibles agafa una d'aquestes tasques pendents quan acaba de processar l'actual.

Aquest mapatge es pot fer basant-se en molts criteris diferents. No obstant això, els aspectes introduïts al llarg d'aquesta unitat didàctica s'haurien de considerar en arquitectures multifil heterogènies. Depenent de com s'assignin les tasques als fils i com estiguin assignats els fils als nuclis, el rendiment de les aplicacions varia molt.

Lectura recomanada

Un treball de recerca molt interessant sobre les tècniques de mapatge és el presentat per Philbin i altres. Els autors presenten tècniques de mapatge i gestió de fils per tal de mantenir la localitat a les diferents memòries cau:

J. Philbin; J. Edler; O. J. Anshus; C. Douglas; K. Li (1996). "Thread scheduling for cache locality". *Seventh international conference on Architectural support for programming languages and operating systems*.

Exemple de mapatge de tasques

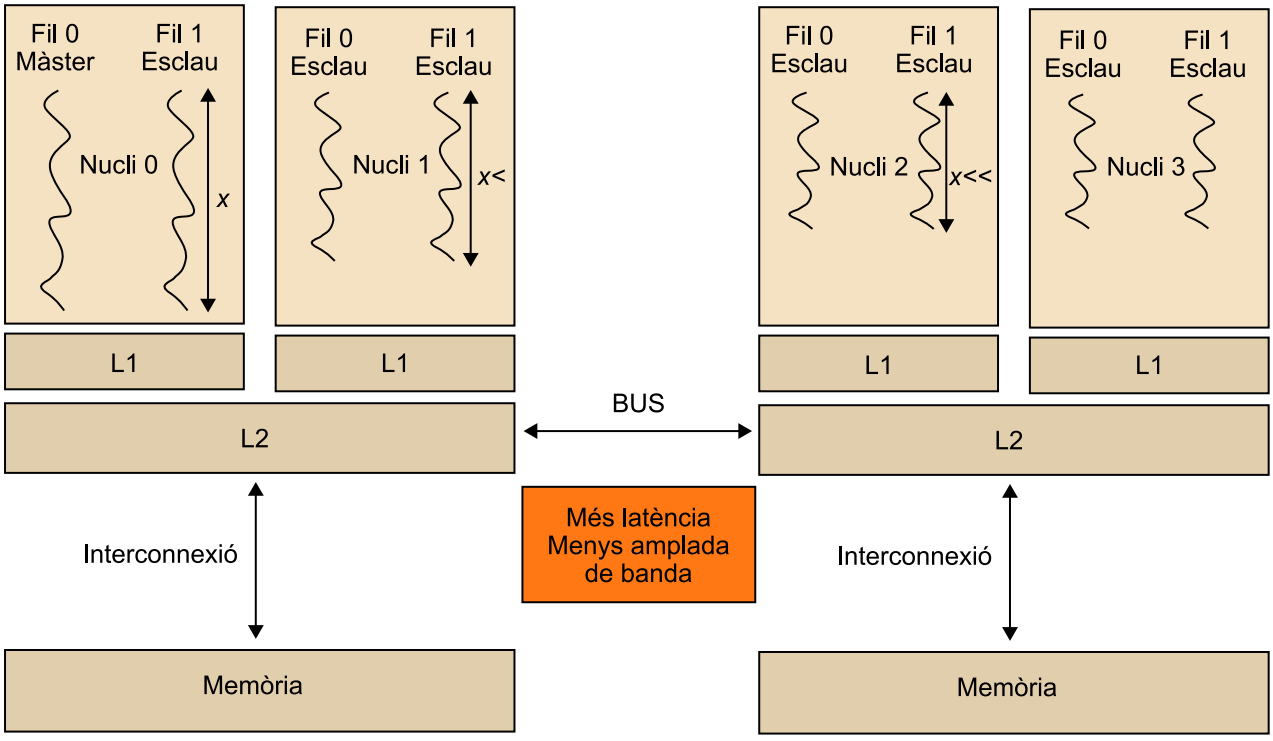
La figura 19 presenta un exemple d'aquesta situació. Una aplicació multifil s'executa sobre una arquitectura composta per dos processadors. Cadascun amb accés a memòria i tots dos connectats per un bus. Els fils que s'estan executant al nucli 2 i al nucli 3 accedeixen a les dades i a la informació que el màster els facilita.

Respecte d'una xarxa d'interconnexió local, el bus acostuma a tenir una latència més elevada i menys amplada de banda. Aquest és el motiu pel qual els fils que s'executen al nucli 2 i al nucli 3 tenen cert desbalanceig respecte dels que ho fan al nucli 0 i al nucli 1. Aquests factors cal considerar-los a l'hora de decidir com assignar les tasques i la feina a cadascun dels fils.

En l'exemple considerat el rendiment del processador i de l'aplicació és força menor del que potencialment es podria assolir. En l'instant de temps T, el fil esclau del nucli 1 ha acabat de fer la feina X. Els fils 0 i 1 del nucli 1 tarden un cert temps (T') més a finalitzar la seva tasca. I els fils dels nuclis 2 i 3 tarden un temps (T'')força major a T fins a acabar. Per tant, els nuclis 0 i 1 no es fan servir durant $T - T''$ i $T' - T''$ respectivament.

No només la utilització del processador és més baixa, sinó que aquest desbalanceig fa finalitzar l'aplicació més tard. En aquest cas, a l'hora de dissenyar el repartiment de tasques cal tenir en compte en quina arquitectura s'executa l'aplicació, com també quins són els elements que potencialment poden causar desbalancejos i quins fils poden fer més feina per unitat de temps.

Figura 19. Patró *master/slave* en un multinucli amb dos processadors



3.2.3. Definició i implementació de mecanismes de sincronització

Sovint les aplicacions multifil usen mecanismes per a sincronitzar les tasques que els diferents fils estan duent a terme. Un exemple el trobem en la figura de l'exemple anterior (figura 19), en què el fil màster s'espera que els esclaus acabin mitjançant funcions d'espera.

En general s'acostumen a fer servir tres tipus de mecanismes diferents de sincronització:

- L'ús de variables per a controlar l'accés a determinades parts de l'aplicació o a determinades dades de l'aplicació (com un comptador global). Exemples d'aquest tipus de variables ho són les dels semàfors o les d'exclusió mútua.

- L'ús de barreres per tal de controlar la sincronització entre els diferents fils d'execució. Aquestes ens permeten assegurar que tots els fils no passen d'un determinat punt fins que tots hi han arribat.

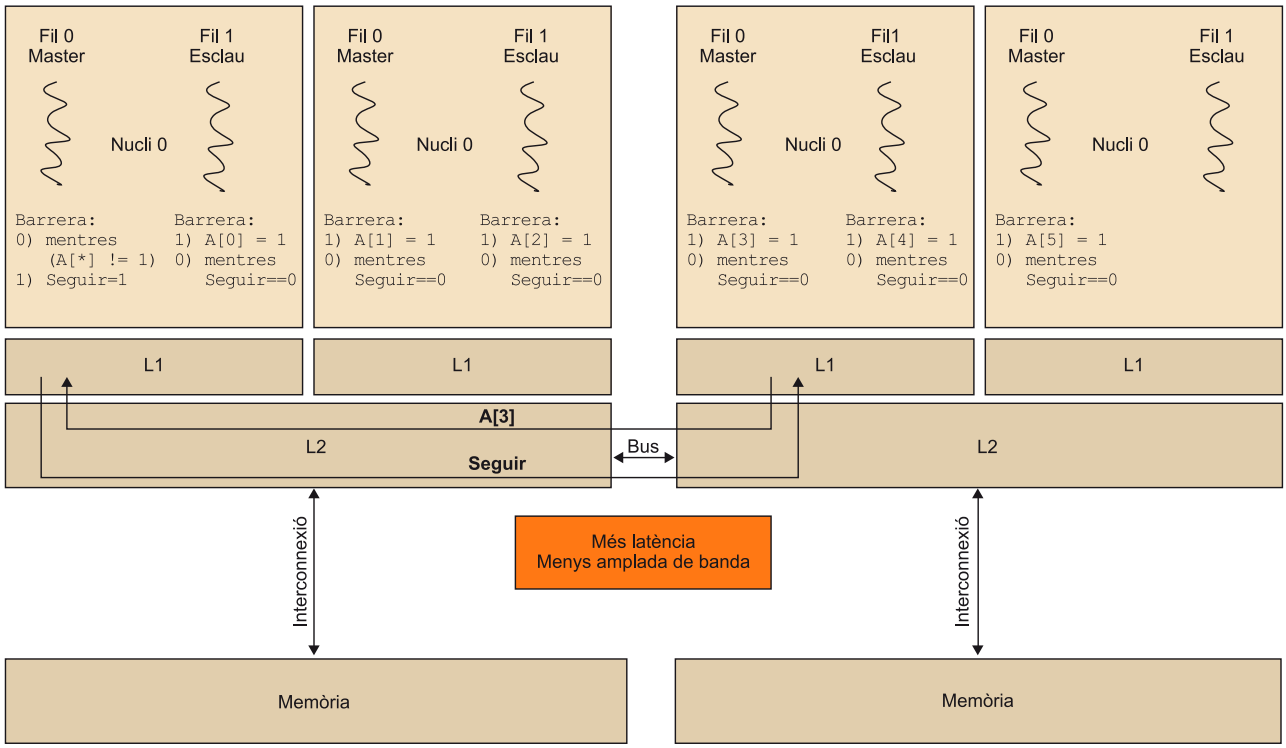
- L'ús de mecanismes de creació i espera de fils. Com a l'exemple anterior, el fil màster espera la finalització dels diferents fils esclaus mitjançant crides a funcions d'espera.

Des del punt de vista d'arquitectures multifil/nucli, el segon i el tercer punt són menys intrusius al rendiment de l'aplicació (Villa, Palermo i Silvano, 2008). Com es veurà a continuació, les barreres són mecanismes que s'empren en només certes parts de l'aplicació i que es poden implementar de manera eficient dins d'un multifil/nucli. En canvi, un ús excessiu de variables de control pot provocar un descens significatiu en el rendiment de les aplicacions.

Barreres en arquitectures multinucli

La figura 20 presenta un exemple de possible implementació de barrera i com es comportaria en un multinucli. En aquest cas, un fil s'encarrega de controlar el nombre de nuclis que han arribat a la barrera. Cal fer notar que el nucli 0 per tal d'accedir al vector A té totes aquestes dades a la L1, en estat compartit o en exclusiu.

Figura 20. Funcionament d'una barrera en un multinucli



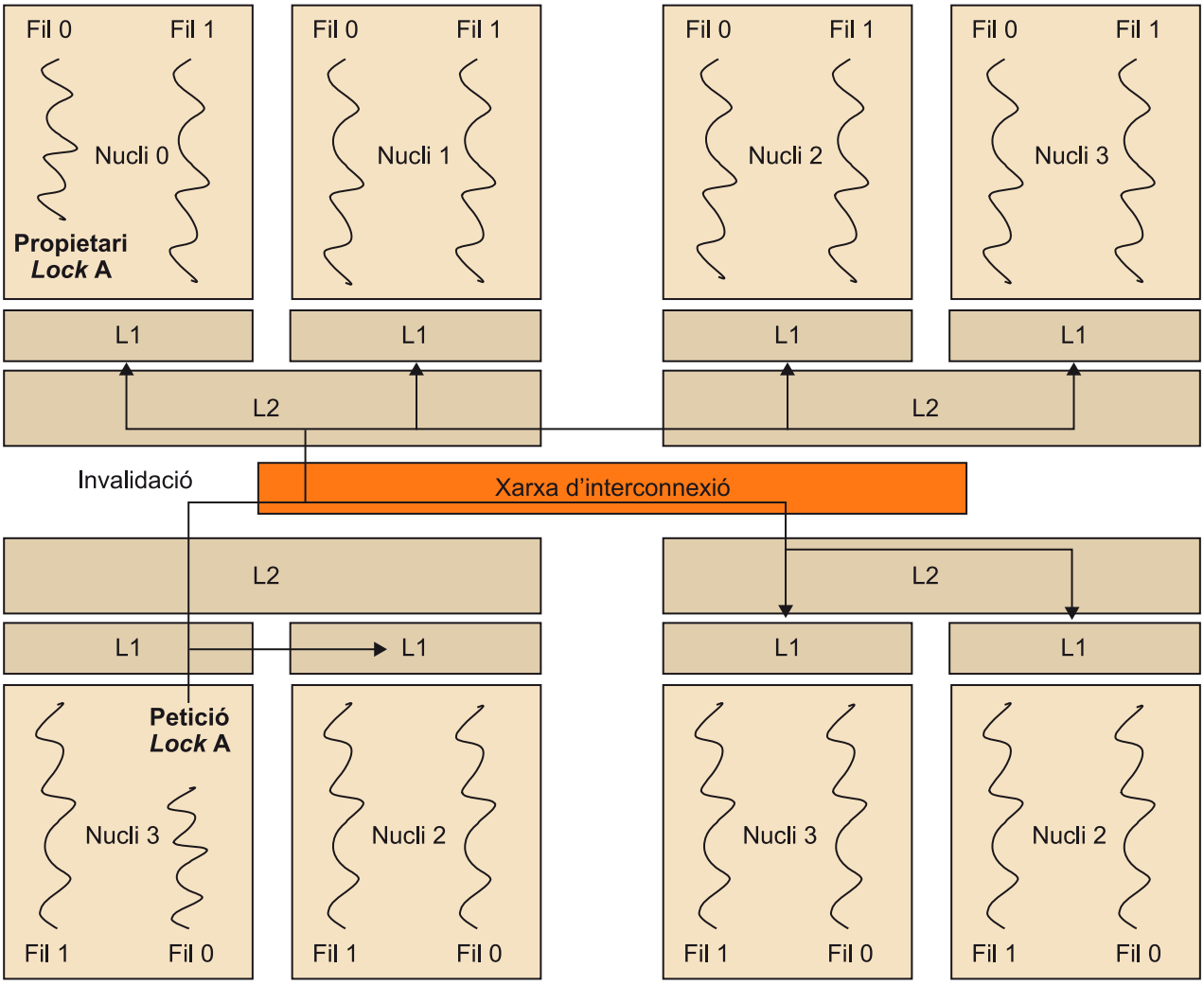
Cada vegada que un fil arriba a la barrera, vol modificar la posició corresponent del vector. Per tant, invalida la línia corresponent (la que conté A[i]) de tots els nuclis i la modifica. Quan el nucli 0 torna a llegir l'adreça d'A[i], ha de demanar el valor al nucli i. Segons el tipus de protocol de coherència que implementi el processador, el nucli 0 invalida la línia del nucli i o bé la mou a estat compartit.

Com es pot deduir d'aquest exemple, el rendiment d'una barrera pot ser més o menys eficient segons la seva implementació i l'arquitectura sobre la qual s'està executant. Així, una implementació en què, en lloc d'un vector, hi ha una variable global que compta els que han acabat seria més ineficient, ja que els diferents nuclis haurien de competir per agafar la línia, invalidar els altres nuclis i escriure'n el valor nou.

Mecanismes d'exclusió mútua en arquitectures multinucli

Aquests mecanismes s'empren per tal de poder accedir de manera exclusiva a certes parts del codi o a certes variables de l'aplicació. Són necessaris per a mantenir l'accés coordinat als recursos compartits i evitar condicions de carreres, *deadlocks* o situacions similars. Alguns d'aquest tipus de recursos són semàfors, *mutex-locks* o *read-writerlocks*.

Figura 21. Adquisició d'una variable d'exclusió mútua



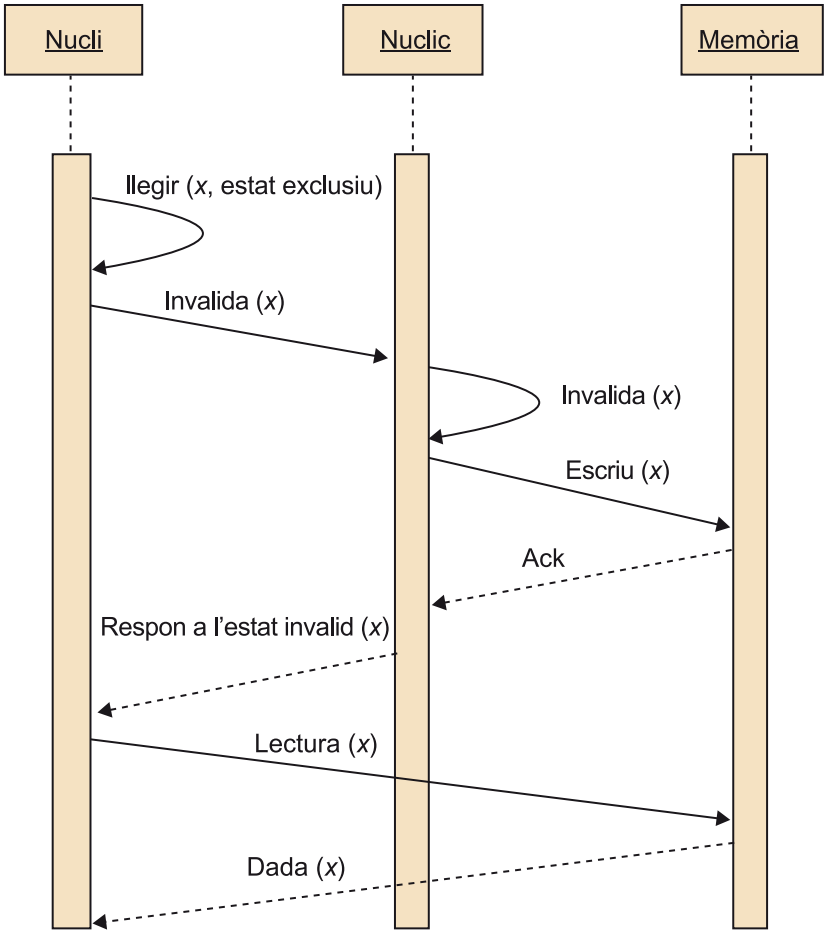
En arquitectures d'un sol nucli, l'accés a aquests tipus d'estructures pot tenir un impacte relativament menor. Amb alta probabilitat tots els fils estan compartint els accessos a les mateixes línies de la L1 que les desen. No obstant això, en un sistema multinucli l'accés concurrent de diferents fils a aquestes estructures pot dur a problemes d'escalabilitat i de rendiments greus. De manera similar al que s'ha mostrat amb les barreres, l'accés a les línies de memòria que contenen les variables emprades per a gestionar l'exclusió mútua impliquen invalidacions i moviments de línies entre els nuclis del processador.

La figura 21 mostra un escenari en què l'ús freqüent d'accés a zones d'exclusió mútua entre els diferents fils pot reduir substancialment el rendiment de l'aplicació. En aquest exemple s'assumeix un protocol de coherència entre els diferents nuclis de tipus *snoop*; per tant, cada vegada que un dels fils d'un nucli vol agafar la propietat de la variable d'exclusió mútua (*lock*) ha d'invalidar tota la resta de nuclis. Cal fer notar que la línia de la memòria cau que conté la variable en qüestió es troba en estat modificat cada cop que el nucli l'actualitzi.

Cada vegada que un fil agafa el *lock*, modifica la variable per tal de marcar-la com a pròpia. En el cas que el fil només l'estigui consultant, no caldria invalidar els altres nuclis.

Es pot assumir que la variable es reenvia del nucli que la té (el nucli 0) al nucli que la demana (el nucli 3) en estat modificat. Ara bé, depenent del tipus de protocol de coherència que implementés el processador, la línia s'escriuria primer a memòria i, llavors, el nucli 3 la podria llegir. En aquest cas el rendiment seria extremadament baix: per cada lectura del *lock x*, s'invalidarien tots els nuclis; així, el que la tingués en estat modificat l'escriuria a memòria i finalment el nucli que l'estigués demanant la llegiria de memòria (figura 22).

Figura 22. Lectura del *lock x* en estat exclusiu



Com s'ha pogut veure, és recomanable un ús moderat d'aquest tipus de mecanismes en arquitectures amb molts nuclis i també en arquitectures heterogènies. Així doncs, a l'hora de decidir quin tipus de mecanismes de sincronització es fan servir, cal considerar l'arquitectura sobre la qual s'executa l'aplicació (jerarquia de memòria, protocols de coherència i interconnexions entre nuclis) i com s'implementen tots aquests mecanismes. D'altra banda, és també recomanable reduir al màxim la quantitat de dades que comparteixen els di-

Lectura recomanada

Per tal d'aprofundir en la creació de mecanismes d'exclusió mútua escalables en arquitectures multinucli es recomana llegir l'article:

M. Chynoweth i M. R. Lee (2009). "Implementing Scalable Atomic Locks for Multi-Core".

ferents nuclis. D'aquesta manera es disminueix la quantitat de missatges de protocol de coherència que s'envien entre nuclis i el nombre d'invalidacions i *snoops* que han de processar els nuclis.

Un dels punts més importants quan es fa el disseny d'arquitectures multinucli eficients, per a aplicacions de supercomputació o HPC, és precisament que facilitin mecanismes per a poder implementar de manera eficient barreres entre tots els fils de les aplicacions. Depenent del nombre de nuclis i dels mecanismes que el processador faciliti una barrera pot tardar des d'un centenar de cicles de processador fins a milers de cicles.

Lectura recomanada

Els autors de l'article següent tracten d'implementacions escalables de mecanismes de sincronització entre fils:

J. M. Mellor-Crummey; M. L. Scott (1991). "Algorithms for scalable synchronization on shared-memory multiprocessors". *ACM Transactions on Computer Systems*.

3.2.4. Gestió d'accés concurrent a dades

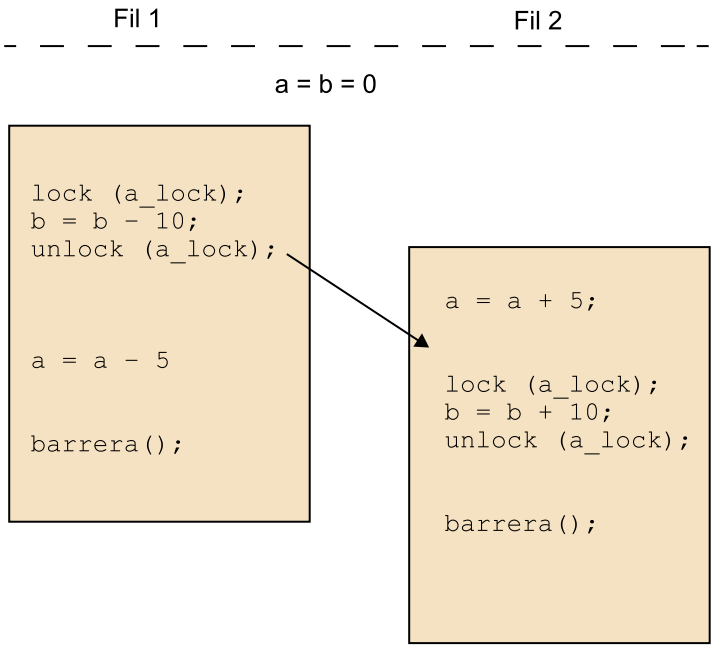
Acabem de veure les implicacions que té usar diferents tècniques d'exclusió mútua en arquitectures multiprocessador. Aquest tipus de tècniques són emprades per tal d'assegurar que l'accés a les dades entre diferents fils és controlat. Si no s'usen aquestes tècniques de manera adequada les aplicacions poden acabar tenint carreres d'accés[3].

(3)En anglès, *race accesses*.

En general es poden distingir dos tipus de carreres d'accés que cal evitar quan es desenvolupa un codi paral·lel:

1) Carreres d'accés a dades: succeeixen quan diferents fils estan accedint de manera paral·lela a les mateixes variables sense cap tipus de protecció. Per tal que hi hagi una carrera d'aquest tipus, un dels accessos ha de ser en forma d'escriptura. La figura 23 mostra un codi que potencialment pot tenir un accés a carrera de dades. Suposant que els dos fils s'estan executant al mateix nucli, quan tots dos fils arriben a la barrera, quin valor té *a*? Com que l'accés a aquesta variable no es troba protegit i s'hi accedeix tant en mode lectura com en escriptura (afegint-hi 5 i –5) aquesta variable pot tenir els valors següents: 0, 5 i –5.

Figura 23. Carrera d'accés a dades



2) Carreres d'accés general: aquest tipus de carreres succeeixen quan dos fils diferents han de seguir un ordre específic en l'execució de diferents parts del seu codi, però no tenim estructures que forcin aquest ordre. L'exemple següent mostra un d'aquest tipus de carreres. Tots dos fils usen una estructura guardada a memòria compartida en què el primer fil posa un treball i el segon el processa. Com es pot observar, si no s'hi afegeix cap tipus de control o variable de control, és possible que el fil 2 s'acabi d'executar sense processar el treball *a*.

Exemple carrera d'accés general

```
Fil 1:
Treball a = nou_treball();
Configurar_Treball(a);
EncuaTreball(a,Cua);
PostProces();
Fil 2:
Treball b  = AgafaTreball(Cua);
si(b != INVALID)
ProcessaTreball(b);
Acaba();
```

Carrera d'accés a dades

Cal esmentar que una carrera d'accés a dades és un tipus específic de carrera d'accés general. Tots dos tipus poden ser evitats emprant els mecanismes d'accés introduïts en l'anterior subapartat (barreres, variables d'exclusió mútua, etc.). Sempre que es dissenyi una aplicació multifil cal considerar que els accessos en mode escriptura i lectura a parts de memòries compartides han d'estar protegits; si els diferents fils assumeixen un ordre específic en l'execució de diferents parts del codi cal forçar-ho via mecanismes de sincronització i espera.

El primer dels dos exemples presentats (figura 23) es pot evitar afegint una variable d'exclusió mútua que controli l'accés a la variable *a*. D'aquesta manera, independentment de qui accedeixi primer a modificar el valor de la variable, el valor d'aquesta un cop arribat a la barrera és 0. El segon dels exemples de

la figura 23 podria ser evitat amb mecanismes d'espera entre fils, és a dir, com mostra l'exemple següent, el segon fil hauria d'esperar que el primer fil notifiqui que li ha facilitat el treball.

Evitar la carrera d'accés mitjançant sincronització

```
Fil 1:
Treball a = nou_treball();
Configurar_Treball(a);
EncuaTreball(a,Cua);
NotificaTreballDisponible();
PostProces();
Fil 2:
EsperaTreball();
Treball b  = AgafaTreball(Cua);
si(b != INVALID)
ProcessaTreball(b);
Acaba();
```

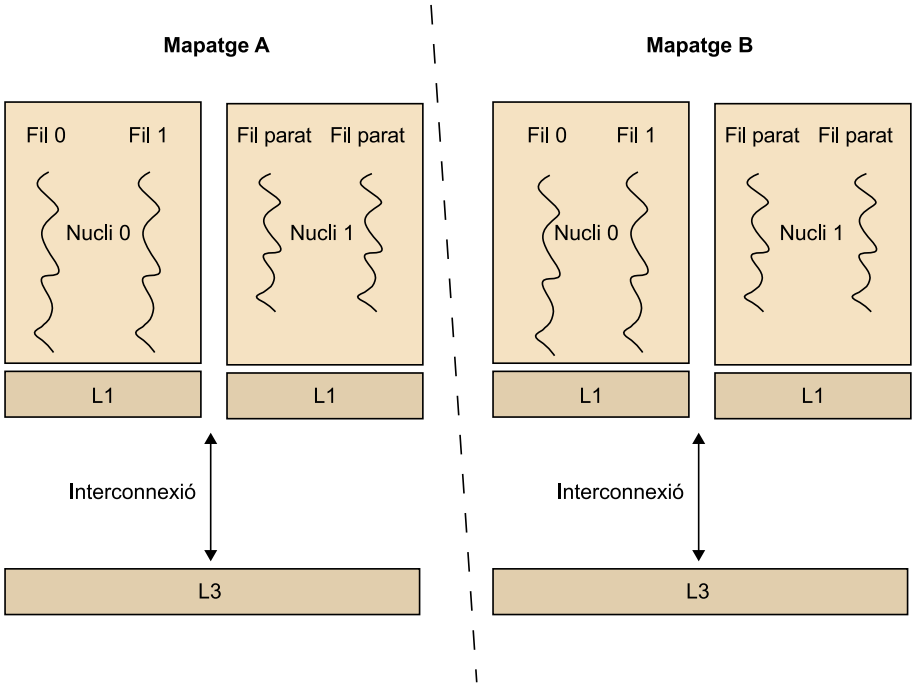
Condicions de carrera en arquitectures multinuclis

En el subapartat anterior s'han introduït diverses situacions en què l'ús de memòria compartida entre diferents fils és inadequat. El primer, les carreres d'accés a dades, esdevé quan dos fils diferents estant llegint i escrivint de manera descontrolada a una zona de memòria. S'ha mostrat com el valor d'una variable pot ser funcionalment incorrecte quan tots dos fils finalitzen els seus fluxos d'instruccions. Ara bé, aquesta cadena d'esdeveniments succeeix independentment de l'arquitectura i del mapatge de fils sobre el qual s'executa l'aplicació?

En aquest subapartat es vol mostrar com el mateix codi es pot comportar de manera diferent dins d'un mateix processador segons com s'assignin els fils als nuclis, ja que, depenent d'aquest aspecte, la condició de carrera analitzada en el subapartat anterior succeeix o no.

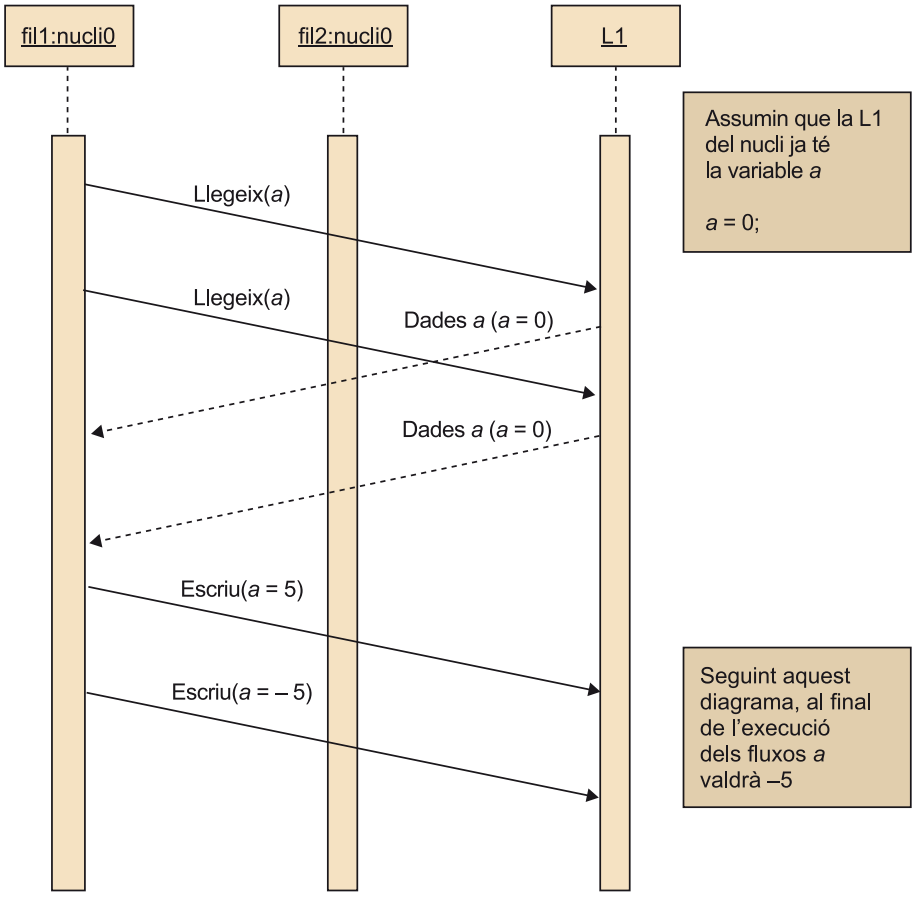
Suposem els dos escenaris que mostra la figura següent:

Figura 24. Dos mapatges de fils diferents executant l'exemple de carrera d'accés general



En el primer dels dos escenaris, els dos fils s'estan executant en el mateix nucli. Tal com mostra la figura 25, els dos fils accedeixen al contingut de la variable *a* de la mateixa memòria cau de nivell dos. Primer el fil 1 en llegeix el valor.

Figura 25. Accés compartit a *a* en un mateix nucli

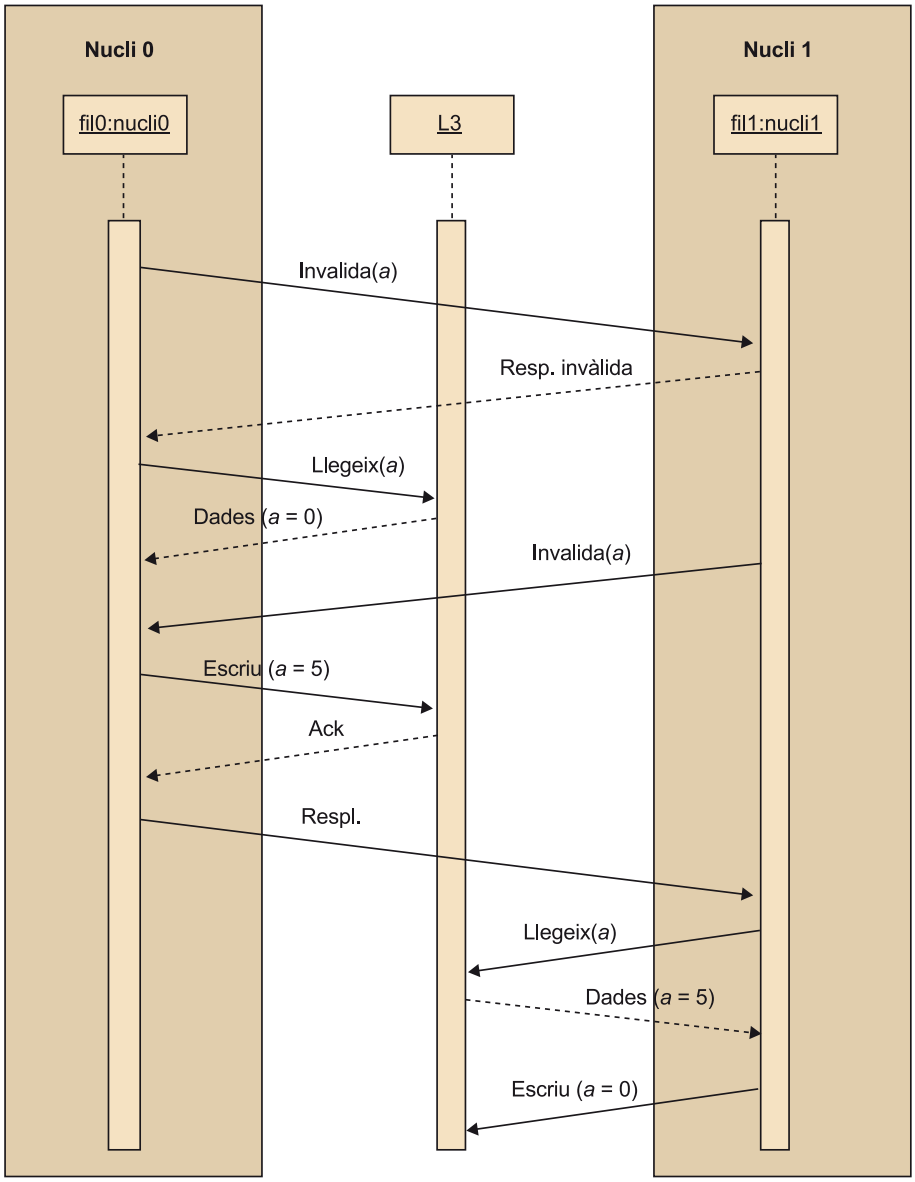


A continuació, tot i que el fil 1 ja està modificant la variable, el segon fil en llegeix el valor original. Finalment, tots dos fils escriuen els valors a memòria. No obstant això, el segon fil sobreescriu el valor actual (5) pel valor resultant de l'operació aritmètica que el fil ha aplicat (–5). Aquest flux d'esdeveniments, com ja s'ha vist, és funcionalment incorrecte, és a dir, el resultat de l'execució dels diferents fils no és l'esperat per l'aplicació en qüestió.

Assumim ara el segon dels escenaris, en què cadascun dels fils s'executa en un nucli diferent. En les arquitectures de processadors amb memòria coherent, si dos nuclis diferents estan accedint en un mateix moment a una mateixa línia de memòria, el protocol de coherència assegura que només un nucli pot modificar el valor d'aquesta línia. Per tant, en un instant de temps tan sols un nucli pot tenir la línia en estat exclusiu per tal de modificar-la.

En aquest escenari nou, i gràcies al protocol de coherència, la carrera d'accés mostrada en el cas anterior no succeeix. Com mostra la figura 26, el protocol de coherència protegeix l'accés en mode exclusiu a la variable *a*. Quan el primer fil vol llegir el valor de la variable en mode exclusiu per a ser modificat, n'invalida totes les còpies dels nuclis del sistema (en aquest cas només un). Un cop el nucli 0 notifica que té la línia en estat invàlid, en demana el valor a la memòria cau de tercer nivell. Per tal de simplificar l'exemple, suposem que modifica el valor de *a* tan aviat com la rep.

Figura 26. Accés compartit a *a* en nuclis diferents



A continuació, el fil 0 que s'està executant en el nucli 1, per tal d'agafar la línia en mode exclusiu, invalida la línia dels altres nuclis del sistema (nucli 0). Quan el nucli 0 en rep la petició d'invalidació, aquest en valida l'estat. Com que es troba en estat modificat, n'escriu el valor a la memòria cau de tercer nivell. Un cop en rep l'*acknowlegement*[4], respon al nucli 1 que la línia es troba en estat invàlid. Arribat a aquest punt, el nucli 1 en demana el contingut a la L3, rep la línia, la modifica i l'escriu de tornada amb el valor correcte.

(4)D'ara en endavant ack.

D'una banda, és important remarcar que si bé en aquest cas no es dóna la carrera d'accés, la implementació paral·lela segueix tenint un problema de sincronització. Depenent de quin tipus de mapatge s'apliqui als fils de l'aplicació es troba l'error ja estudiat.

D'altra banda, cal tenir present que, depenent de quin tipus de protocol de coherència implementi el processador, el comportament de l'aplicació podria variar. Per aquest motiu, emprar les tècniques de sincronització per tal de fer que l'execució sigui determinista i no lligada a factors arquitectònics o de mapatge és realment rellevant.

3.2.5. Altres factors que cal considerar

Aquest subapartat presenta diferents factors que cal considerar a l'hora de dissenyar, desenvolupar i executar aplicacions paral·leles en arquitectures multifil, com també les característiques principals i les implicacions que aquestes aplicacions tenen sobre les arquitectures multifil.

Com ja s'ha esmentat, el disseny d'aplicacions paral·leles és un camp en què s'ha fet molta recerca (tant acadèmica com industrial). És, per tant, aconsellable aprofundir en algunes de les referències facilitades. Altres factors que no s'han esmentat, però que també són importants, són els següents:

- Els *deadlocks*. Succeeixen quan dos fils es bloquegen esperant un recurs que té l'altre. Tots dos fils queden bloquejats per sempre, per tant bloquegen l'aplicació (Kim i Jun, 2009).

- Composició dels fils paral·lels. És a dir, la manera en què s'organitzen els fils d'execució. Aquesta organització depèn del model de programació (com OpenMP o MPI) i de com es programa l'aplicació (per exemple, depèn de si els fils estan gestionats per l'aplicació amb *PosixThreads*).

- Escalabilitat del disseny. Factors com ara el nombre de fils, baixa concurrència en el disseny o bé massa contenció en accessos als mecanismes de sincronització poden reduir substancialment el rendiment de l'aplicació.

Lectura recomanada

Per tal d'aprofundir en la escalabilitat del disseny, es recomana la lectura següent:

S. Prasad (1996). *Multithreading Programming Techniques*. Nova York: McGraw-Hill, Inc.

3.3. Factors lligats a l'arquitectura

Aquest subapartat presenta alguns factors que poden impactar en el rendiment de les aplicacions paral·leles que no estan directament lligades al model de programació. Com es veurà a continuació, alguns d'aquests factors apareixen depenent de les característiques del processador multifil sobre el qual s'executa l'aplicació. Així doncs, tractarem alguns dels factors més importants que cal que tenir en compte quan es desenvolupen aplicacions paral·leles per a arquitectures multifil.

D'una banda, la compartició de recursos entre diferents fils pot portar a situacions de conflictes que degraden molt el rendiment de les aplicacions. Un cas de compartició és el de les mateixes entrades d'una memòria cau.

D'altra banda, el disseny de les arquitectures multifil implica certes restriccions que cal considerar en implementar les aplicacions. Per exemple, un processador multinucli té un sistema de jerarquia de memòria en què els nivells inferiors (per exemple, la L3) es comparteixen entre diferents fils i els superiors es troben separats pel nucli (per exemple, la L1). Les fallades als nivells superiors són força menys costoses que als nivells inferiors. No obstant això, la mida d'aquestes memòries és força menor, per tant, cal adaptar les aplicacions perquè tinguin el màxim de local possible als nivells superiors.

Aquest subapartat està centrat en aspectes lligats als protocols de coherència i gestió de memòria, tot i que hi ha molts altres factors que cal considerar si es vol treure el màxim rendiment de l'algorisme que s'està dissenyant.

Exemple

Optimitzacions del compilador (Lo, Eggers, Levy, Parekh i Tullsen, 1997), característiques de les memòries cau (Hily i Seznec, 1998) o el joc d'instruccions del processador (Kumar, Farkas, Jouppi, Ranganathan i Tullsen, 2003).

3.3.1. Compartició falsa

En moltes situacions es vol que diferents fils de l'aplicació comparteixin zones de memòria concretes. Com hem vist, sovint es donen situacions en què diferents fils comparteixen comptadors o estructures en els quals es desen dades resultants de càlculs fets per cadascun, en què s'usen variables d'exclusió mútua per tal de protegir aquestes zones.

Compartició d'entrades

En la figura 25, tots dos fils accedeixen a la mateixa entrada de la L1 que desa la variable *a*.

Les dades que els diferents fils estan usant en un instant de temps concret es desen en les diferents memòries cau de la jerarquia (des de la memòria cau de darrer nivell, fins a la memòria cau de primer nivell del nucli que l'està fent servir). Per tant, quan dos fils estan compartint una dada, també comparteixen les mateixes entrades de les diferents memòries cau que desen aquesta dada.

Des del punt de vista de rendiment el que es busca és que els accessos dels diferents fils encertin alguna de les memòries cau (com més propera al nucli millor), ja que l'accés a memòria és molt costós. D'això se'n diu mantenir la localitat en els accessos (Grunwald, Zorn i Henderson, 1993).

Memòria 2-associativa

Una memòria 2-associativa (Seznec, 1993) divideix la memòria cau en conjunts de dues entrades. Primer es calcula en quin dels conjunts pertany l'adreça i després s'escull quina de les dues entrades conjunt es fa servir.

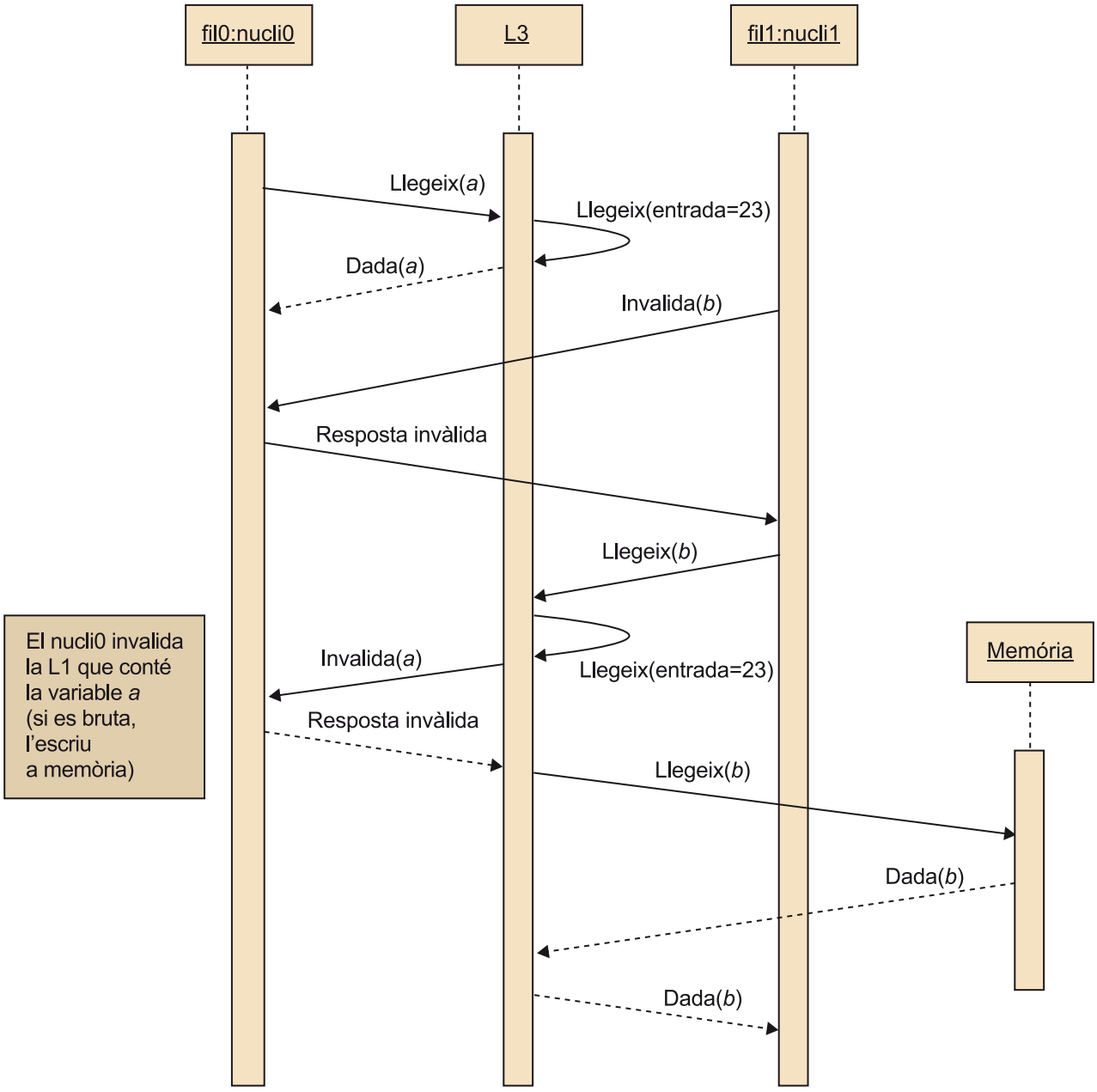
Ara bé, hi ha situacions en què les mateixes entrades de la memòria cau les comparteixen dades diferents. El càlcul de quina entrada d'una memòria cau es fa servir es defineix segons l'associativitat i el tipus de mapatge de la memòria (Handy, 1998).

Pot passar que dos fils accedeixin a dos blocs de memòria diferents, que es mapegin al mateix conjunt d'una memòria cau. En aquest cas, els accessos d'un fil al seu bloc de memòria invaliden les dades de l'altre fil desades a les mateixes entrades de la memòria. Tot i que les adreces coincideixen en les mateixes entrades de la memòria, les dades i les adreces són diferents. Per tant, per a cada accés s'invaliden les dades de l'altre fil i es demana la dada al següent nivell de la jerarquia de memòria (per exemple, la L3 o la memòria principal). Com es pot deduir, aquest aspecte implica una reducció important en el rendiment de l'aplicació. Això s'anomena *compartició falsa*[5].

(5)En anglès, *false sharing*.

La figura següent mostra un exemple del que podria passar en una arquitectura multinucli en què dos fils de nuclis diferents estan experimentant *falsesharing* al mateix set de la L3. Per a cada accés d'un dels fils s'invalida una línia de l'altre fil, que és desada al mateix conjunt. Com s'observa per a cada accés es generen un nombre important d'accions: invalidació d'adreça a l'altre nucli, lectura a la L3, es victimitza la dada de l'altre fil de l'altre nucli i s'escriu a memòria si és necessari, es llegeix la dada a memòria i s'envia al nucli que l'ha demanada.

Figura 27. Compartició falsa a la L3



En el cas que tots dos fils no tinguin conflicte en els mateixos conjunts de la L3, amb una probabilitat força alta encerten a L3 i s'estalvien la resta de transaccions que s'esdevenen per culpa de la compartició falsa.

La compartició falsa es pot detectar quan una aplicació mostra un rendiment molt baix i el nombre d'invalidacions d'una memòria cau concreta i misos és molt més elevat del que s'esperava. En aquest cas, és molt probable que dos fils competeixin pels mateixos recursos de la memòria cau. Hi ha eines com el Vtune d'Intel (Intel, *Boost Performance Optimization and Multicore Scalability*, 2011) que permeten detectar aquest tipus de problemes.

Evitar la compartició falsa és relativament més senzill del que pot semblar. Un cop s'han detectat quines són les estructures que probablement causen aquest efecte, cal afegir un desplaçament per tal que caiguin en posicions de memòria diferent per a cadascun dels fils. D'aquesta manera les dades que abans compartien un mateix set de la memòria cau, aquest cop es desen en sets diferents.

3.3.2. Penalitzacions per a fallades a la L1 i tècniques de *prefetch*

Moltes aplicacions paral·leles tenen una alta localitat a les memòries cau del nucli sobre les quals s'executen. És a dir, la majoria d'accessos que es fan a memòria encerten la memòria cau de primer nivell.

Ara bé, els casos en què les peticions a memòria no encerten la memòria cau del nucli han de fer un procés molt més llarg fins que la dada és disponible per al fil: petició a la L3, fallada a la L3, petició a memòria, etc. Evidentment, com més alt és el percentatge de fallades, més penalitzada es troba l'aplicació. La causa és que un accés a memòria que falla a la L1 té una latència molt més elevada que una que l'encerta (un o dos ordres de magnituds més elevats, depenent de l'arquitectura i protocol de coherència).

Una de les tècniques que s'usa per a mirar d'amagar la latència més llarga de les peticions que fallaran a la memòria cau s'anomena *prefetching*. Aquesta tècnica consisteix a demanar la dada a memòria molt més aviat del que l'aplicació la necessita. D'aquesta manera, quan realment la necessita, aquesta ja es troba a la memòria local del nucli (L1). Per tant, l'encerta, la latència de la petició a memòria és molt més baixa i l'aplicació no es bloqueja.

Hi ha dos tipus de tècniques de *prefetching* usades en les arquitectures multinucli:

1) ***Hardware prefetching***. És una tècnica que s'implementa en les peces dels nuclis que s'anomenen *hardware prefetchers*. Aquests intenten predir quines són les properes adreces que l'aplicació demanarà i les demanen de manera proactiva abans que l'aplicació ho faci.

Aquest component es basa en tècniques de predicció que no requereixen una lògica gaire complexa, com les cadenes de Markov (Joseph i Grunwald, 1997). El problema principal d'aquest tipus de tècniques és que són agnòstiques res-

pecte del que l'aplicació està executant. Per tant, tot i que per algunes aplicacions pot funcionar força bé, per a altres les prediccions poden ser errònies i fer que el rendiment de l'aplicació empitjori.

2) *Software prefetching*. Alguns dels sistemes multinucli faciliten instruccions per tal que les aplicacions puguin demanar de manera explícita les dades a memòria, usant la instrucció de *prefetch*.

La instrucció *vprefetch*

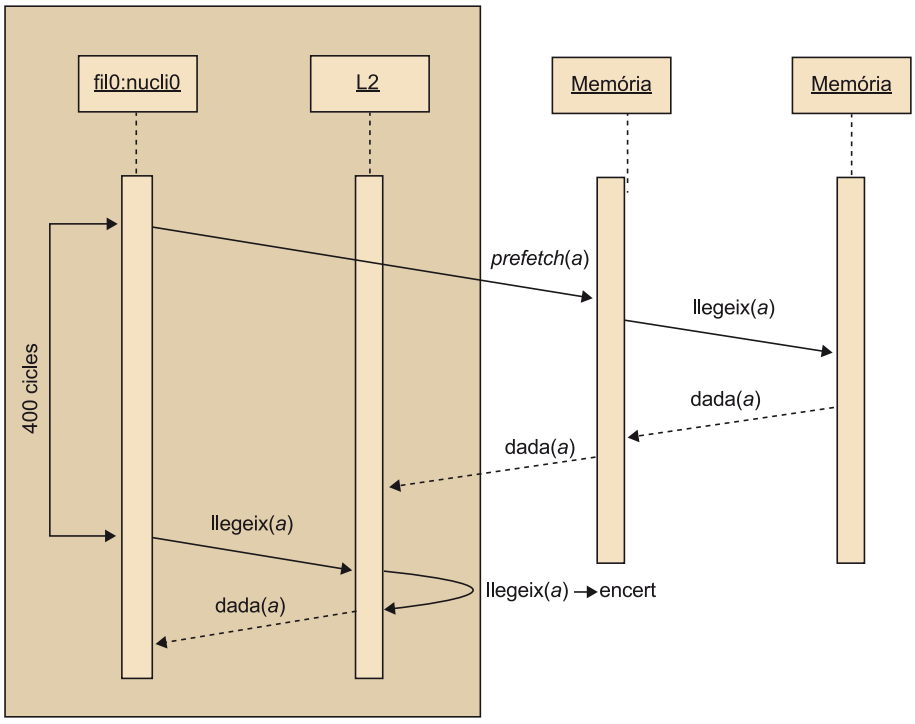
En l'arquitectura Intel es fa servir la instrucció *vprefetch* (Intel, *Use Software Data Prefetch on 32-Bit Intel / IA-32 Intel® Architecture Optimization Reference Manual,* 2011). L'aplicació és responsable de demanar les dades a memòria de manera anticipada usant aquesta instrucció.

L'avantatge d'aquest mecanisme és que l'aplicació sap exactament quan necessitarà les dades. Per tant, basant-se en la latència d'un accés a memòria, ha de demanar les dades amb el nombre de cicles suficient per tal que quan la necessiti la tingui. L'ús d'aquest tipus de *prefetch* és el que acostuma a permetre obtenir el rendiment més elevat de l'aplicació.

El desavantatge principal és que el codi es troba lligat a una arquitectura concreta. És a dir, la majoria d'arquitectures multinucli tenen latències diferents. Per tant, cada vegada que es vol executar aquesta aplicació en una plataforma nova cal calcular les distàncies de *prefetch* adequades al sistema nou.

La figura següent presenta un exemple de com funcionaria un *prefetch* en un sistema format per un multinucli amb una memòria cau de tercer nivell. En aquest cas l'aplicació necessita la dada *a* al cicle *X*; assumint que la latència d'un accés a memòria que falla a la L3 és de 400 cicles, ha de fer el *prefetchX*-400 cicles abans.

Figura 28. Exemple de *prefetch*



Cal tenir, però, les consideracions següents:

- Els *prefetchs* acostumen a poder ser eliminats del *pipeline* del processador si no hi ha prou recursos per a dur-lo a terme (per exemple, si la cua que desa les dades que tornen de la L3 és plena). Per tant, si el nombre de *prefetchs* que fa una aplicació és massa elevat, poden ser eliminats del sistema.

- La latència de les peticions a memòria poden variar depenent del camí que segueixin. Per exemple, un encert a la L3 fa que una dada estigui disponible a la mateixa memòria L3 molt més aviat del previst, i en el cas que hi hagi una víctima interna serà molt més tardana. Quan es fan servir aquest tipus de peticions cal tenir en compte l'ús de tota la jerarquia de memòria, com també les característiques de l'aplicació que s'usa.

- Els *prefetchs* poden tenir impactes de rendiment tant positius com negatius en l'aplicació desenvolupada. Cal estudiar quins requisits té l'aplicació desenvolupada i com es comporten en l'arquitectura que executa l'aplicació.

Lectura recomanada

L'ús de tècniques de *prefetch* en arquitectures multifil és un recurs freqüent per tal de treure rendiment en aplicacions paral·leles. Per aquest motiu, es recomana llegir l'article:

Intel (2007). *Optimizing Software for Multi-core Processors*. Portland: Intel Corporation - White Paper.

3.3.3. Impacte del tipus de memòria cau

Cada vegada que un nucli accedeix a una línia de memòria per primera vegada, l'ha de demanar al nivell de memòria següent. En els exemples estudiats, les fallades a la L2 es demanen a la L3, i les fallades a la L3 es demanen a memòria. Cadascuna d'aquestes fallades genera un conjunt de víctimes en les diferents memòries cau: cal alliberar una entrada per la línia que s'està demanant.

Per tal de treure rendiment a les aplicacions paral·leles que es desenvolupen és important mantenir al màxim la localitat en els accessos a les memòries cau. És a dir, maximitzar el reús (percentatge d'encert) a les diferents memòries cau (L1,L2, L3, etc.). Per aquest motiu és important considerar les característiques de la jerarquia de memòria cau: inclusiva, no inclusiva/exclusiva i mides.

A continuació es discuteixen els diferents punts esmentats des del punt de vista de l'aplicació.

Inclusivitat

Si dues memòries (Lx i Lx – 1) són inclusives vol dir que qualsevol adreça @X que sigui a Lx – 1 es troba sempre a la Lx. Per exemple, si la L1 és inclusiva amb la memòria L2, aquesta inclou la L1 i altres línies. En aquest cas cal considerar el següent:

1) Quan una línia de la Lx – 1 es victimitza, al final de la transacció aquesta està disponible al nivell següent de memòria Lx.

2) Quan una línia de la Lx es victimitza, al final de la transacció aquesta línia ja no està disponible al nivell superior Lx – 1. Per exemple, en el cas de victimitzar la línia @X a L3, aquesta s'invalida també a la memòria cau L2. I si la L1 és inclusiva amb la L2, la primera també invalida la línia en qüestió.

3) Quan un nucli llegeix una adreça @X que no es troba a la memòria Lx – 1, al final de la transacció aquesta també es troba inclosa a la Lx. Per exemple, en el cas que tinguem una L1, L2 i L3 inclusives, @X s'escriu a totes les memòries. Cal remarcar que cada una d'aquestes entrades usades potencialment ha generat una víctima. És a dir, una adreça @Y que usa el *way* i el *set* en el qual s'ha desat @X. La selecció d'aquesta posició depèn de la política de gestió de cada memòria cau, com també de la mida.

Els punts 2 i 3 poden causar un impacte força important en el rendiment de l'aplicació. Per tant, és important dissenyar les aplicacions per tal que el nombre de víctimes generades en nivells superiors sigui el més baix possible i maximitzar el percentatge d'encerts de les diferents jerarquies de memòria cau més properes al nucli (per exemple, L1).

Exclusivitat i no inclusivitat

Que dues memòries cau siguin exclusives implica que si la memòria cau Lx té una línia @X, la memòria cau Lx – 1 no la té. No s'acostumen a tenir arquitectures en què tots els nivells són exclusius. Habitualment les jerarquies que tenen memòries cau exclusives acostumen a ser híbrides.

Processadors amb memòria exclusiva

Un exemple de processador amb memòria exclusiva és l'AMD Athlon (AMD, 2011) i l'Intel Nehalem (Intel, *Nehalem Processor*, 2011). El primer té una L1 exclusiva amb la L2. El segon té una L2 i L1 no inclusives i la L3 és inclusiva de la L2 i la L1.

En els casos en què una memòria cau Lx és exclusiva amb Lx – 1 cal considerar el següent:

- Quan s'accedeix a una línia @Y a la memòria Lx – 1, aquesta es mou de la memòria Lx.

- Quan una línia es victimitza de la memòria Lx – 1, aquesta es mou a la memòria cau Lx. En aquests casos, com que la memòria és exclusiva, cal trobar una entrada en la memòria Lx. Per tant, cal victimitzar la línia que hi hagi desada al *way* i al *set* seleccionat.

- Quan una línia es victimitza de la memòria Lx no cal victimitzar la memòria cau Lx – 1.

Hi ha certes situacions en què cal saber detalladament quin tipus de jerarquia de memòria i protocol de coherència implementa el processador. Per exemple, si es considera una arquitectura multinucli en què la L1 és exclusiva amb la L2, en els casos en què els diferents fils estiguin compartint l'accés a un conjunt

elevat d'adreces el rendiment de l'aplicació es pot veure deduït. En aquesta situació, cada accés a una dada @X en un nucli podria implicar la victimització d'aquesta mateixa adreça en un altre nucli i demanar l'adreça a memòria.

Algunes memòries cau exclusives permeten que en certes situacions algunes dades es trobin en dues memòries que són exclusives entre si per defecte. Per exemple, en les línies de memòria compartides per diferents nuclis.

Mida de la memòria

El rendiment de l'aplicació depèn en gran mesura de la localitat dels accessos dels diferents fils en les memòries cau. En situacions en què els fils demanen diverses vegades les mateixes línies a memòria per mala praxi de programació, el rendiment de l'aplicació cau substancialment. Això passa quan un nucli demana una adreça @X, aquesta línia es victimitza i es torna a demanar més tard.

Un exemple senzill és a accedir a una matriu d'enters de 8 bytes per files quan aquesta s'ha emmagatzemat per columnes. En aquest cas, cada 8 enters consecutius d'una mateixa columna es trobarien mapats a la mateixa línia. Ara bé els elements i i $i + 1$ d'una fila es trobarien desats en línies diferents. Per tant, en el cas de recórrer a la matriu per columnes el nombre de fallades a la L1 serà molt més elevat.

Per aquest motiu, és important usar tècniques d'accés a les dades que intentin mantenir localitat a les diferents memòries cau.

No obstant això, en molts casos les aplicacions paral·leles dissenyades no segueixen cap dels patrons que hem analitzat en altres estudis acadèmics (per exemple, en tècniques de partició de matrius). Per a aquests problemes hi ha aplicacions disponibles que permeten analitzar com es comporten les aplicacions paral·leles i veure de quina manera es poden millorar.

Exemple

Alguns exemples són Vtune (Kishan Malladi, 2011) o Cachegrind (Valgrind, 2011).

3.3.4. Arquitectures multinucli i multiprocessador

En algunes situacions, les arquitectures en què s'executen les aplicacions paral·leles no només donen accés a múltiples fils i múltiples nuclis, sinó a múltiples processadors. En la majoria de casos, aquestes arquitectures contenen una placa mare o més en què cada una conté un conjunt de processadors que estan connectats per mitjà d'una connexió d'alta velocitat. Si l'arquitectura de computació està formada per més d'una placa, acostumen a estar connectades per xarxes de connexió més lentes.

Lectures recomanades

De manera semblant a d'altres factors ja introduïts anteriorment, s'ha fet molta recerca en aquest àmbit. Una referència en tècniques de partició en blocs per la multiplicació de matrius és la següent:

K. Kourtis; G. Goumas; N. Koziris (2008). "Improving the Performance of Multithreaded Sparse Matrix-Vector Multiplication Using Index and Value Compression". *37th International Conference on Parallel Processing*.

I en tècniques de compressió en el procés de grafs:

R. Jin; T. S. Chung (2010). "Node Compression Techniques Based on Cache-Sensitive B+-Tree". *9th International Conference on Computer and Information Science* (ICIS) (pàg. 133-138).

Tipus de connexions

Un exemple de connexió d'alta velocitat és l'Intel Quickpath Interconnect (Intel, *Intel® Quickpath Interconnect Maximizes Multi-Core Performance*, 2012), també conegut com a *QPI*. I com a exemple de connexió més lenta, tenim la Myrinet (Flich, 2000).

En aquestes arquitectures es poden assignar els diferents fils de l'aplicació a cadascun dels fils disponibles en cadascun dels nuclis dels diferents processadors connectats a una mateixa placa. Tots els diferents fils acostumen a compartir un espai de memòria coherent. Des del punt de vista de l'aplicació, aquesta té accés a *N* nuclis diferents i a un espai de memòria comú.

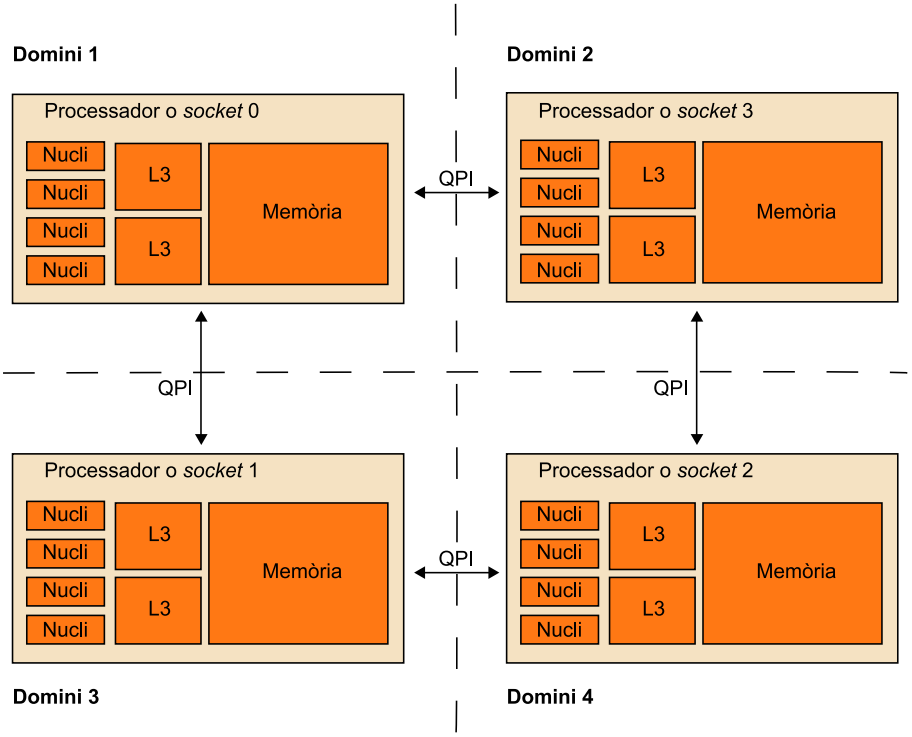
D'altra banda, els fils que s'han ubicat en plaques diferents no acostumen a compartir l'espai d'adreces de memòria. Per tant, si es vol que comparteixin informació, cal emprar un entorn que permeti fer-ho. Hi ha models de programació que ho permeten. El model de pas de missatges[6] n'és l'exemple més conegut. Aquest model de programació permet la comunicació de processos ubicats en plaques diferents (que, per tant, no comparteixen l'espai d'adreces) per mitjà d'enviament de missatges. En molts casos, aquest model es combina amb OpenMP. Aquest darrer permet definir paral·lelisme dins d'una mateixa placa assumint que els diferents fils comparteixen el mateix espai d'adreces.

(6)En anglès, *message passing interface* (MPI).

Tot i que aquest espai de memòria pot ser compartit per tots fils d'una mateixa placa, l'accés d'un fil a determinades zones de memòria pot tenir latències diferents. Cada processador disposa d'una jerarquia de memòria pròpia (des de la L1 fins a la memòria principal) i aquest gestiona un rang d'adreces de memòria concret. Així, si un processador disposa d'una memòria principal de 2 GB, aquest processador gestiona l'accés de l'espai d'adreces assignat a aquests 2 GB (per exemple, de 0x a FFFFFFFF).

Cada vegada que un fil accedeixi a una adreça que es troba assignada a un espai que gestiona un altre processador, ha d'enviar la petició de lectura d'aquesta adreça al processador que la gestiona a través de la xarxa d'interconnexió (com en una QPI). Aquests accessos són molt més costosos ja que han d'enviar la petició a través de la xarxa, arribar a l'altre processador i fer l'accés (seguint el protocol de coherència que segueixi l'arquitectura).

Figura 29. Arquitectura multifil



Aquest model de programació segueix el paradigma del que s'anomena *non-uniform memory access* (NUMA), en què un accés de memòria pot tenir diferents tipus de latència depenent d'on es trobi assignat.

Quan es programi per aquest tipus d'arquitectura, cal considerar tots els factors que s'han explicat en aquest apartat, però en una escala superior. Així doncs, en l'accés a variables d'exclusió mútua s'ha de considerar que, per cada vegada que s'agafi el *lock*, cal invalidar tota la resta de nuclis del sistema. Els que siguin a fora del processador van per la xarxa d'interconnexió i tarden més a retornar la resposta. Val a dir que el comportament depèn del protocol de coherència i de la jerarquia de memòria del sistema.

En aquest mòdul s'han introduït alguns dels aspectes més importants que cal tenir en compte a l'hora de dissenyar i desenvolupar aplicacions per a arquitectures de computació d'altes prestacions. Com s'ha pogut veure amb la gran quantitat de referències facilitades, aquest àmbit és molt complex i, per tal d'aprofundir-hi, cal dedicar-hi molts esforços i dedicació.

Lectures recomanades

Dins l'àmbit de programació multifil, per aquest tipus d'arquitectures s'hi poden trobar moltes referències interessants. Algunes són les següents:

G. R. Andrews (1999). *Foundations of Multithreaded, Parallel, and Distributed Programming*. Reading, Massachusetts: Addison-Wesley.

M. Herlihy; N. Shavit (2008). *The Art of Multiprocessor Programming*. Burlington, Massachusetts: Morgan Kaufmann.

D. E. Culler; J. Pal Singh (1999). *Parallel computer architecture: a hardware/software approach*. Burlington, Massachusetts: Morgan Kaufmann.

Resum

En el primer apartat s'han discutit alguns dels factors més importants que cal tenir en compte a l'hora de desenvolupar aplicacions en arquitectures multifil.

En el primer subapartat s'han presentat diferents factors associats al model de programació emprat, i també a l'aplicació que es desenvolupa. S'ha discutit l'impacte de la creació de tasques i de l'assignació d'aquestes als diferents fils d'execució que el sistema facilita. S'ha presentat l'impacte de la gestió de dades compartides entre els diferents fils i mecanismes per a la sincronització, usats per tal de mantenir-hi un accés coherent. I finalment també s'han descrit alguns altres factors importants que cal considerar des del punt de vista de l'aplicació, a l'hora de desenvolupar-la.

En el segon subapartat s'han presentat aspectes que, tot i que també es troben associats al model de programació, estan fortament lligats a l'arquitectura sobre la qual s'executa l'aplicació. Aquests són factors en què el comportament de l'aplicació paral·lela està fortament lligat a l'arquitectura sobre la qual s'està executant. S'han analitzat quatre factors associats al comportament de la jerarquia de memòria: falsa compartició, tècniques de *prefetch*, impacte en el tipus d'arquitectura de memòria i consideracions en l'àmbit d'arquitectures multiprocessador.

Durant el primer apartat s'ha remarcat la gran quantitat de recerca feta en aquest àmbit, ja que és molt elevada. A més a més, s'han facilitat referències a articles de recerca o recursos web que són rellevants des del punt de vista del model de programació emprat per tal de desenvolupar de manera eficient les aplicacions paral·leles.

En el segon apartat s'han estudiat alguns dels factors més importants que cal tenir en compte a l'hora de desenvolupar aplicacions paral·leles que s'executen sobre arquitectures multifil. Com s'ha pogut veure, depenent del tipus de model de programació emprat i de les característiques de l'arquitectura usada, el rendiment obtingut pot variar substancialment.

Primerament s'han analitzat els factors associats al model de programació i l'estil de programació. Juntament amb aquests factors s'ha presentat l'impacte que la manera com es desenvolupa l'aplicació té en el seu rendiment. Per exemple, l'accés massa freqüent a zones de memòria compartida o una assignació de tasques o dades erroni.

A continuació, s'han presentat factors que, tot i poder ser solucionats mitjançant tècniques de programació, es troben lligats a les característiques de l'arquitectura sobre la qual s'executen les aplicacions. En particular, s'ha analitzat l'impacte de les característiques de la memòria cau (falsa compartició o *false sharing*).

Finalment, s'han presentat tres models de programació orientats a desenvolupar aplicacions multifil. Per una banda, s'han analitzat els POSIX Threads. Aquest és un estàndard de programació per a arquitectures paral·leles que va ser presentat el 1995. Per l'altra, s'han presentat dos models de programació proposats per l'empresa Intel: el Cilk i els Thread Control Blocks. El primer és un model molt senzill orientat a facilitar una interfície molt simple i fàcil d'emprar. El segon és un model molt més complex que permet treure un rendiment més elevat.

Aquí també hem remarcat que dins aquest àmbit s'ha fet molta recerca. Per tant, tot i que s'han cobert alguns dels aspectes més representatius, és recomanable mirar d'estendre els coneixements amb les referències bibliogràfiques facilitades.

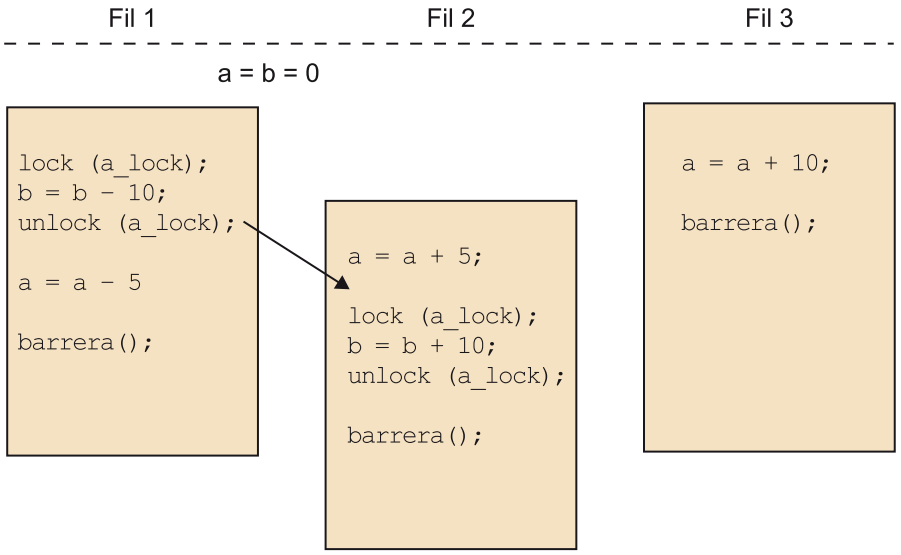
Així mateix, cal tenir present que el món de les arquitectures multifil està en evolució constant. Per tant, és recomanable mantenir-se al dia amb les noves propostes i arquitectures disponibles, atès que conceptes que han sigut vàlids fins un moment específic, poden no ser aplicables a arquitectures o models de programació nous.

Activitats

1. Per tal d'entendre més en detall quins són els reptes de la implementació de mecanismes de sincronització en arquitectures multinucli, feu un estudi de l'article següent:

Villa, O.; Palermo, G.; Silvano, C. (2008). "Efficiency and scalability of barrier synchronization on NoC based many-core architectures". *CASES '08 Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*.

2. Esteneu els diagrames de seqüència presentats en el subapartat 1.1.4.1 "Condicions de carrera en arquitectures multinucli", assumint que hi afegim un tercer fil tal com es mostra en la figura següent:



Assumiu que els fils 1 i 2 s'executen al nucli 0 i el fil 3 al nucli 1. Podeu assumir l'ordre en els accessos que cregueu convenient.

3. En el subapartat 1.1.1 "Definició i creació de les tasques paral·leles" s'ha presentat l'ús de la localitat com un dels factors més importants a l'hora de treure bon rendiment de les aplicacions paral·leles. Escriviu dos diagrames de seqüència en què es vegi la diferència entre l'impacte d'una memòria inclusiva o no inclusiva quan s'invalida una línia de la memòria cau de nivell 1, similars al diagrama de la figura 10.

4. Un article de recerca força interessant relacionat amb els efectes de la programació d'arquitectures multifil i les memòries cau és el presentat per Kwak i altres:

Kwak, H.; Lee, B.; Hurson, A.; Suk-Han, Y.; Woo-Jong, H. (1999). "Effects of multithreading on cache performance". *IEEE Transactions on Computer* (pp. 176-184).

En aquest article acadèmic es presenta un model teòric per tal d'estudiar l'impacte en el rendiment d'aplicacions multifil de les jerarquies de memòria cau. Per tal d'aprofundir en aquest àmbit, es recomana la lectura d'aquesta dissertació.

5. Implementeu un aplicació amb *pthreads* que implementi els funcionaments següents:

a) Un fil crea un vector.

b) Crea dos fils: un inicialitza la primera part del vector amb '12 i el segon la segona part amb '22.

c) El fil creador s'espera que acabin.

d) Crea dos fils d'execució: cada fil afegirà a la seva part del vector $v[i] = v[i - 1]$. El primer fil avisarà amb un senyal condicional al segon per tal que el segon comenci un cop el primer hagi fet la suma en qüestió.

6. S'ha presentat el model de programació d'Intel Cilk. D'altra banda, Intel també proporciona accés al model de programació anomenat Thread Building Blocks (TBB). Enumereu deu 10 diferències principals amb el model Cilk i els POSIX Threads.

7. En l'últim subapartat s'ha introduït l'ús dels *maps* de la biblioteca dels TBB. En aquest exercici es proposa aprofundir en les referències facilitades d'aquest entorn i implementar un codi que sumi dos *maps* diferents i n'insereixi el resultat en un tercer *map*. Tot i que no s'ha introduït, cal que la solució inclogui la creació i inicialització del *map*.

Bibliografia

AMD (2011). *AMD Athlon™ Processor*. Recuperat el 4 de gener del 2012: http://www.amd.com/us/products/desktop/processors/athlon/Pages/AMD-athlon-processor-for-desktop.aspx

Andrews, G. R. (1999). *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley.

Chapman, B.; Huang, L.; Biscondi, E.; Stotzer, E.; Shrivastava, A. G. (2008). "Implementing OpenMP on a High Performance Embedded Multicore MPSoC". *IPDPS*.

Chynoweth, M.; Lee, M. (2009). *Implementing Scalable Atomic Locks for Multi-Core*. Recuperat el 28 de desembre del 2011: http://software.intel.com/en-us/articles/implementing-scalable-atomic-locks-for-multi-core-intel-em64t-and-ia32-architectures/

Culler, D. E.; Pal Singh, J. (1999). *Parallel computer architecture: a hardware/software approach.* Morgan Kaufmann.

Gibbons, A.; Rytte, W. (1988). *Efficient Parallel Algorithms*. Cambridge: Cambridge University Press.

Grunwald, D.; Zorn, B.; Henderson, R. (1993). "Improving the cache locality of memory allocation". *ACM SIGPLAN 1993, conference on Programming language design and implementation*.

Handy, J. (1998). *The cache memory book*. Londres: Academic Press Limited.

Herlihy, M.; Shavit, N. (2008). *The Art of Multiprocessor Programming*. Morgan Kaufmann.

Hewlett-Packard (1994). *Standard Template Library Programmer's Guide*. Recuperat el 9 de gener del 2012: http://www.sgi.com/tech/stl/

Hily, S.; Seznec, A. (1998). "Standard Memory Hierarchy Does Not Fit Simultaneous Multithreading". *Workshop on Multithreaded Execution, Architecture, and Compilation*.

IEEE (2011). *IEEE POSIX 1003.1c standard*. Recuperat el 11 de gener del 2012: http://standards.ieee.org/findstds/interps/1003-1c-95_int/index.html

Intel (2007). *Intel® Threading Building Blocks Tutorial*. Intel.

Intel (2007). *Optimizing Software for Multi-core Processors*. Portland: Intel Corporation - White Paper.

Intel (2011). *Boost Performance Optimization and Multicore Scalability*. Recuperat el 3 de gener del 2012: http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/

Intel (2011). *Intel Cilk Plus*. Recuperat el 21 de desembre del 2011: http://software.intel.com/en-us/articles/intel-cilk-plus/

Intel (2011). *Intel® Array Building Blocks 1.0 Release Notes*. Intel.

Intel (2011). *Nehalem Processor*. Recuperat el 4 de gener del 2012: http://ark.intel.com/products/codename/33163/Nehalem-EP

Intel (2012). *Intel Sandy Bridge - Intel Software Network*. Recuperat el 8 de gener del 2012: http://software.intel.com/en-us/articles/sandy-bridge/

Intel (2012). *Intel® Quickpath Interconnect Maximizes Multi-Core Performance*. Recuperat el 8 de gener del 2012: http://www.intel.com/content/www/us/en/io/quickpath-technology/quickpath-technology-general.html

Jin, R.; Chung, T.-S. (2010). "Node Compression Techniques Based on Cache-Sensitive B +-Tree". *9th International Conference on Computer and Information Science (ICIS)* (pp. 133-138).

Joseph, D.; Grunwald, D. (1997). "Prefetching using Markov predictors". *24th Annual International Symposium on Computer Architecture*.

Kernel.org (2010). *Linux Programmer's Manual*. Recuperat el 3 de gener del 2012: http://www.kernel.org/doc/man-pages/online/pages/man7/pthreads.7.html

Kim, B.-C.; Jun, S.-W. H.-K. (2009). "Visualizing Potential Deadlocks in Multithreaded Programs". *10th International Conference on Parallel Computing Technologies*.

Kourtis, K.; Goumas, G.; Koziris, N. (2008). "Improving the Performance of Multithreaded Sparse Matrix-Vector Multiplication Using Index and Value Compression". *37th International Conference on Parallel Processing*.

Kumar, R.; Farkas, K. I.; Jouppi, N. P.; Ranganathan, P.; Tullsen, D. M. (2003). *Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction* (pp. 81-92).

Kwak, H.; Lee, B.; Hurson, A.; Suk-Han, Y.; Woo-Jong, H. (1999). "Effects of multithreading on cache performance". *IEEE Transactions on Computer* (pp. 176-184).

Lo, J. L.; Eggers, S. J.; Levy, H. M.; Parekh, S. S.; Tullsen, D. M. (1997). "Tuning Compiler Optimizations for Simultaneous Multithreading". *International Symposium on Microarchitecture* (pp. 114-124).

Martorell, X.; Corbalán, J.; González, M.; Labarta, J.; Navarro, N.; Ayguadé, E. (1999). "Thread Fork/Join Techniques for Multi-level Parallelism Exploitation in NUMA Multiprocessors". *13th International Conference on Supercomputing*.

ParaWise (2011). *ParaWise: the Computer Aided Parallelization Toolkit*. Recuperat el 27 de desembre del 2011: www.parallelsp.com/parawise.htm

Philbin, J.; Edler, J.; Anshus, O. J.; Douglas, C.; Li, K. (1996). "Thread scheduling for cache locality". *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*.

Prasad, S. (1996). *Multithreading Programming Techniques*. Nova York: McGraw-Hill, Inc.

Reinders, J. (2007). *Intel Threading Building Blocks*. O' Reilly.

Seznec, A. (1993). "A case for two-way skewed-associative caches". *20th Annual International Symposium on Computer Architecture*.

Valgrind (2011). *Cachegrind: a cache and branch-prediction profiler*. Recuperat el 3 de gener del 2012: http://valgrind.org/docs/manual/cg-manual.html

Villa, O.; Palermo, G.; Silvano, C. (2008). "Efficiency and scalability of barrier synchronization on NoC based many-core architectures". *CASES '08 Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*.