Introducció a les arquitectures paral·leles

Daniel Jiménez-González

PID_00215405

Universitat Oberta de Catalunya

Índex

Introducció

Són moltes les àrees d'investigació, i sobretot, programes d'aplicació real en què la capacitat de còmput d'un únic processador no és suficient. Un exemple el trobem en el camp de la bioinformàtica i, en particular, en la branca de la genòmica, en què els seqüenciadors de genomes són capaços de produir milions de seqüències en un dia. Aquests milions de seqüències s'han de processar i avaluar amb tal de formar el genoma de l'ésser viu que s'hagi analitzat. Aquest processament i avaluació requereixen un temps de còmput i capacitats de memòria molt grans, només a l'abast de computadors amb més d'un *core* (nucli) i una capacitat d'emmagatzemament significativa. Altres àmbits en què ens podem trobar amb aquestes necessitats de còmput són en el disseny de fàrmacs, estudis del cosmos, detecció de petroli, simulacions d'aeronaus, etc.

Durant anys, l'augment de la freqüència dels processadors havia estat la via d'augmentar el rendiment de les aplicacions, d'una manera transparent al programador. No obstant això, encara que la freqüència dels monoprocessadors ha anat augmentant, s'ha observat que aquest augment no es pot mantenir indefinidament. No podem fer moure els electrons i els protons a més velocitat que la de la llum. D'altra banda, aquest augment de freqüència comporta un problema de dissipació de calor, i fa que els monoprocessadors hagin d'incorporar mecanismes de refrigeració d'última generació. Finalment, la millora de la tecnologia i, per tant, la reducció conseqüent de la mida del transistor, ens permeten incorporar més components al monoprocessador sense augmentar-ne l'àrea (més memòria, més *pipelines*, etc.). No obstant això, aquesta disminució de la mida del transistor per a incorporar més millores i capacitats al monoprocessador tampoc no és una solució que es pugui portar a l'infinit (principi d'incertesa de Heisenberg). Aquests problemes tecnològics van contribuir al fet que, per a poder tractar amb problemes tan grans evitant les limitacions tecnològiques, els arquitectes de computadors comencessin a centrar els seus esforços en arquitectures paral·leles.

Principi d'incertesa de Heisenberg

"The position and momentum of a particle cannot be simultaneously measured with arbitrarily high precision. There is a minimum for the product of the uncertainties of these two measurements. There is likewise a minimum for the product of the uncertainties of the energy and time."

El suport maquinari per a processar en paral·lel es pot introduir en diversos nivells; des del més baix nivell, el monoprocessador, al més alt nivell amb les `grids` de multiprocessadors o multicomputadores, en què la xarxa d'interconnexió és Internet. Aquests nivells abasten el paral·lelisme a escala d'instrucció (`ILP`), de dades (`DLP`) i de `threads` (`TLP`).

Nivells de paral·lelisme

ILP: *instruction level parallelism.*
DLP: *data level parallelism.*
TLP: *thread level parallelism*

A escala d'un monoprocessador, el paral·lelisme es va incorporar amb l'execució segmentada* i els processadors superescalars (processadors amb múltiples unitats funcionals d'un mateix tipus que es poden usar alhora). Aquestes dues característiques van permetre que diverses instruccions es poguessin executar al mateix temps en un monoprocessador. Amb la mateixa finalitat d'executar més d'una instrucció alhora, però alleugerint el maquinari necessari, van aparèixer alguns processadors que permetien executar instruccions molt llargues (*very long instruction word*). En aquest cas, el pa-

* ***Pipeline*** **en anglès**

per del compilador és important per a organitzar adequadament les instruccions en el codi.

Una altra manera en la qual es va afegir paral·lelisme a escala de monoprocessador va ser permetent processar més d'una dada alhora amb una única instrucció (instruccions vectorials o instruccions *SIMD*), amb la qual cosa es va començar a explotar el paral·lelisme a escala de dades. Finalment, es va afegir suport maquinari per a explotar el paral·lelisme a escala de *thread*, per a poder executar més d'un *thread* alhora en un monoprocessador amb un cost petit de canvi de context.

A partir de l'any 2000, a causa dels problemes de dissipació de calor i rendiment final dels monoprocessadors, es van començar a explorar noves formes de paral·lelisme dins d'un mateix xip amb els processadors *multicores*. Els processadors *multicores* permeten aprofitar les millores tecnològiques de reducció de mida dels transistors, mantenint o disminuint la freqüència de cada CPU que forma part del *multicore*. La disminució de freqüència permet la reducció del consum energètic, i com a conseqüència, la necessitat de dissipació de calor. D'altra banda, en tenir més d'una CPU es poden aconseguir millors rendiments en les aplicacions.

En qualsevol cas, per a certes aplicacions amb necessitats d'un factor de paral·lelisme de més de 1000x (1.000 vegades més ràpid) respecte als monoprocessadors, cal unir centenars o milers de CPU, que connectades d'alguna manera, puguin treballar eficientment. Això ha portat als grans multiprocessadors i multicomputadors, que són computadors de memòria compartida i distribuïda respectivament, formats per un gran nombre de processadors connectats amb xarxes d'interconnexió molt ràpides. De fet, aquest paral·lelisme, com a tal, ja porta molts anys essent utilitzat en l'àrea d'investigació i d'aplicacions numèriques, i no ha estat només conseqüència de les limitacions dels monoprocessadors.

Finalment, una altra possible via d'adquirir més paral·lelisme és unint diversos sistemes multiprocessador o multicomputadors a través de la xarxa. En aquest cas tenim el que anomenem *grids*, que són multicomputadores.

Objectius

Els objectius generals d'aquest mòdul didàctic són els següents:

1. Conèixer els diferents suports *maquinari* per a explotar paral·lelisme en un monoprocessador.
2. Saber fer programes petits que explotin el suport *maquinari* per al paral·lelisme a escala de dades d'un monoprocessador.
3. Conèixer la taxonomia de Flynn.
4. Conèixer les diferents estratègies de paral·lelització, i saber fer programes paral·lels basats en models de programació basats, al seu torn, en variables compartides i pas de missatges.
5. Saber utilitzar les diferents mètriques per a mesurar el rendiment de programes paral·lels.

En particular, els objectius específics seran que l'estudiant:

1. Sigui capaç d'enumerar i descriure breument els diferents nivells de paral·lelisme que podem explotar en els monoprocessadors.
2. Sigui capaç d'analitzar si un codi es pot o no vectoritzar (explotar el paral·lelisme a escala de dades amb instruccions vectorials o SIMD).
3. Sigui capaç de vectoritzar un codi vectoritzable.
4. Sigui capaç de definir la taxonomia de Flynn, i detallar les diferents subcategories de les arquitectures MIMD.
5. Sigui capaç de detectar possibles problemes de concurrència en un programa que s'ha pensat paral·lelitzar.
6. Sigui capaç de determinar quina és la millor manera de distribuir un programa en tasques per a aprofitar una màquina concreta.
7. Sigui capaç d'analitzar una estratègia de paral·lelització d'un codi mitjançant les diferents mètriques de rendiment.
8. Sigui capaç de modelitzar una estratègia de paral·lelització basant-se en un model bàsic de comunicació.
9. Sigui capaç d'analitzar diferents estratègies de paral·lelització amb tal de minimitzar T_P, i per tant, es maximitzi l'*speed up* (acceleració del programa paral·lel pel que fa al seqüencial).

1. Paral·lelisme en monoprocessadors

En aquest apartat repassarem els mecanismes maquinari que s'han incorporat en els monoprocessadors amb tal d'explotar el paral·lelisme a escala d'instrucció, a escala de `thread` i a escala de dades. Alguns d'aquests mecanismes maquinari permeten explotar el paral·lelisme sense necessitat de cap esforç per part del programador ni del compilador. Altres, en canvi, necessiten el programador o el compilador per a poder explotar-los. Així, per exemple, la segmentació i els processadors superescalars permeten, de manera transparent, explotar el paral·lelisme a escala d'instrucció. En canvi, en el cas dels processadors *very long instruction word* (VLIW), que també tenen com a objectiu explotar el paral·lelisme a escala d'instrucció, necessiten el compilador per a explotar adequadament aquest paral·lelisme. En el cas de voler explotar el paral·lelisme a escala de dades o a escala de `threads`, també és necessari que el compilador o el programador generin un binari o desenvolupin el programa, respectivament.

Els processadors VLIW

Aquests processadors suposen que les instruccions ja estan en l'ordre adequat, i per això necessiten compiladors específics que coneguin l'arquitectura del processador.

1.1. Paral·lelisme a escala d'instrucció

1.1.1. Processador segmentat

La segmentació en un processador consisteix en la divisió de l'execució d'una instrucció en diverses etapes, en què cada etapa normalment es fa en un cicle de CPU. El nombre d'etapes i els noms que reben poden variar d'un processador a un altre.

Execució segmentada

L'origen de l'execució segmentada es pensa que ve o bé del projecte ILLIAC II o del projecte IBM Strech.

Amb la segmentació de l'execució de les instruccions s'aconsegueix augmentar la ràtio d'execució, és a dir, el nombre mitjà d'instruccions que acaben per cicle (*IPC-instructions per cycle*). L'augment de ràtio és gràcies al fet que se sobreposen les etapes d'execució de més d'una instrucció*. D'aquesta manera, podem estar executant en paral·lel tantes instruccions com el nombre d'etapes que tingui l'execució de les instruccions, i després d'emplenar totes les etapes amb tantes instruccions com etapes hi hagi en la segmentació, aconseguirem finalitzar una instrucció per cicle.

*** El paral·lelisme a escala d'instruccions que s'aconsegueix amb la segmentació coincideix amb el nombre d'etapes.**

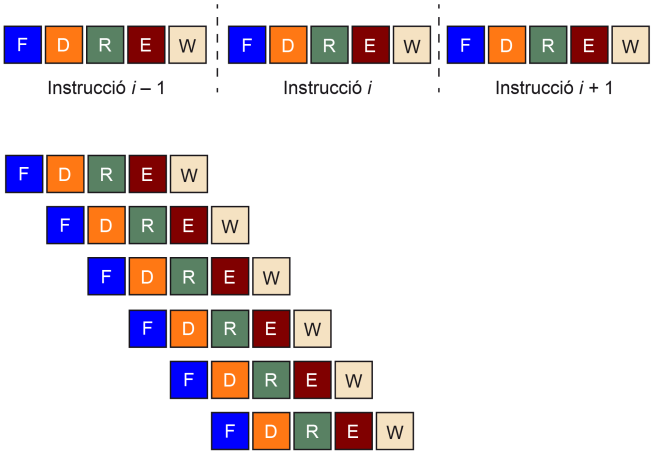
Etapes d'un processador

Les etapes clàssiques d'un processador `RISC` (`reduced instruction set computing`) varien una mica pel que fa a les explicades en aquest mòdul, i són: `instruction fetch`, `decode`, `execute`, `mem` o escriptura en memòria, i `writeback`. Alguns exemples de processadors que tenien aquest tipus de segmentació són: MIPS, SPARC, Motorola 88000, i DLX.

La figura 1 mostra un exemple de la segmentació de 5 etapes: `Instruction fetch` o anar a buscar una instrucció a la memòria, `Instruction decode` o descodificar la instrucció i els operands, `Operand fetch unit` en què es van a buscar els operands, `instruction execution unit` per a l'execució de les operacions, i `writeback`, en què s'escriuen les dades en els registres o memòria. En la part superior de la figura observem l'execució en un processador en què no hi ha segmentació, i per tant, cada instrucció ha d'esperar que la instrucció anterior s'acabi d'executar. En canvi, segmentant l'execució, tal com es mostra en la part inferior de la figura, podem observar que diverses instruccions es poden estar executant en paral·lel; en

concret cinc instruccions. També observem que després de tenir ple el *pipeline* del processador, acabarà una instrucció a cada cicle.

Figura 1. Execució en un processador no segmentat (a dalt) i un processador segmentat (a baix)



Tal com hem comentat, el nombre d'etapes pot variar molt d'un processador a un altre. Per exemple, el processador Pentium 4 tenia 20 etapes; en canvi, el processador Prescott té 31 etapes, i el processador Cell Broadband Engine (BE) té 17 etapes en els SPE. L'etapa més lenta determinarà la freqüència del processador. Per tant, una estratègia que s'ha seguit per a augmentar la velocitat dels processadors és la d'incrementar el nombre d'etapes (la profunditat del *pipeline*), reduint-ne la mida. No obstant això, hi ha alguns inconvenients a l'hora de tenir segmentacions tan profundes.

Cell Broadband Engine

Processador de Sony, Toshiba i IBM, format per un PowerPC i 8 *synergistic processing element* (SPE), que podem trobar en les PlayStation 3.

Què farem quan ens trobem amb un salt? I quina és la destinació del salt?

Hi ha dues opcions: agafar-los o no agafar-los. Normalment, per no parar l'execució i per tant, aprofitar al màxim el paral·lelisme de la segmentació, es prediu el sentit del salt, i la destinació d'aquest. Això es fa amb predictors de salts, implementats en maquinari, que normalment tenen una taxa d'encert en la predicció del salt elevada (més gran que el 90%). Però, en cas de predicció incorrecta de salt, la penalització de desfer totes les instruccions que no s'haurien d'haver iniciat és proporcional al nombre d'etapes en la segmentació de la instrucció.

Aquesta penalització limita el nombre d'etapes que podem posar en la segmentació. És per això que una manera d'aconseguir augmentar la ràtio d'instruccions d'execució, i d'aquesta manera, accelerar els programes, és aconseguir executar més instruccions per cicle. Per a aconseguir-ho, es va afegir maquinari per a poder llançar l'execució de més d'una instrucció per cicle. Aquest tipus de processador es diu *superscalar*. Aquests processadors tenen normalment unitats funcionals duplicades, que permeten executar més d'una instrucció del mateix tipus en el mateix cicle.

Processadors superescalars

Els processadors superescalars permeten llançar l'execució de més d'una instrucció en el mateix cicle.

1.1.2. Processador superescalar

El terme de processador *superscalar* va aparèixer per primera vegada el 1987, en el sentit que el processador tenia unitats funcionals duplicades, i per tant, podia llançar més d'una instrucció alhora. Quaranta anys abans, el computador CDC 6600 ja tenia 10 unitats funcionals en les quals podia estar executant diverses instruccions. No obstant això, aquest processador no podia fer el llançament de més d'una instrucció alhora, i per tant, no es considera un processador superescalar.

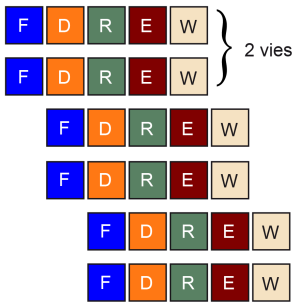
Lectura complementària

Sobre els processadors superescalars podeu llegir: **T. Agerwala; D. A. Wood** (1987). *High Performance Reduced Instruction Set Processors*. IBM T. J. Watson Research Center Technical Report RC12434.

La figura 2 mostra un exemple d'un processador que té l'habilitat de començar a executar (llançar) dues instruccions* a cada cicle. Amb aquest processador superescalar de dues vies o `pipelines` es pot arribar a tenir una ràtio d'execució de fins a 2 instruccions per cicle. No obstant això, hi ha una sèrie de problemes que poden limitar aquesta ràtio: dependència de veritat**, dependència de recursos, i dependència de salts o de procediments/funcions.

*** *Two-way*, en anglès**
**** *True data dependency*, en anglès**

Figura 2. Execució en un processador superescalar amb dues vies



La dependència de veritat és aquella que es produeix quan una instrucció necessita un operand que està calculant una instrucció prèvia al programa, i que està essent executada. Aquestes dependències s'han de resoldre abans de fer el llançament de les dues instruccions en paral·lel per a executar-se. Això té dues implicacions:

1) Hi ha d'haver maquinari dedicat per a la resolució d'aquesta dependència en temps d'execució.

2) El grau de paral·lelisme aconseguit està limitat per la manera en què es codifica el programa.

La primera implicació comporta introduir una complexitat maquinari en el processador per a poder tractar-ho. La segona, un compilador que sigui conscient del maquinari pot millorar el rendiment reordenant les instruccions.

La dependència de recursos prové del fet que no hi ha recursos infinits. El nombre d'unitats funcionals d'un tipus pot limitar el nombre d'instruccions per poder llançar, encara tenint suficients *pipelines*. Per exemple, si tenim un processador superescalar amb dos *pipelines*, però una única unitat funcional de coma flotant, no podrem llançar més d'una instrucció que vagi a la unitat funcional.

La tercera dependència és la dependència per salt, o bé de procediments, que és la mateixa que es va comentar per a la segmentació d'un processador. La destinació del salt només és coneguda en el moment de l'execució, i per tant, continuar executant instruccions sense saber quin és el resultat del salt pot portar a errors. No obstant això, com hem vist, s'usa l'especulació amb predicció per a determinar la decisió del salt i la destinació. En cas d'error, es desfà tot el que s'havia fet. La freqüència d'instruccions de salts en el codi és d'aproximadament una de cada cinc o sis instruccions. Això el fa un factor important per considerar.

Limitacions per dependències

El paral·lelisme a escala d'instrucció en els processadors superescalars està limitat per les dependències de veritat, de recursos i de salt.

Per a sobreposar-se a totes aquestes limitacions per dependències, i augmentar la ràtio de les instruccions executades per cicle, una de les millores en el disseny dels processadors va ser la de reordenar instruccions en execució, per a poder executar en paral·lel totes aquelles que no tinguessin dependències. Noteu que reordenen les instruccions per a executar-les fora d'ordre, però, en el moment d'haver d'escriure en registres o memòria es fa en ordre. Aquests processadors s'anomenen *fora d'ordre**. Fins al moment els processadors que havíem estat mirant eren en ordre**.

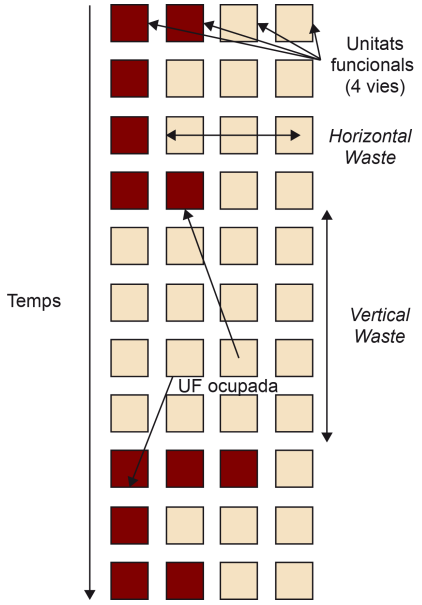
* `Out-of-order`, en anglès
** `In-order`, en anglès

Malgrat aquestes millores, ens podem trobar que el *pipeline* del processador queda buit durant diversos cicles, o que algunes de les nostres unitats funcionals no s'estan usant durant un cicle concret. Aquests casos es coneixen, en anglès, com a *vertical waste* i *horizontal waste*, respectivament. La figura 3 mostra les quatre vies (una per columna) d'execució en un processador superescalar. Cada fila representa els cicles d'execució en aquest processador. Cada requadre per fila representa una unitat funcional, i que estigui acolorida significa que està ocupada. Com podem observar, s'està desaprofitant el processador tant verticalment com horitzontalment (quadrats no acolorits). D'aquesta manera, el processador superescalar no s'està aprofitant al màxim.

Processador fora d'ordre

Els processadors fora d'ordre són aquells processadors que reordenen l'execució de les instruccions amb tal d'evitar dependències. El primer microprocessador amb execució fora d'ordre va ser el POWER1, d'IBM.

Figura 3. `Waste` vertical i horitzontal en un processador superescalar



Aprofitar millor o pitjor el processador dependrà, en part, del maquinari que tinguem dedicat a la reordenació de les instruccions en temps d'execució. Aquest maquinari té un cost, que és funció quadràtica del nombre de *pipelines* del processador, i per tant

pot tenir un cost elevat. Per al cas típic d'un *four-way* té un cost el 5% al 10% del maquinari del processador.

Una opció alternativa és deixar al compilador el treball de buscar les instruccions que es poden executar alhora en el processador. El compilador organitzarà el binari de tal manera que aquelles instruccions que es poden executar alhora estaran consecutives al programa. Així, el processador les podrà llegir d'una vegada, com si fos una instrucció llarga, formada per diverses instruccions, i enviar-les a executar sense haver de mirar si hi ha dependència o no. Aquests processadors es coneixen com a *very long instruction word* o VLIW.

1.1.3. Processador *very long instruction word* (VLIW)

El concepte de `VLIW` va ser usat per primera vegada amb la sèrie TRACI creada per Multiflow i posteriorment com una variant de l'arquitectura Intel IA64. Aquests processadors tenen l'avantatge d'estalviar-se el maquinari dedicat a la reordenació en temps d'execució de les instruccions, i depenen molt del tipus d'optimitzacions que faci el compilador. Normalment també disposen d'instruccions dedicades per a controlar el flux d'execució del programa. D'altra banda, no disposen de predicció de salts basada en la història de l'execució del programa, i els és molt difícil detectar situacions de fallades d'accés en la memòria cau, i per tant s'ha de parar l'execució del programa. Amb això l'ordenació que va fer el compilador força que l'execució de moltes instruccions es pari.

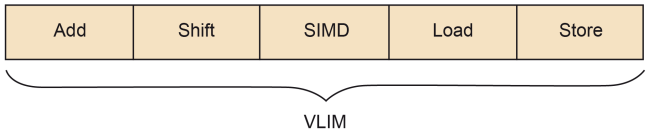
Processadors VLIW

Els processadors VLIW no tenen suport maquinari per a detectar paral·lelisme entre les instruccions en temps d'execució, però depenen totalment de la compilació feta.

La figura 4 mostra l'esquema d'una instrucció llarga formada per unes 5 instruccions: una suma, un desplaçament binari, una instrucció `SIMD`, una operació de lectura i una d'escriptura. El codi binari generat pel compilador específic ha d'estar organitzat en paraules de 5 instruccions que es puguin executar en paral·lel. En cas de no poder obtenir 5 instruccions, s'emplenaran amb `NOP`*.

* ***No operation*, en anglès**

Figura 4. VLIW formada per 5 instruccions



1.2. Paral·lelisme a escala de *threads*

Fins al moment hem vist diverses maneres d'explotar el paral·lelisme a escala d'instrucció (ILP). Aquest paral·lelisme, no obstant això, es perd en el moment en què ens trobem amb errors d'accés als diferents nivells de memòria cau. Això és a causa que el processador ha de parar l'execució de noves instruccions fins que la dada (línia de memòria cau) es rebi. Una solució per a aconseguir que el processador continuï executant instruccions és que pugui executar un altre programa o `thread`, emmascarant aquesta situació de bloqueig. Això és el que es diu, en anglès, *on-chip multithreading*.

ILP

El paral·lelisme a escala de `thread` ajuda a explotar millor el paral·lelisme a escala d'instrucció ocultant els errors d'accés a memòria.

Hi ha diverses aproximacions al *on-chip multithreading*: *multithreading* de gra fi, *multithreading* de gra gruixut, i *simultaneous multithreading*, que veurem a continuació.

1.2.1. *Multithreading* de gra fi

Aquest tipus de paral·lelisme a escala de *threads* intenta ocultar els bloquejos del processador fent una execució en *round robin* de les instruccions de *threads* diferents, en cicles consecutius.

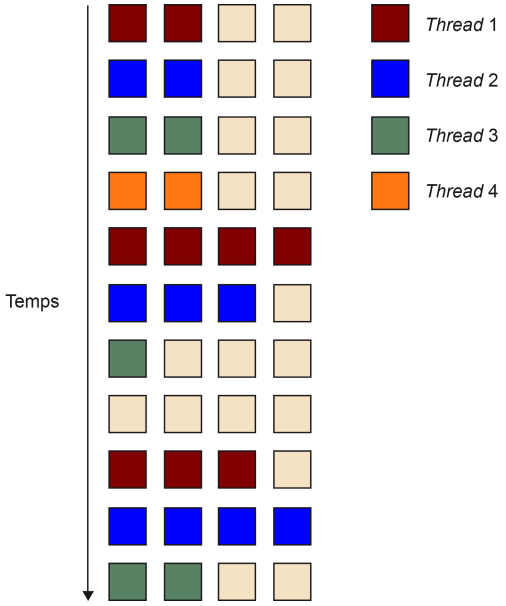
Round robin

És la política d'assignació de cicles del processador en la qual es donen un nombre fix de cicles a un *thread* darrere d'un altre.

D'aquesta manera, si tenim tants *threads* executant-se com cicles de bloqueig que necessita, per exemple, un error d'accés a memòria, podrem reduir la probabilitat que el processador es quedi sense fer res en aquests cicles.

La figura 5 mostra com 4 *threads*, amb colors diferents, es van executant en les diferents unitats funcionals del processador superescalar. A cada cicle d'execució, es canvia de *thread*.

Figura 5. *Multithreading* amb gra fi



Per a poder fer aquesta execució de diversos *threads*, independents entre ells, cadascun necessita un conjunt de registres associats. Així, cada instrucció té associada informació per a saber quin banc de registres cal usar a cada moment. Això fa que el nombre de *threads* que es poden executar alhora (en *round robin*) dependrà del maquinari de què disposem.

Threads independents

Si els *threads* no fossin independents haurien de compartir variables en memòria i podrien necessitar algun mètode de sincronització per a poder accedir-hi si alguns haguessin d'actualitzar la memòria.

Un altre motiu de bloqueig del processador és la decisió que han de prendre quant als salts que es trobin. Això dificulta saber si es fa o no el salt, i cap a on. Una solució és tenir tants *threads* que es puguin executar en *round robin* com etapes en el *pipeline*, amb la qual cosa sabríem que sempre s'estaria executant un *thread*. Però, tal com hem comentat, això representaria, a escala de suport maquinari, molts bancs de registres.

Bancs de registres

El nombre de bancs de registres limita el nombre de *threads* als quals es pot donar suport maquinari en gra fi.

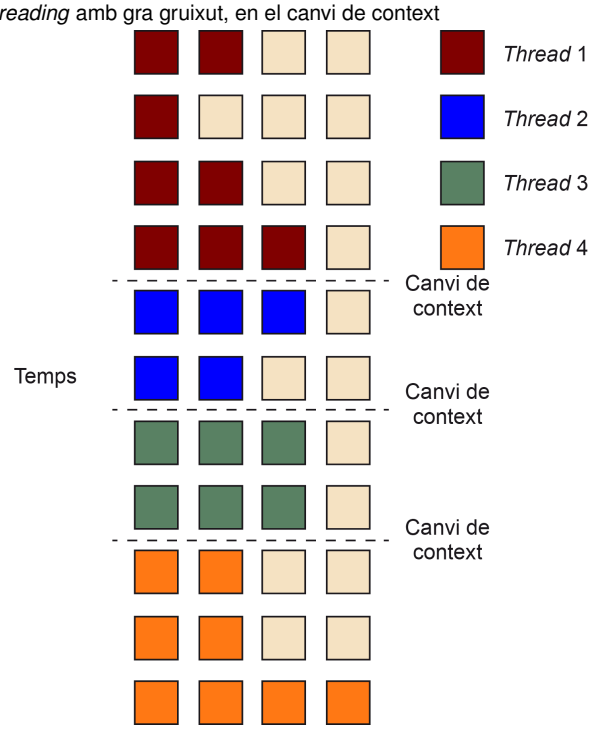
1.2.2. *Multithreading* de gra gruixut

Aquesta aproximació consisteix a poder executar més d'una instrucció d'un *thread* en cicles consecutius. La figura 6 mostra una execució de 4 *threads* en un sistema *multithreading* amb gra gruixut.

Multithreading de gra gruixut

En un processador amb suport *multithreading* de gra gruixut els *threads* es poden executar més d'un cicle consecutiu, i es redueix així la necessitat de molts bancs de registres.

Figura 6. *Multithreading* amb gra gruixut, en el canvi de context



En aquest cas, el que normalment es fa és que un *thread* continua l'execució d'instruccions fins que es produeix un bloqueig a causa d'un salt, un conflicte de dades, etc. Amb aquesta estratègia de deixar executar més d'un cicle a un *thread* no són necessaris tants *threads* actius com passava amb el *multithreading* de gra fi amb tal d'aprofitar al màxim el processador. En contrapartida, com sempre s'espera que hi hagi un bloqueig per a canviar de *thread*, els cicles que es necessitin per a adonar-se del bloqueig i canviar de *thread* es perdran. En qualsevol cas, si tenim un nombre suficient de *threads* actius, podrem aconseguir més bon rendiment que amb el gra fi.

Una altra possibilitat per a reduir encara més el nombre de cicles en els quals el processador està bloquejat, és que es faci el canvi de *thread* cada vegada que una instrucció pugui provocar un bloqueig, i no quan aquest es produeixi.

Amb aquest tipus de *multithreading* de gra gruixut també pot tenir més sentit buidar el *pipeline* cada vegada que canviem de *thread*. D'aquesta manera no hem de tenir la informació sobre quin banc de registres s'ha d'usar al llarg de tota l'execució d'una instrucció.

1.2.3. *Simultaneous multithreading*

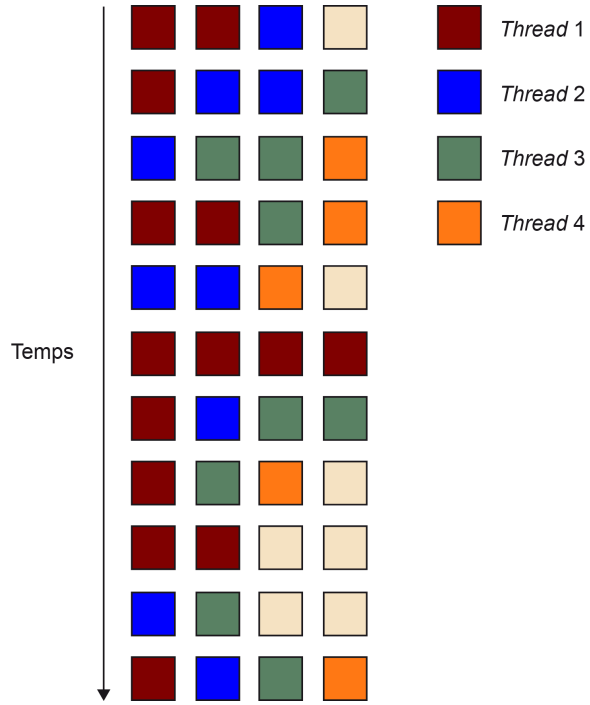
Processadors amb *Simultaneous multithreading*

Els processadors amb *simultaneous multithreading* redueixen el desaprofitament horitzontal de les unitats funcionals, permetent que dos *threads* diferents es puguin executar en el mateix cicle de processador.

Aquesta última aproximació intenta reduir l'*horizontal waste* en els processadors superescalars amb *multithreading*. Es pot considerar com un refinament del gra gruixut, de tal manera que si en un cicle del processador una instrucció d'un *thread* es bloqueja, una instrucció d'un altre *thread* es pot usar per a mantenir el processador i totes les unitats funcionals ocupades. També és possible que una instrucció d'un *thread* pugui quedar bloquejada perquè hi ha un nombre limitat d'unitats funcionals d'un tipus. En aquest cas també es podria agafar una instrucció d'un altre *thread*.

La figura 7 mostra un exemple d'execució de 4 *threads* en un sistema *Simultaneous Multithreading*. En aquest cas, a cada cicle d'execució pot haver instruccions de diferents *threads*.

Figura 7. *Simultaneous multithreading*



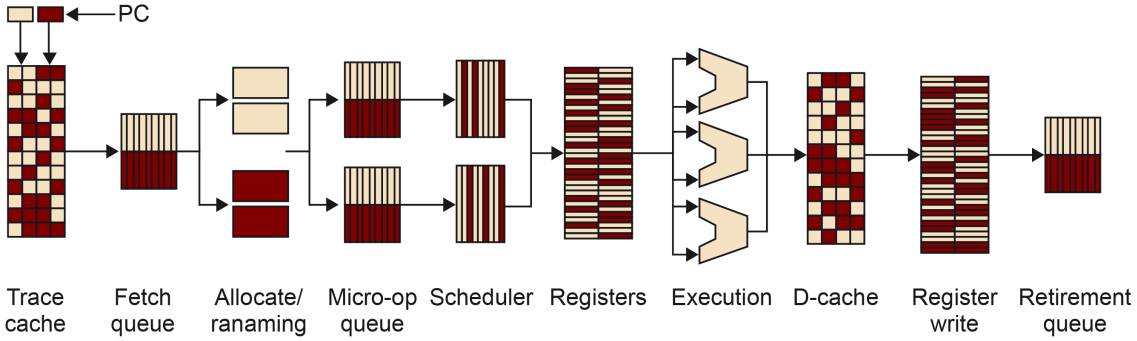
Multicore

Es tracta d'un xip amb més d'un *core* a dins.

El primer processador que va incorporar el *simultaneous multithreading*, conegut com a *hyperthreading*, va ser el Pentium 4. Aquest tipus de suport ha tingut continuïtat en altres processadors, com per exemple l'Intel Corei7, un *multicore* que incorpora en cadascun dels seus *cores* el *simultaneous multithreading*.

Per al sistema operatiu, un processador amb *simultaneous multithreading* és com un processador amb dos *cores*, que comparteixen memòria cau i memòria. En canvi, en maquinari, s'han de preveure quins recursos es comparteixen i com s'han de gestionar. Intel preveiea quatre estratègies diferents. Per a comentar-les ens basarem en l'estructura bàsica del *pipeline* d'aquests processadors, que es mostra en la figura 8.

Figura 8. Estructura del *pipeline* d'un processador Pentium 4 amb *hyperthreading*



Aquestes estratègies són:

1) Duplicació de recursos: el comptador de programa, i també la taula de mapatge dels registres (*eax*, *ebx*, etc.) i controlador d'interrupció, han de ser duplicats.

2) Particionament de recursos compartits: particionar permet distribuir els recursos entre els *threads*, de tal manera que uns no interfereixen en els altres. Això permet evitar *overheads* de control, però també, que alguns d'aquests recursos quedin sense ser utilitzats en algun moment. Un exemple de recurs particionat és la cua d'instruccions que es llançaran per mitjà de dos *pipelines* separats. Un per cada `thread`.

3) Compartició total dels recursos compartits: en aquest cas s'intenta solucionar el desavantatge de tenir un dels recursos particionats poc aprofitat. Per exemple, un *thread* lent d'execució podria omplir la seva part de cua d'instruccions per executar, fent que un altre *thread* més ràpid no pugui aprofitar que es pot executar més ràpid, per a inserir més instruccions en la cua. En el cas de la figura, les etapes de `renaming` són totalment compartides.

4) Compartició de compromís*: en aquest cas no hi ha una partició fixa, sinó que es van demanant recursos i adquirint-los de manera dinàmica, fins a un cert màxim. En la figura, el *scheduler* és dinàmicament compartit, amb un *threshold* màxim.

*** *Threshold*, en anglès**

1.3. Paral·lelisme a escala de dades

El paral·lelisme a escala de dades es refereix bàsicament a la possibilitat d'operar sobre dues o més dades amb una única instrucció; en anglès es refereixen a instruccions *single instruction multiple data* (SIMD).

SIMD

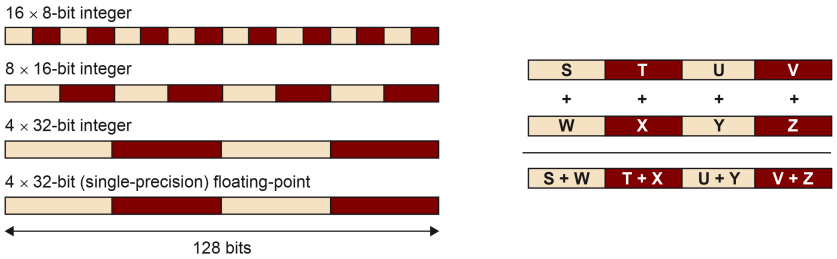
Un monoprocessador amb instruccions SIMD és capaç de fer una mateixa operació, en paral·lel, amb més d'una dada alhora.

El suport maquinari necessari per a poder executar instruccions SIMD inclou tenir registres més grans, busos cap a memòria que permetin l'accés a dades de la mida dels registres, i unitats funcionals que suportin operar amb més d'una dada alhora. A més, depenent de la semàntica de la instrucció es podrà operar amb més o menys operands.

La mida del registre, de 128 i 256 bits, es reparteix equitativament entre els elements que indiqui la semàntica de la instrucció. La figura 9 mostra la típica distribució dels bits del registre segons la semàntica de la instrucció, per a un registre de 128 bits.

Les operacions que s'apliquen a cada element només afecten aquests elements, tret que la semàntica de la instrucció permeti que s'operi entre dos elements consecutius d'un mateix registre. La figura 9 mostra, a la dreta, una operació de sumar sobre dos registres vectorials, cadascun dels quals té 4 elements de tipus enter de 32 bits.

Figura 9. Instruccions vectorials



Per a explotar aquest tipus de paral·lelisme es necessita que el compilador o el programador utilitzin les instruccions SIMD. El compilador, de vegades, és capaç de detectar patrons de codis en els quals es poden utilitzar aquestes instruccions. No obstant això, la majoria de les vegades són els programadors els que han de fer una anàlisi de les dependències existents al programa, i en funció d'això, veure si es poden usar aquestes instruccions.

Web complementària

El compilador `gcc` té una pàgina dedicada (http://gcc.gnu.org/projects/tree-ssa/vectorization.html) que indica quines millores ha fet en la utilització d'instruccions SIMD de manera automàtica.

Des que van aparèixer els primers processadors amb instruccions SIMD el 1997, l'evolució ha estat progressiva, i en general, els processadors han anat incorporant unitats funcionals de tipus SIMD. Les primeres a aparèixer van ser les conegudes com a `MMX` (Intel). Aquestes instruccions solament operaven amb enters de 32 bits, i no suportaven instruccions de tipus *float*. Els registres que s'usaven eren els del banc de registres reals, per la qual cosa no es permetia barrejar operacions de coma flotant i `MMX`.

Les següents van ser les 3D-Now d'AMD, que sí que suportaven els *floats*. A partir d'això van començar a aparèixer les extensions SSE, amb la incorporació de *floats*, l'ampliació del nombre de registres i de bits fins a 128 bits. Els processadors Sandy Bridge, apareguts el 2011, disposen de les extensions AVX que poden operar fins a 256 bits.

La taula 1 mostra la relació d'any, empresa i característiques més importants de l'aparició de les instruccions SIMD en els processadors des de 1997 fins a 2011, any d'aquesta publicació.

Taula 1. Relació d'instruccions SIMD d'Intel i AMD, i les seves característiques principals.

Companyia	SIMD extension	Característiques
Intel (1997)	MMX,MMX2	64-bit, 8 vectors, enters, no barreja *MMX-floats*
AMD (1998)	3DNow	64-bit, 8 vectors, FP i enters, barreja *MMX-float*
Intel (1999)	SSE/SSE1	128-bit, banc de registres específic, FP i enters
Intel (2001)	SSE2	S'eviten totalment els registres MMX
Intel (2004)	SSE3	Instruccions per a treballar horitzontalment amb els registres
Intel (2006)	SSSE3	Instruccions vectorials més complexes
Intel (2007)	SSE4	Instruccions específiques sobre multimèdia
Intel (2011)	AVX (*advanced vector extensions*)	Extensió a registres de 256-bit
Intel (per aparèixer)	VPU (*wide vector processing units*)	Extensió a registres de 512-bit

1.3.1. Exemples de vectorització

En aquest apartat analitzarem alguns codis amb l'objectiu de saber determinar si es pot o no explotar el paral·lelisme a escala de dades i, per tant, si es poden o no utilitzar les instruccions SIMD (vectorització d'un codi).

Lectura complementària

Manual d'Intel d'optimitzacions per a IA32, capítols 4, 5 i 6.

Primer analitzarem un codi que, en no tenir dependències de veritat entre les seves instruccions, es pot vectoritzar (codi 1.1). Després veurem un altre codi que no es pot vectoritzar (codi 1.3) a causa que té dependències de veritat entre les instruccions que no ho permeten. I finalment veurem dos codis que es poden vectoritzar: un que, fins i tot tenint dependències de veritat entre les instruccions, la distància entre les instruccions en execució permet que es pugui vectoritzar (codi 1.4), i un altre que es pot vectoritzar després de reordenar les instruccions del codi original (codi 1.6).

1.3.2. Codi vectoritzable sense dependències

El primer d'aquests (codi 1.1) és un codi que no té dependències de veritat (una lectura d'una dada després de l'escriptura d'aquesta dada), i per tant es pot vectoritzar.

Dependència de veritat

true data dependency: una lectura d'una dada després d'una escriptura sobre la mateixa dada.

```
1 char A[16], B[16], C[16];
2
3 for (i=0; i<16; i++)
4     A[i] = B[i] + C[i];
```

Codi 1.1: Suma de vectors

El codi vectoritzat, amb instruccions SSE2 de l'Intel, es mostra en el codi 1.2. Per a programar amb les SSE2 utilitzarem les funcions intrínseques (`_mm_load_si128`, `_mm_add_epi8`, `_mm_store_si128`) i els tipus que ens ofereix aquesta extensió. El tipus de les variables enteres, perquè es desin en registres vectorials, és `__m128i`. Per a això hem d'incloure la capçalera `emmintrin.h`. En aquesta implementació ens hem assegurat que els vectors `A`, `B` i `C` estiguin alineats a 16 bytes. Això és perquè les operacions de memòria que hem utilitzat (lectura: `_mm_load_si128`, i escriptura: `_mm_store_si128`) necessiten aquesta alineació o per contra es produirà un accés no alineat no permès. Si no poguéssim assegurar aquest alineament en la declaració de les variables, es poden utilitzar les funcions intrínseques `_mm_loadu_si128` per a les lectures no alineades (*o* de *unaligned*), i `_mm_storeu_si128` per a les escriptures no alineades. Aquestes operacions són significativament més lentes que les operacions de lectura i escriptura normals. L'operació `_mm_add_epi8` indica que es farà la suma vectorial (`_add`) d'un paquet d'elements enters (`_epi`) de mida 8 bits (`_epi8`).

Funcions intrínseques

Són funcions que es tradueixen a una o poques instruccions d'assemblador. No hi ha salt a un codi d'una funció, ni pas de paràmetres en la traducció a assemblador.

Lectura i escriptura

Els accessos a memòria en les operacions de lectura i escriptura en les extensions d'Intel han de ser alineades a 16 bytes. En cas contrari s'han d'usar operacions no alineades de lectura i escriptura.

```
1 #include <emmintrin.h>
2 ...
3
4 char A[16] __attribute__ (( __aligned__ (16)));
5 char B[16] __attribute__ (( __aligned__ (16)));
6 char C[16] __attribute__ (( __aligned__ (16)));
7
8 __m128i a, b, c;
9
```

```
10 ...
11 a = _mm_load_si128((__m128i*) &A[i]);
12 b = _mm_load_si128((__m128i*) &B[i]);
13 c = _mm_add_epi8(a, b);
14 _mm_store_si128((__m128i*)&C[i], c);
```

Codi 1.2: Suma de vectors amb instruccions SIMD

En operar amb elements de 8 bits, podem desar fins a 16 elements de tipus *char* en cada registre vectorial. Això implica que fent l'operació `_mm_add_epi8` aconseguim fer totes les operacions que hi havia en el bucle, i per tant, el bucle del codi no vectorial desapareix.

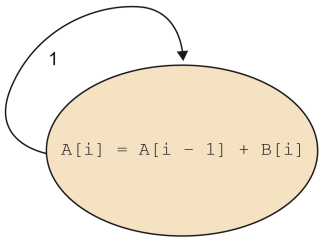
Codi no vectoritzable amb dependències

El codi 1.3 no es pot vectoritzar. Aquest codi té una dependència de veritat, que es mostra en la figura 10, que no permet que sigui vectoritzat. En el graf de dependències el sentit de l'aresta indica l'origen i la destinació de la dependència, i l'etiqueta de l'aresta és la distància en nombre d'iteracions de la dependència. En l'exemple hi ha una dependència de veritat de distància 1 amb origen i final com l'única instrucció que té el cos del bucle. L'origen és l'escriptura `A[i]=...` ja que es produeix abans en l'espai d'iteracions del bucle, i la destinació és la lectura `A[i-1]`. La distància de la dependència es calcula com a $d = (i - (i - 1)) = 1$. Aquesta distància és més petita que el nombre d'elements que podem carregar en un registre vectorial, i per tant, no podem operar amb tots aquests elements alhora sense trencar aquesta dependència.

```
1 char A[16], B[16], C[16];
2 for (i=1; i<16; i++)
3     A[i] = A[i-1] + B[i];
```

Codi 1.3: Suma de vectors amb dependència de veritat

Figura 10. Graf de dependències del codi 1.3



Codi vectoritzable amb dependències a una distància suficient

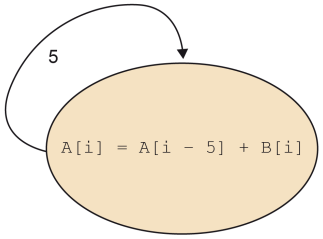
El codi 1.4 és un codi que també té una dependència de veritat. No obstant això, la distància de la dependència ($d = (i-(i-5)) = 5$) és més gran que nombre d'elements que cap en un registre vectorial (4 enters de 32 bits). El graf de dependències es mostra en la figura 11.

```
int A[N], B[N];

for (i=16; i<N; i++)
    A[i] = A[i-5] + B[i];
```

Codi 1.4: Suma de vectors amb una dependència de veritat amb distància més gran que el nombre d'elements sobre els quals s'opera

Figura 11. Graf de dependències del codi 1.4



El codi vectoritzat amb instruccions SSE2 de l'Intel es mostra en el codi 1.5.

Vectoritzar codi

Un codi es pot arribar a vectoritzar malgrat tenir dependències de veritat. Per exemple, quan la distància de les dependències de veritat és més gran que el nombre d'elements que caben en el registre vectorial.

```
int A[N] __attribute__ (( __aligned__ (16)));
int B[N] __attribute__ (( __aligned__ (16)));
...

for (i=16; i<N-3; i=i+4) {
  __m128i a, a5, b;
  a5 = _mm_loadu_si128((__m128i*) &A[i-5]);
  b = _mm_load_si128((__m128i*) &B[i]);
  a = _mm_add_epi32(a5, b);
  _mm_store_si128((__m128i*)&A[i], a);
}

for (; i<N; i++)
    A[i] = A[i-5] + B[i];

...
```

Codi 1.5: Vectorització del codi 1.4

Noteu que en el codi vectorial 1.5 hem hagut d'usar la lectura no alineada (`_mm_loadu_si128((__m128i) &A[i-5])`) fins i tot havent alineat els vectors* `A` i `B` en el seu primer element amb `__attribute__ (( __aligned__ (16)))`. Això és a causa que hem alineat el seu primer element a 16 bytes, però $i - 5$ està a una distància $i\%4 + 1$, que no està alineada. També és necessari fer un epíleg** amb tal de fer les iteracions que no s'hagin efectuat de manera vectorial.

*** Un bucle vectoritzat pot necessitar accessos alineats i no alineats a un vector si hi ha accessos desplaçats respecte a la variable d'inducció del bucle.**
**** Normalment és necessari fer epílegs dels codis vectoritzats. Això és perquè el nombre d'iteracions no és múltiple del nombre d'elements per registre vectorial.**

Codi vectoritzable amb dependències salvables

El codi 1.6 és un codi que té una dependència de veritat i dues antidependències* (escriptura després de lectura). La figura 12 mostra aquestes dependències entre les dues instruccions del bucle. Les antidependències s'indiquen amb una aresta amb una línia que la talla.

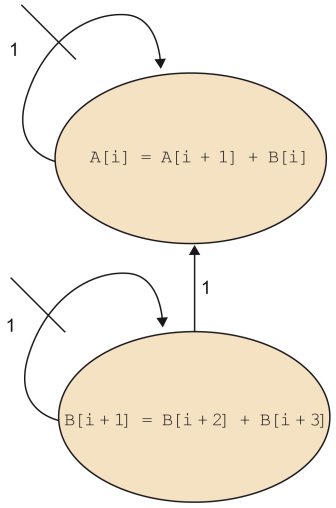
*** Una escriptura d'una dada després d'una lectura sobre la mateixa dada.**

```
int A[N], B[N];

for (i=0; i<(N-3); i++) {
  A[i] = A[i+1] + B[i];
  B[i+1] = B[i+2] + B[i+3];
}
```

Codi 1.6: Suma de vectors amb dependències de veritat que es poden reorganitzar

Figura 12. Graf de dependències del codi 1.6



Les antidependències no afecten la vectorització. En canvi, la dependència de veritat (lectura de `B[i]` en la primera instrucció del bucle després de l'escriptura de `B[i+1]`, en la segona instrucció) sí que trenca la possible vectorització.

No obstant això, la dependència de veritat existent ens la podríem evitar si canviem l'ordre seqüencial de les instruccions. Per exemple, si fem primer les 4 iteracions de la primera instrucció, i posteriorment, les 4 iteracions de la segona instrucció, tal com vam mostrar en el codi 1.7.

```
int A[N], B[N];

for (i=0; i<(N-3)-3; i+=4) {
  A[i]   = A[i+1] + B[i];
  A[i+1] = A[i+2] + B[i+1];
  A[i+2] = A[i+3] + B[i+2];
  A[i+3] = A[i+4] + B[i+3];
  B[i+1] = B[i+2] + B[i+3];
  B[i+2] = B[i+3] + B[i+4];
  B[i+3] = B[i+4] + B[i+5];
  B[i+4] = B[i+5] + B[i+6];
}

for (; i<(N-3); i++) {
  A[i] = A[i+1] + B[i];
  B[i+1] = B[i+2] + B[i+3];
}
```

Codi 1.7: Suma de vectors amb dependències de veritat una vegada reorganitzat el codi

Amb aquesta disposició de les instruccions en el nou bucle podríem vectoritzar els dos grups de quatre instruccions escalars del codi, sense trencar la dependència de veritat.

Activitat

Feu la vectorització del codi 1.7 amb funcions intrínseques d'SSE2.

1.4. Limitacions del rendiment dels processadors

Hi ha dos factors que limiten el rendiment final que es pot obtenir amb els processadors. Un és l'accés a memòria i l'altre és el consum energètic i la llei de Moore.

1.4.1. Memòria

La diferència entre la velocitat a la qual el processador pot processar dades i aquella a la qual la memòria pot subministrar aquestes dades és un factor que limita el rendiment potencial de les aplicacions. En concret, tant la latència com l'amplada de banda d'accés a memòria poden influir en el rendiment dels programes.

La latència és el temps que triga a obtenir-se una dada de memòria. I l'amplada de banda és la quantitat de bytes per segon que la memòria pot oferir.

Latència

És el temps necessari per a obtenir una dada de memòria. Amplada de banda: quantitat de bytes que la memòria pot subministrar per unitat de temps (segons).

La introducció de la memòria cau, una memòria molt més ràpida amb una latència molt petita, i una amplada de banda gran, va ser una manera d'alleujar la diferència de velocitat entre processadors i memòria. Aquestes memòries tenen l'objectiu de tenir les dades més usades prop del processador amb tal d'evitar accessos a la memòria llunyana i lenta. Quan ocorre que estem aprofitant una dada que es troba en la memòria cau, es diu que s'ha produït un encert en la memòria cau, i per tant, que s'està explotant la localitat temporal. En cas contrari, es diu que s'ha produït un error en l'accés a la memòria cau. Cada vegada que tinguem un encert, ens estarem aprofitant de la baixa latència de la memòria cau.

D'altra banda, a més de la dada que es demana, es produeix una transferència de totes les dades que es troben en la mateixa línia de memòria cau. Amb això s'intenta apropar al processador les dades a les quals possiblement s'accedirà en un futur proper. En cas que realment es referenciïn més endavant, es dirà que s'està explotant la localitat espacial. En aquest cas, augmentar la mida de les línies de memòria cau pot contribuir a millorar la localitat espacial, a més d'ajudar a explotar millor l'amplada de banda que pugui oferir la memòria.

Línia de memòria cau

Una línia de memòria cau és la unitat bàsica de moviment entre memòria i memòria cau.

En qualsevol cas, hi ha aplicacions que poden aprofitar millor la localitat temporal que altres, fet que augmenta la tolerància de memòries amb amplades de banda baixes. De la mateixa manera, les polítiques de reemplaçament d'una línia de memòria cau per una altra, l'associativitat de la memòria cau (*full associative*, *n-set associative*, *direct mapped*, etc.), etc. poden afectar la localitat temporal d'una aplicació.

Altres mecanismes per a millorar el rendiment final d'un programa en un processador, des del punt de vista de la memòria, són ocultar d'alguna manera la latència existent en els accessos a memòria. Un d'aquests mecanismes és programar diversos *threads* (*multithreading* de gra gruixut), de tal manera que en processadors amb suport per a

*** *Outstanding requests*, en anglès**

executar més d'un *thread*, si un d'aquests *threads* es para per un error de memòria cau, l'altre pot continuar. Perquè realment es pugui superposar s'ha de complir que la memòria pugui suportar diverses peticions de memòria en curs*, i el processador pugui fer un canvi de context en un cicle (com el cas dels processadors de *multithreading* de gra gruixut).

Una altra manera d'ocultar la latència d'accés és fer `prefetch` de les dades que es necessiten. Hi ha processadors que són capaços de fer `prefetch` a escala de maquinari quan detecten patrons d'accessos a adreces consecutives (o amb una certa distància) de memòria. Hi ha compiladors que són capaços d'avançar les lectures, pel que fa a la instrucció que la necessita, amb tal de no pagar la latència d'accés. El programador i el compilador també poden inserir instruccions de `prefetch` si és que el processador en disposa.

`Prefetch` en el compilador GCC

El programador disposa de `__builtin_` de `prefetch` per al compilador de C de GNU (GCC).

En qualsevol cas, totes dues maneres d'ocultar la latència tenen els seus inconvenients. Per exemple, el fet de tenir més d'un *thread* executant-se i demanant dades de memòria pot requerir un augment en l'amplada de banda que hauria d'oferir la memòria. D'altra banda, el `prefetch` sobre registres implica que hem de disposar de més registres per a poder desar les dades, i a més, fer els càlculs que volíem fer, o en cas contrari, necessitaríem demanar una altra vegada les dades.

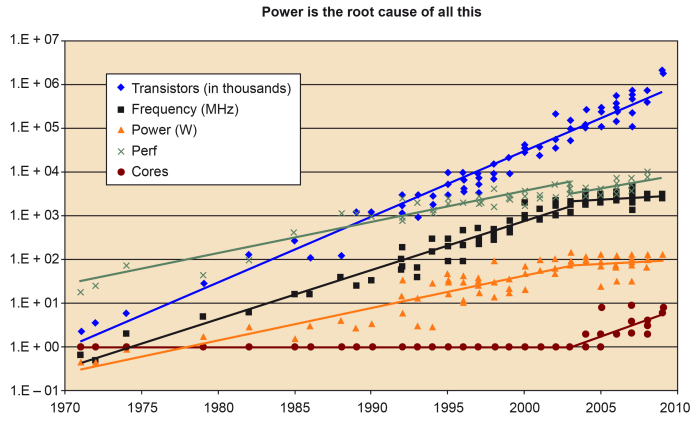
1.4.2. Llei de Moore i consum dels processadors

El 1965, Gordon Moore va fer l'observació següent:

> "The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years. That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000".

I més tard, el 1975, va fer una revisió de la ràtio d'augment en la complexitat dels circuits. En aquesta ocasió, Moore va predir que aquesta complexitat s'augmentaria cada 18 mesos. Aquesta corba és la que coneixem com a llei de Moore, i observant la figura 13, ens adonem que realment s'ha complert.

Figura 13. Evolució del nombre de transistors segons la llei de Moore



Font: K. Olukotun, L. Hammond, H. Sutter, B. Smith, C. Batten, i K. Asanovic.

No obstant això, encara que el nombre de transistors que podem integrar en un circuit augmenta cada any, hi ha un consum energètic màxim que el xip pot suportar, que limita el creixement infinit del nombre dels transistors en un *core* per a augmentar l'ILP i la velocitat d'aquest. De fet, es va arribar a un punt en què aquest consum començava a afectar negativament el rendiment de les aplicacions. Aquesta pèrdua de rendiment va fer que, a partir de l'any 2000 aproximadament, els dissenyadors de processadors comencessin a optar per no intentar incrementar l'ILP d'un *core* o la seva velocitat, i en canvi, augmentar el nombre de *cores* dins d'un xip (*multicores*).

2. Taxonomia de Flynn i altres

En l'apartat anterior hem fet una revisió de les principals innovacions de maquinari que s'han anat fent a escala d'un monoprocessador.

En aquest apartat es descriu la taxonomia de Flynn, que és una classificació de les màquines en funció del paral·lelisme que s'exploti. La taula 2 mostra la classificació que va fer Flynn.

Michael J. Flynn

Nascut el 1934 a Nova York i professor emèrit de la Universitat d'Standford. Va cofundar Palyn Associates amb Max Paley i és president de Maxeler Technologies.

Taula 2. Taxonomia de Flynn.

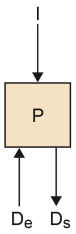
Flux d'instruccions	Flux de dades	Nom	Exemples
1	1	SISD	Von Neumann
1	$\geq$1	SIMD	Vectorials
$\geq$1	1	MISD	No coneguts
$\geq$1	$\geq$1	MIMD	Multiprocessadors/Multicomputadors

Flynn classifica les màquines segons el programa i les dades que tracta. Així, ens podem trobar que un programa pot tenir més d'una seqüència d'instruccions, en què cada seqüència d'instruccions necessita un comptador diferent de programa. Una màquina amb diverses CPU disposa de diversos comptadors de programa, i per tant, pot executar diverses seqüències d'instruccions. Quant a les dades, una seqüència de dades es pot definir com un conjunt d'operands.

Tenir una o diverses seqüències d'instruccions i una o diverses seqüències de dades és independent. Per aquest motiu podem distingir quatre tipus de màquina, que detallem a continuació:

- **SISD**: les màquines SISD (*single instruction single data*) són les màquines Von Neumann. Són màquines que executen una única seqüència d'instruccions sobre una seqüència de dades, tractats d'una en una. La figura 14 mostra un esquema bàsic d'aquest tipus d'arquitectura. Un processador executa una seqüència d'instruccions (*I*) i les aplica a una seqüència d'entrada de dades *D_e* (*e* d'*entrada*) i retorna una seqüència de resultats *D_s* (*s* de *sortida*).

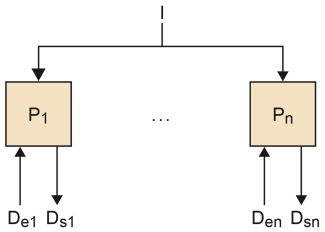
Figura 14. Arquitectura SISD



- **SIMD**: aquestes màquines processen en paral·lel més d'una seqüència de dades ($D_{e1} \ldots D_{en}$) amb una única seqüència d'instruccions (*I*). Així, tots els processa-

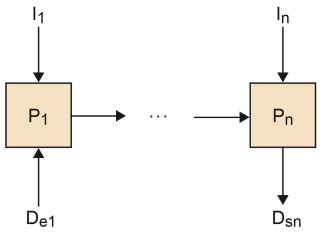
dors que formen la màquina SIMD prenen la mateixa seqüència d'instruccions que apliquen a les diferents entrades i generen tantes sortides com entrades. La figura 15 mostra un esquema bàsic d'aquest tipus d'arquitectura.

Figura 15. Arquitectura SIMD



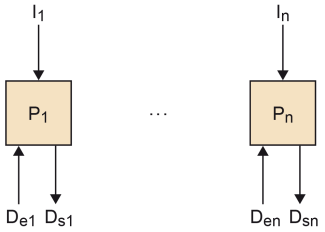
- **MISD**: aquesta categoria és una mica difícil d'imaginar, i no es coneix cap màquina que compleixi amb el fet de tenir més d'una seqüència d'instruccions que operin sobre les mateixes dades. La figura 16 mostra un esquema bàsic d'aquest tipus d'arquitectura.

Figura 16. Arquitectura MISD



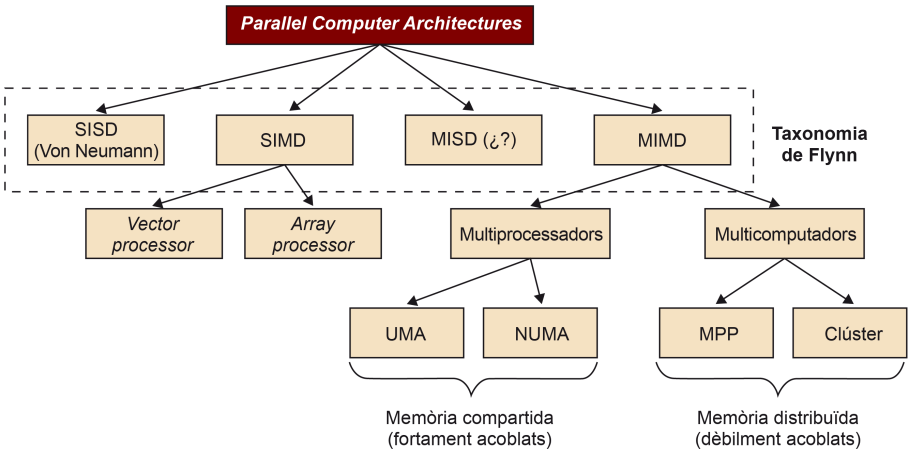
- **MIMD**: aquesta és l'última categoria que distingeix Flynn. En aquesta categoria múltiples processadors independents operen sobre diferents seqüències de dades. Cadascun dels processadors aplica una seqüència diferent d'instruccions a una seqüència d'entrada, per a obtenir una seqüència de sortida. La majoria de les màquines paral·leles cauen en aquesta categoria. La figura 17 mostra un esquema bàsic d'aquest tipus d'arquitectura.

Figura 17. Arquitectura MIMD



D'altra banda, Andrew S. Tanenbaum, en el seu llibre sobre l'organització de l'estructura del computador amplia aquesta classificació. La figura 18 mostra aquesta ampliació. Així, SIMD la torna a dividir en processadors vectorials i processadors en *array*. Els primers són processadors numèrics que processen vectors, i fan una operació en cada element del vector (per exemple, Convex Machine). La segona té més a veure amb una màquina paral·lela, com per exemple la ILLIAC IV, en la qual utilitzen múltiples ALU independents quan cal fer una instrucció determinada. Per a això, una unitat de control fa que totes funcionin cada vegada.

Figura 18. Classificació de les arquitectures segons Andrew S. Tanembaum.



Per al cas de MISD, tal com hem comentat, no s'ha trobat cap màquina que compleixi amb aquesta classificació.

Finalment, per a les màquines de tipus MIMD, aquestes es poden classificar en dos tipus: multiprocessadors* (o màquines de memòria compartida) i muticomputadors** (o màquines basades en pas de missatges).

*** Multiprocessadors: màquines MIMD amb memòria compartida.**
**** Multicomputadors: màquines MIMD que no tenen memòria compartida i necessiten pas de missatges per a compartir les dades.**

Dins de les màquines de memòria compartida, segons el temps d'accés i com està compartida la memòria, podem distingir entre les crides UMA (*uniform memory access* o memòries d'accés uniforme) i les crides NUMA (*nonuniform memory access* o memòries d'accés no uniforme).

Les màquines UMA* es caracteritzen perquè tots els accessos a memòria triguen el mateix. Això ajuda que el programador pugui predir el comportament del seu programa en els accessos a memòria i programi codis eficients.

*** UMA (*uniform memory access*) són multiprocessadors en els quals tots els accessos a memòria triguen el mateix temps.**

Per contra, les màquines NUMA* són màquines en les quals els accessos a memòria poden trigar temps diferents. Així, si una dada està en la memòria propera a un processador, es trigarà menys que si s'accedeix a una dada que està lluny del processador.

*** NUMA (*nonuniform memory access*) són multiprocessadors en els quals els accessos a memòria poden trigar temps diferents.**

D'altra banda tenim els multicomputadors, que es distingeixen bàsicament pel fet que el sistema operatiu no pot accedir a la memòria d'un altre processador únicament utilitzant operacions d'accés a memòria `load/store`. Quant als multicomputadors, distingim entre MPP (*massively parallel processors*) i clústers de processadors. Els primers són supercomputadors formats per CPU que estan connectades per xarxes d'interconnexió d'alta velocitat. Els segons consisteixen en PC o estacions de treball, connectades amb xarxes d'interconnexió *off-the-shelf*. Dins d'aquesta última categoria, podem trobar els clústers d'estacions de treball o COW (*cluster of workstations*).

off-the-shelf

Que es pot comprar i no és específic per a una màquina determinada.

Finalment podem tenir sistemes híbrids: clústers en els quals els nodes són màquines amb memòria compartida, i sistemes *grids*: màquines de memòria compartida o de memòria distribuïda que estan connectades via LANs o WAN.

A continuació detallarem més la divisió feta de les màquines MIMD, que són objectiu d'aquest mòdul.

2.1. MIMD: memòria compartida

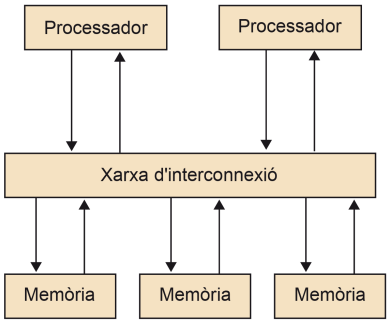
En una màquina de memòria compartida tots els processadors comparteixen un mateix espai d'adreces. En aquests sistemes, els *threads* es poden comunicar uns amb els altres per mitjà de lectures i escriptures a variables/dades compartides.

Una de les classes de màquines de memòria compartida més conegudes són els SMP (*symmetric multiprocessors*), que són màquines UMA. Tots els processadors són iguals i van a la mateixa velocitat, i tots comparteixen una connexió per a accedir a totes les posicions de memòria. La figura 19 mostra un exemple d'un SMP amb dos processadors que es connecten a un `bus` compartit per a accedir a la memòria principal.

SMP

Els SMP (*symmetric multiprocessors*) són multiprocessadors UMA que comparteixen un mateix `bus` per a accedir a la memòria compartida.

Figura 19. Arquitectura SMP



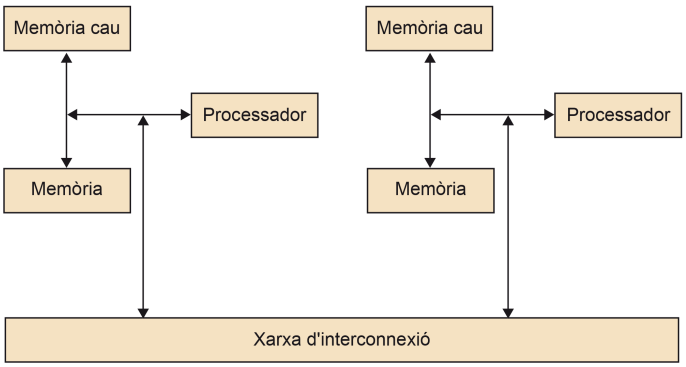
A causa que tots els processadors comparteixen la connexió, aquest tipus de màquines no escalen amb un gran nombre de processadors. La connexió compartida es converteix en un coll d'ampolla. Malgrat això, són els sistemes més fàcils de muntar i de programar, ja que el programador no es preocupa d'on van les dades.

L'altre tipus de màquina són les NUMA, que permeten escalar amb més processadors, ja que la seva connexió a la memòria no és compartida per tots els processadors. En aquest cas hi ha memòries que estan a prop i lluny d'una CPU, i per tant, es pot trigar un temps d'accés diferent depenent de la dada a la qual s'accedeix. La figura 20 mostra un esquema bàsic d'una arquitectura `NUMA` formada per dos processadors.

Multiprocessadors NUMA

En els multiprocessadors NUMA, el programador ha de ser conscient d'on es poden desar les dades compartides. Per exemple, la inicialització de les dades en un processador pot fer que aquest tingui les dades en la seva memòria propera, mentre que els altres la tenen en la llunyana.

Figura 20. Arquitectura multiprocessador de tipus NUMA



Per a reduir els efectes de la diferència de temps entre accés a memòria propera i accés a memòria llunyana, cada processador disposa d'una memòria cau. Per contra,

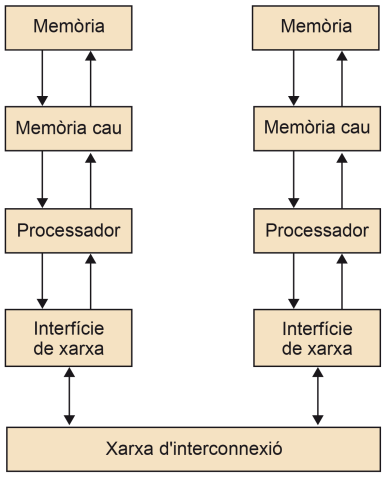
per a mantenir la coherència d'una dada en totes les memòries cau, calen protocols de coherència de memòria cau. De vegades es coneixen com a màquines ccNUMA (*cache-coherent nonuniform memory access*). La manera de programar aquestes màquines és tan fàcil com una màquina UMA, però aquí el programador ha de pensar bé on posa les dades per temes d'eficiència.

2.2. MIMD: memòria distribuïda

En les màquines de memòria distribuïda cada processador té el seu espai d'adreces propi i, per tant, els processos s'han de comunicar via pas de missatges (punt a punt o col·lectives).

La figura 21 mostra com cada *core* (processador) disposa de la seva memòria i que per a obtenir les dades de la memòria d'un altre *core*, aquests s'han de comunicar entre ells via una xarxa d'interconnexió. La latència i amplada de banda de la xarxa d'interconnexió pot variar molt, i pot ser tan ràpida com l'accés a memòria compartida, o tan lenta com anar a través d'una xarxa Ethernet.

Figura 21. Arquitectura de memòria distribuïda



Des del punt de vista del programador, aquest s'ha de preocupar de distribuir les dades i, a més, de fer la comunicació entre els diferents processos.

Multicomputadors

En els multicomputadors, el programador ha de distribuir les dades entre els diferents processos i fer explícita la compartició de dades entre aquests processos mitjançant la comunicació amb pas de missatges.

2.3. MIMD: Sistemes híbrids

Segons alguns autors, com Dongarra i van der Steen, els sistemes híbrids estan formats per clústers d'SMP connectats amb una xarxa d'interconnexió d'alta velocitat. A més, aquests mateixos autors, en l'any de la publicació, indicaven que era una de les tendències actuals. De fet, l'any 2003, quatre dels 5 principals computadors més potents en el Top500 eren d'aquest tipus.

Lectura complementària

Aad J. van der Steen; Jack J. Dongarra (2003). *Overview of Recent Supercomputers.*

2.4. MIMD: *grids*

Les *grids* són sistemes que combinen sistemes distribuïts, de recursos heterogenis, que estan connectats per mitjà de LAN o WAN, que solen ser Internet.

Lectura complementària

Ian Foster; Carl Kesselman (2003). *The Grid 2: Blueprint for a New Computing Infraestructure* (2a. ed.). Morgan Kaufmann.

Al principi, els sistemes *grids* es van veure com una manera de combinar grans supercomputadors per a resoldre problemes molt grans. Després s'han derivat més envers una visió d'un sistema per a combinar recursos heterogenis com servidors, emmagatzematge de grans volums de dades, servidors d'aplicacions, etc. En qualsevol cas, la distinció bàsica entre un clúster i una *grid* és que en aquest últim no hi ha un punt comú d'administració dels recursos. Cada organització que està dins d'una *grid* manté el control de la gestió dels seus recursos, la qual cosa repercuteix en els mecanismes de comunicació entre els computadors de les organitzacions.

3. Mesures de rendiment

En aquest apartat treballarem amb mètriques que ens ajudaran a analitzar el paral·lelisme potencial d'un programa. Analitzarem com afecta el gra de paral·lelisme en l'estratègia de paral·lelització el paral·lelisme potencial, i per tant, l'*speedup* (quantes vegades més ràpid) ideal que podem assolir. Finalment, treballarem un model senzill de temps que ens ajudi a analitzar quin és el paral·lelisme i *speedup* real que podem aconseguir en un computador determinat.

Gra de paral·lelització

El gra de paral·lelització (treball que ha de fer cada tasca) determina el paral·lelisme potencial que podrem explotar en la nostra estratègia de paral·lelització.

3.1. Paral·lelisme potencial

Per a mesurar el paral·lelisme potencial que assolirem en la resolució d'un problema, hem de pensar primer en la distribució de treball que farem. Hi ha dues maneres de fer la distribució del treball:

Distribució del treball

Es poden fer dos tipus de distribució del treball: *task decomposition* si pensem en les operacions que cal fer, i *data decomposition* si pensem en les dades que cal tractar.

1) Segons una distribució des d'un punt de vista de tasques (*task decomposition* en anglès).

2) Segons un punt de vista de dades (*data decomposition*).

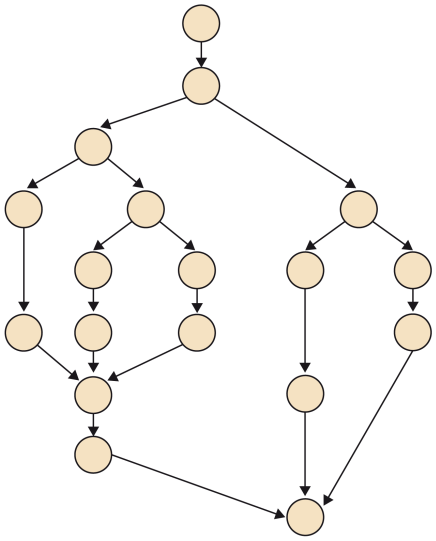
En el primer cas, la distribució podria ser cada crida a funció o cada iteració d'un bucle que és una tasca. En el segon cas, en el cas d'haver de tractar una matriu, podríem dividir la matriu entre els diferents processos/*threads*, i després assignar una tasca a cada fragment.

Una vegada feta la distribució de treball entre tasques, ja sigui després d'una *task decomposition* o una *data decomposition*, hem de crear el graf de dependències entre tasques amb tal de poder analitzar el paral·lelisme existent. Aquest graf està format per nodes (tasques), i les arestes entre nodes indiquen les dependències entre si (l'origen de l'aresta és la font de la dependència, és a dir, qui genera la dada). Cada dependència indica que un node només pot procedir a fer el seu càlcul quan tots els nodes dels quals depèn ja han generat les dades. La figura 22 mostra un possible graf de dependències entre tasques. Aquest tipus de grafs són grafs acíclics dirigits (DAG, *directed acyclic graph*). Per simplificar, anem a suposar que disposem de P processadors, i cadascun podrà executar un node a cada moment.

Graf de tasques

El graf de tasques resulta d'analitzar les dependències entre les diferents tasques per fer.

Figura 22. Graf de dependències de les tasques



A partir del graf de dependències de tasques podem fer l'anàlisi del programa paral·lel i quin paral·lelisme potencial podem obtenir en comparació del programa seqüencial. Definirem primer una sèrie de conceptes, i després, amb un exemple simple, veurem com aplicar-los.

- T_1: és el temps de l'execució seqüencial de tots els nodes del graf, és a dir, la suma del cost de cadascuna de les tasques del graf de dependències:

$$T_1 = \sum_{i=1}^{nodes} (cost\_node_i)$$

- T_∞: o *span* en anglès, és el temps mínim necessari per a poder acabar totes les tasques que s'executen amb un nombre infinit de processadors. Des del punt de vista del graf de dependències, és el camí crític des de la primera i l'última tasca per executar-se.

- Paral·lelisme: definit com a $\frac{T_1}{T_\infty}$. Aquest és independent del nombre de processadors existents, i depèn únicament de les dependències entre tasques. És el paral·lelisme potencial que podem explotar si tinguéssim un nombre infinit de processadors.

- Folgança de paral·lelisme*: es defineix com a $\frac{T_1}{T_\infty}/P$, en què P és el nombre de processadors. Ens indica un límit inferior del nombre de processadors necessaris per a assolir el paral·lelisme potencial. Això no significa que amb aquest nombre de processadors puguem assolir T_∞.

Paral·lelisme

El paral·lelisme existent en un programa és independent del nombre de processadors dels quals es disposa. Depèn únicament de la distribució de tasques feta i les dependències entre aquestes.

* ***Parallel slackness*** **en anglès**

En el codi 3.1 tenim un codi que no té més propòsit que el de desenvolupar una anàlisi del paral·lelisme potencial segons la granularitat en la definició de tasques. La granularitat és la quantitat de treball que volem assignar a cada tasca. Aquesta pot ser fina si li assignem un treball petit, i gruixuda si li donem més quantitat de treball. Per a l'e-

xemple en qüestió suposarem que el cost de fer la instrucció del bucle més intern del bucle niat és $O(1)$. Per al `print` del bucle *ij* posterior també suposarem que aquesta instrucció té cost $O(1)$.

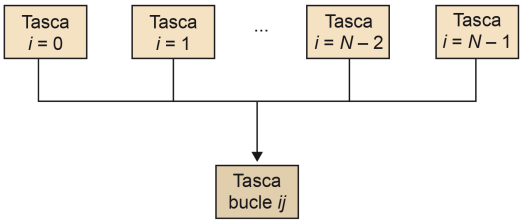
```
1
2    for (i=0; i<N; i++)
3      for (j=0; j<M; j++)
4        C[i][j] = A[i][j]*B[i][j];
5
6    ptr = (int *)C;
7    for (ij=0; ij<N*M; ij++, ptr++)
8       printf("%d\n", *ptr);
```

Codi 3.1: Codi molt simple per a analitzar el paral·lelisme potencial

Una primera distribució de treball seria tenir dues tasques de gra gruixut: (tasca 1) els dos bucles imbricats (bucles *i* i *j*), i (tasca 2) el bucle *ij*. El cost de la primera i segona tasca és $O(N*M)$. Per tant, T_1 és $2*O(N*M)$. Per a calcular el T_∞ hem d'analitzar el graf de dependències de tasques: la tasca 2 s'ha d'esperar que la tasca 1 acabi per a poder imprimir per pantalla les dades correctes. Per tant, el T_∞ és exactament igual que el T_1.

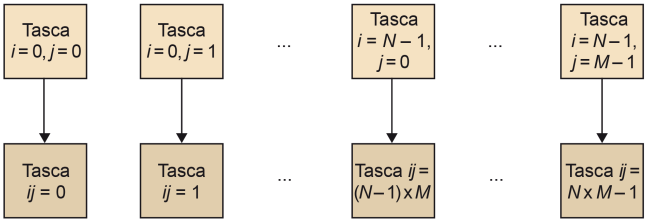
Una altra distribució possible del treball en tasques seria tenir una tasca per a cada iteració del bucle *i*, i deixar l'antiga tasca 2 igual. D'aquesta manera, hem fet una distribució de tasques d'un gra més fi. En aquest cas T_1 continua essent el mateix, ja que no hem canviat l'algorisme seqüencial. No obstant això, T_∞ s'ha reduït. El graf de dependències de tasques ha canviat significativament. La figura 23 mostra que ara totes les tasques que corresponen amb una iteració del bucle *i* es poden fer en paral·lel. Cadascuna d'aquestes tasques del bucle *i* té un cost de $O(M)$. La tasca 2 no canvia. Per tant, T_∞ és ara $O(M) + O(N*M)$.

Figura 23. Graf de dependències de les tasques quan cada iteració *i* dels bucles niats és considerada una tasca



Finalment, si ens decantem per un gra més fi en la distribució de tasques, tant del bucle niat com del bucle *ij*, podríem definir cada tasca com cada iteració del bucle *j* i cada iteració del bucle *ij*. T_1 continuarà essent el mateix, ja que no hem hagut de fer cap canvi en l'algorisme. T_∞, no obstant això, s'ha reduït significativament. Suposant que podem fer un `print` de cada dada de manera paral·lela, totes les tasques del bucle *j* i les del bucle *ij* es poden fer en paral·lel. Cadascuna té un cost de $O(1)$. Per tant, T_∞ s'ha reduït a $2 \times O(1)$. Això és així, ja que el graf de dependències de tasques queda com a mostra la figura 24, amb la qual cosa cada element per imprimir solament depèn del còmput de la dada corresponent en el bucle anterior.

Figura 24. Graf de dependències de les tasques quan cada iteració j dels bucles més interns és considerada una tasca



Tal com hem vist, una granularitat fina afavoreix el T_∞. Per tant, un podria pensar, naturalment, que la idea és fer tasques de gra molt fi, amb la qual cosa podríem augmentar el nombre de tasques que es poden fer en paral·lel. No obstant això, un primer límit que ens podem trobar és el nombre màxim de tasques de gra molt fi que es poden definir. En el cas anterior no podem definir més de $2 \times N \times M$ tasques. D'altra banda, en el cas d'haver d'intercanviar dades entre un nombre elevat de tasques, això podria significar un cost addicional excessiu. Altres costos addicionals, en el moment de crear el programa paral·lel amb totes aquestes tasques, són: el cost de creació de cadascuna de les tasques, la sincronització d'aquestes, la destrucció de les tasques, etc. És a dir, que hi ha d'haver un compromís entre el gra fi, i per tant, el nombre de tasques que es volen crear, i els costos addicionals que representa haver de gestionar totes aquestes tasques i l'intercanvi de dades i sincronització entre aquestes.

Granularitat fina

La granularitat molt fina afavoreix el T_∞, però hem de tenir en compte que els costos de gestió i sincronització d'un nombre elevat de tasques poden afectar el rendiment real del programa paral·lel.

3.2. *Speedup* i eficiència

En el subapartat anterior calculàvem el paral·lelisme potencial (T_∞) d'una aplicació després de determinar la distribució en tasques i el camí crític del graf de dependències existent entre elles. No obstant això, moltes vegades el T_∞ és difícil d'assolir en una màquina paral·lela MIMD amb un nombre de processadors P limitat.

Definirem T_P com el temps real que triguem a fer els nodes del graf de dependència de tasques amb P processadors. Sabem que T_P és normalment més gran o igual que T_1/P i T_∞*, és a dir, com a molt podrem dividir el treball entre els P processadors, i no baixarem dels T_∞ que havíem calculat.

*** T_∞ és límit inferior a T_P**

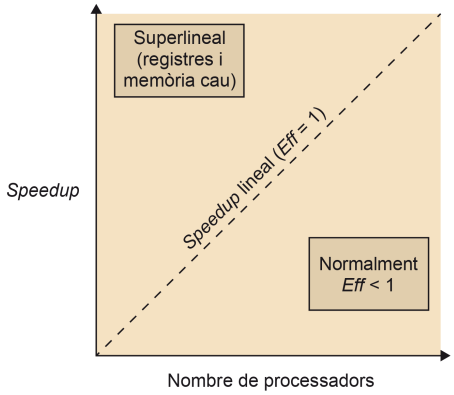
Així, l'*speedup* aconseguit per un programa paral·lel pel que fa al programa en seqüencial es defineix com:

$$S_P = \frac{T_1}{T_P}$$

En concret, podem definir *speedup* com la reducció de temps d'execució relativa, en processar una mida fixa de dades quan usem P processadors, pel que fa al temps d'execució del programa seqüencial. La corba de l'*speedup* hauria de ser idealment una funció lineal de pendent 1, és a dir, que anéssim P vegades més ràpid amb P processadors. La figura 25 mostra les tres situacions amb les quals ens podem trobar quan fem un programa paral·lel i analitzem l'*speedup*. La situació normal és que els nostres programes, una vegada paral·lelitzats, assoleixin un *speedup* més petit que el

lineal, ja que els costos de comunicació, sincronització i creació/destrucció de tasques solen fer que el cost de paral·lelització no sigui a cost zero. No obstant això, es poden donar casos en els quals obtenim un *speedup* superior al lineal. Aquest és el cas de programes paral·lels que ajuden a explotar la jerarquia de memòria, o els registres del processador, cosa que abans no es podia amb el programa seqüencial.

Figura 25. Eficiència lineal i situacions en què és més petit o superior: superlineal



Una altra mesura que ens ajuda a determinar com de bona és la paral·lelització feta és l'eficiència. L'eficiència es defineix com la mesura de la fracció de temps en la qual cada processador és usat per a resoldre el problema en qüestió de manera útil. Si els nostres processadors s'utilitzen de manera eficient, això significarà que el temps dedicat per cada processador pel nombre de processadors hauria de ser T_1. Així, eficiència (Eff_P) es defineix com:

$$Eff_P = \frac{T_1}{T_P \times P}$$
$$Eff_P = \frac{S_P}{P}$$

3.3. Llei d'Amdahl

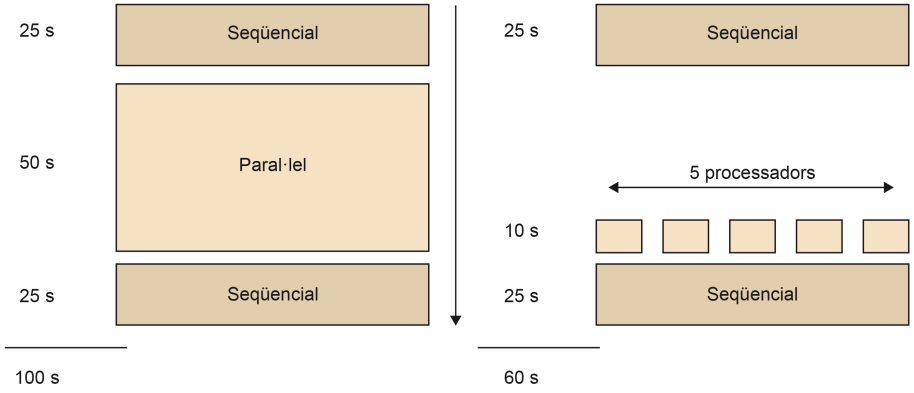
Llei d'Amdahl

La fracció de temps invertida en aquesta part seqüencial limitarà la millora de rendiment que obtindrem de la paral·lelització.

En la paral·lelització d'una aplicació normalment hi ha alguna part que no es pot paral·lelitzar. Aquesta fracció de temps invertida en aquesta part seqüencial limitarà la millora de rendiment que obtindrem de la paral·lelització, i per tant, l'eficiència obtinguda.

En la figura 26 mostrem el temps que triguem en un codi totalment executat en seqüencial (esquerra) i un codi en el qual hi ha una part, definida com a paral·lela, que han paral·lelitzat perfectament 5 processadors. Si calculem l'*speedup* que podem aconseguir, obtenim que $S_P = 100/60 = 1.67$, fins i tot havent aconseguit una eficiència d'1 en part paral·lela del programa.

Figura 26. Execució en seqüencial i en paral·lel



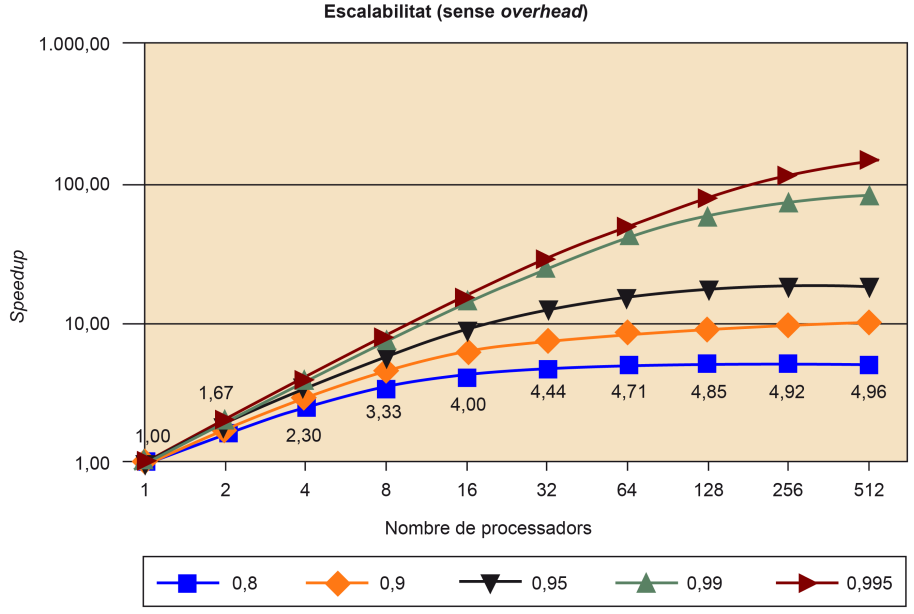
La llei d'Amdahl indica que la millora de rendiment està limitada per la fracció de temps que el programa s'està executant en paral·lel. Si anomenem aquesta fracció ϕ, tindrem que:

$$\begin{aligned} T_1 &= T_{seq} + T_{par} = (1-\phi) \times T_1 + \phi \times T_1 \\ T_P &= (1-\phi) \times T_1 + \phi \times T_1/P \\ S_P &= \frac{T_1}{T_P} \\ S_P &= \frac{1}{(1-\phi) + \phi/P} \end{aligned}$$

De tal manera que si el nombre de processadors tendeix a infinit, l'*speedup* màxim que es podrà obtenir únicament dependrà de la fracció que no es pot paral·lelitzar, tal com mostra la fórmula següent, i com gràficament ens mostra la figura 27:

$$S_P \rightarrow \frac{1}{(1-\phi)} \quad per\ a \quad P \rightarrow \infty$$

Figura 27. Corba de *speedup* que es pot obtenir per a una determinada ϕ



En aquesta figura podem veure que fins i tot disposant d'un 80% ($\phi = 0.8$) del temps d'execució per a paral·lelitzar, el màxim *speedup* que podem aconseguir amb 512 processadors és de $4.96\times$, sempre que no afegim cap tipus de cost addicional per a aconseguir la paral·lelització. No obstant això, normalment la paral·lelització no és gratis, i hem de pagar alguns costos.

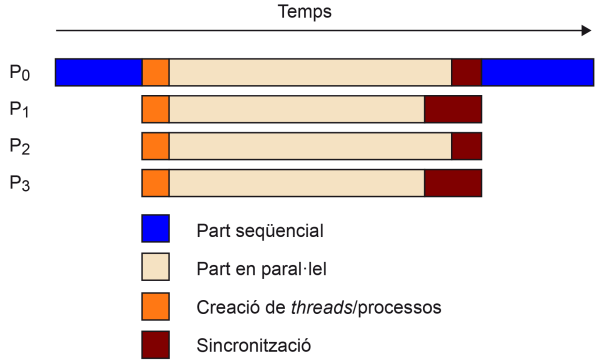
Alguns dels costos que s'han de pagar en fer la paral·lelització d'una aplicació són deguts al següent:

- La creació i la terminació de processos/*threads*. Aquí tenim un processament extra en iniciar i finalitzar les tasques. Noteu que la creació de processos és molt més cara que la creació de *threads*.
- Sincronització. Per a poder assegurar les dependències entre tasques existents en el graf de dependències de les tasques.
- Compartició de dades. Aquesta comunicació pot ser que s'hagi de fer amb missatges explícits o bé via jerarquia de memòria.
- Càlcul. Hi ha càlculs que es repliquen amb tal de no haver de fer la comunicació entre les tasques.
- Contenció quan es fan accessos a recursos compartits, com la memòria, la xarxa d'interconnexió, etc.
- La inactivitat d'alguns processos/*threads*, a causa de les dependències entre tasques, el desequilibri de càrrega, una superposició pobra entre comunicació i càlculs, etc.
- Estructures de dades necessàries extra, a causa de la paral·lelització de l'algorisme.

La figura 28 mostra un gràfic d'execució, per al cas de 4 processadors, en què apareixen detallats alguns d'aquests costos de paral·lelització. Aquests costos poden ser constants, o linears pel que fa al nombre de processadors. T_P per al cas d'un cost addicional és:

$$T_P = (1 - \phi) \times T_1 + \phi \times T_1/P + sobrecoste(P)$$

Figura 28. Costos addicionals que poden aparèixer en l'execució d'un programa paral·lel



En el primer dels casos, amb un cost constant, fa que el rendiment ideal, per a una determinada fracció d'execució de temps per paral·lelitzar, segueixi la forma de la figura 29. Si comparem el rendiment final per a $\phi = 0.9$ quan no tenim cost de paral·lelització i quan en tenim de cost constant, observem que el rendiment ideal baixa significativament. És més, si comparem el cas de cost zero amb el cost lineal en funció del nombre de processadors, aquest cost addicional* fa contraproduent augmentar el nombre de processadors, i fa que l'*speedup* sofreixi una retrocés, com podem observar en la figura 30.

*** El sobrecost de paral·lelització pot influir significativament en l'*speedup* ideal.**

Figura 29. Amdalh quan tenim un cost addicional de paral·lelització no menyspreable i constant

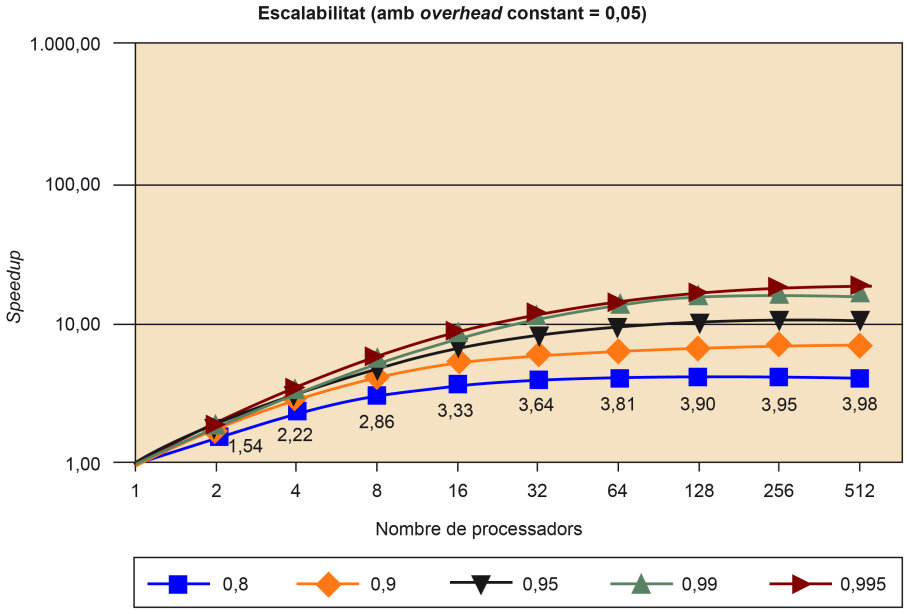
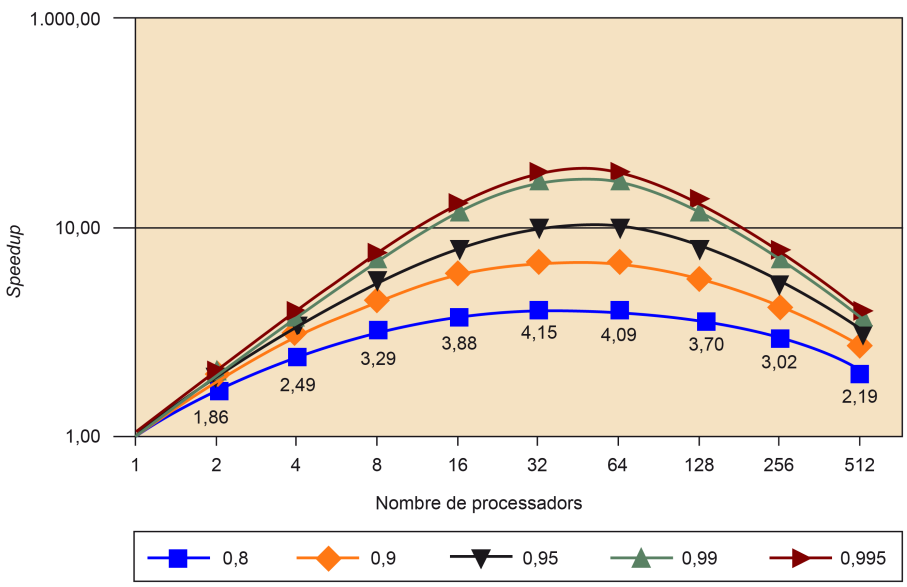


Figura 30. Amdalh quan tenim un cost addicional de paral·lelització no menyspreable i lineal

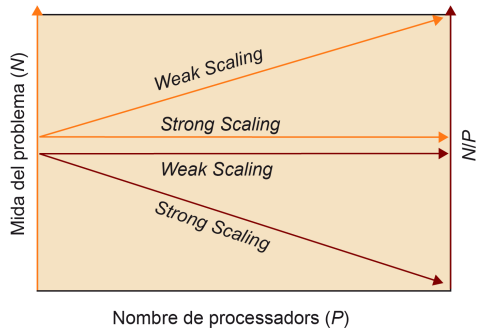


3.4. Escalabilitat

L'escalabilitat és una mesura de com es comporta un programa paral·lel quan augmentem la mida del problema proporcionalment al nombre de processadors, o augmentem el nombre de processadors sense variar la mida del problema per tractar.

La figura 31 mostra les dues mesures d'escalabilitat que es poden fer, segons ho mirem des del punt de vista de la mida total del problema per tractar (eix y esquerre), i la mida del problema per tractar per processador (eix y dret). Així, la *weak scalability* és una mesura que manté fixa la mida que toca a cada processador, i per tant, la mida augmenta amb el nombre de processadors. Per contra, la *strong scalability* és una mesura que manté la mida fixa del problema per tractar per a qualsevol nombre de processadors per tractar.

Figura 31. Escalabilitat *weak* i *strong*



Normalment, quan s'analitza l'*speedup* estem analitzant també la *strong scalability*.

Una altra mesura que ens indica com d'adaptable és la nostra estratègia de paral·lelització a problemes petits és $N_{\frac{1}{2}}$, que ens indica quina és la mida mínima necessària per a assolir una eficiència (Eff_P) del 0.5 per a un nombre determinat de processadors P. Un valor gran de $N_{\frac{1}{2}}$ indica que el problema és difícil de paral·lelitzar amb l'estratègia usada.

3.5. Model de temps d'execució

El cost de compartició de les dades depèn del tipus d'arquitectura que tinguem i de la paral·lelització usada. La compartició en una màquina de memòria compartida és molt més senzilla, però és molt més difícil de modelitzar. En canvi, en el cas de memòria distribuïda és més difícil de programar, però més senzill de modelitzar.

En aquest subapartat ens centrarem en els costos de comunicació amb pas de missatges. En aquest temps de comunicació tenim:

- Temps d'inicialització* (t_s): que consisteix a preparar el missatge, determinar el camí del missatge a través de la xarxa, i el cost de la comunicació entre el node local i el *router*.

* ***Start up*, en anglès**

- Temps de transmissió del missatge, que al seu torn consta del següent:

– per *hop* (t_h): temps necessari perquè la capçalera del missatge es transmeti entre dos nodes directament connectats a la xarxa.

– per *byte* (t_w): temps de transmissió d'un element (*word*).

Així, depenent del camí que prengui el missatge i el mecanisme d'encaminament que segueixi, tindrem un cost de comunicació o un altre. Distingirem entre tres mecanismes d'encaminament:

1) Encaminament *store-and-forwarding*: en aquest cas els missatges viatgen d'un node a un altre. Cada node rep el missatge, l'emmagatzema i després fa la reexpedició del missatge.

El cost de comunicació en aquest cas és:

$$T_{comm} = t_s + (m \times t_w + t_h) \times l$$

en què l és el nombre de nodes que visitem (connectats directament), i m és el nombre de *words* de què està format el missatge.

No obstant això, com que t_h és normalment molt petit, podem simplificar la fórmula i quedar-nos amb la següent:

$$T_{comm} = t_s + m \times t_w \times l$$

2) Encaminament *packet*: en aquest cas el missatge es divideix en parts de tal manera que puguem explotar millor els recursos de comunicació, ja que poden seguir camins diferents, i evitar la contenció d'alguns camins. A més, amb la divisió de paquets reduïm el nombre d'errors.

Per contra, la mida del missatge global s'incrementarà a causa que hem d'afegir les capçaleres, informació de control d'errors i de seqüenciació del missatge. A més, hi ha un temps invertit en la divisió en parts. Això fa que s'augmenti el t_s.

3) Encaminament *cut-through*: aquest tipus d'encaminament intenta reduir el cost addicional de divisió en parts. Per a això totes les parts són forçades a tenir el mateix encaminament i informació d'error i seqüenciació. Així, s'envia un traçador per a decidir el camí des del node origen i el node destinació.

Els missatges, quan es comuniquen d'un node a un altre, no són copiats i reexpedits com es feia amb l'*store-and-forward*. De fet, no s'espera que tot el missatge arribi per a començar a reexpedir-ho.

El cost de comunicació en aquest tipus d'encaminament és:

$$T_{comm} = t_s + m \times t_w + t_h \times l \quad (1)$$

en què podem observar que el cost de transmissió no es multiplica pel nombre de *links* que s'han de travessar.

La majoria de la màquines paral·leles solen tenir aquest tipus d'encaminament.

Suposant el cost de comunicació de l'encaminament *Cut-through*, ens podríem preguntar: com podem reduir el cost de comunicació?

Hi ha diverses maneres d'aconseguir-ho:

Reduir el cost

Maneres de reduir el cost de comunicació: agrupar missatges, reduir el volum per comunicar, i reduir la distància de comunicació.

- Agrupar dades per comunicar quan sigui possible. Normalment t_s és molt més gran que t_h i t_w. Així, si podem ajuntar missatges, reduïm el nombre de missatges i el pes de t_s.

- Reduint el volum total de dades per comunicar. Així minimitzem el cost que paguem per *word* comunicat (t_w). Aconseguir-ho dependrà de cada aplicació.

- Reduint la distància de comunicació entre els nodes. Aquest és el més difícil d'aconseguir, ja que depèn de l'encaminament fet. D'altra banda, normalment es té molt poc control del mapatge dels processos en els processadors físics.

En qualsevol cas, podem fer una simplificació de la fórmula anterior tenint en compte que:

- El pes del factor t_s és molt més gran que t_h quan els missatges són petits.

- El pes del factor t_w és molt més gran que t_h quan els missatges són grans.

- l no sol ser gaire gran, i t_h sol ser petit.

Pel això el terme $l \times t_h$ el podem ignorar, i simplificar el model en la fórmula següent:

$$T_{comm} = t_s + m \times t_w$$

La manera d'obtenir t_s i t_w d'una màquina, perquè ajusti el model fet, es podria fer mitjançant *minimal mean square error**.

*** Raj Jain (1991). "The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling". *Wiley- Interscience*. Nova York.**

Per a fer un model per al cas de treballar amb una màquina de memòria compartida, s'haurien de tenir molts factors en compte, com per exemple: com és l'organització física de la memòria, les memòria cau de les quals disposem i la seva capacitat, el cost de mantenir sistemes de coherència, la localitat espacial i temporal, el *prefetching*, la compartició falsa, la contenció de memòria, etc., que fan molt difícil fer un model raonable. No obstant això, fent una sèrie de suposicions, i amb l'únic objectiu de tenir un model de compartició de dades, podríem considerar que aquest model de comunicació es podria prendre també per a programes de memòria compartida, considerant com a unitat bàsica de comunicació la línia de memòria cau.

3.6. Casos d'estudi

En aquest subapartat farem el model de temps d'execució per a dos problemes: un *edge-detection* i un *stencil*. El primer el farem sense aplicar la tècnica d'optimització *blocking*, i el segon aplicant-la. *Blocking* permet distribuir el treball en blocs de tal manera que ajudi a millorar el rendiment de la jerarquia de memòria en explotar millor la localitat temporal de dades (no analitzat aquí), i afavorir el paral·lelisme entre processos/*threads*.

Nota

En els casos d'estudi estem suposant que cada procés s'executa en un processador independent.

3.6.1. Exemple sense *blocking*: *edge-detection*

En aquest primer exercici farem el següent:

- Modelitzar el temps invertit en l'execució en paral·lel.
- Calcular el `speedup`.
- Obtenir l'eficiència d'una paral·lelització de l'*edge-detection*.

Lectura complementària

A. Grama i altres (2003). *Introduction to Parallel Computing*. Boston: Addison Wesley.

El codi 3.2 mostra una aproximació de l'*edge-detection*. Aquest codi reflecteix com s'aplica un *template* de 3×3 a totes les posicions de la variable `in_image`, deixant el resultat en `out_image`.

```
1
2  int apply_template_ij(int i, int j, int in_image[N+2][N+2], int
       template[3][3])
3  {
4     int ii, jj;
5     int apply = 0;
6
7     for (ii=i-1; ii<i+2; ii++)
8       for (jj=j-1; jj<j+2; j++)
9         apply += in_image[ii][jj] * template[ii-i+1][jj-j+1];
10
11    return apply;
12 }
13
14 void Edge-Detection(int in_image[N+2][N+2], int out_image[N+2][N+2],
       int template[3][3])
15 {
16   int i, j;
17
18   for (i=1; i<N+1; i++)
19     for (j=1; j<N+1; j++)
20       out_image[i][j] = apply_template_ij(i,j,in_image,template);
21 }
```

Codi 3.2: Codi de l'*edge-detection*.

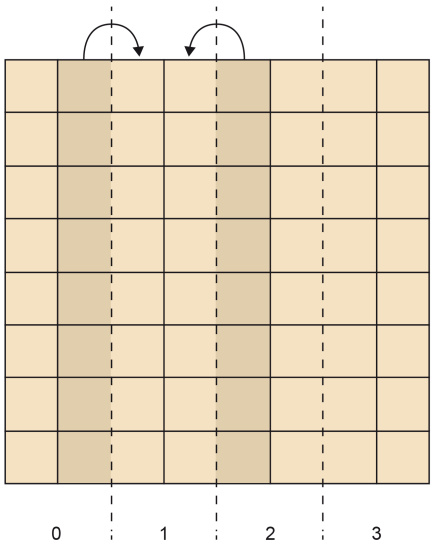
Per a fer el càlcul de l'`speedup` hem de calcular el temps d'execució seqüencial i el temps d'execució en paral·lel. Anem a suposar que el temps de cada multiplicació i suma (una única instrucció) en la funció `apply_template_ij` és de t_c.

Començarem primer pel càlcul del temps seqüencial, T_1. Si analitzem el codi, observem que per a cada element de la matriu cridem la funció `apply_template_ij`, i que cada crida significa fer 9 operacions de multiplicació i suma. És a dir, el temps total d'execució en seqüencial és:

$$T_1 = 9 \times t_c \times N^2$$

Per al càlcul del temps de l'execució en paral·lel necessitem plantejar una estratègia de paral·lelització i de distribució de treball. Les dades les distribuirem per columnes. A cada procés assignarem $\frac{N}{P}$ columnes senceres, és a dir, $\frac{N}{P} \times N \rightarrow \frac{N^2}{P}$ píxels (un segment). Cada procés necessitarà dues columnes (*boundaries*), pertanyents als processos esquerre i dret, per a fer el càlcul del seu segment. Cada *boundary* té N píxels. La figura 32 mostra la distribució de les dades i les *boundaries* esquerra i dreta del procés 1.

Figura 32. Distribució de les dades i *boundaries* per a l'*edge-detection*



Estratègia de paral·lelització:

Per a fer la paral·lelització d'aquest codi seguirem l'estratègia següent per a cada procés:

1) Primer, intercanviar les *boundaries* amb els dos processos adjacents.

2) Segon, aplicar la funció `apply_template` al seu segment.

A continuació calcularem el temps d'execució d'aquesta paral·lelització (T_P). Per a això primer calcularem el temps de comunicació de les *boundaries*. Cada procés fa dos missatges, de N píxels cadascun, i tots els processos en paral·lel. El temps de comunicació és:

$$T_{comm} = 2(t_s + t_w N)$$

D'altra banda, cada procés, de manera independent i en paral·lel amb la resta de processos, aplicarà el *template* a cadascun dels píxels del seu segment. El temps de còmput és:

$$T_{cmput} = 9t_c \frac{N^2}{P}$$

Això ens porta al temps d'execució en paral·lel amb P processadors (T_P):

$$T_P = 9t_c \frac{N^2}{P} + 2 \times (t_s + t_w N)$$

I l'*speedup* i l'eficiència són:

$$\begin{aligned} S_P &= \frac{9 \times t_c \times N^2}{9 \times t_c \times \frac{N^2}{P} + 2 \times (t_s + t_w N)} \\ Eff_P &= \frac{1}{1 + \frac{2 \times P \times (t_s + t_w N)}{9 \times t_c \times N^2}} \end{aligned}$$

3.6.2. Exemple amb *blocking*: *Stencil*

En aquest cas analitzarem el temps d'execució en paral·lel del codi de l'*Stencil* (codi 3.3).

```
1  #include <math.h>
2  void compute( int N, double *u) {
3      int i, k;
4      double tmp;
5
6      for ( i = 1; i < N-1; i++ ) {
7         for ( k = 1; k < N-1; k++ ) {
8            tmp = u[N*(i+1) + k] + u[N*(i-1) + k] + u[N*i + (k+1)] + u[N*
                 i + (k-1)] - 4 * u[N*i + k];
9            u[N*i + k] = tmp/4;
10        }
11     }
12 }
```

Codi 3.3: Codi de *Stencil*.

La distribució de les dades entre els processos és per files. Cada procés ha de tractar N/P files consecutives de la matriu (o N^2/P elements, un segment). Perquè l'execució sigui eficient, cada procés processarà les files assignades en blocs de B columnes, tal com detallarem en l'estratègia de paral·lelització. En aquest cas també tenim *boundaries* per a cada procés, la fila immediatament anterior al seu segment (*boundary* superior) i la fila immediatament posterior al seu segment (*boundary* inferior).

Estratègia de paral·lelització:

En primer lloc, i només una vegada, cada procés farà la comunicació de la *boundary* inferior al procés predecessor. Aquesta comunicació es pot fer abans de començar el càlcul, ja que no hi ha dependències entre processos. A continuació, cada procés farà, per a cada bloc de B columnes, els passos següents:

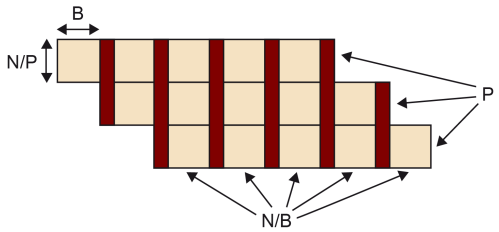
1) Esperar els B elements de l'última fila del bloc processat per part del procés superior (B elements de la *boundary* superior). En el cas del procés zero, aquest no té aquesta comunicació.

2) Aplicar l'algorisme de Stencil al bloc de $B \times \frac{N}{P}$ elements.

3) Enviar els B elements de l'última fila del bloc que s'acaba de processar al procés inferior. A excepció de l'últim procés.

La figura 33 mostra com es processen en el temps els blocs de cada processador. Les bandes verticals vermelles reflecteixen la comunicació de les *boundaries*. En la figura no es mostra la comunicació de les *boundaries* inferiors.

Figura 33. Execució paral·lela, usant blocs, de codi *Stencil*



Fent una simplificació de $N-2 \rightarrow N$, ens porta que el temps de comunicació de totes les *boundaries* inferiors en paral·lel és de:

$$T_{comm} = (t_s + Nt_w)$$

D'altra banda, el temps de còmput i comunicació per al conjunt de blocs per tractar és:

$$T_{blocs} = (\frac{N}{P}B)(\frac{N}{B} + P - 1)t_c + (\frac{N}{B} + P - 2)(t_s + t_w B)$$

El primer terme correspon al càlcul de tots els blocs del procés zero (N/B, en paral·lel amb tot el còmput superposat dels blocs en altres processos) més els blocs que s'han de fer per a acabar de processar tots els blocs per part de la resta de processadors ($P-1$). El segon terme correspon a la comunicació que hi ha entre bloc i bloc per processar. Aquestes comunicacions s'han de fer entre cada bloc processat ($N/B+P-1$ blocs) menys per a l'últim bloc. Cada comunicació és de B elements.

Llavors el temps total T_P, assumint que $P >>> 2$, és de:

$$\begin{aligned} T_p &\simeq (t_s + Nt_w) + (\frac{N}{P}B)(\frac{N}{B}+P)t_c + (\frac{N}{B}+P)(t_s+t_wB) \\ &= (t_s+Nt_w) + \frac{N^2}{P}t_c + NBt_c + t_s\frac{N}{B} + t_wN + t_sP + t_wPB \end{aligned}$$

Finalment, es podria calcular la mida de bloc òptim B_{opt} que fes minimitzar el temps d'execució T_P. Per a això, derivem la fórmula anterior i, assumint que $N >>> P$, la igualem a zero.

$$\begin{aligned} \frac{\partial T}{\partial B} &= Nt_c - t_s\frac{N}{B^2} + t_wP = 0 \\ B_{opt} &= \sqrt{\frac{t_sN}{Nt_c+t_wP}} = \sqrt{\frac{t_s}{t_c+t_w\frac{P}{N}}} \\ &\simeq \sqrt{\frac{t_s}{t_c}} \end{aligned}$$

Llavors, el temps paral·lel òptim T_{opt}* és de:

$$T_{opt} = t_s + Nt_w + (\frac{N^2}{P})t_c + 2N\sqrt{t_st_c} + t_wN + t_sP + t_wP\sqrt{\frac{t_s}{t_c}}$$

*** El temps calculat és un temps teòric que segurament s'hauria d'ajustar utilitzant la bibliografia complementària aconsellada per a la realització d'un model adaptat a l'arquitectura del computador i de l'algorisme.**

4. Principis de programació paral·lela

Quan paral·lelitzem una aplicació, dos dels principals objectius que es busquen són:

1) Millorar el rendiment de l'aplicació, maximitzant la concurrència i reduint els costos addicionals d'aquesta paral·lelització (amb la qual cosa maximitzarem l'*speedup* obtingut), tal com hem analitzat en l'apartat anterior.

2) Productivitat en el moment de programar: llegibilitat, portabilitat, independència de l'arquitectura de destinació.

Per a això, primer cal buscar una distribució adequada del treball/dades de l'aplicació (trobar la concurrència), després hem de triar l'esquema d'aplicació paral·lela més adequat (*task parallelism*, *divide and conquer*, etc.), i finalment, adaptar aquest algorisme a les estructures d'implementació conegudes (SPMD, *fork/join*, etc), i als mecanismes de creació, sincronització i finalització de les unitats de processament.

4.1. Concurrència en els algorismes

La idea és que, des de l'especificació del problema original, podem trobar una distribució del problema per al següent:

- Identificar tasques (parts del codi que poden ser executades concurrentment), com vam fer, per exemple, en els dos casos d'estudi de l'apartat anterior.

- Distribuir i analitzar les estructures de dades per a saber quines són les dades que necessiten la tasques, les dades que produeixen les tasques, i quines dades es comparteixen. Amb aquesta anàlisi podem intentar minimitzar els moviments de dades entre tasques, o les dades compartides.

- Determinar les dependències entre les tasques per a intentar determinar l'ordre entre aquestes i les sincronitzacions necessàries. És a dir, hem de determinar quin és el graf de dependència de tasques i que el resultat final de la nostra paral·lelització sigui el mateix que el del codi seqüencial.

Per a trobar la concurrència/paral·lelisme en un algorisme, primer hem de determinar si són les dades, les tasques, o el flux de dades els que determinen aquest paral·lelisme. Depenent del que determinem què és més important per a distribuir el treball, farem una distribució o una altra:

- *Data decomposition*: en aquest cas, les estructures de dades es particionen de tal manera que assignarem una tasca a cada partició feta.

- *Task decomposition*: en aquest cas, s'identifiquen parts del codi com a tasques, i això implicarà una distribució de les dades.

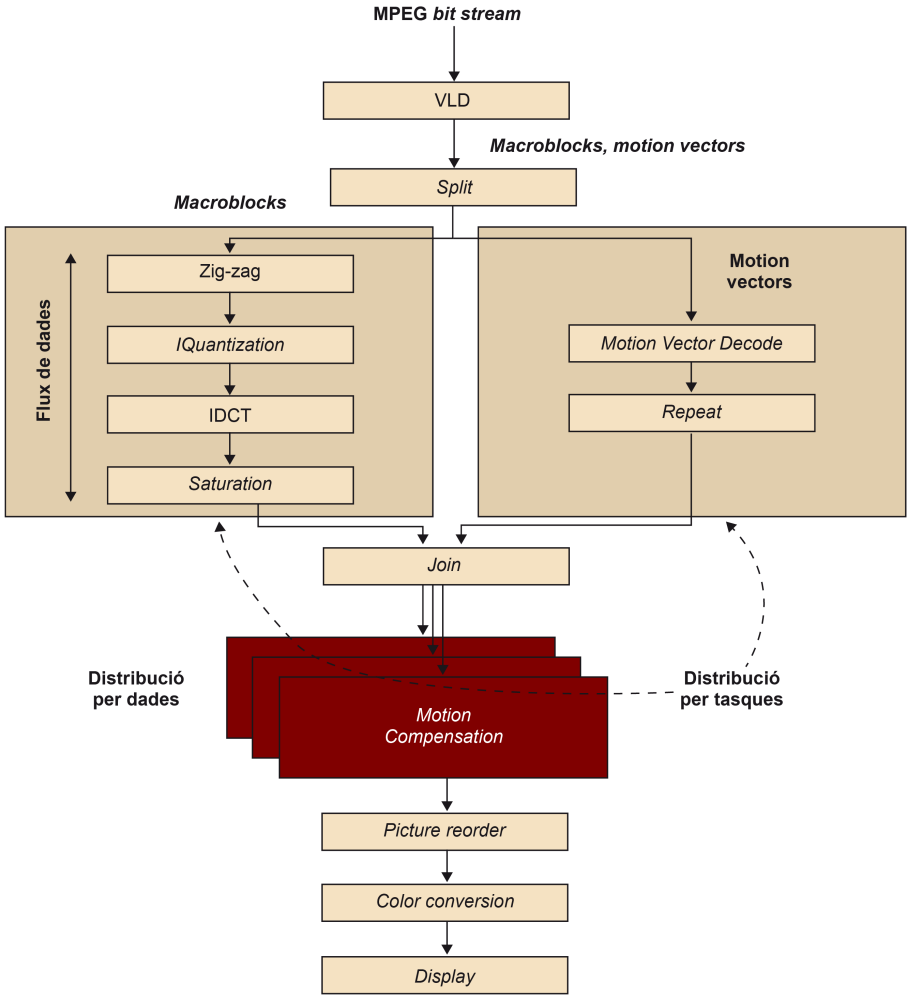
- *Data-flow decomposition*: en aquest cas, el flux de les dades comportarà que unes tasques s'activin i facin el procés d'aquestes dades.

Combinació de formes

Una mateixa aplicació pot combinar diferents maneres de distribuir el treball per fer: *data decomposition*, *task decomposition* i *data-flow decomposition*.

En qualsevol cas, al final tindrem una sèrie de tasques, que possiblement poden tenir dependències de control i de dades entre si. A més, és possible que en un mateix problema tinguem tots tres tipus de distribucions, tal com mostrem en la figura 34. En aquesta figura mostrem com el programa de processament de vídeo MPEG pot tenir una distribució de treball per dades en la part de *motion*, distribució en tasques en la part macroblocs i vectors, i flux de dades, entre les funcions que estan dins del processament dels macroblocs.

Figura 34. MPEG i la distribució en tasques, dades i flux de dades



4.1.1. Bones pràctiques

Una vegada hem identificat les distribucions per fer. Seria bo seguir algunes bones practiques quan fem la distribució en tasques:

- **Flexibilitat:** ser flexibles en nombre i mida de les tasques generades. Per exemple, el nombre de tasques no hauria de ser gaire dependent de l'arquitectura específica triada. A més, aquestes tasques seria bo que anessin parametritzables.

- **Eficiència:** pensar en l'eficiència del nostre paral·lelisme, en el sentit que cadascuna d'aquestes tasques tingui el treball suficient per a poder amortitzar la creació i l'administració d'aquestes tasques. A més, aquestes tasques haurien de ser prou independents perquè el fet d'haver de gestionar aquestes dependències no fos un coll d'ampolla.

- **Simplicitat:** fer una paral·lelització que sigui fàcil de mantenir i de depurar si es dóna el cas.

Quant a la distribució en dades, algunes guies de bones pràctiques són:

- **Flexibilitat:** ser flexibles en el nombre i mida de particions* de les dades.

*** `Chunks`, en anglès**

- **Eficiència:** pensar també en l'eficiència de la nostra paral·lelització com a resultat d'aquesta distribució. Per exemple, hauríem de veure si aquest particionament de les dades significa tenir o no desequilibri en la càrrega de treball de les tasques. A més, en el cas de distribució per dades, se sol considerar l'arquitectura del nostre computador per a poder tenir en compte la jerarquia de memòria d'aquest.

- **Simplicitat:** fer una distribució no gaire complexa, ja que en cas contrari la depuració del programa pot ser difícil.

4.1.2. Ordre i sincronització de tasques

Una vegada enunciades aquestes bones pràctiques, el primer pas que hem d'efectuar després de fer la distribució en tasques és analitzar quin és l'ordre entre aquestes, i a més, quines són les restriccions de compartició de dades d'aquestes tasques. En cas de tenir algun tipus de restricció d'ordre o de compartició de dades, haurem de veure quins mecanismes de sincronització s'han d'usar. També pot ser que ens trobem que hi hagi una paral·lelització en què no s'hagi de fer cap tipus de sincronització, i totes la tasques són completament independents; en aquest cas estem parlant d'un paral·lelisme enutjós. Això no treu que hàgim de tenir en compte aspectes d'equilibri de càrrega o de localitat de dades que es poden traduir en una paral·lelització poc eficient del programa.

Paral·lelisme enutjós

Embarransingly parallel o paral·lelisme enutjós és quan la paral·lelització d'un programa es pot fer de tal manera que les tasques no necessiten cap tipus de comunicació o sincronització entre elles. El fet que tinguem un paral·lelisme enutjós no treu que puguem tenir problemes de desequilibri de càrrega o de localitat de dades.

Per a aconseguir la sincronització entre tasques, podem seqüencialitzar l'execució d'aquestes tasques, o bé fer algun tipus de sincronització global. En l'apartat de models

*** *Runtime* en anglès**

de programació paral·lela veurem exemples de com pot fer el programador aquesta sincronització en un sistema de memòria compartida i distribuïda. També veurem un exemple en què el programador deixa la biblioteca de gestió de recursos i tasques del model de programació* perquè gestioni les sincronitzacions entre tasques segons les dependències d'unes pel que fa a les altres, indicades pel programador.

4.1.3. Compartició de dades entre tasques

Pel que fa a la compartició de dades en memòria compartida, tots els *threads* tenen accés a totes les dades en la memòria compartida. Per tant, han de fer algun tipus de sincronització en l'accés compartit per a evitar *race conditions* o condicions de cursa, que veurem en el subapartat següent. En el cas de memòria distribuïda, cada processador té les dades a les quals pot accedir en la seva memòria local, per la qual cosa no hi pot haver condicions de cursa. En canvi, hi ha d'haver una comunicació explícita per a poder compartir les dades.

Per a la memòria compartida, una manera d'evitar les *race conditions* és usant zones d'exclusió mútua en les seccions crítiques. Una secció crítica és una seqüència d'instruccions en una tasca que pot entrar en conflicte amb una seqüència d'instruccions en una altra tasca, i crear una possible condició de cursa. Per a entrar en conflicte dues seqüències d'instruccions, una, almenys, ha de modificar algun dada, i l'altra llegir-la. Una zona d'exclusió mútua és un mecanisme que assegura que solament una tasca alhora executa el codi que es troba en una secció crítica.

En el moment de fer una exclusió mútua ens podem trobar que hem provocat un *deadlock*, que explicarem en el subapartat següent. A més, ens podem trobar amb un problema de rendiment si fem un gran nombre d'exclusions mútues o bé la mida de codi que abasten és extens.

Deadlock

Un *deadlock* es produeix quan una tasca necessita i espera un recurs que té una altra tasca, al mateix temps que aquesta última necessita i espera un recurs de la primera.

En l'apartat dedicat als models de programació veurem exemples de com podem fer zones d'exclusió mútua en el cas de memòria compartida.

Vegeu també

Els models de programació es tracten en l'apartat 5.

4.2. Problemes que apareixen en la concurrència

Quan apareix la concurrència en el càlcul o comunicació d'una sèrie d'accions ens podem trobar amb quatre problemes típics, que comentem a continuació.

4.2.1. *Race condition*

Race condition o condició de cursa: múltiples tasques llegeixen i escriuen una mateixa dada, el resultat final de la qual depèn de l'ordre relatiu de l'execució. La manera de solucionar aquest problema és forçar algun mecanisme per a ordenar/sincronitzar els accessos a aquestes dades.

Un exemple típic del problema de *race condition* és extreure diners d'un mateix compte d'un banc des de dos caixers diferents. En aquest cas, si el banc no ofereix un mecanisme per a sincronitzar els accessos al compte en qüestió, ens podríem trobar amb la situació que dues persones podrien treure tants diners com els diners que disposem en un cert moment en el nostre compte. Això implicaria que podríem treure fins al doble de la quantitat de què es disposava en el compte.

4.2.2. *Starvation*

Starvation o mort per inanició: una tasca no pot aconseguir accedir a un recurs compartit i, per tant, no pot avançar. Aquest problema és més difícil de solucionar, i normalment se sol fer amb mecanismes de control del temps que es passa en el bloqueig. Una situació en la qual ens podríem trobar en *starvation* és justament en la situació anterior dels caixers. Imaginem que una de les persones accedeix al compte des d'un caixer. Una altra persona accedeix des d'un altre caixer i es queda bloquejada mentre que la primera persona es pensa els moviments que vol fer. En aquest cas, la persona que s'ha quedat bloquejada es pot mantenir a l'espera per un temps indeterminat, tret que se'n vagi i ho provi en un altre moment.

4.2.3. *Deadlock*

Deadlock o condició de bloqueig: dues o més tasques no poden continuar perquè unes estan esperant que les altres facin alguna cosa. Una manera d'evitar condicions de bloqueig és mitjançant una ordenació adequada de les sincronitzacions que provoquen aquest bloqueig. Per exemple, pensem en la transferència de diners d'un compte d'una persona al compte d'una altra persona. En aquesta transferència hem de bloquejar el compte de la persona origen i el compte de la persona destinació, per a poder actualitzar els diners de tots dos comptes. Si es donés el cas que la persona del compte destinació de la transferència intenta al seu torn, i en paral·lel, fer una transferència des del seu compte al compte de la primera persona, podríem trobar una condició de bloqueig. Això és a causa que cada persona, per separat, bloqueja primer el seu compte, i després intenta el bloqueig del compte destinació. En el moment d'intentar bloquejar el compte destinació, totes dues persones es queden bloquejades, ja que s'ha forçat un cicle en els bloquejos.

Una manera d'evitar aquest problema, per a aquesta situació concreta, és ordenar els bloquejos per fer segons un criteri que faci que les dues persones intentin bloquejar primer el **mateix** compte, i després l'altre compte. Així, una persona podrà fer la transferència i l'altra es quedarà bloquejada en el primer intent de bloquejar un compte.

4.2.4. *Livelock*

Livelock o condició de bloqueig però amb continuació: dues o més tasques canvien contínuament d'estat en resposta dels canvis produïts en altres tasques però no aconsegueixen fer cap treball útil.

L'exemple típic és el dels 5 filòsofs asseguts en una taula rodona. Entre cada dos filòsofs contigus hi ha un cobert. Cada filòsof fa dues coses: pensar i menjar. Així, cada filòsof pensa durant una estona, i quan els filòsofs tenen gana, paren de pensar i, en aquest ordre, agafen el cobert de l'esquerra i després el de la dreta. Si, pel que fos, un d'aquests coberts l'hagués agafat ja un filòsof, llavors haurà d'esperar fins a poder disposar dels dos coberts. Quan un filòsof té els dos coberts, llavors pot menjar. Una vegada que ha menjat, pot tornar a deixar els coberts i continuar pensant.

En el cas que tots els filòsofs agafin un cobert alhora, tots els filòsofs tindran un cobert en una mà i hauran d'esperar a tenir l'altre cobert amb tal de poder menjar. Amb la qual cosa, estem en una situació de bloqueig *deadlock*. Per a evitar el problema de *deadlock* podem fer que cada filòsof deixi el cobert, esperi 5 minuts i després ho intenti una altra vegada. Com podem observar, els filòsofs canviaran d'estat entre bloqueig i no-bloqueig, però no aconseguiran menjar. En aquest cas estan en un *livelock*.

Una possible solució és fer el mateix que fèiem per a solucionar el problema del *deadlock*: determinar un ordre en el moment d'agafar els coberts que faci que els filòsofs no es puguin quedar bloquejats.

4.3. Estructura dels algorismes

Segons el tipus de tasques concurrents, i en funció del tipus de distribució que hàgim fet (en tasques o en dades), podem classificar les estratègies de paral·lelització segons el seu patró.

Segons la distribució feta tenim:

- Distribució en tasques: patrons *task parallelism* i *divide & conquer*.
- Distribució en dades: patrons linear o *geometric decomposition* i *recursive decomposition*.
- Distribució per flux de dades: patrons *pipeline* i *event based*.

4.4. Estructures de suport

Patrons de programació

Els patrons de programació SPMD, *fork/join*, *master/workers* i *loop parallelism* es poden combinar en la implementació d'un programa.

Per a poder implementar tots aquests patrons, normalment es treballa amb uns esquemes de programació que bàsicament són l'esquema *single program multiple data* (SPMD), l'esquema *fork/join*, l'esquema *master/workers* i finalment l'esquema *loop parallelism*. En qualsevol cas, és possible que dos o més esquemes es puguin combi-

nar, com per exemple un patró *master/worker* podria ser implementat amb un esquema SPMD o *fork/join*.

4.4.1. SPMD

En aquest esquema, tots els *threads*/processos o unitats d'execució (UE) executen, bàsicament, el mateix programa, però sobre diferents dades. Encara que tots executen el mateix programa poden variar en el camí d'execució seguit, segons l'identificador de la UE (per exemple, l'identificador de procés o l'identificador de *thread*), a l'estil del codi 4.1.

```
1    ...
2    if (my_id==0)
3    {
4        /* Master */
5    }
6    else
7    {
8        /* Worker */
9    }
10   ...
```

Codi 4.1: Exemple bàsic d'un esquema SPMD

4.4.2. *Master/workers*

Un procés/*thread* `master` crea un conjunt de *threads*/processos perquè facin una borsa de tasques. Aquest conjunt de *threads* o processos van prenent tasques de la borsa de treball i les fan fins que no en quedin més en la borsa de tasques.

En el cas que les tasques es generessin de manera dinàmica o bé el nombre total de tasques no es conegués *a priori*, seria necessari algun mecanisme per a indicar als *threads*/processos que no hi ha més tasques per fer.

4.4.3. *Loop parallelism*

Aquest esquema és típic de programes seqüencials amb bucles de còmput intensiu. En aquest cas, moltes vegades fem que les iteracions del bucle es puguin executar en paral·lel.

4.4.4. *Fork/join*

L'esquema *Fork/join* consisteix que un procés o *thread* principal crearà un conjunt de processos/*threads* per a fer una porció de treball, i normalment s'espera que aquests acabin.

5. Models de programació paral·lela

A continuació farem una breu explicació d'MPI i OpenMP, que són els models de programació més acceptats per a memòria distribuïda i memòria compartida, respectivament. També mostrarem algunes pinzellades d'una extensió del model OpenMP (OpenMP Superescalar, OmpSs) amb tal de poder fer control de dependències entre tasques en temps d'execució. Per a aprofundir en aquests models de programació paral·lela aconsellem la lectura de la bibliografia bàsica.

5.1. OpenMP

OpenMP consisteix en una extensió API dels llenguatges C, C++ i Fortran per a escriure programes paral·lels per a memòria compartida. OpenMP es troba inclòs en el compilador GCC, igual que molts altres compiladors de propietat, com l'ICC d'Intel, el XLC d'IBM, etc. Bàsicament inclou suport per a crear automàticament *threads*, compartir treball entre aquests, i sincronitzar els *threads* i la memòria.

En aquest subapartat solament donarem alguns exemples sense entrar en un gran detall, ja que no és objectiu exclusiu d'aquest mòdul.

5.1.1. Un programa simple

El codi 5.1 mostra un "hola món" escrit en OpenMP, en què tots els *threads* creats escriuran aquest missatge.

```
1  #include <omp.h>
2
3  int rank;
4  int nproc;
5
6  int main( int argc, char* argv[] ) {
7
8
9      #pragma omp parallel
10     {
11        int thread_id;
12        int num_threads;
13
14        /* Nothing to do */
15
16        thread_id   = omp_get_thread_num();
17        num_threads = omp_get_num_threads();
18        printf("thread: %d de un total de %d: Hola mundo!\n",thread_id,
              num_threads);
19
```

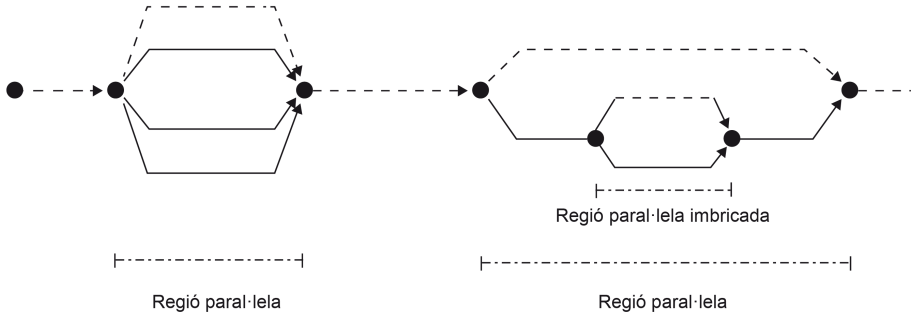
```
20         }
21
22 }
```

Codi 5.1: Programa simple en OpenMP

La directiva `omp parallel` fa que el compilador insereixi codi per a generar un conjunt de *threads*, tants com indiqui la variable d'entorn `OMP_NUM_THREADS`. D'aquesta manera, si compilem i executem aquest codi amb `OMP_NUM_THREADS=4 ./omp-program`, haurien d'aparèixer quatre missatges "hola món". Hi ha altres maneres d'indicar el nombre de *threads*. Una és utilitzant la funció d'OpenMP `omp_set_num_threads(number)`. L'altra és, en el moment de posar la directiva `parallel`, indicar la clàusula `num_threads(number)`.

El model de programació OpenMP és un model *fork/join*, en què podríem tenir directives `parallel` imbricades, tal com observem en la figura 35.

Figura 35. Model *fork/join* del model de programació OpenMP



En la directiva `parallel` es pot especificar si les variables del programa, declarades fora del context del `parallel` són privades (`private(var1,var2, ...)`), compartides (`shared(var1,var2, ...)`), o són privades però s'ha de copiar el valor que tenen en aquest moment (`firstprivate(var1,var2, ...)`).

5.1.2. Sincronització i *locks*

El fet de tenir diversos *threads* que poden estar compartint una mateixa posició de memòria pot portar a tenir condicions de cursa. Per a això, necessitem algun tipus de sincronització o *locks* per a aconseguir un ordre en els accessos.

Les tres directives OpenMP de sincronització són:

- `barrier`: els *threads* no poden passar el punt de sincronització fins que tots arribin a la barrera i tot el treball anterior s'hagi completat. Algunes construccions tenen una barrera implícita al final, com per exemple el `parallel`.

- `critical`: crea una regió d'exclusió mútua en què solament un *thread* pot estar treballant en un instant determinat. Es pot assignar un nom a la zona d'exclusió amb `critical (name)`. Per defecte totes tenen el mateix nom, i per tant, qualsevol punt del programa en què ens trobem un `critical` voldrà dir que solament un *thread* pot entrar.

- `atomic`: implica que l'operació simple a la qual afecti (operació del tipus llegir i actualitzar) es faci de manera atòmica.

El codi 5.2 mostra la versió OpenMP del càlcul de `pi` en el qual s'usa el `critical`. Per a l'`atomic` seria canviar la directiva `critical` per `atomic`.

```
1
2 void main ()
3 {
4    int i, id;
5    double x, pi, sum=0.0;
6
7    step = 1.0/(double) num_steps;
8    omp_set_num_threads(NUM_THREADS);
9    #pragma omp parallel private(x, i, id)
10   {
11      id = omp_get_thread_num();
12      for (i=id+1; i<=num_steps; i=i+NUM_THREADS) {
13         x = (i-0.5)*step;
14         #pragma omp critical
15         sum = sum + 4.0/(1.0+x*x);
16      }
17   }
18   pi = sum * step;
19 }
20 }
```

Codi 5.2: Exemple de sincronització amb OpenMP

En aquest codi, cada *thread* fa les iteracions del bucle `for` en funció del seu identificador dins del conjunt de *threads*, creat amb la directiva `parallel`. La variable `sum`, que és la variable en què els *threads* faran el càlcul de `pi`, està compartida per defecte. Per a evitar una condició de cursa en el càlcul, s'ha d'actualitzar via `critical` o `atomic`. D'altra banda, les variables `id, x, y` també serien compartides per defecte si no s'haguessin privatitzat explícitament. En cas de compartir-se, el codi no seria correcte, ja que tots estarien actualitzant la variable *inducció*.

Una altra manera de fer el càlcul de la variable `sum`, sense necessitat de fer una exclusió mútua és fent una reducció amb la clàusula `reduction`. El codi 5.3 mostra com quedaria.

```
1 void main ()
2 {
3    int i, id;
4    double x, pi, sum;
5
6    step = 1.0/(double) num_steps;
```

```
7     omp_set_num_threads(NUM_THREADS);
8     #pragma omp parallel private(x, i, id) reduction(+:sum)
9     {
10       id = omp_get_thread_num();
11       for (i=id+1; i<=num_steps; i=i+NUM_THREADS) {
12          x = (i-0.5)*step;
13          sum = sum + 4.0/(1.0+x*x);
14       }
15    }
16    pi = sum * step;
17 }
```

Codi 5.3: Exemple de reducció amb OpenMP

La clàusula `reduction` crea una còpia privada de la variable, de tal manera que s'actualitza aquesta variable local, i després s'actualitza de manera sincronitzada la variable compartida `sum`, i es fa així la reducció. En la clàusula `reduction` s'indica quin tipus d'operació s'ha de fer.

Una altra manera de sincronitzar-se és utilitzant *locks*. OpenMP disposa de primitives *lock* per a sincronitzacions de baix nivell. Les funcions són les següents:

- `omp_init_lock`: inicialitza el *lock*.
- `omp_set_lock`: adquireix el *lock*, i es pot quedar bloquejat.
- `omp_unset_lock`: allibera el *lock*, i pot desbloquejar *threads* que estiguin bloquejats.
- `omp_test_lock`: prova a adquirir un *lock*, però no es bloqueja.
- `omp_destroy_lock`: allibera els recursos del *lock*.

El codi 5.4 mostra un exemple d'ús de les operacions de *lock*.

```
1  #include <omp.h>
2  void foo ()
3  {
4     omp_lock_t lock;
5
6     omp_init_lock(&lock);
7     #pragma omp parallel
8     {
9         omp_set_lock(&lock);
10        // Región de exclusión mútua.
11        omp_unset_lock(&lock);
12    }
13    omp_destroy_lock(&lock);
14 }
```

Codi 5.4: Exemple de sincronització amb *locks* amb OpenMP

5.1.3. Compartició de treball en bucles

Les construccions de compartició de treball divideixen l'execució d'una regió de codi entre els *threads* d'un equip. D'aquesta manera els *threads* cooperen per a fer un treball sense necessitat d'haver d'identificar-se, tal com passava en el codi 5.2.

OpenMP té quatre construccions *worksharing: loop worksharing*, *single*, *section* i *master*. D'aquestes veurem exemples de *loop* i *single*. Les altres dues no se solen usar.

En el codi 5.5 mostrem un exemple d'ús del *loop worksharing* (`omp for`). Les iteracions del bucle associades a aquesta directiva es divideixen entre els *threads* del conjunt de *threads* creat, i es fan totalment en paral·lel. És responsabilitat de l'usuari que les iteracions siguin independents, o que d'alguna manera se sincronitzin si és necessari. A més, el bucle ha de ser un bucle `for` en què es pugui determinar quantes iteracions hi ha. La variable d'inducció (la `i` en el cas de l'exemple) ha de ser de tipus enter, punter o iteradors (C++). Aquesta variable és automàticament privatitzada. La resta de variables són *shared* per defecte.

```
1  #include <omp.h>
2  static long num_steps = 100000;
3  double step;
4  #define NUM_THREADS 2
5
6  void main ()
7  {
8     int i, id;
9     double x, pi, sum;
10
11    step = 1.0/(double) num_steps;
12    omp_set_num_threads(NUM_THREADS);
13    #pragam omp parallel
14    {
15      #pragma omp for private(x) reduction(+:sum)
16      for (i=1; i<=num_steps; i++) {
17          x = (i-0.5)*step;
18          sum = sum + 4.0/(1.0+x*x);
19      }
20    }
21    pi = sum * step;
22 }
```

Codi 5.5: Exemple de la directiva `omp for` d'OpenMP.

La manera de distribuir els *threads* és, per defecte, *schedule* (*static*), de tal manera que en temps de compilació es determina quines iteracions corresponen als *threads* existents. Aquest tipus de distribució consisteix a donar, en el cas de l'exemple, `num_steps/#threads` iteracions consecutives a cada *thread*. Hi ha altres tipus de *schedule* que es poden especificar a la directiva `for`. Aquests tipus són: *dynamic* i *guided*.

Dynamic i guided

El *schedule* (*dynamic*) o *schedule* (*guided*) són tipus de distribució dinàmica de les iteracions, i intenten reduir el desequilibri de càrrega entre els *threads*.

El *loop worksharing*, per defecte, té un *barrier* implícit al final. Si volguéssim que aquesta barrera desaparegués li hauríem d'indicar la clàusula `nowait` en l'`omp for`. Una altra clàusula interessant és la de `collapse`, que és per a quan tenim dos bucles

imbricats perfectes (el cos del bucle extern és únicament el bucle intern). En aquest cas, l'espai d'iteracions es col·lapsa com si fos un únic *loop* amb tantes iteracions com el producte de les iteracions dels dos bucles.

Finalment, una altra construcció que cal destacar per al cas d'haver de compartir treball és la d'indicar que solament volem que un dels *threads* faci una cosa determinada. Això ho podem fer amb la directiva `single`. En veurem un exemple en el subapartat següent.

5.1.4. Tasques en OpenMP

Les *tasks* són unitats de treball que poden ser executades immediatament, o bé deixades en una cua perquè siguin executades posteriorment. Són els *threads* del conjunt de *threads* creats amb el `parallel` els que cooperen en l'execució d'aquestes tasques.

En realitat, quan es fa un `parallel` es creen tasques implícites, a les quals s'assigna treball per fer. En OpenMP es poden crear tasques explícites amb la directiva *task*. Amb les tasques *tasks*, quan un *thread* la troba, empaqueta el codi i les dades, i crea una nova tasca perquè sigui executada per un *thread*.

Normalment s'utilitza aquest *pragma* quan es fan tasques que no estan dins d'un *loop* amb una variable inducció que permeti utilitzar `omp for`.

El codi 5.6 mostra una manera d'utilitzar la directiva `task`.

```
1  ...
2  ...
3  #pragma omp parallel
4  #pragma omp single
5  traverse_list ( l );
6  ...
7  ...
8  void traverse_list ( List l )
9  {
10   Element e ;
11   for ( e = l->first ; e ; e = e ->next  )
12        #pragma omp task
13          process ( e );
14 }
```

Codi 5.6: Exemple de creació de tasques explícites en OpenMP

En aquest codi es creen un conjunt de *threads* amb l'`omp parallel`. Per a evitar que tots els *threads* executin el codi `traverse_list`, i per tant, dupliquin el treball, s'ha d'utilitzar la directiva `omp single`, tal com mostrem en el codi 5.6. D'aquesta manera solament un *thread* es recorrerà la llista i crearà les tasques.

Per a cada crida a la funció `process(e)` es crea una tasca explícita que algun dels *threads* del `parallel` executarà. I com ens esperem que acabin? Hi ha dues construccions que ens poden ajudar a esperar-les:

1) `#pragma omp barrier`: amb aquesta directiva fem que tots els *threads* del `parallel` s'esperin que tot el treball anterior, incloent-hi les tasques, es completi.

2) `#pragma omp taskwait`: en aquest cas, es fa suspendre la tasca actual fins que totes les tasques fill (directe, no descendents) s'hagin completat.

pragma omp task

Quan utilitzem el `pragma omp task` hem de vigilar si volem que tots els *threads* d'un `omp parallel` creïn les tasques o no. En cas que no, hauríem d'usar el `pragma omp single`.

El codi 5.7 mostra com podem esperar les tasques.

```
1
2  void traverse_list ( List l )
3  {
4    Element e ;
5    for ( e = l->first ; e ; e = e->next )
6         #pragma omp task
7           process ( e );
8
9    #pragma omp taskwait
10   //  En este punto todas las tareas han acabado.
11 }
```

Codi 5.7: Exemple de sincronització de tasques en OpenMP

Resum

En aquest mòdul hem repassat els diferents nivells de paral·lelisme que s'han incorporat al maquinari d'un uniprocessador amb la intenció de millorar el rendiment de les aplicacions. No obstant això, també hem vist que, a causa que els guanys de rendiment no eren els esperats i a més l'increment del consum energètic era significatiu, l'estratègia seguida fins a l'any 2000 de millorar els uniprocessadors no es podia mantenir. A partir d'aquest punt, hem descrit la classificació de les màquines paral·leles segons Flynn, i ens hem centrat a veure com es poden paral·lelitzar les aplicacions i com se'n mesura el rendiment i l'eficiència.

En particular, hem vist que per a paral·lelitzar una aplicació necessitem mecanismes de creació de tasques, i també tenir en compte els problemes de concurrència que sorgeixen com a conseqüència d'aquesta paral·lelització. Com a part dels mecanismes de creació de tasques, hem introduït els models de programació paral·lela més coneguts per a memòria compartida (OpenMP).

També hem detallat els passos per a analitzar el paral·lelisme potencial en una aplicació, i diferents estratègies de distribució de treball en tasques i/o en dades, amb la finalitat de paral·lelitzar les aplicacions de manera eficient i adequada.

Finalment, hem presentat un cas d'estudi senzill amb l'objectiu de veure com es pot paral·lelitzar un mateix programa utilitzant diferents estratègies i models de programació paral·lela, la qual cosa introdueix conceptes essencials per a la programació d'arquitectures *many-core* introduïdes en l'assignatura, com per exemple el cas d'Intel Xeon Phi o arquitectures tipus GPU.

Exercicis d'autoavaluació

1. Hem de fer la suma dels elements d'un vector $X[0]$ ... $X[n-1]$. L'algorisme seqüencial es mostra en el codi 5.8.

```
1  sum = 0;
2  for (i=0; i<n ; i++)
3     sum += X[i]
```

Codi 5.8: Suma d'elements d'un vector

Dibuixeu el graf de dependències de tasques, calculeu T_1, T_∞ i el paral·lelisme potencial.

2. per a solucionar el mateix problema de l'exercici anterior, és a dir, la suma dels elements d'un vector $X[0]$ ... $X[n-1]$, es podria plantejar una solució en arbre, de tal manera que primer se suma per parells, després grups de dos parells, etc. Plantegeu una solució iterativa i una solució recursiva per a solucionar el problema d'aquesta manera. Després dibuixeu el graf de dependències de tasques, calculeu T_1, T_∞ i el paral·lelisme potencial.

3. Feu les versions paral·leles amb OpenMP de les solucions de l'exercici anterior.

4. Hem de fer un algorisme que calculi el producte matriu per vector, que es mostra en el codi 5.9.

```
1  for (i=0; i<n; i++)
2      y[i]=0;
3
4  for (i=0; i<n ; i++)
5    for (j=0; j<n ; j++)
6      for (k=0; k<n ; k++)
7        y[i]+= A[i][k] * b[k];
```

Codi 5.9: Producte matriu per vector

Per a les granularitats següents en la paral·lelització, calculeu T_1, T_∞ i el paral·lelisme.

- Gra molt fi: cada iteració dels bucles més interns és una tasca.
- Gra fi: el càlcul de tot un element $y[i]$ és una tasca.
- Gra gruixut: el càlcul de tres elements consecutius de $y[i]$ és una tasca.

5. Analitzeu els codis següents i indiqueu si hi ha dependència de dades. Per a cada dependència de dades indiqueu de quin tipus és, la distància i d'on a on va. Finalment, una vegada analitzades les dependències de dades, indiqueu quins codis són vectoritzables i escriviu-ne el codi vectorial. N és una constant que pot tenir qualsevol valor.

```
\textbf{a)}
char *a, *b, i;
...
for (i=16; i<N; i++)
    a[i] = a[i-16] + b[i];
```

```
\textbf{b)}
double *a, i;
...
for (i=N-2; i>=3; i--)
     a[i] = a[i+2] + a[i-3];
```

```
\textbf{c)}
int *a, *b, i;
...
for (i=1; i<(N-1); i++)
     a[i] = a[i-1] + b[i];
```

Bibliografia

Chapman, Barbara; Jost, Gabriele; Pas, Ruud van der (2008). *Using `OpenMP`*. The MIT Press.

Grama, Ananth; Gupta, Anshul; Karypis, George; Kumar, Vipin (2003). *Introduction to Parallel Computing. Second Edition.*. Pearson: Addison Wesley.

Jost, Gabriele; Jin, H.; Mey, D.; Hatay, F. (2003). *Comparing `OpenMP`, `MPI`, and Hybrid Programming*. Proc. Of the 5th European Workshop on OpenMP.

Mattson, Timothy G.; Sanders, Beverly A.; Massingill, Berna L. (2009). *Patterns for Parallel Programming*. Pearson: Addison Wesley.

Pacheco (2011). *An Introduction to Parallel Programming*. Morgan Kaufmann.

Quinn, M. J. (2008). *Parallel Programming in C with `MPI` and `OpenMP`*. McGraw-Hill.

Tanenbaum, Andrew S. (2006). *Structured Computer Organization*. Pearson, Addison Wesley.