# Introducció a la computació d'altes prestacions

PID_00250607

Ivan Rodero Castro
Francesc Guim Bernat

Temps mínim de dedicació recomanat: 5 hores

Universitat Oberta de Catalunya

**Ivan Rodero Castro**

Enginyer d'informàtica i doctor per la Universitat Politècnica de Catalunya. Ha impartit docència a la Facultat d'Informàtica de Barcelona (UPC) d'assignatures dels àmbits d'arquitectura de computadors, sistemes operatius i sistemes paral·lels i distribuïts, tant a grau com a màster i doctorat. Des del 2009 fa docència i recerca a Rutgers University (Nova Jersey), on és també el director associat del Rutgers Discovery Informatics Institute. Consultor dels Estudis d'Informàtica, Multimèdia i Telecomunicació de la Universitat Oberta de Catalunya des del 2010. Centra la seva recerca en l'àrea dels sistemes paral·lels i distribuïts, incloses la computació d'altes prestacions, la *green computing*, la informàtica en núvol i les dades massives.

**Francesc Guim Bernat**

Enginyer d'Informàtica i doctor per la Universitat Politècnica de Catalunya. Ha impartit docència a la Facultat d'Informàtica de Barcelona (UPC) en assignatures dels àmbits d'arquitectura de computadors, sistemes operatius i sistemes paral·lels i distribuïts, tant a grau, com a màster i doctorat. Des de 2008 fa docència com a consultor dels Estudis d'Informàtica, Multimèdia i Telecomunicació de la Universitat Oberta de Catalunya. Des de l'any 2008 és arquitecte de processadors a la companyia Intel Corporation.

# Índex

# Introducció

En aquest primer mòdul didàctic estudiarem les motivacions i característiques de la computació d'altes prestacions i en repassarem els fonaments, tant pel que fa a l'arquitectura com a la programació, per tal de posar en context i vertebrar els continguts dels següents mòduls didàctics.

Estudiarem les arquitectures dels sistemes paral·lels i altres conceptes de rellevància, com ara les xarxes d'interconnexió. També estudiarem conceptes fonamentals relacionats amb el rendiment de sistemes d'altes prestacions, i també mètriques i el paper que tenen les eines d'anàlisi de rendiment.

Un cop presentats els sistemes paral·lels, ens centrarem en la programació d'aquests sistemes. Estudiarem els conceptes fonamentals dels models de programació, tant per a memòria compartida (per exemple, OpenMP) com per a memòria distribuïda (per exemple, MPI).

Finalment estudiarem els reptes actuals més importants de la computació d'altes prestacions, com ara els relacionats amb la gestió de paral·lelisme massiu o les limitacions quant a la disponibilitat d'energia.

# Objectius

Els materials didàctics d'aquest mòdul contenen les eines necessàries per a assolir els objectius següents:

**1.** Entendre les motivacions de la computació d'altes prestacions i del paral·lelisme.

**2.** Conèixer els fonaments del paral·lelisme i les arquitectures paral·leles, tant els relacionats amb sistemes de memòria compartida com de memòria distribuïda.

**3.** Conèixer les característiques dels sistemes paral·lels, com ara la jerarquia de memòria i les xarxes d'interconnexió.

**4.** Entendre les diferències entre sistemes paral·lels i distribuïts.

**5.** Conèixer els models de programació per a sistemes d'altes prestacions, tant de memòria compartida com de memòria distribuïda.

**6.** Conèixer els fonaments relacionats amb el rendiment de sistemes d'altes prestacions i de l'anàlisi de rendiment.

**7.** Conèixer el reptes actuals de la computació d'altes prestacions.

# 1. Motivacions de la computació d'altes prestacions

En general, la computació d'altes prestacions ha estat motivada per la contínua i creixent demanda de prestacions i velocitat de computació necessàries per a resoldre problemes computacionals cada cop més complexos en un temps raonable. Aquesta gran demanda de computació és requerida per àrees com ara models numèrics i simulació de problemes de ciència i enginyeria.

## 1.1. Utilitat de la simulació numèrica

La simulació numèrica es considera el tercer pilar de la ciència, a més de l'experimentació o observació i la teoria. El paradigma tradicional de la ciència i enginyeria està basat en la teoria o disseny en paper per a fer experiments després o crear el sistema. Aquest paradigma té nombroses limitacions envers el problema que cal resoldre, com ara:

- El problema és molt difícil, com per exemple construir túnels de vent complexos.
- El problema és molt car econòmicament, com per exemple fabricar un avió només per a experimentar.
- El problema és massa lent, com per exemple esperar l'efecte del canvi climàtic o l'evolució de galàxies.
- El problema és molt perillós, com per exemple prova d'armes, disseny de fàrmacs o experimentació amb el medi ambient.

El paradigma de la ciència computacional està basat a **utilitzar sistemes d'altes prestacions** per a simular el fenomen volgut. Per tant, la ciència computacional es basa en les lleis de la física i mètodes numèrics eficients.

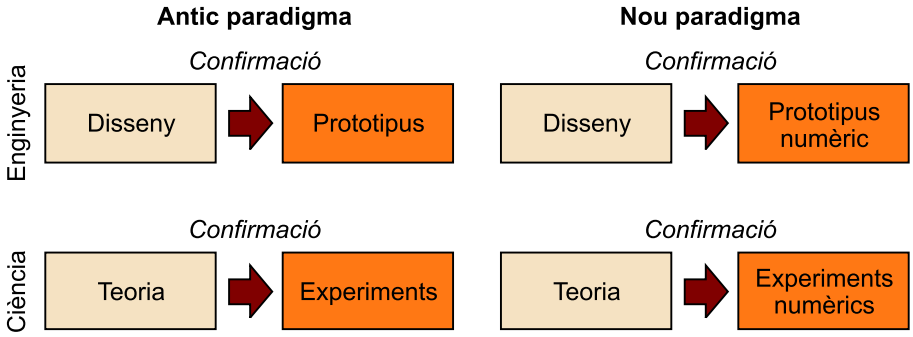
Algunes de les aplicacions que tradicionalment han estat un repte per a la comunitat i que necessiten computació d'altes prestacions, ja que no es poden executar amb un altre tipus de computador (per exemple, computadors personals), són:

- Aplicacions científiques, com per exemple el canvi climàtic, els models astrofísics, l'anàlisi genòmica o el plegament de proteïnes (disseny de fàrmacs).
- Aplicacions d'enginyeria, com per exemple la simulació de tests de xoc, el disseny de semiconductors o els models estructurals de terratrèmols.
- Aplicacions de negoci, com per exemple els models financers i econòmics, el procés de transaccions, els serveis web o els motors de cerca.

- Aplicacions de defensa i militars, com per exemple les simulacions de proves amb armes nuclears o la criptografia.

En aplicacions d'enginyeria la computació d'altes prestacions permet disminuir la utilització de prototipus en el procés de disseny i optimització de processos. Així, doncs, permet disminuir costos, augmentar la productivitat –ja que disminueix el temps de desenvolupament– i augmentar la seguretat. En aplicacions científiques, la computació d'altes prestacions permet la simulació de sistemes a gran escala, sistemes a molt petita escala i l'anàlisi de la validesa d'un model matemàtic. La figura següent il·lustra el canvi de paradigma en aplicacions d'enginyeria i ciència.

Figura 1. Canvi de paradigma en aplicacions d'enginyeria i ciència




## 1.2. Tipus bàsics d'aplicacions orientades a altes prestacions

Les aplicacions que requereixen altes prestacions es poden dividir en dos grans grups: **aplicacions d'alta productivitat o** HTC[1] i **aplicacions d'altes prestacions o** HPC[2].

(1)De l'anglès. *high throughput computing.*

(2)De l'anglès *high performance computing.*

També ho podem veure des del punt de vista de la manera de tractar el problema. La computació paral·lela permet el desenvolupament d'aplicacions que aprofitin la utilització de múltiples processadors de manera col·laborativa amb l'objectiu de resoldre un problema comú. De fet, l'objectiu fonamental que persegueixen aquestes tècniques és aconseguir reduir el temps d'execució d'una aplicació mitjançant la utilització de múltiples processadors. Addicionalment, també és possible voler resoldre problemes més grans mitjançant l'aprofitament de les diferents memòries dels processadors involucrats en l'execució. Podem parlar bàsicament de dos conceptes fonamentals:

**1) Particionament de tasques**: consisteix a dividir una tasca gran en un nombre de diferents subtasques que puguin ser complementades per diferents unitats de procés.

**2) Comunicació entre tasques**: tot i que cada procés faci una subtasca, generalment farà falta que aquests processos es comuniquin per poder cooperar en l'obtenció de la solució global del problema.

### 1.2.1. Aplicacions HTC

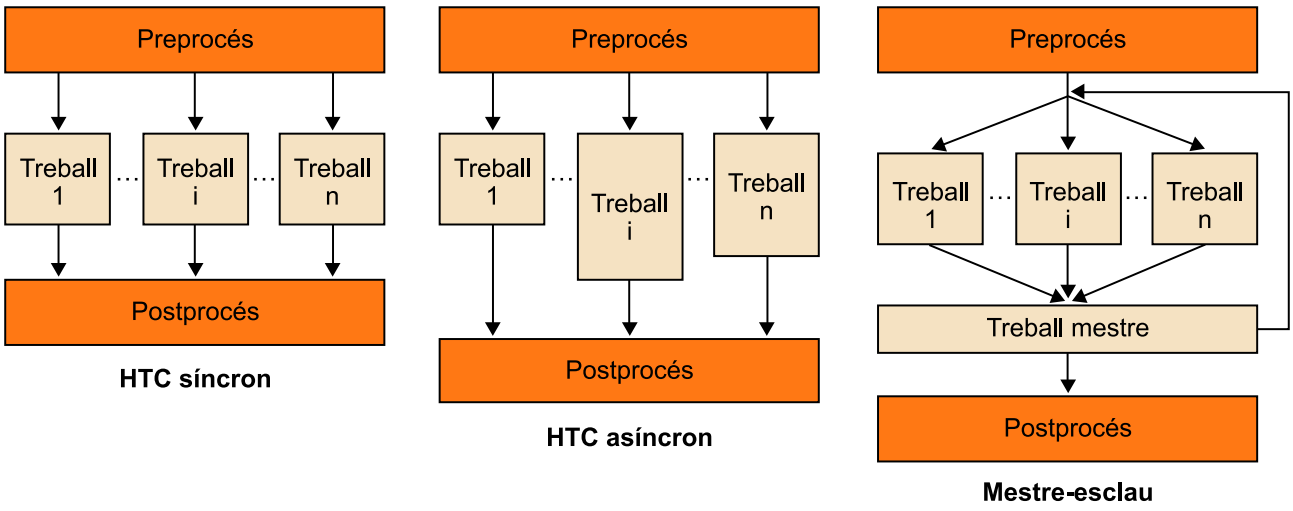
L'objectiu de les aplicacions HTC és augmentar el nombre d'execucions per unitat de temps. El rendiment d'aquestes aplicacions es mesura en nombre de treballs executats per unitat de temps (per exemple, treballs per segon).

**Aplicacions HTC**

Algunes de les àrees que acostumen a requerir aquest tipus d'aplicacions són la bioinformàtica i les finances.

A la figura següent es mostren tres tipus de models d'aplicacions HTC suposant que la tasca que cal executar es pot descompondre en diferents treballs (normalment anomenats *jobs*). En el primer (HTC síncron), els treballs estan sincronitzats i acaben a la vegada; en el segon (HTC asíncron), els treballs acaben a diferents instants, i en l'últim (mestre-esclau), hi ha un treball especialitzat (mestre) que s'encarrega de sincronitzar la resta de treballs (esclaus).

Figura 2. Models d'aplicacions HTC




### 1.2.2. Aplicacions HPC

L'objectiu de les aplicacions HPC és reduir el temps d'execució d'una única aplicació paral·lela. El rendiment d'aquestes aplicacions es mesura en nombre d'operacions en punt flotant per segon[3] (flop/s) i normalment s'acostuma a mesurar en milions, bilions, trilions, etc., tal com mostra la taula 1.

(3)En anglès, *floating point operations per second.*

Taula 1. Unitats de mesura de rendiment

| Nom | Flops |
|---|---|
| Megaflops | $10^6$ |
| Gigaflops | $10^9$ |

| Nom | Flops |
|---|---|
| Teraflops | $10^{12}$ |
| Petaflops | $10^{15}$ |
| Exaflops | $10^{18}$ |
| Zettaflops | $10^{21}$ |
| Yottaflops | $10^{24}$ |

Algunes àrees que acostumen a requerir aquest tipus d'aplicacions són:

**a)** Estudi de fenòmens a escala microscòpica (dinàmica de partícules). La resolució és limitada per la potència de càlcul del computador. Com més graus de llibertat (punts), més bé es reflecteix la realitat.

**b)** Estudi de fenòmens a escala macroscòpica (sistemes descrits per equacions diferencials fonamentals). La precisió és limitada per la potència de càlcul del computador. Com més punts, més s'apropa la solució discreta a la contínua.

Actualment la computació d'altes prestacions és sinònim de paral·lelisme i, per tant, de l'augment del nombre de processadors. En general es vol augmentar el nombre de processadors per a:

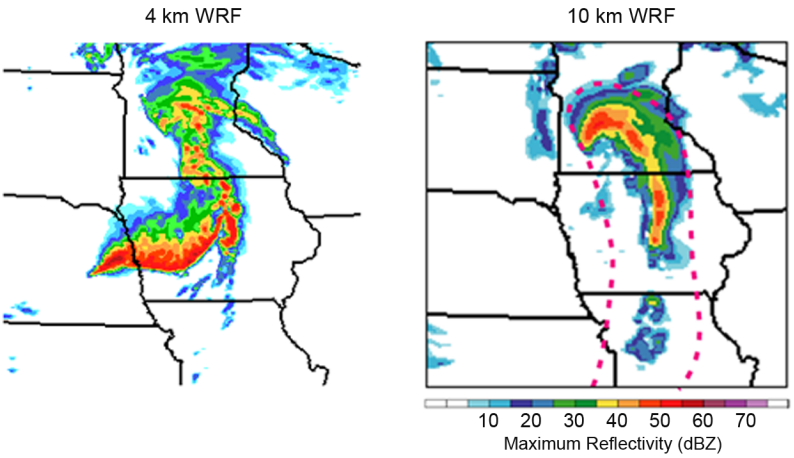
- Resoldre problemes en un temps d'execució inferior, utilitzant més processadors.
- Resoldre problemes amb més precisió, utilitzant més memòria.
- Resoldre problemes més reals, utilitzant models matemàtics més complexos.

**La predicció meteorològica**

Un exemple és la predicció meteorològica, la qual requereix gran capacitat de càlcul i memòria. La predicció meteorològica és una funció de la longitud, la latitud, l'altura i el temps. Per a cadascun d'aquests punts s'han de calcular diversos paràmetres, com ara la temperatura, pressió, humitat i velocitat del vent. Hi ha casos en els quals la predicció meteorològica es necessita en "temps real" (en qüestió d'hores, no de dies), com en el cas de la predicció de la formació, de la trajectòria i del possible impacte d'un huracà. Cal tenir en compte que fer les simulacions amb una precisió insuficient pot provocar perdre detalls importants i arribar a conclusions poc encertades que poden tenir conseqüències devastadores. La figura 3 il·lustra el resultat obtingut amb el model WRF[4] amb dos nivells de precisió diferents:

[(4)]Weather Research and Forecasting.

Figura 3. Prediccions meteorològiques amb WRF de diferents precisions

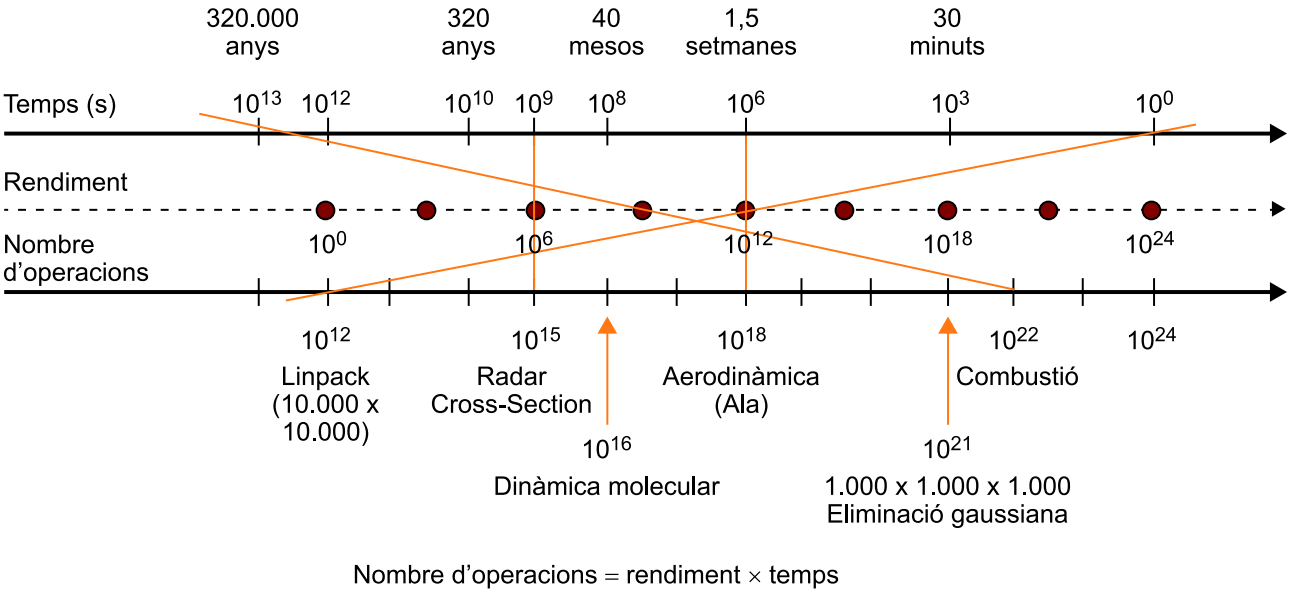



Font: The National Center for Atmospheric Research

En la simulació del gràfic de l'esquerra la graella és de 4 quilòmetres quadrats, mentre que en el de la dreta la graella és de 10 quilòmetres quadrats. Es pot apreciar clarament que hi ha certs fenòmens que no es poden observar amb claredat en la figura de la dreta. Cal tenir en compte que una predicció acurada d'aquests fenòmens, a part de l'impacte directe en la població, afecta l'economia (per exemple, s'estima que als Estats Units el 40% de petites i mitjanes empreses tanquen en els següents trenta-sis mesos si es veuen forçades a tancar tres dies o més després d'un huracà). Així, doncs, disposar de computació d'altes prestacions pot arribar a salvar vides o l'economia de regions.

La figura 4 mostra altres exemples clàssics que requereixen gran potència de càlcul i, per tant, computació d'altes prestacions i els quantifica pel que fa a la quantitat de càlcul requerit per a dur-los a terme.

Figura 4. Exemples de la necessitat d'altes prestacions



# 2. Paral·lelisme i arquitectures paral·leles

## 2.1. Necessitat de la computació paral·lela

Durant les darreres dècades, el rendiment dels microprocessadors s'ha incrementat de mitjana un 50% per any. Aquest increment en rendiment i prestacions significa que els usuaris i els desenvolupadors de programes podien esperar simplement la següent generació de microprocessadors per tal d'obtenir una millora substancial al rendiment dels seus programes. En canvi, des del 2002 la millora de rendiment dels processadors de manera individual s'ha frenat un 20% per any. Aquesta diferència és molt gran, ja que si tenim en compte un increment de rendiment del 50% per any, en qüestió de deu anys el factor és aproximadament de seixanta vegades, mentre que amb un increment del 20% el factor només és de sis vegades.

Un dels mètodes més rellevants per a millorar el rendiment dels computadors ha estat l'augment de la velocitat del rellotge del processador a causa de l'augment de la densitat dels processadors. A mesura que disminueix la mida dels transistors, se'n pot augmentar la velocitat i, en conseqüència, augmenta la velocitat global del circuit integrat.

Això està motivat per la llei de Moore, una llei empírica que diu que "la densitat de circuits en un xip es dobla cada 18 mesos" (Gordon Moore, Chairman, Intel, 1965). La taula 2 mostra l'evolució de la tecnologia en els processadors Intel. Tot i això la llei de Moore té límits evidents.

**Els límits de la llei de Moore**

Per exemple, si tenim en compte la velocitat de la llum com a velocitat de referència pel moviment de les dades en un xip, per a desenvolupar un computador que tingui un rendiment d'1 Tflop amb 1 TB de dades (1 THz), la distància (r) per a accedir a la dada en memòria hauria de ser inferior a $c/10^{12}$. Això vol dir que la mida del xip hauria de ser de 0,3 × 0,3 mm. Per a encabir 1 TB d'informació, una paraula de memòria hauria d'ocupar 3 àngstroms × 3 àngstroms, que és la mida d'un àtom!

Taula 2. Evolució de la tecnologia de microprocessadors (família Intel no complerta)

| Processador | Any | Nombre de transistors | Tecnologia |
|---|---|---|---|
| 4004 | 1971 | 2.250 | 10 µm |
| 8008 | 1972 | 3.500 | 10 µm |
| 8080 | 1974 | 6.000 | 6 µm |
| 8086 | 1978 | 29.000 | 3 µm |
| 286 | 1982 | 134.000 | 1,5 µm |

| Processador | Any | Nombre de transistors | Tecnologia |
|---|---|---|---|
| 386 | 1985 | 275.000 | 1 µm |
| 486 DX | 1989 | 1.200.000 | 0,8 µm |
| Pentium | 1993 | 3.100.000 | 0,8 µm |
| Pentium II | 1997 | 7.500.000 | 0,35 µm |
| Pentium III | 1999 | 28.000.000 | 180 nm |
| Pentium IV | 2002 | 55.000.000 | 130 nm |
| Core 2 Duo | 2006 | 291.000.000 | 65 nm |
| Core i7 (Quad) | 2008 | 731.000.000 | 45 nm |
| Core i7 (Sandy Bridge) | 2011 | 2.300.000.000 | 32 nm |

També cal tenir en compte que a mesura que augmenta la velocitat dels transistors, també ho fa el consum elèctric. La majoria d'aquest consum es transforma en calor, i quan un circuit integrat s'escalfa molt, no és fiable. Fins i tot podem trobar limitacions en el nivell de paral·lelisme a nivell d'instrucció[5], ja que fent el *pipeline* més i més gran es pot acabar obtenint un pitjor rendiment.

(5)En anglès, *instruction level parallelism*.

Així, doncs, actualment no és viable continuar incrementant la velocitat dels circuits integrats. En canvi, encara podem continuar incrementant la densitat dels transistors, però cal destacar que només per un cert temps.

La manera que ens queda per a treure partit de l'augment en la densitat dels transistors és mitjançant el paral·lelisme. En comptes de construir processadors monolítics cada cop més ràpids i complexos, la solució que la indústria va adoptar var ser el desenvolupament de processadors amb múltiples nuclis, centrant-se en el rendiment d'execució d'aplicacions paral·leles en canvi de programes seqüencials. D'aquesta manera, en els darrers anys s'ha produït un canvi molt significatiu en la indústria de la computació paral·lela.

Actualment gairebé tots els ordinadors de consum incorporen processadors multinucli[6] i el concepte de *nucli*[7] s'ha convertit en sinònim de CPU. Des de la incorporació dels processadors multinucli en dispositius quotidians, des de processadors duals per a dispositius mòbils fins a processadors amb més de 70 nuclis per a servidors i estacions de treball, la computació paral·lela ha deixat de ser exclusiva de supercomputadors i sistemes d'altes prestacions. Així, aquests dispositius proporcionen funcionalitats més sofisticades que els predecessors mitjançant computació paral·lela.

(6)En anglès, *multicore*.

(7)En anglès, *core*.

Aquest canvi també té conseqüències dràstiques amb vista als programadors, ja que les noves generacions de circuits integrats que incorporen més processadors no milloren automàticament el rendiment de les aplicacions seqüen-

cials com passava fins llavors. Tal com veurem més endavant, caldran noves tècniques i models de programació per a treure profit de les noves generacions de processadors.

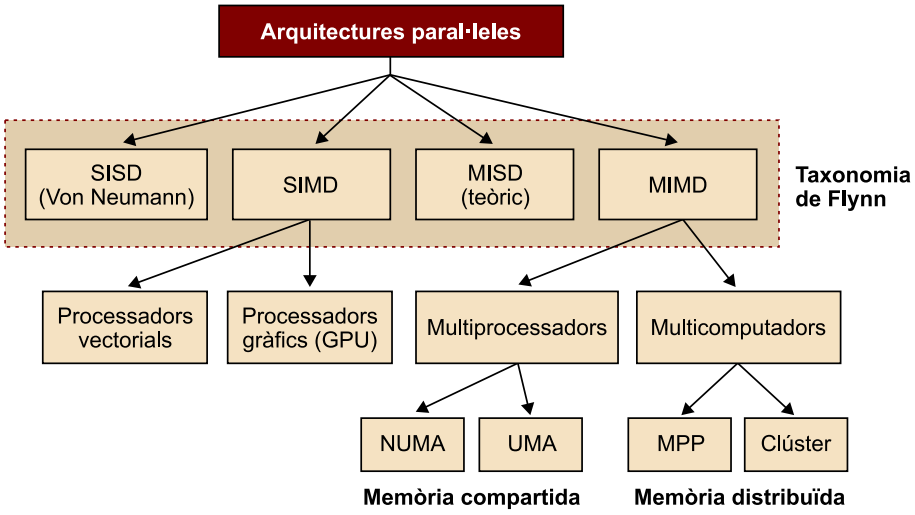
## 2.2. Arquitectures de computació paral·lela

En computació paral·lela normalment s'acostuma a utilitzar la taxonomia de Flynn per a classificar les arquitectures de computadors. Aquesta taxonomia classifica un sistema segons el nombre de fluxos de dades i d'instruccions que poden gestionar simultàniament. Tot i que estudiarem aquesta taxonomia en més detall en un altre mòdul didàctic, els dos tipus fonamentals s'exposen a continuació. Tot i això, la figura següent mostra la classificació de les arquitectures paral·leles, incloent-hi la taxonomia de Flynn.

**Vegeu també**

De la taxonomia de Flynn en tractem amb profunditat en el mòdul "Processadors i models de programació" d'aquesta assignatura.

Figura 5. Classificació de les arquitectures paral·leles




### 2.2.1. Una instrucció, múltiples dades (SIMD)

En els sistemes SIMD[8], una mateixa instrucció s'aplica sobre diverses dades, de manera que es podria veure com tenir una única unitat de control i múltiples ALU[9].

[8]De l'anglès *single instruction, multiple data stream.*

[9]De l'anglès *arithmetic logic unit.*

[10]De l'anglès *graphical processing unit.*

Una instrucció s'envia des de la unitat de control cap a totes les ALU i cadascuna de les ALU o bé aplica la instrucció al conjunt de dades o bé queda sense feina si no té dades que hi estiguin associades. Els sistemes SIMD són ideals per a certes tasques, com ara per a paral·lelitzar bucles senzills que operen sobre vectors de dades de gran mida. Aquest tipus de paral·lelisme s'anomena *paral·lelisme de dades*. El problema principal dels sistemes SIMD és que no sempre funciona bé per a tots els tipus de problemes. Aquets sistemes han evolucionat d'una manera una mica peculiar durant la seva història. A principis dels anys noranta la majoria dels supercomputadors paral·lels estaven basats en sistemes SIMD. En canvi, a finals de la mateixa dècada els sistemes SIMD eren

bàsicament processadors vectorials. Més recentment els processadors gràfics o GPU[10] i els processadors de gran consum fan servir aspectes de l'arquitectura SIMD.

### Processadors vectorials

Tot i que el que constitueix un processador vectorial ha canviat amb els anys, la característica clau que té és que **opera sobre vectors de dades**, mentre que els processadors convencionals operen sobre elements de dades individuals o escalars.

Aquest tipus de processadors són ràpids i fàcils d'utilitzar per a força tipus d'aplicacions. Els compiladors que vectoritzen el codi són molt bons identificant codi que es pot vectoritzar i també bucles que no es poden vectoritzar. Els sistemes vectorials tenen una amplada de banda a memòria molt elevada i totes les dades que es carreguen s'utilitzen en contra dels sistemes basats en memòria cau que no fan ús de tots els elements d'una línia de memòria cau. La part negativa dels sistemes vectorials és que no poden treballar amb estructures de dades irregulars i, per tant, tenen limitacions importants respecte a l'escalabilitat, ja que no poden tractar problemes gaire grans. Els processadors vectorials actuals tenen les característiques següents:

- Registres vectorials: són registres que són capaços de guardar vectors d'operands i operar simultàniament sobre els seus continguts. La mida del vector la fixa el sistema i pot oscil·lar des de 4 elements fins a, per exemple, 128 elements de 64 bits.

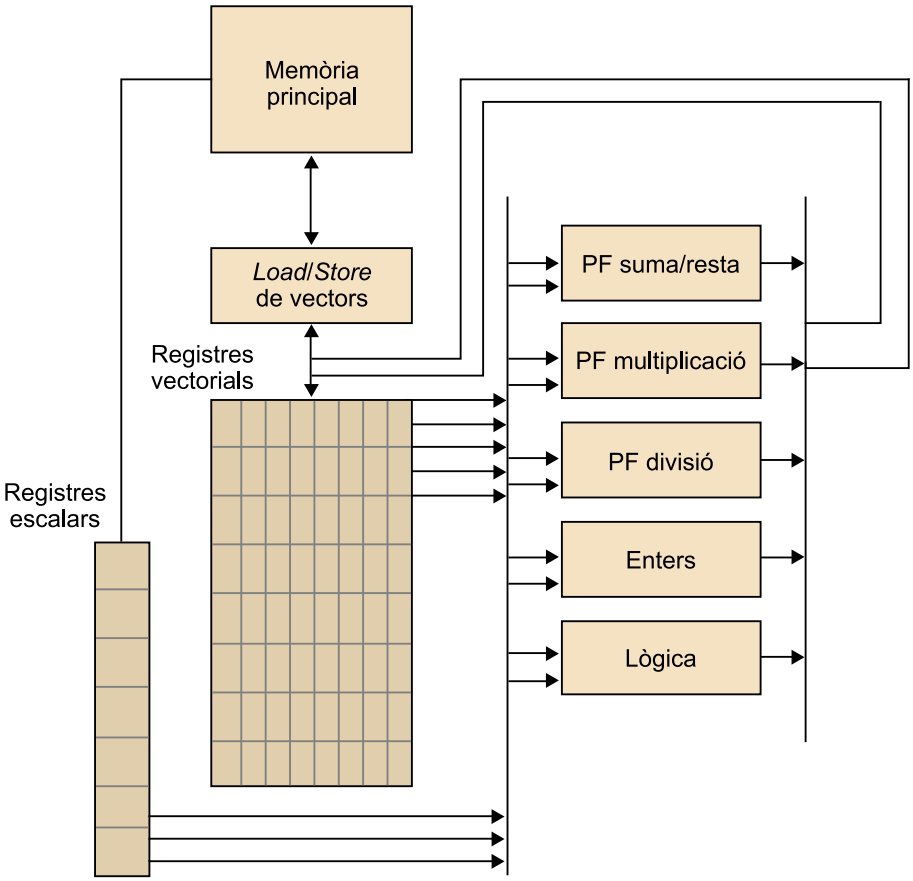
- Unitats funcionals vectoritzades: cal tenir en compte que la mateixa operació s'aplica a cada element del vector, o en operacions com ara la suma la mateixa operació s'aplica a cada parell d'elements dels dos vectors d'una sola vegada. Per tant, les operacions són SIMD.

- Instruccions sobre vectors: són instruccions que operen sobre vectors en comptes d'escalars. Això fa que, per a realitzar les operacions, en comptes de fer-ho individual per a cada element del vector (per exemple, *load*, *add* i *store* per a incrementar el valor de cada element del vector) es pugui fer per blocs.

- Accés a memòria per intervals: amb aquest tipus d'accés el programa accedeix a elements d'un vector localitzat a intervals. Per exemple, accedint al segon element, al sisè, al desè, etc., seria accés a memòria en un interval de 4.

La figura següent mostra un processador vectorial simple, on es pot apreciar que els blocs més bàsics poden actuar sobre un conjunt de registres vectorials.

Figura 6. Arquitectura vectorial simple




## Processadors gràfics (GPU)

El processadors gràfics tradicionalment **funcionen mitjançant un *pipeline* de processament** format per etapes molt especialitzades en les funcions que desenvolupen, i que s'executen en un ordre preestablert. Cadascuna de les etapes del *pipeline* rep la sortida de l'etapa anterior i proporciona la sortida a l'etapa següent.

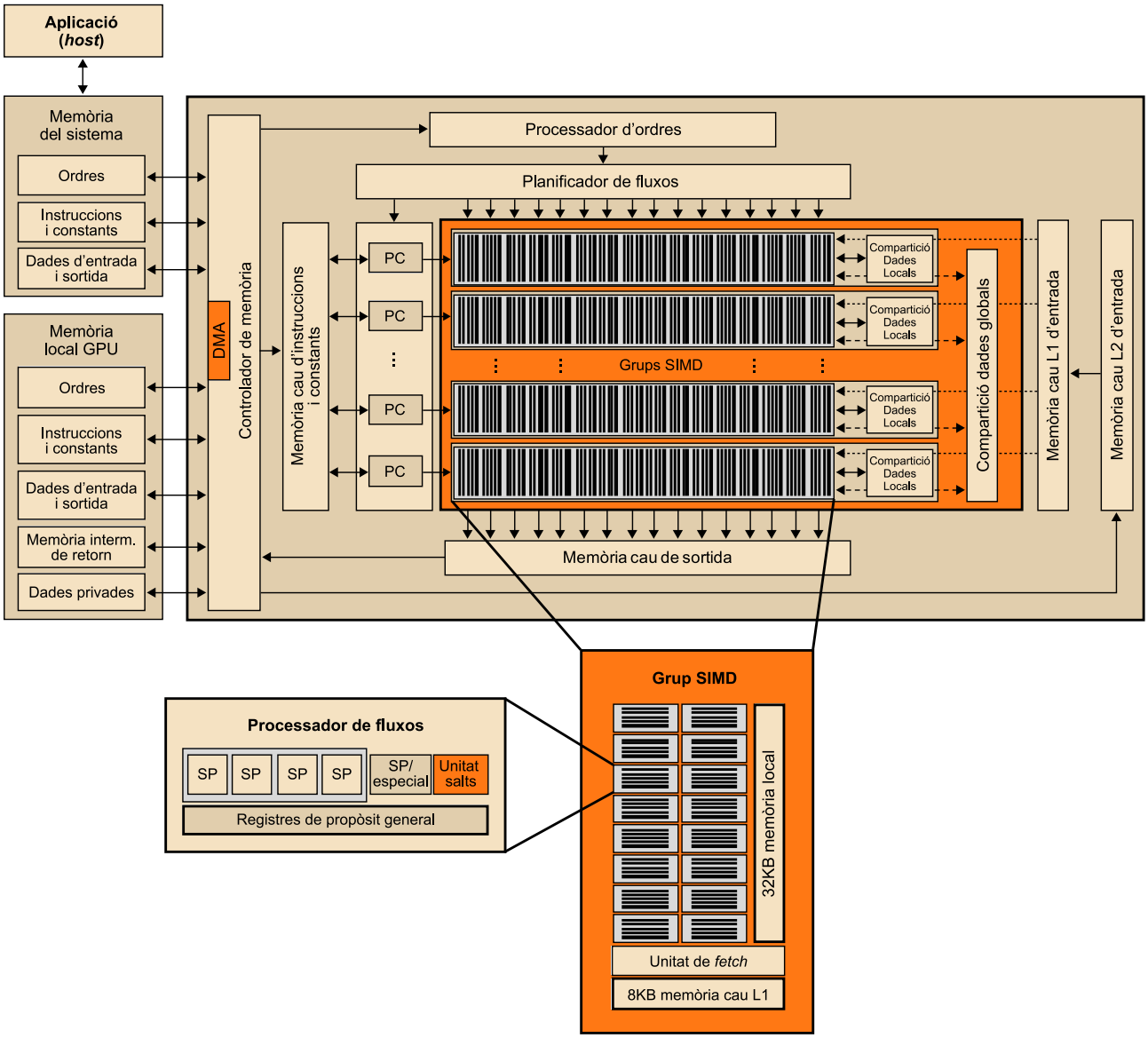
**Nombre elevat d'ALU**

En l'arquitectura Evergreen, d'AMD, es posen 1.600 ALU.

Gràcies a la implementació mitjançant una estructura de *pipeline*, el processador gràfic pot executar diverses operacions en paral·lel. Com que aquest *pipeline* és específic per a la gestió de gràfics, normalment s'anomena *pipeline gràfic* o *pipeline de renderització*. L'operació de renderització consisteix a projectar una representació en 3 dimensions a una imatge en 2 dimensions (que és l'objectiu d'un processador gràfic). Tot i això, hi ha etapes del *pipeline* gràfic que es poden programar i fins i tot en GPU actuals orientades a propòsit general hi ha gran flexibilitat de programació mitjançant els anomenats *kernels*. Els *kernels* són normalment codis força curts en els quals el paral·lelisme està implícit, ja que, quan s'instanciïn els *kernels*, aquests s'encarreguen de processar la part de les dades que li pertoquin. De fet, les GPU també poden utilitzar paral·lelisme SIMD per a millorar el rendiment i les generacions actuals de GPU ho fan posant un nombre elevat d'ALU (vegeu la figura 7) en cada nucli. Els processadors

gràfics estan esdevenint molt populars en la computació d'altes prestacions i s'han desenvolupat diversos llenguatges per a facilitar als usuaris la programació d'aquests processadors.

Figura 7. Diagrama de blocs de l'arquitectura Evergreen d'AMD




### 2.2.2. Múltiples instruccions, múltiples dades (MIMD)

Els sistemes MIMD[11] normalment consisteixen en un conjunt d'unitats de processament o **nuclis completament independents** que tenen la seva pròpia unitat de control i la seva pròpia ALU.

(11)De l'anglès *multiple instruction, multiple data stream.*

A diferència dels sistemes SIMD, els MIMD normalment són asíncrons, és a dir, que poden operar per la seva banda. En molts sistemes MIMD, a més, no hi ha un rellotge global i pot ser que no hi hagi relació entre els temps dels sistemes de dos processadors diferents. De fet, si no és que el programador imposi

certa sincronització, fins i tot encara que els processadors executin la mateixa seqüència d'instruccions, en un moment de temps concret poden executar parts diferents.

Tal com il·lustren les figures 8 i 9, hi ha dos tipus principals de sistemes MIMD: sistemes de memòria compartida i sistemes de memòria distribuïda. En un sistema de memòria compartida un conjunt de processadors autònoms es connecten al sistema de memòria mitjançant una xarxa d'interconnexió i cada processador pot accedir a qualsevol part de la memòria. Els processadors normalment es comuniquen implícitament mitjançant estructures de dades compartides a la memòria. En un sistema de memòria distribuïda, cada processador està lligat a la seva pròpia memòria privada i els conjunts processador-memòria es comuniquen mitjançant una xarxa d'interconnexió. Per tant, en un sistema de memòria distribuïda els processadors normalment es comuniquen explícitament enviant missatges o utilitzant funcions especials que proporcionen accés a la memòria d'un altre processador.

**RDMA (*remot direct memory access*)**

Un exemple d'aquest últim tipus és l'accés a la memòria d'un altre processador mitjançant la xarxa (RDMA), una característica d'interfícies de xarxa que permeten a un computador accedir directament a la memòria d'un altre computador sense passar pel sistema operatiu. Una implementació que s'utilitza en computació d'altes prestacions és sobre InfiniBand.

### Activitat

Us proposem que cerqueu informació sobre Infiniband i de la seva implementació d'RDMA.

Figura 8. Sistema de memòria compartida

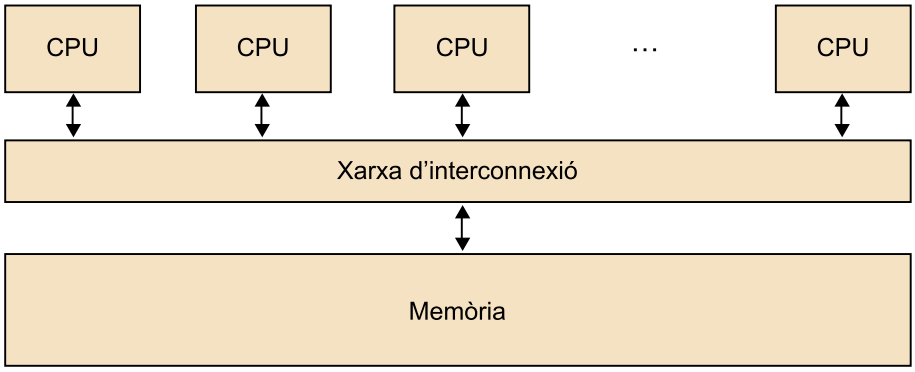
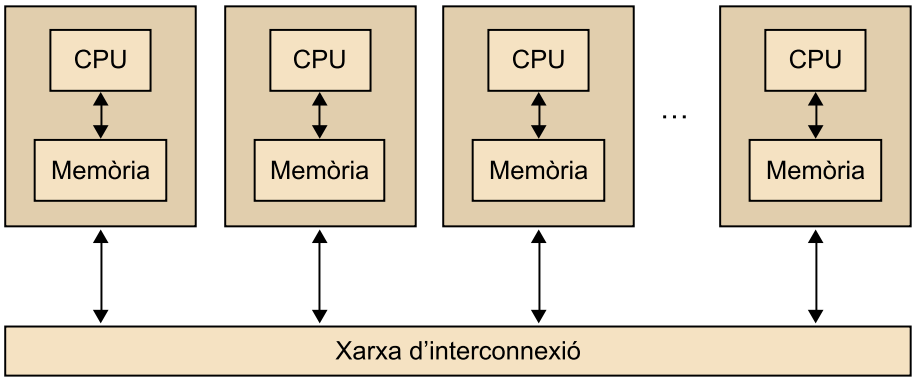



Figura 9. Sistema de memòria distribuïda



### Sistemes de memòria compartida

Els sistemes de memòria compartida més utilitzats utilitzen un o més processadors multinucli, els quals utilitzen una o més CPU en un mateix xip. Típicament, els nuclis tenen una memòria cau privada de nivell 1, mentre que hi ha altres memòries cau que poden estar compartides o no entre els diversos nuclis.

En sistemes de memòria compartida amb múltiples processadors multinucli, la xarxa d'interconnexió o bé pot connectar tots els processadors directament amb la memòria principal o bé cada processador pot tenir accés directe a un bloc de la memòria principal, i els processadors poden accedir als blocs de memòria dels altres processadors mitjançant maquinari especialitzat incorporat en els processadors, tal com mostren les figures 10 i 11.

Figura 10. Sistema multinucli UMA

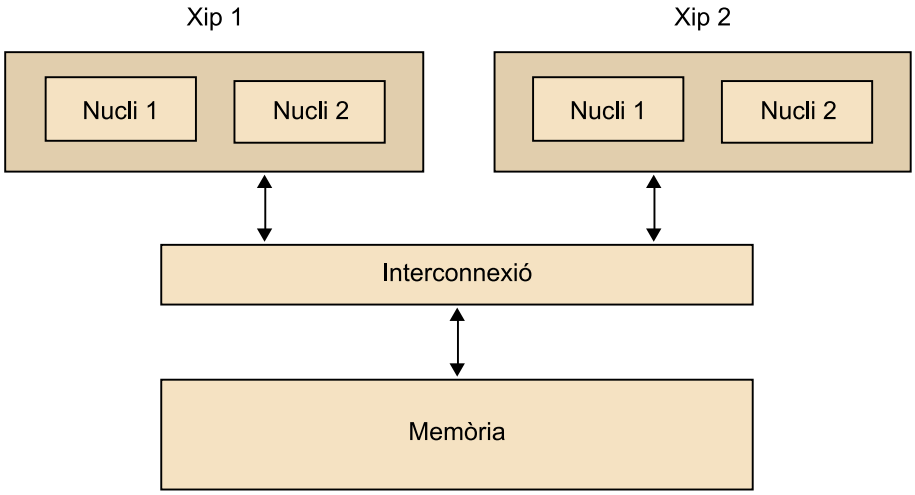
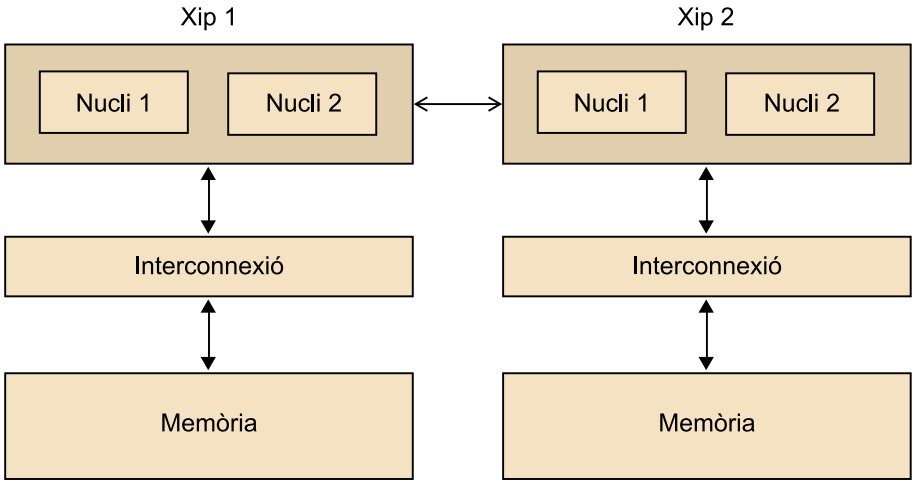



Figura 11. Sistema multinucli NUMA




En el primer tipus de sistema el temps d'accés a totes les posicions de memòria és el mateix per a tots els nuclis, mentre que en el segon tipus el temps d'accés a la memòria, a la qual el nucli està directament connectat, serà més ràpid que el d'accedir a la memòria d'un altre xip. Per tant, el primer tipus

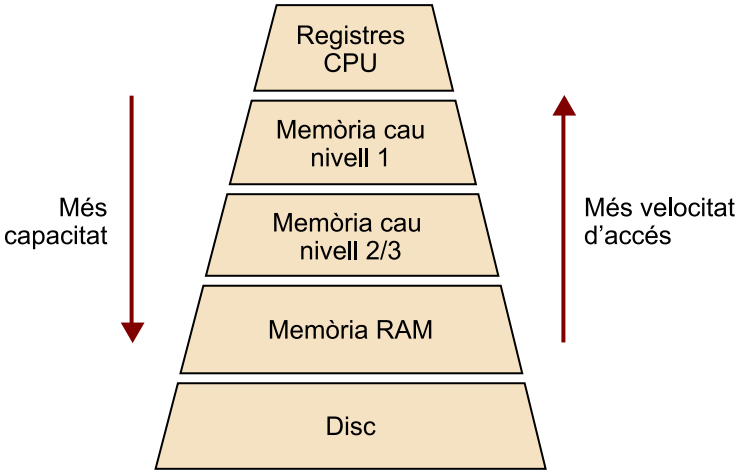
(12)De l'anglès *uniform memory access*.

de sistema s'anomena *UMA*[12] i el segon s'anomena *NUMA*[13]. Els sistemes UMA normalment són molt més fàcils de programar, ja que el programador no s'ha de preocupar del temps d'accés a les dades i, per tant, de la localitat de les dades. Aquest avantatge, però, queda en certa manera compensat per l'accés més ràpid a la memòria a la qual està connectat el nucli en els sistemes NUMA i també pel fet que aquests sistemes acostumen a disposar de més quantitat de memòria que els UMA.

(13)De l'anglès *non-uniform memory access*.

Per tant, podem dir que la jerarquia de memòria i la seva gestió eficient és clau per a la computació d'altes prestacions. La figura següent mostra els nivells clàssics de la jerarquia de memòria d'un computador emfatitzant el fet que com més ràpida és una memòria, més petita i costosa acostuma a ser. L'existència de diferents nivells de memòria implica que el temps dedicat a fer una operació d'accés a memòria d'un nivell augmenta amb la distància al nivell més ràpid, que és l'accés als registres del processador. Una mala gestió de la memòria provoca moltes fallades de pàgina, que acaben fent un ús excessiu del sistema de memòria virtual, la qual cosa comporta realitzar costosos accessos al disc. Per tant, el disseny d'aplicacions eficients requereix un profund coneixement i saber explotar l'estructura de la memòria del computador.

Figura 12. Jerarquia de memòria d'un computador




L'optimització del procés de càlcul seqüencial implica, entre altres coses, una selecció adequada de les estructures de dades que cal utilitzar per a representar la informació relativa al problema, atenent criteris d'eficiència computacional. Per exemple, la utilització de tècniques d'emmagatzemament de matrius disperses pertany a aquesta categoria, i permet emmagatzemar únicament els elements no nuls d'una matriu. A més, aquest nivell inclou la selecció dels mètodes eficients per a la resolució del problema. Una fase important de les aplicacions, especialment per a aquelles orientades a processos de simulació, correspon a la gravació de dades a disc. Com que l'accés a un disc dur presenta temps d'accés molt superiors als corresponents a memòria RAM, resulta indispensable fer una gestió eficient del procés d'E/S[14] perquè tingui el mínim impacte sobre el procés de simulació.

(14)Es refereix a un procés d'entrada/sortida.

Finalment, i per sobre de la resta de les capes, apareix la utilització de computació paral·lela. En aquest sentit, un bona estratègia de partició de tasques que garanteixi una distribució equitativa de la càrrega entre els processadors involucrats, i també la minimització del nombre de comunicacions entre aquests processadors, permet reduir els temps d'execució. A més, amb la participació de les principals estructures de dades de l'aplicació entre els diferents processadors es pot aconseguir la resolució de problemes més grans.

### Sistemes de memòria distribuïda

Els sistemes de memòria distribuïda més estesos i populars són els anomenats *clústers*, que estan compostos per un conjunt de sistemes de consum[15] com per exemple servidors, connectats a una xarxa d'interconnexió també de consum, com ara Ethernet. De fet, els nodes d'aquests clústers, que són unitats de computació individuals interconnectades mitjançant una xarxa, són normalment sistemes de memòria compartida amb un o més processadors multinucli. Per a diferenciar aquests sistemes dels que són purament de memòria distribuïda, a vegades s'anomenen *sistemes híbrids*.

(15)En anglès, *commodity*.

Actualment s'entén que un clúster està compost de nodes de memòria compartida. A més, s'entén que els grans sistemes de computació paral·lels són clústers per definició. La diferència principal entre grans centres de processament o de dades[16] que utilitza la indústria (per exemple, Google i Facebook) i els supercomputadors és la xarxa d'interconnexió. La clau en aquests sistemes està a poder oferir una latència molt reduïda en la xarxa d'interconnexió perquè el pas de missatges sigui molt ràpid, ja que les aplicacions que requereixen supercomputació són bàsicament fortament acoblades[17], és a dir, que els diferents processos que s'executen en diferents nodes distribuïts tenen dependències de dades amb els altres processos i s'han de comunicar molt sovint mitjançant pas de missatges.

(16)En anglès, *datacenters*.

(17)En anglès, *tightly coupled*.

## 2.2.3. Coherència de memòria cau

Cal recordar que les memòries cau les gestionen sistemes maquinari, i per tant els programadors no tenen control directe sobre aquestes memòries. Això té conseqüències importants per als sistemes de memòria compartida. Per a entendre això suposem que tenim un sistema de memòria compartida amb dos nuclis, cadascun dels quals té la seva pròpia memòria cau de dades. No hi hauria cap problema mentre els dos nuclis només llegeixin dades compartides.

Per exemple, suposem dos nuclis que comparteixen memòria i que $x$ és una variable compartida que s'ha inicialitzat a 2, $y0$ és privada i propietat del nucli 0, i $y1$ i $z1$ són privades i propietat del nucli 1. Ara suposem el següent escenari que mostra la taula 3.

Taula 3. Escenari il·lustratiu de coherència de memòria cau

| Temps | Nucli 0 | Nucli 1 |
|---|---|---|
| 0 | $y0 = x;$ | $y1 = 3 * x;$ |

| Temps | Nucli 0 | Nucli 1 |
|---|---|---|
| 1 | $x = 7;$ | Codi que no involucra $x$ |
| 2 | Codi que no involucra $x$ | $z1 = 4 * x;$ |

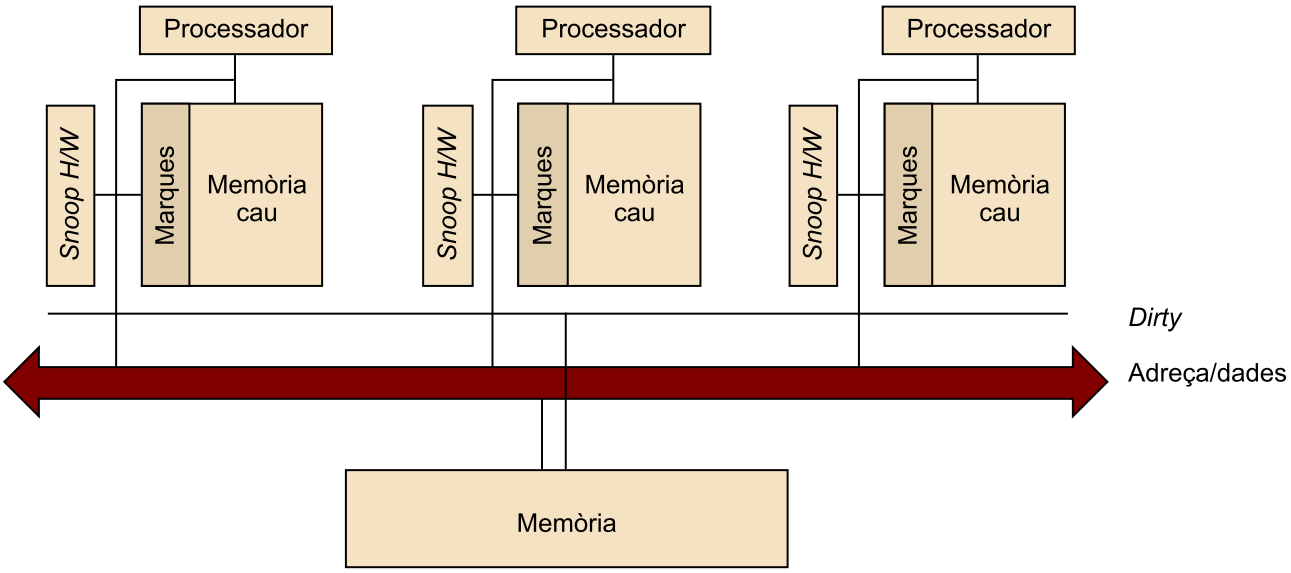
Llavors, la posició de memòria de $y0$ finalment prendrà el valor 2 i la posició de memòria de $y1$ prendrà el valor 6. En canvi, no és clar quin valor prendrà $z1$. Inicialment podria semblar que, com que el nucli 0 actualitza $x$ a 7 abans de l'assignació a $z1$, $z1$ tindrà el valor $4 * 7 = 28$. En canvi, a l'instant de temps 0, $x$ és a la memòria cau del nucli 1. Per tant, si no és que per alguna raó $x$ sigui desallotjada de la memòria cau del nucli 0 i després reallotjada a la memòria cau del nucli 1, en realitat es podrà utilitzar el valor original $x = 2$ i $z1$ prendrà el valor $4 * 2 = 8$.

Clarament això és un problema important i el programador no ha pas de tenir control directe de quan s'actualitzen les memòries cau i, per tant, el seu programa no podrà suposar en el cas de l'exemple anterior quin valor tindrà $z1$. Aquí hi ha uns quants problemes, però el que es vol destacar és el fet que les memòries cau de sistemes d'un únic processador no tenen mecanismes per a assegurar que, quan les memòries cau de múltiples processadors guarden la mateixa variable, una modificació feta per un dels processadors a la variable en memòria cau sigui vista per la resta de processadors. Això vol dir que els valors en memòria cau en altres processadors siguin també actualitzats i, per tant, parlem del problema de coherència de memòria cau.

### Tècnica de coherència de memòria cau *snooping*

Hi ha dues alternatives bàsiques per a assegurar coherència en la memòria cau: la tècnica de *snooping* i la basada en directori. La idea de la tècnica *snooping* ve dels sistemes basats en bus: quan els nuclis comparteixen un bus, qualsevol senyal transmès pel bus pot ser consultat per tots els nuclis connectats al bus. Per tant, quan el nucli 0 actualitza la còpia de $x$ que hi ha a la seva memòria cau, si també es distribueix aquesta informació pel bus i el nucli 1 està escoltant el bus, aquest podrà veure que $x$ s'ha actualitzat i podrà marcar la seva pròpia còpia de $x$ com a invàlida. Així és bàsicament com funciona la coherència per *snooping*, tot i que en la implementació real quan es distribueix la informació d'algun canvi es fa informant els altres nuclis que la línia de memòria cau que conté $x$ s'ha actualitzat i no que $x$ s'ha actualitzat. La figura següent mostra un esquema del sistema de coherència per *snooping*:

Figura 13. Sistema senzill de coherència de memòria per *snooping*

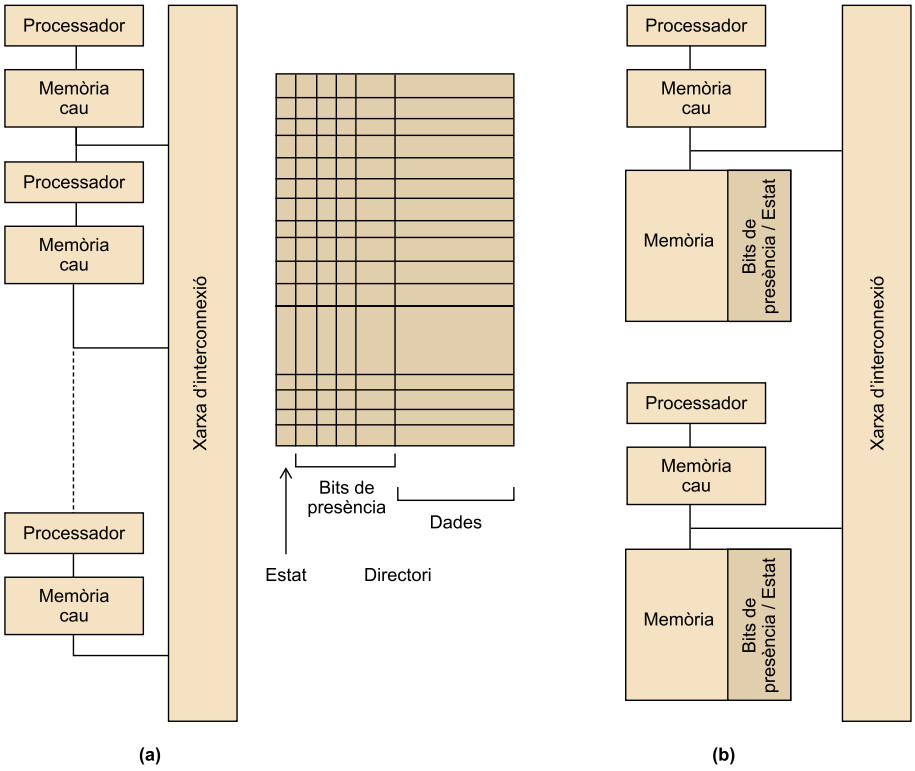



### Tècnica de coherència de memòria basada en directori

En xarxes d'interconnexió grans fer servir la tècnica de *snooping* no és viable, ja que cal distribuir dades per la xarxa cada cop que hi ha un canvi i això clarament no és escalable, ja que el rendiment es degradaria molt.

Els protocols de memòria cau basats en directori intenten resoldre aquest problema mitjançant una estructura de dades que s'anomena *directori*. El directori emmagatzema l'estat de cada línia de memòria cau. Típicament, aquesta estructura de dades és distribuïda, de manera que cada parell de processador-memòria és responsable d'emmagatzemar la part de l'estructura corresponent a l'estat de les línies de memòria cau en la seva memòria local. Quan s'actualitza una variable, es consulta el directori, i els controladors de memòria cau dels nuclis que tenen la línia de memòria cau on hi ha aquesta variable en la seva memòria cau són invalidats.

Figura 14. Sistemes de coherència de memòria basats en directori: (a) amb un directori centralitzat i (b) amb un directori distribuït




Amb aquesta tècnica clarament es necessita més espai de memòria per al directori, però quan s'actualitza una variable en memòria cau, només han de ser contactats els nuclis que emmagatzemen aquesta variable i, per tant, es redueixen les comunicacions per les xarxes i en conseqüència millora el rendiment respecte a la tècnica de *snooping*.

### Falsa compartició

És important recordar que les memòries cau dels processadors estan implementades en maquinari, i per tant operen sobre línies de memòria cau i no pas sobre variables individuals. Això pot tenir conseqüències catastròfiques en relació amb el rendiment. Com a exemple, suposem que volem cridar repetidament una funció `f(i,j)` i afegir el valor retornat a dins d'un vector tal com mostra el codi següent:

```
int i, j, m, n;
double y[m];
/* Assignar y = 0 */
...
for(i=0; i<m; i++)
   for(j=0; j<n; j++)
      y[i] += f(i,j);
```

Ho podem paral·lelitzar dividint les iteracions del bucle exterior entre els diferents nuclis. Si tenim `num_cores` nuclis, podem assignar les primeres `m/num_cores` iteracions al primer nucli, les següents `m/num_cores` iteracions al segon nucli, i així successivament, tal com es mostra en el codi següent:

```
/* Variables privades*/
int i, j, num_iter;
/* Variables privades inicialitzades per un nucli */
int m, n, num_cores;
double y[m];
num_iter = m/num_cores;
/* El nucli 0 fa el següent */
for(i=0; i<num_iter; i++)
   for(j=0; j<n; j++)
      y[i] += f(i,j);
/* El nucli 1 fa el següent */
for(i=num_iter+1; i<2*num_iter; i++)
   for(j=0; j<n; j++)
      y[i] += f(i,j)
...
double y[m];
/* Assignar y = 0 */
...
```

Suposem que tenim un sistema de memòria compartida que té dos nuclis, `m=8`, el tipus *double* és de 8 bytes, les línies de memòria cau són de 64 bytes i `y[0]` està emmagatzemat al principi d'una línia de memòria cau. Una línia de memòria cau pot emmagatzemar vuit *doubles* i `y` ocupa una línia de memòria cau completa. Si els nuclis 0 i 1 executen simultàniament els seus codis, com que tot el vector y està emmagatzemat en una única línia de memòria cau, cada cop que un dels nuclis executa la línia `y[i] += f(i,j)`, la línia serà invalidada i el pròxim cop que l'altre nucli intenti executar aquesta línia haurà d'agafar la línia una altra vegada de memòria principal. Per tant, si es fan moltes `(n)` iteracions podem suposar que un gran percentatge de cops que s'executa `y[i] += f(i,j)` s'accedirà a memòria principal tot i que tant el nucli 0 com el nucli 1 mai no accedeixen als elements de `y` que corresponen a l'altre nucli. Això és l'anomenada *falsa compartició*[18], en què el sistema es comporta com si els elements de `y` fossin compartits pels nuclis.

[18] En anglès, *false sharing*.

Cal remarcar que la falsa compartició no causa resultats incorrectes, sinó que pot degradar el rendiment d'un programa perquè pot ser que accedeixi a memòria principal moltes més vegades del que és necessari. Podem mitigar aquest efecte utilitzant emmagatzemament temporal que sigui local al flux o procés i després copiant l'emmagatzemament temporal a l'emmagatzemament compartit. També es poden aplicar altres tècniques, com per exemple *padding*.

**Activitat**

Us proposem que cerqueu informació sobre la tècnica de *padding*.

## 2.3. Sistemes distribuïts

Tot i que el model de computació d'altes prestacions és tradicionalment el que hem vist en les arquitectures paral·leles, en la darrera dècada s'han desenvolupat sistemes distribuïts que han aparegut com una font viable per a certes aplicacions de computació d'altes prestacions, com ara els sistemes *grid*.

Els **sistemes *grid*** proporcionen la infraestructura necessària per a transformar xarxes de computadors distribuïts geogràficament per Internet en un sistema unificat de memòria distribuïda.

En general, aquests sistemes són molt heterogenis, ja que individualment els nodes que els componen poden ser de diferents tipus de maquinari i fins i tot de diferents tipus de programari. Tot i això, mitjançant una capa intermèdia[19] de programari, la infraestructura *grid* proporciona les interfícies i abstraccions necessàries per a treballar amb computadors distribuïts i de diferents institucions de manera transparent com si fos un sistema convencional.

(19)En anglès, *middleware*.

També s'han desenvolupat altres tipus de sistemes altament distribuïts que inicialment no estaven dissenyats per a la computació d'altes prestacions, com el d'igual a igual (*peer-to-peer*) o la informàtica en núvol (*cloud computing*).

**Vegeu també**

Dels sistemes d'igual a igual i la informàtica en núvol, juntament amb la computació en *grid*, en tractarem en el mòdul 4 d'aquesta assignatura i també tractarem de la relació que tenen amb la computació d'altes prestacions.

# 3. Programació d'aplicacions paral·leles

La majoria de programes que s'han desenvolupat per a sistemes convencionals amb un únic nucli no poden explotar els múltiples nuclis dels processadors actuals. Podem executar diferents instàncies d'un programa –per exemple deixant que el sistema operatiu planifiqui aquestes instàncies en els diferents processadors–, però normalment aquesta solució no és suficient, especialment en la computació d'altes prestacions, en què es requereix paral·lelisme massiu. Les solucions principals són o bé reescriure els programes o bé utilitzar eines que paral·lelitzin automàticament els programes.

La manera d'escriure programes paral·lels depèn de la manera amb què es vulgui dividir el treball. N'hi ha de dos tipus principals: paral·lelisme de tasques i paral·lelisme de dades.

En el **paral·lelisme de tasques** el treball es reparteix en diverses tasques que es distribueixen pels diferents nuclis. En el **paral·lelisme de dades** les dades es divideixen per a resoldre el problema entre els diferents nuclis, els quals fan operacions similars sobre les dades que els pertoca.

**Paral·lelisme de dades**

Un exemple de paral·lelisme de dades és la computació gràfica o basada en GPU.

Actualment els programes paral·lels més potents s'escriuen utilitzant construccions de paral·lelisme explícit utilitzant extensions de llenguatges, com ara C/C++. Aquests programes inclouen instruccions que són específiques per a gestionar el paral·lelisme com ara "el nucli 0 executa la tasca 0, el nucli 1 executa la tasca 1, ..., sincronitzar tot els nuclis" i, per tant, els programes esdevenen moltes vegades molt complexos. Hi ha altres opcions per a escriure programes paral·lels, com ara utilitzar llenguatges de més alt nivell, però aquests llenguatges acostumen a sacrificar rendiment per tal de facilitar el desenvolupament i millorar la productivitat.

Ens centrarem en els programes que són explícitament paral·lels. Els principals models de programació són pas de missatges o MPI[20], fluxos (per exemple, fluxos POSIX o Pthreads) i OpenMP. MPI i Pthreads són biblioteques amb definicions de tipus, funcions i macros que es poden utilitzar per exemple en programes escrits en C. OpenMP consisteix en una biblioteca i també certes modificacions del compilador, com per exemple de C. Addicionalment també tractarem d'altres models de programació, com ara els de computació gràfica o d'altres que proporcionen abstraccions més modernes.

[20] De l'anglès *message passing interface*.

Aquests models de programació donen resposta als dos tipus principals de sistemes paral·lels: sistemes de memòria compartida i sistemes de memòria distribuïda. En un sistema de memòria compartida els nuclis poden compartir

l'accés a la memòria, i per tant cal coordinar l'accés dels diferents nuclis a memòria havent d'examinar i actualitzar l'espai compartit de la memòria. En un sistema de memòria distribuïda cada nucli té el seu propi espai de memòria privat i els nuclis s'han de comunicar explícitament fent pas de missatges per la xarxa d'interconnexió. Pthreads i OpenMP es van dissenyar per a programar sistemes de memòria compartida i proporcionen mecanismes per a accedir a posicions de memòria compartida. En canvi, MPI es va dissenyar per a programar sistemes de memòria distribuïda i proporciona mecanismes per a pas de missatges.

## 3.1. Models de programació de memòria compartida

### 3.1.1. Programació amb fluxos

Tot i que es poden considerar diferents tipus de fluxos per a programar programes paral·lels, com ara fluxos propis d'Unix o de Java, ens centrarem en els Pthreads, que és una biblioteca que compleix els estàndards POSIX i que ens permet treballar amb diferents fluxos al mateix temps (concurrentment). Pthreads són més efectius en un sistema multiprocessador o en un sistema multinucli, en què el flux d'execució es pot planificar en un altre processador o nucli, i així guanyar velocitat gràcies al paral·lelisme.

Els Pthreads tenen menys sobrecost que la creació d'un nou procés (per exemple, les crides fork o exec) perquè el sistema no inicialitza un nou espai de memòria virtual i entorn per al procés. Tot i que són més efectius en sistemes amb múltiples processadors o nuclis, també es pot obtenir benefici en un sistema uniprocessador, ja que permet explotar la latència de l'entrada/sortida i altres funcions del sistema que poden aturar l'execució del procés en execució. Els Pthreads s'utilitzen en sistemes de memòria compartida i tots els fluxos d'un procés comparteixen el mateix espai d'adreces. Per a crear un flux cal definir una funció i els arguments d'aquesta funció que seran utilitzats pel flux. L'objectiu principal d'utilitzar Pthreads és aconseguir més velocitat (més bon rendiment).

El codi següent mostra un exemple en què es pot apreciar com s'han de crear dos fluxos (cadascun amb diferents paràmetres) i esperar-ne la finalització.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void *print_message_function( void *ptr );
main() {
   pthread_t thread1, thread2;
   char *message1 = "Thread 1";
   char *message2 = "Thread 2";
   int  iret1, iret2;
```

```
    /* Crea fluxos independents, els quals executaran la funció */
    iret1 = pthread_create( &thread1, NULL, print_message_function, (void*) message1);
    iret2 = pthread_create( &thread2, NULL, print_message_function, (void*) message2);
    /* Espera que tots els fluxos acabin abans de continuar amb el programa principal */
    pthread_join( thread1, NULL);
    pthread_join( thread2, NULL);
    printf("Thread 1 returns: %d\n",iret1);
    printf("Thread 2 returns: %d\n",iret2);
    exit(0);
}
void *print_message_function( void *ptr ) {
    char *message;
    message = (char *) ptr;
    printf("%s \n", message);
}
```

Un dels elements clau en la gestió de fluxos Pthread és la sincronització i exclusió mútua que tenen, que habitualment es porta a terme mitjançant semàfors.

### 3.1.2. OpenMP

Tot i que tant Pthreads com OpenMP són interfícies de programació per a memòria compartida tenen moltes diferències importants. Amb Pthreads es requereix que el programador especifiqui explícitament el comportament de cada flux. D'altra banda, OpenMP permet que el programador només hagi d'especificar que un bloc de codi s'executi en paral·lel, i determinar les tasques i quin flux les executarà es deixa en mans del compilador i del sistema d'execució[21]. Això suggereix una altra diferència important amb Pthreads: Pthreads és una biblioteca de funcions que es pot afegir en muntar un programa en C, mentre que OpenMP necessita suport de compilador i, per tant, no necessàriament tots els compiladors de C han de compilar programes OpenMP com a programes paral·lels. Per tant, podem dir que OpenMP permet millorar la productivitat mitjançant una interfície de més alt nivell que la de Pthreads, però d'altra banda no ens permet controlar certs detalls relacionats amb el comportament dels fluxos.

[21] En anglès, *runtime*.

OpenMP va ser concebut per un grup de programadors i científics informàtics que pensaven que escriure programes d'altes prestacions a gran escala utilitzant interfícies com la de Pthreads era massa difícil. De fet, OpenMP es va dissenyar explícitament per a permetre als programadors paral·lelitzar programes seqüencials existents progressivament i no haver-los de tornar a escriure des de zero.

OpenMP proporciona una interfície de memòria compartida basada en les anomenades *directives*. Per exemple, en C/C++, això vol dir que hi ha certes instruccions especials per al preprocessador anomenades directives o *pragmes*

en anglès. Normalment s'afegeixen pragmes a un programa per a especificar comportaments que no són part de la mateixa especificació del llenguatge de programació. Això vol dir que els compiladors que no entenen aquests pragmes els poden ignorar. Això permet que un programa es pugui executar en un sistema amb el qual no funciona.

Els pragmes, en C/C++, comencen per `#pragma` igual que altres directives de processament. A continuació es mostra un programa OpenMP molt senzill que implementa el típic "hello, world". En aquest programa, podem apreciar la utilització d'una directiva (o pragma) d'OpenMP i també algunes funcions típiques associades a OpenMP.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
void Hello(void);
int main(int argc, char* argv[]) {
   /* Obtenir el nombre de fluxos de l'intèrpret d'ordres */
   int thread_count = strtol(argv[1], NULL, 10);
#  pragma omp parallel num_threads(thread_count)
   Hello();
   return 0;
}
void Hello(void( {
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();
   printf("Hello from thread %d of %d\n", my_rank, thread_count);
}
```

### 3.1.3. CUDA i OpenCL

CUDA[22] és una especificació inicialment de propietat desenvolupada per Nvidia com a plataforma per als seus productes de computació gràfica (GPU). CUDA inclou les especificacions de l'arquitectura i un model de programació que hi està associat.

(22)De l'anglès *compute unified device architecture*.

CUDA es va desenvolupar per a augmentar la productivitat en el desenvolupament d'aplicacions de propòsit general per a computació gràfica. Des del punt de vista del programador, el sistema està compost per un processador principal[23], que és una CPU tradicional, com per exemple un processador d'arquitectura Intel, i d'un o més dispositius[24], que són GPU.

(23)En anglès, *host*.

(24)En anglès, *devices*.

CUDA pertany al model SIMD; per tant, està pensat per a explotar el paral·lelisme de dades. Això vol dir que un conjunt d'operacions aritmètiques es poden executar sobre un conjunt de dades de manera simultània. Afortunadament, hi ha moltes aplicacions que tenen parts amb un nivell molt elevat de paral·lelisme de dades.

Un programa en CUDA consisteix en una o més fases que es poden executar o bé en el processador principal (CPU) o bé en el dispositiu GPU. Les fases en les quals hi ha molt poc o gens de paral·lelisme de dades s'implementen en el codi que s'executarà en el processador principal, i les fases amb un nivell de paral·lelisme de dades elevat s'implementen en el codi que s'executarà al dispositiu.

Els elements fonamentals de CUDA són:

**a) Model de memòria**. És important remarcar que la memòria del processador principal i del dispositiu són espais de memòria completament separats (tot i que algunes arquitectures actuals i models de programació inclouen memòria compartida entre l'amfitrió i el dispositiu). Això reflecteix la realitat que els dispositius són típicament targetes que tenen la seva pròpia memòria DRAM. Per a executar un *kernel* al dispositiu GPU normalment cal seguir els passos següents:

- Reservar memòria al dispositiu.
- Transferir les dades necessàries des del processador principal a l'espai de memòria assignat al dispositiu.
- Invocar l'execució del *kernel* en qüestió.
- Transferir les dades amb els resultats des del dispositiu cap al processador principal.
- Alliberar la memòria del dispositiu (si ja no és necessària), un cop finalitzada l'execució del *kernel*.

**Activitat**

Busqueu les noves extensions d'OpenCL que Intel ha proposat per compartir memòria entre l'amfitrió i el dispositiu.

**b) Organització de fluxos**. En CUDA, un *kernel* s'executa mitjançant un conjunt de fluxos (per exemple, un vector i una matriu de fluxos). Com que tots els fluxos executen el mateix *kernel* (model SIMT) es necessita un mecanisme que permeti diferenciar-los, i així poder assignar la part corresponent de les dades a cada flux d'execució. CUDA incorpora paraules clau per a fer referència a l'índex d'un flux.

c) ***Kernels***. El codi que s'executa en el dispositiu (*kernel*) és la funció que executen els diferents fluxos durant la fase paral·lela, cadascun en el rang de dades que li correspon. Cal dir que CUDA segueix el model SPMD[25] i, per tant, tots els fluxos executen el mateix codi.

[25]De l'anglès *single program, multiple data*.

A continuació es mostra la funció o *kernel* de la suma de matrius i la seva crida.

```
__global__ matrix_add_gpu (fload *A, float *B, float *C, int N)
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   int j = blockIdx.y * blockDim.y + threadIdx.y;
   int index = i + j*N;
   if (i<N && j<N){
      C[index] = A[index] + B[index];
   }
}
int main(){
   dim3 dimBlock(blocksize, blocksize);
   dim3 dimGrid(N/dimBlock.x, N/dimBlock.y);
   matrix_add_gpu<<<dimGrid, dimBlock>>>(a, b, c, N);
}
```

OpenCL és una interfície estàndard, oberta, lliure i multiplataforma per a la programació paral·lela. La principal motivació per al desenvolupament d'OpenCL va ser la necessitat de simplificar la tasca de programació portable i eficient de la creixent quantitat de plataformes heterogènies, com ara CPU multinucli, GPU o fins i tot sistemes encastats. OpenCL va ser concebuda per Apple, tot i que la va acabar desenvolupant el grup Khronos, que és el mateix que va impulsar OpenGL i n'és responsable.

OpenCL consisteix en tres parts: l'especificació d'un llenguatge multiplataforma, una interfície a escala d'entorn de computació i una interfície per a coordinar la computació paral·lela entre processadors heterogenis. OpenCL utilitza un subconjunt de C99 amb extensions per al paral·lelisme i utilitza l'estàndard de representació numèrica IEEE 754 per a garantir la interoperabilitat entre plataformes.

Hi ha moltes similituds entre OpenCL i CUDA, tot i que OpenCL té un model de gestió de recursos més complex, ja que funciona amb múltiples plataformes i té portabilitat entre diferents fabricants. OpenCL funciona amb models de paral·lelisme tant de dades com de tasques.

De la mateixa manera que en CUDA, un programa en OpenCL està format per dues parts: els *kernels* que s'executen en un o més dispositius i un programa en el processador principal que invoca i controla l'execució dels *kernels*. Quan es fa la invocació d'un *kernel*, el codi s'executa en tasques elementals[26]. que

[26]En anglès, *work items*.

[27]En anglès, *NDRanges*.

corresponen als fluxos de CUDA. Les tasques elementals i les dades associades a cada tasca elemental es defineixen a partir del rang d'un espai d'índexs de dimensió N[27]. Les tasques elementals formen grups de tasques[28], que corresponen als blocs de CUDA. Les tasques elementals tenen un identificador global que és únic. A més, els grups de tasques elementals s'identifiquen dins del rang de dimensió N i, per a cada grup, cadascuna de les tasques elementals té un identificador local, que anirà des de 0 fins a la mida del grup-1. Per tant, la combinació de l'identificador del grup i de l'identificador local dins del grup també identifica de manera única una tasca elemental.

(28) En anglès, *work groups*.

OpenCL també té el seu propi model de memòria i de gestió de *kernels* i de dispositius. A continuació es mostra la funció o *kernel* de la suma de matrius en OpenCL.

```
__kernel void matrix_add_opencl ( __global const float *A,
                  __global const float *B,
                  __global float *C,
                  int N) {
   int i = get_global_id(0);
   int j = get_global_id(1);
   int index = i + j*N;
   if (i<N && j<N){
      C[index] = A[index] + B[index];
   }
}
```

## 3.2. Models de programació de memòria distribuïda

### 3.2.1. MPI

MPI (Message Passing Interface) és una biblioteca de pas de missatges. Es pot utilitzar bàsicament en programes C o Fortran, des dels quals es fan crides a funcions d'MPI per a la gestió de processos i per a comunicar els processos entre si.

MPI no és l'única biblioteca disponible de pas de missatges, però es pot considerar l'estàndard actual pel paradigma de memòria distribuïda. Abans d'MPI hi van haver altres models com ara PVM[29] tot i que MPI es va acabar imposant.

(29) De l'anglès *parallel virtual machine*.

Aquesta especificació va ser desenvolupada per l'MPI Forum, una agrupació d'universitats i empreses que van especificar les funcions que hauria de tenir una biblioteca de pas de missatges. A partir de l'especificació els diferents fabricants de multicomputadors van incloure implementacions d'MPI específi-

ques per als seus equips i van aparèixer diverses implementacions lliures, les més esteses de les quals són MPICH i OpenMPI (que és una distribució de codi lliure d'MPI2).

Resumint, MPI ha aconseguit una sèrie de fites, entre les quals podem remarcar les següents:

**Altes prestacions**

IBM, per exemple, té la seva pròpia biblioteca d'MPI per als seus sistemes BlueGene.

- **Estandardització**. Les implementacions de l'especificació han fet que es convertís en l'estàndard de pas de missatges i que no sigui necessari desenvolupar programes diferents per a màquines diferents.

- **Portabilitat**. Els programes MPI funcionen sobre multiprocessadors de memòria compartida, multicomputadors de memòria distribuïda, clústers de computadors, sistemes heterogenis, etc., sempre que hi hagi una versió d'MPI per a ells.

- **Altes prestacions**. Els fabricants han desenvolupat implementacions eficients per als seus equips.

- **Àmplia funcionalitat**. MPI inclou gran quantitat de funcions per a dur a terme de manera senzilla les operacions que acostumen a aparèixer en programes de pas de missatges.

Quan es posa en marxa un programa MPI es creen diversos processos, tots executant el mateix codi, cadascun amb les seves pròpies variables (model SPMD). A diferència d'OpenMP, no hi ha un procés mestre que controla la resta.

A continuació es mostra un programa que implementa el "hello, world" en MPI.

```
#include <stdio.h>
#include <mpi.h>
int main ( int argc, char *argv[])
{
   int rank, size;
   MPI_Init (&argc, &argv); /* starts MPI */
   MPI_Comm_rank (MPI_COMM_WORLD, &rank); /* Obté l'identificador del procés */
   MPI_Comm_size (MPI_COMM_WORLD, &size); /* Obté el nombre de processos */
   printf( "Hello world from process %d of %d\n", rank, size );
   MPI_Finalize();
   return 0;
}
```

Tot i que es tracta d'un codi molt simple, es poden observar alguns components essencials d'MPI:

- Cal incloure la biblioteca d'MPI (`mpi.h`).

- Tots els processos executen el mateix codi des del principi, de manera que tots tenen variables `myrank` i `size`. A diferència d'OpenMP, aquestes variables són diferents i poden ser en memòries diferents en processadors diferents.

- Els processos treballen de manera independent fins que s'inicialitza MPI amb la funció `MPI_Init`. A partir d'aquest punt, els processos poden col·laborar intercanviant dades, sincronitzant-se, etc.

- La funció `MPI_Finalize` es crida quan ja no és necessari que els processos col·laborin entre si. Aquesta funció allibera tots els recursos reservats per a MPI.

- Les funcions MPI tenen la forma `MPI_Nom(paràmetres)`. De la mateixa manera que en OpenMP és necessari que els processos sàpiguen l'identificador que tenen (entre 0 i el nombre de processos menys 1) i el nombre de processos que s'han posat en marxa. Les funcions `MPI_Comm_rank` i `MPI_Comm_size` s'utilitzen per a aconseguir això.

- Veiem que aquestes funcions tenen un paràmetre `MPI_COMM_WORLD` que és una constant MPI i que identifica el comunicador al qual estan associats tots els processos. Un comunicador és un identificador d'un grup de processos, i les funcions MPI han d'identificar en quin comunicador s'estan fent les operacions que es criden.

Tal com veurem en un altre mòdul d'aquesta assignatura, MPI ofereix una àmplia interfície on podem trobar operacions tant punt a punt (per exemple, per a enviar una dada d'un procés a un altre) com col·lectives (per exemple, per a sincronitzar tots els processos en un punt concret o reduir un conjunt de dades que estan distribuïdes entre els diferents processos del programa MPI).

**Vegeu també**

De la interfície d'MPI en tractem en el mòdul 3 d'aquesta assignatura.

### 3.2.2. PGAS

Molts programadors troben els models de programació de memòria compartida molt més atractius i pràctics que els de pas de missatges. Per donar resposta a això, hi ha diferents grups que estan desenvolupant llenguatges de programació paral·lela que permeten utilitzar algunes tècniques de memòria compartida per a programar sistemes que realment són de memòria distribuïda. Això no és tan simple com sembla ja que si, per exemple, escrivim un compilador que gestioni un conjunt de memòries distribuïdes com si en fossin una de compartida, els nostres programes tindrien un rendiment extraordinàriament baix i difícil de predir, ja que no és clar quan s'accedeix a la memòria local o distribuïda. Accedir a la memòria d'un altre node pot ser centenars o fins i tot milers de vegades més lent que accedir a la memòria local.

Els llenguatges PGAS[30] proporcionen alguns dels mecanismes dels programes de memòria compartida, però també proporcionen eines al programador per tal de poder controlar la localitat de les dades dins de l'espai (abstracte) de memòria compartida. Les variables privades s'allotgen en la memòria local del nucli en el qual s'està executant el procés, i el programador pot controlar la distribució de les dades en estructures de dades compartides. D'aquesta manera, per exemple, el programador pot saber quins elements d'un vector compartit hi ha a la memòria local de cada procés.

(30)De l'anglès *partitioned global address space*.

Hi ha diverses implementacions de llenguatges PGAS, les més importants de les quals són UPC[31], Co-Array Fortran i Titanium, les quals estenen C, Fortran i Java, respectivament.

(31)De l'anglès *unified parallel C*.

### 3.3. Models de programació híbrids

Els models de programació vistos més amunt, com OpenMP i MPI, donen solucions concretes a sistemes de memòria compartida i distribuïda, respectivament. En canvi, els actuals sistemes d'altes prestacions combinen sistemes de memòria distribuïda (bàsicament clústers) amb sistemes de memòria compartida (processadors o fins i tot multiprocessadors multinucli).

Així, doncs, també s'utilitzen models de programació híbrids, com ara MPI +OpenMP, per a aplicacions que tenen més d'un nivell de paral·lelisme (multinivell). Tot i que hi ha diferents maneres de combinar dos tipus de models de programació, en aquests casos s'acostuma a utilitzar MPI per als bucles (*loops*) més exteriors i OpenMP per als bucles interns, i així s'aprofita la localitat de les dades.

A continuació es mostra un programa híbrid MPI+OpenMP molt senzill que implementa una versió del "hello, world" vist més amunt. En aquest exemple no hi ha realment pas de missatge però il·lustra la manera de consultar més d'un nivell de paral·lelisme.

```
#include <stdio.h>
#include "mpi.h"
#include <omp.h>
int main(int argc, char *argv[]) {
   int numprocs, rank, namelen;
   char processor_name[MPI_MAX_PROCESSOR_NAME];
   int iam = 0, np = 1;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Get_processor_name(processor_name, &namelen);
   #pragma omp parallel default(shared) private(iam, np)
   {
```

```
        np = omp_get_num_threads();
        iam = omp_get_thread_num();
        printf("Hello from thread %d out of %d from process %d out of %d on %s\n",
           iam, np, rank, numprocs, processor_name);
    }
    MPI_Finalize();
}
```

Una altra motivació dels models de programació híbrids és que els diferents models de programació tenen avantatges i inconvenients i la combinació de més d'un model de programació pot aprofitar els avantatges i deixar de banda els inconvenients. Podem comparar diferents aspectes d'MPI i OpenMP, els dos paradigmes més habituals:

- **Gra de paral·lelisme**. OpenMP és més apropiat per a paral·lelisme de gra més fi, és a dir, quan hi ha poca computació entre sincronitzacions entre processos.

- **Arquitectura d'execució**. OpenMP funciona bé en sistemes de memòria compartida, però en sistemes distribuïts és preferible MPI.

- **Dificultat de programació**. En general els programes OpenMP són més semblants als seqüencials que no pas els MPI i és més fàcil desenvolupar programes OpenMP.

- **Eines de depuració**. Es disposa de més eines de desenvolupament, depuració, autoparal·lelització, etc., per a memòria compartida que no pas per a pas de missatges.

- **Creació de processos**. En OpenMP els fluxos es creen dinàmicament, mentre que en MPI els processos es creen estàticament, tot i que algunes versions d'MPI permeten la gestió dinàmica.

- **Ús de la memòria**. L'accés a la memòria és més senzill amb OpenMP pel fet d'utilitzar memòria compartida. Tot i això la localització de les dades en la memòria i l'accés simultani pot reduir el rendiment. La utilització de memòries locals amb MPI pot resultar en accessos més ràpids.

Així, doncs, alguns dels avantatges de la programació híbrida són:

- Pot ajudar a millorar l'escalabilitat i a millorar el balanceig de les tasques.

- Es pot utilitzar en aplicacions que combinen paral·lelisme de gra fi i gruixut o que barregen paral·lelisme funcional i de dades.

- Es pot utilitzar per a reduir el cost de desenvolupament; per exemple, si es tenen funcions que utilitzen OpenMP i s'utilitzen dins de programes MPI.

Tot i això, també cal tenir en compte que utilitzar dos paradigmes de manera híbrida en pot complicar la programació.

Finalment, els models híbrids també són una solució per a determinades funcionalitats que no s'acostumen a donar amb un únic model de programació. Un exemple d'això és el model de programació MPI+CUDA. CUDA permet aprofitar el gran potencial de còmput dels processadors gràfics, mentre que MPI permet distribuir les tasques en diversos nodes i així poder fer servir simultàniament més acceleradors gràfics dels que pot tenir un únic node físicament.

**Vegeu també**

En el mòdul didàctic 3 tractarem dels diferents models de programació breument introduïts en aquest subapartat.

# 4. Rendiment d'aplicacions paral·leles

Quan escrivim programes paral·lels normalment podrem apreciar una millora en el rendiment, ja que en molts casos es pot treure profit de disposar de més nuclis. En canvi, escriure un programa paral·lel és un art, ja que hi ha moltes maneres de fer-ho, perquè no hi ha una norma general. Així, doncs, cal saber què és el que podem esperar de la nostra implementació i per tal de poder saber si se'n pot millorar el rendiment i com s'ha de fer ens caldrà poder-la avaluar. A continuació veurem les principals mètriques utilitzades per a determinar el rendiment d'un programa paral·lel, veurem els estàndards principals en relació amb la computació d'altes prestacions i comprovarem que hi ha eines molt convenients que permeten analitzar el rendiment de les nostres aplicacions de tal manera que podem millorar la nostra productivitat en desenvolupar programes paral·lels.

## 4.1. *Speedup* i eficiència

Normalment la millor manera d'escriure el nostres programes paral·lels és dividint el treball entre els diferents nuclis a parts iguals, a la vegada que no s'introdueix cap treball addicional per als nuclis. Si realment podem aconseguir executar el programa en $p$ nuclis mitjançant un flux o procés a cadascun dels nuclis, llavors el nostre programa paral·lel anirà $p$ vegades més ràpid que el programa seqüencial. Si anomenem el temps d'execució seqüencial $T_{seq}$ i el temps d'execució del nostre programa paral·lel $T_{par}$, llavors el límit que podem esperar és $T_{par} = T_{seq}/p$. Quan això passa diem que el programa paral·lel té *speedup* lineal.

A la pràctica és molt difícil poder obtenir *speedup* lineal perquè només pel fet d'utilitzar múltiples processos o fluxos s'afegeix un sobrecost[32].

[32] En anglès, *overhead*.

Per exemple, en programes amb memòria compartida cal afegir mecanismes d'exclusió mútua com ara semàfors i en programes de memòria distribuïda en algun moment caldrà transmetre dades per la xarxa per a implementar el pas de missatges. A més, cal tenir en compte que els costos addicionals augmentaran a mesura que s'incrementi el nombre de processos o fluxos.

Així, doncs, si definim l'*speedup* d'un programa paral·lel de la manera següent:

$$S = \frac{T_{seq}}{T_{par}} \qquad 1.1$$

llavors l'*speedup* lineal (rendiment proporcional al nombre d'unitats de còmputs usades per a resoldre el problema) té $S = p$, que no és gens habitual. A més, a mesura que $p$ augmenti, hem d'esperar que $S$ sigui una fracció cada cop més petita de l'*speedup* lineal $p$, que és l'ideal. Una altra manera de veure-ho és

que $S/p$ serà probablement cada cop més petit, ja que $p$ augmenta. La taula 4 mostra un exemple de la manera com canvien $S$ i $S/p$ a mesura que $p$ augmenta. Aquest valor $S/p$ s'anomena *eficiència del programa paral·lel*. Si substituïm la formula_ per $S$ veiem que l'eficiència és:

$$E = \frac{S}{P} = \frac{\left(\frac{T_{seq}}{T_{par}}\right)}{p} = \frac{T_{seq}}{p.T_{par}} \qquad 1.2$$

Taula 4. *Speedups* i eficiències d'un programa paral·lel

| $p$ | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| $S$ | 1,0 | 1,9 | 3,6 | 6,5 | 10,8 |
| $E = S/p$ | 1,0 | 0,95 | 0,90 | 0,81 | 0,68 |

És clar que $T_{par}$, $S$ i $E$ depenen del nombre de processos o fluxos ($p$). També cal tenir present que $T_{par}$, $S$, $E$ i $T_{seq}$ depenen de la mida del problema.

Per exemple, si executem l'aplicació amb què es va obtenir la taula 4 però amb la meitat i el doble de la mida del problema, podrem obtenir els *speedups* i les eficiències que es mostren a les figures 15 i 16, respectivament.

Figura 15. *Speedups* del programa paral·lel amb diferent mida del problema

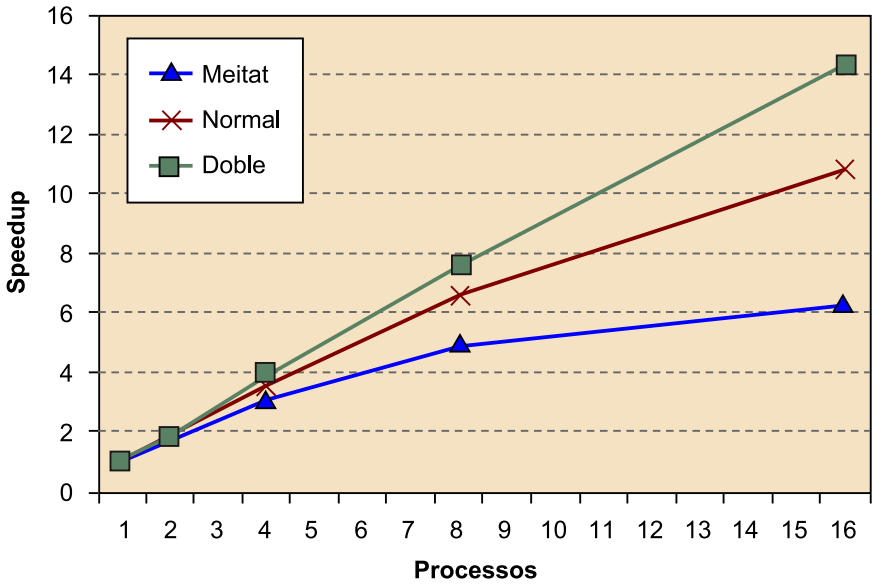
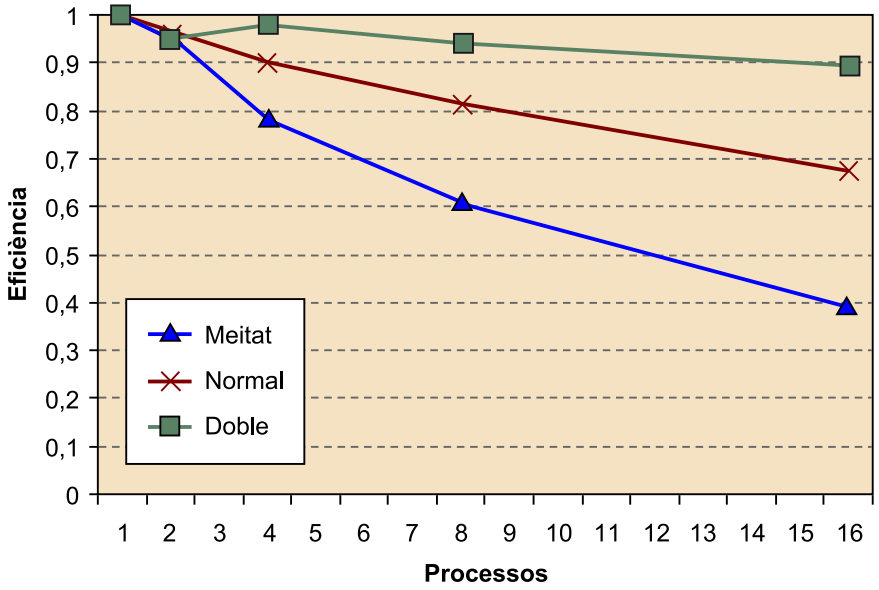


Figura 16. Eficiències del programa paral·lel amb diferent mida del problema




Tal com veiem en aquest exemple, quan la mida del problema és més gran, augmenten els *speedups* i les eficiències, mentre que, quan la mida del problema és més petita, disminueixen. Aquest comportament és molt habitual.

Hi ha molts programes paral·lels que es desenvolupen dividint el treball de la versió seqüencial entre els diferents processos o fluxos i afegint el sobrecost necessari per tal de manegar el paral·lelisme, com ara per l'exclusió mútua o per les comunicacions. Així, doncs, si anomenem aquest sobrecost del paral·lelisme $T_{overhead}$, tenim que:

$$T_{par} = \frac{T_{seq}}{p} + T_{overhead} \qquad 1.3$$

També cal tenir en compte que habitualment, a mesura que augmenta la mida del problema $T_{overhead}$ augmenta més lentament que $T_{seq}$. Si succeeix això, augmentaran tant l'*speedup* com l'eficiència. En altres paraules, si hi ha més treball per fer per part dels diferents processos o fluxos, la proporció de temps destinada a coordinar les tasques serà inferior que si la mida del problema és més petita.

### 4.2. Llei d'Amdahl

La llei d'Amdahl (Gene Amdahl, 1967) diu que, si no és que un programa seqüencial es pugui paral·lelitzar completament, l'*speedup* que s'obtindria serà molt limitat, independentment del nombre de nuclis disponibles.

Suposem que hem pogut paral·lelitzar el 90% del codi del programa seqüencial i que, a més, l'hem paral·lelitzat perfectament. Segons la llei d'Amdahl, independentment del nombre de nuclis $p$ que utilitzem, l'*speedup* d'aquesta part del programa serà $p$. Si la part seqüencial de l'execució és $T_{seq} = 20$ segons, el temps d'execució de la part paral·lela serà $0,9 \times T_{seq}/p = 18/p$ i el temps d'execució de la part no paral·lela serà $0,1 \times T_{seq} = 2$. El temps d'execució de la part paral·lela serà:

$$T_{par} = 0,9 \times \frac{T_{seq}}{p} + 0,1 \times T_{seq} = \frac{18}{p} + 2 \qquad 1.4$$

i l'*speedup* serà:

$$S = \frac{T_{seq}}{0,9 \times \frac{T_{seq}}{p} + 0,1 \times T_{seq}} = \frac{20}{\frac{18}{p} + 2} \qquad 1.5$$

Si el nombre de nuclis $p$ es fes més i més gran (si $p \to \infty$), llavors $0,9 \times T_{seq}/p = 18/p$ tendiria a 0 i, per tant, el temps d'execució total no podria ser més petit que $0,1 \times T_{seq} = 2$ Per tant, l'*speedup* hauria de ser més petit que:

$$S \leq \frac{T_{seq}}{0,1 \times T_{seq}} = \frac{20}{2} = 10 \qquad 1.6$$

Això vol dir que encara que fem una paral·lelització perfecta del 90% del programa, mai no podrem obtenir un *speedup* millor que 10, encara que disposéssim de milions de nuclis.

De manera general, si una fracció $r$ del programa seqüencial no es pot paral·lelitzar, llavors la llei d'Amdahl diu que no podrem obtenir un *speedup* millor que $1/r$. L'aplicació de la llei d'Amdahl pot semblar molt pessimista, però també hi ha la llei de Gustafson, que considera que la part no paral·lela d'un programa decreix quan augmenta la mida del problema. Per tant, segons la llei de Gustafson, qualsevol problema prou gran es pot paral·lelitzar eficientment i l'*speedup* passa a ser:

$$S = p - r + (p - 1) \qquad 1.7$$

### 4.3. Escalabilitat

De manera general, podem dir que una tecnologia o aplicació és **escalable** si és capaç de manegar un problema de mida creixent. Més formalment, diem que un programa és escalable si l'eficiència es manté constant, independentment de si augmenta la mida del problema.

Hi ha dues nocions comunes de l'escalabilitat. La primera és escalat fort, en el qual si s'incrementa el nombre de processos o fluxos es pot conservar l'eficiència sense la necessitat d'incrementar la mida del problema. El segon és escalat feble, en el qual si volem conservar l'eficiència fixa cal incrementar la mida del problema de la mateixa manera que s'augmenta el nombre de processos o fluxos.

## 4.4. Anàlisi de rendiment d'aplicacions paral·leles

En el moment de programar les nostres aplicacions paral·leles podem seguir alguns estàndards generals o seguir procediments de bones pràctiques, però és difícil saber si la nostra aplicació podrà complir els requeriments de rendiment que volem. En cas de voler millorar el rendiment de la nostra aplicació o simplement si obtenim un rendiment que considerem pobre per a la mateixa experiència, és molt difícil poder identificar la font del problema i trobar-hi solució sense analitzar-la detingudament.

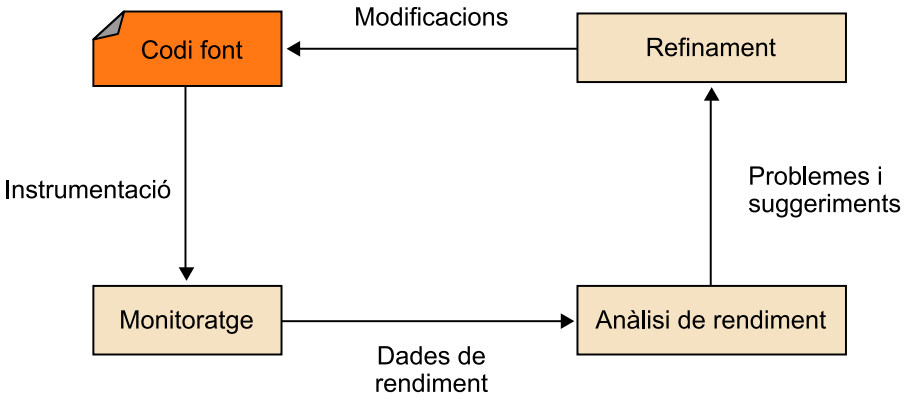
En general l'anàlisi i optimització de rendiment d'aplicacions paral·leles és un procés cíclic, tal com mostra la figura següent:

- La fase d'instrumentació i monitoratge consisteix a afegir a l'aplicació una certa informació d'instrumentació que permet recopilar informació respecte al comportament de l'aplicació (per exemple, nombre de fallades a la memòria cau de nivell 2).

- La fase d'anàlisi consisteix a inspeccionar la informació recopilada en la fase anterior per tal de trobar problemes de rendiment, deduir les possibles causes i determinar solucions.

- La fase de refinament consisteix a aplicar els canvis oportuns al codi de l'aplicació per tal de poder corregir els possibles problemes de rendiment identificats.

Figura 17. Procés cíclic de millora del rendiment



Per a analitzar el rendiment d'aplicacions paral·leles es poden utilitzar diferents mecanismes, i d'aquests mecanismes n'hi ha de més simples i de més complexos, de més generals i de més acurats, tal com veurem en els subapartats següents. Cal remarcar que la utilització d'aquests mecanismes no és sempre igual per a tots els casos i l'anàlisi de rendiment d'aplicacions paral·leles normalment esdevé un art i s'aprèn mitjançant l'experimentació.

### 4.4.1. Mesures de temps

Aquesta és la manera més simple i tradicional d'instrumentació i monitoratge d'aplicacions però que en certes ocasions pot ser molt eficaç i ràpida. De fet hi ha diverses raons per a prendre mesures de temps, com per exemple per a determinar si el programa que s'està desenvolupant es comporta com ho hauria de fer o, un cop el programa ja està desenvolupat, poder determinar fins a quin punt és bo el rendiment que té.

Cal tenir en compte que no sempre serà important saber el temps d'execució de tota l'aplicació, sinó que serà més important mesurar el temps d'execució de certes parts del programa. Per això, normalment no es podrà utilitzar la crida `time` típica de l'intèrpret d'ordres d'Unix, la qual retorna el temps d'execució del programa des del principi fins al final.

També hem de tenir present que no sempre ens interessarà saber el temps de processador, que és el que retorna la funció estàndard de C `clock`. De fet, es tracta del temps total que el programa passa executant codi del programa. Això pot ser un problema, ja que no inclou el temps que el programa està aturat.

**Programa en espera**

Un exemple d'això és quan un programa de memòria distribuïda està esperant que li arribi un missatge des d'un altre node.

Per tant, per a prendre mesures de programes paral·lels, com a norma general caldrà tenir en compte el temps total d'execució[33]. A continuació es mostra un codi senzill per a mesurar el temps d'execució d'un programa.

[33]També conegut com a *wall clock*.

```
Double start, finish;
...
start = get_current_time();
/* Codi que volem mesurar */
...
finish = get_current_time();
printf("Temps transcorregut = %e segons\n", finish-start);
```

En aquest exemple la funció `get_current_time()` és una hipotètica funció que retorna els segons que han transcorregut des d'un instant de temps prefixat del passat. La funció concreta que cal utilitzar depèn de la interfície de programació.

**Exemple**

Per exemple, MPI disposa de la funció `MPI_Wtime` i OpenMP disposa de la funció `omp_get_wtime`. Ambdues retornen el temps total d'execució i no pas el temps de processador.

Hi ha una qüestió molt important que cal tenir en compte a l'hora de prendre mesures de temps que és la resolució de les funcions de mesura de temps. Sempre caldrà que la resolució, que és l'interval de temps entre dues mesures, sigui més petita que la mesura que volem prendre ja que si no obtindrem una mesura de 0.

**La resolució**

Hi ha moltes funcions de mesura de temps que tenen una resolució de microsegons ($10^{-3}$) i moltes vegades ens caldran resolucions de nanosegons ($10^{-9}$) per a poder identificar esdeveniments d'interès per a l'anàlisi.

Finalment, cal destacar que quan es prenen mesures de temps es poden obtenir diferents valors en diferents execucions. Així, doncs, caldrà tenir en compte aquesta possible variabilitat de les mesures i quan es facin estudis a partir de mesures de temps utilitzar valors estadístics com ara el valor mitjà obtingut de diverses execucions. També cal tenir en compte que ens podem trobar mesures anòmales per culpa de factors puntuals com ara un mal funcionament de la xarxa d'interconnexió.

### 4.4.2. Interfícies d'instrumentació i monitoratge

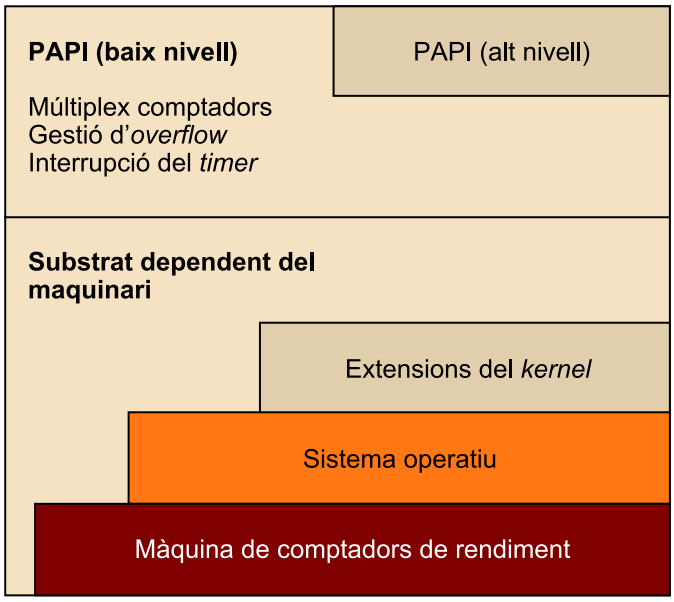
Les eines de monitoratge normalment consten de dues parts: una biblioteca o un conjunt de biblioteques que permeten inserir codi d'instrumentació per a mesurar i emmagatzemar dades, i una sèrie de mòduls que ofereixen la possibilitat de mostrar les dades generades durant el monitoratge de l'aplicació paral·lela. Cal tenir en compte que la instrumentació pot afectar les característiques de rendiment de l'aplicació paral·lela, ja que poden afegir un cert sobrecost relacionat amb la instrumentació.

Tot i que n'hi ha moltes, una de les eines d'instrumentació més utilitzades és PAPI[34]. PAPI és una interfície per a poder accedir als comptadors de maquinari relacionats amb el rendiment que estan disponibles en la majoria de processadors actuals. Aquests comptadors són un conjunt de registres que van comptant les vegades que succeeixen determinats esdeveniments en el processador.

[34]De l'anglès *performance application programming interface*

El *kernel* del sistema operatiu pot necessitar algunes modificacions per a permetre accedir als comptadors de maquinari mitjançant l'ús de PAPI. Depenent del processador i del sistema operatiu, pot arribar a ser necessari aplicar un *patch* (per exemple, `PertCtr`) i haver de recompilar el *kernel*. La figura següent mostra l'arquitectura bàsica de PAPI.

Figura 18. Arquitectura de PAPI




Quan afegim PAPI a una part del nostre codi, es poden passar a l'aplicació com a paràmetres els noms dels comptadors de maquinari dels quals es vol obtenir informació. Cal tenir present que hi ha força comptadors de maquinari als quals es pot accedir mitjançant PAPI.

**Us de PAPI**

A tall d'exemple, per a obtenir el nombre de MIPS o Mflops que l'aplicació està obtenint es necessiten el comptadors de maquinari següents: PAPI_FP_OPS (nombre d'operacions en coma flotant), PAPI_TOT_INS (nombre total d'instruccions) i PAPI_TOT_CYC (nombre total de cicles). Hi ha altres comptadors de màquina que donen, per exemple, informació relacionada amb la memòria, com ara el nombre de fallades a memòria cau de cadascun dels seus nivells. A continuació es mostra un exemple de la utilització de PAPI.

```
#include "papi.h"
#define NUM_EVENTS 2
int Events[NUM_EVENTS] = {PAPI_FP_OPS, PAPI_TOT_CYC};
INT EventSet;
Long long vàlues[NUM_EVENTS];
/* Inicialització de la biblioteca */
retval = PAPI_library_init (PAPI_VER_CURRENT);
/* Cercar espai per al nou element */
retval = PAPI_create_eventSet (&EventSet);
/*Afegeix Flops i altres cicles que cal monitorar */
retval = PAPI_add_eventset (&EventSet, Events, NUM_EVENT);
/* Comencen a funcionar els comptadors */
retval = PAPI_start(EventSet);
fer_treball(); /* El que suposem que monitorem */
/* Atura els comptadors i restaura els valors per defecte */
retval = PAPI_stop(EventSet, vàlues);
```

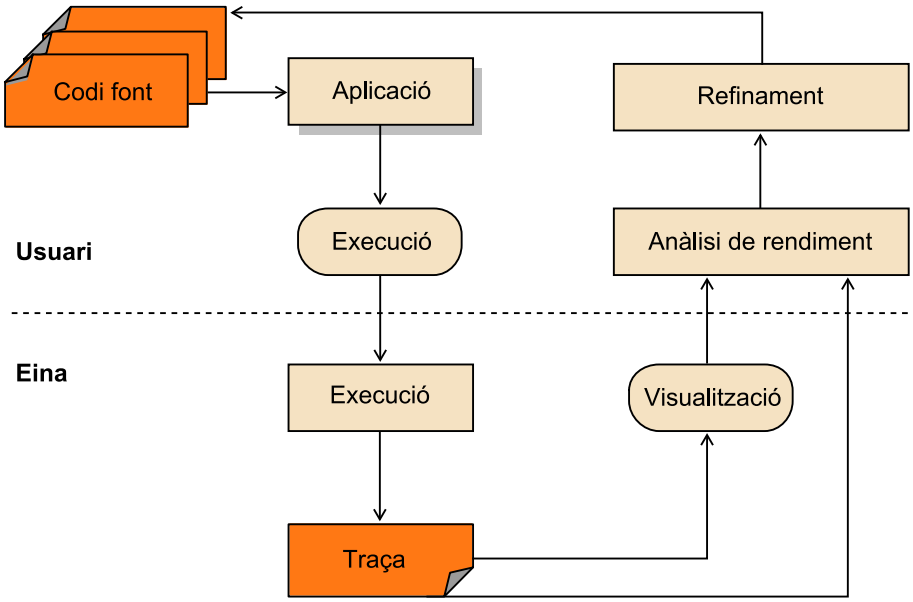
### 4.4.3. Eines d'anàlisi de rendiment

L'objectiu de les eines d'anàlisi de rendiment és analitzar de manera més o menys automàtica la informació generada durant la fase de monitoratge. Mitjançant aquestes eines podem identificar possibles colls d'ampolla i altres pro-

blemes propis de les aplicacions paral·leles, encara que no proporcionen la solució a aquests problemes de manera automàtica, sinó que requereix la intervenció humana per a extreure conclusions i proposar solucions o millores.

La manera clàssica d'anàlisi de rendiment està basada en la visualització de l'execució de l'aplicació paral·lela mitjançant una traça, un cop s'ha acabat d'executar aquesta aplicació. Aquest procés es mostra a la figura següent i s'anomena *post mortem*.

Figura 19. Metodologia clàssica d'anàlisi de rendiment




Normalment les eines que s'utilitzen en aquest procés mostren informació específica sobre el comportament de l'aplicació mitjançant diferents vistes gràfiques i numèriques. Per això, primerament es requereix l'ús de l'eina que faci el monitoratge per a obtenir dades de rendiment de l'execució del programa paral·lel. La inserció de la instrumentació es pot fer de manera estàtica amb l'eina o bé la pot fer manualment l'usuari. El procés de monitoratge es pot fer seguint diverses tècniques:

- Basades en temps d'execució, mitjançant les quals es detecta quina part de l'aplicació paral·lela s'executa més estona.
- Basades en comptadors que indiquen el nombre de vegades d'un determinat esdeveniment en l'aplicació (per exemple, el cas de PAPI).
- Basades en mostreig, les quals generen mesures periòdicament sobre l'estat de l'aplicació.
- Basades en traces d'esdeveniments, que proporcionen informació associada a esdeveniments concrets definits en l'aplicació.

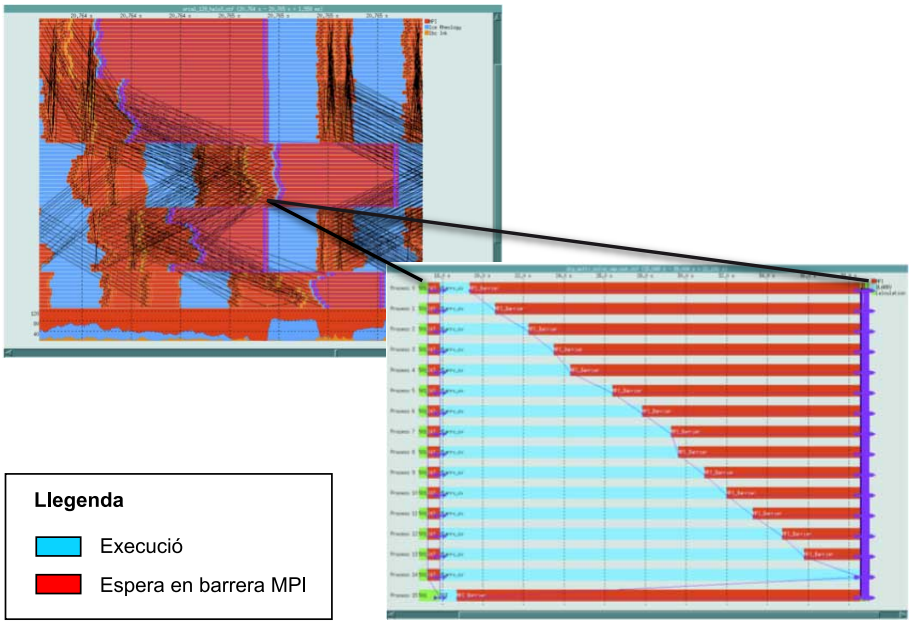
Com que les dades s'emmagatzemen en un fitxer de traça de l'aplicació, les eines de visualització generen gràfics sobre patró que segueix l'aplicació, com per exemple diagrames de Gantt, diagrames circulars o diagrames de barres. La informació mostrada s'ha de correspondre amb aspectes relacionats amb pas

de missatges, comunicacions col·lectives, execució de rutines de l'aplicació, entre d'altres. Finalment, l'usuari ha d'analitzar aquestes representacions gràfiques, buscant problemes de rendiment, determinant les causes d'aquests problemes i canviant el codi font manualment si és necessari. D'aquesta manera el procés global es repeteix tornant a compilar i executar l'aplicació, fins que obtinguem el rendiment que volem.

L'anàlisi de rendiment clàssic requereix un elevat grau d'experiència en programació paral·lela per a dur-lo a terme eficientment, de manera que resulta una tasca especialment difícil per a usuaris no experts. La complexitat d'aquesta tasca es deu principalment a la interpretació i la mida del fitxer de traça, la qual és proporcional a la mida i al temps d'execució de l'aplicació. A més, aquesta aproximació no és fiable quan les aplicacions o els entorns d'execució tenen un comportament dinàmic. Hi ha moltes aplicacions que tenen un comportament diferent segons les dades d'entrada o fins i tot poden variar durant la mateixa execució. A més, hi ha moltes eines de visualització que no escalen bé, de manera que, quan el nombre de processos implicats en l'aplicació és molt elevat, els gràfics generats no són llegibles correctament.

La figura següent mostra una traça on es pot observar un alt nivell de desbalanceig en els temps d'espera de diversos processos MPI en una barrera. En aquesta situació sabem que els nuclis romandran sense feina durant els temps d'espera en la barrera (color vermell del gràfic), però no és tan evident trobar-ne una solució.

Figura 20. Exemple de traça. Mostra un alt nivell de desbalanceig en una barrera MPI



A continuació veurem un parell d'exemples d'eines d'anàlisi de rendiment d'aplicacions paral·leles, Vampir i Taubench. Cal dir, però, que hi ha una gran diversitat d'eines d'anàlisi de rendiment, com per exemple de comercials, com VTune d'Intel, PPW[35] o Paraver, i d'altres de visualització, com per exemple Jumpshot.

(35)De l'anglès *parallel performance wizard*.

Vampir és una eina d'anàlisi de rendiment que permet la visualització gràfica i l'anàlisi de les dades de l'estat d'un programa, missatges punt a punt, operacions col·lectives i comptadors de rendiment de maquinari, juntament amb resums estadístics. Està dissenyada com una eina fàcil d'utilitzar, la qual permet als desenvolupadors visualitzar ràpidament el comportament de l'aplicació en un determinat nivell de detall.

Vampir va començar a desenvolupar-se al centre de matemàtica aplicada del Centre de Recerca de Jülich, a Alemanya, i al Centre de Computació d'Altes Prestacions de la Universitat Tècnica de Dresden, també a Alemanya, i està disponible com a producte comercial des de 1996. Aquesta eina ha estat utilitzada àmpliament per la comunitat de la computació d'altes prestacions durant molt anys. El format de traça que gestiona (actualment Open Trace o OTF) és compatible amb moltes altres eines.

Aquesta eina permet visualitzar les activitats i comunicacions de l'aplicació al llarg del temps en gràfics temporals, que l'usuari pot explorar fent *zooms* i desplaçant-se per l'execució per tal de detectar la causa dels problemes de rendiment, a més de permetre la correcta paral·lelització i balanceig de càrrega. També proporciona gràfics estadístics per caràcters quantitatius. Vampir està disponible per a pràcticament totes les plataformes de 32 i 64 bits, com ara PC, clústers Linux i IBM. També hi ha disponible un programari d'instrumentació i mesura conegut com a VampirTrace.

VampirTrace proporciona una infraestructura que permet el desenvolupament amb instrumentació i facilitats per a recollir mesures en aplicacions HPC. Cobreix l'anàlisi d'aplicacions desenvolupades en MPI i OpenMP. La instrumentació modifica l'aplicació per detectar i emmagatzemar esdeveniments d'interès generats durant l'execució, com per exemple una operació de comunicació MPI o una determinada crida a funció. Això es pot fer a escala de codi font, durant la compilació o en temps d'enllaç mitjançant diverses tècniques. La biblioteca de VampirTrace s'encarrega de la recollida de dades en tots els processos. Aquestes dades inclouen esdeveniments definits per l'usuari, esdeveniments MPI i OpenMP, i també informació sobre temporització o localització. A més, permet obtenir informació amb comptadors de maquinari mitjançant PAPI. La instrumentació automàtica del codi font es pot fer mitjançant diversos compiladors, entre els quals hi ha el de GNU. La instrumentació binària es fa amb Dynist. Finalment, les dades de rendiment recollides s'emmagatzemen en fitxer de format OTF. A la figura 20 es mostrava un exemple de traça visualitzada amb Vampir.

TAU[36] és un sistema d'anàlisi de rendiment paral·lel que integra un entorn i un conjunt d'eines automàtiques per a la instrumentació, mesura, anàlisi i visualització del rendiment d'aplicacions executades en sistemes paral·lels de gran escala. Una de les característiques principals que té és el gran nombre de plataformes de maquinari i programari amb què funciona. TAU es pot executar a la majoria d'entorns d'altes prestacions i permet diversos llenguatges, com ara C/C++, Java, Python, Fortran, OpenMP, MPI i Charm. L'arquitectura de TAU s'organitza en tres capes: instrumentació, mesura i anàlisi.

(36)De l'anglès *tuning and analysis utilities*.

TAU utilitza un model d'instrumentació flexible basat en instrumentació dinàmica, que permet a l'usuari inserir instrumentació de rendiment cridant la interfície de mesures de TAU. El concepte clau de la capa d'instrumentació és que aquesta capa és on es defineixen els esdeveniments de rendiment. El mecanisme d'instrumentació de TAU permet diferents tipus d'esdeveniments que defineixen el rendiment, incloent-hi esdeveniments definits per localitzacions de codi, esdeveniments d'interfície de biblioteca, esdeveniments del sistema i esdeveniments definits per l'usuari mateix. Així, doncs, la sortida de la instrumentació és informació sobre els esdeveniments d'un experiment de rendiment, la qual serà utilitzada per altres eines. La capa d'instrumentació es comunica amb la capa de mesura mitjançant la interfície de mesura de TAU. Els mòduls de TAU es divideixen en components per a fer la depuració del codi i obtenir traces, les quals es poden visualitzar amb eines especialitzades, com ara ParaProf o Vampir.

## 4.5. Anàlisi de rendiment de sistemes d'altes prestacions

De la mateixa manera que podem analitzar el rendiment de les nostres aplicacions paral·leles en un determinat sistema, hem de poder analitzar els sistemes d'altes prestacions en relació amb les aplicacions que volem executar. Això ens serà útil per a decidir o dissenyar les característiques del sistema que necessitem perquè les nostres aplicacions ens puguin donar el rendiment volgut.

Tal com hem vist més amunt, hi ha unitats de mesura de rendiment, com ara els flops, que resulten senzilles d'obtenir des d'un punt de vista teòric, ja que és determinat per l'arquitectura. En canvi, aquest rendiment teòric no es pot assolir a la pràctica, ja que hi ha els sobrecostos relacionats amb el paral·lelisme i amb l'escalabilitat en si de les aplicacions paral·leles que hem vist abans.

**Identificar la xarxa d'interconnexió**

Un exemple d'aquest tipus d'estudi és identificar quina xarxa d'interconnexió necessitarem quant a amplada de banda i latència per a assolir un cert rendiment (per exemple, mesurat en Mflops) d'una aplicació de memòria distribuïda amb pas de missatges. Per a dur a terme aquest tipus d'estudi, normalment es fan servir eines d'anàlisi de rendiment que poden obtenir el patró de comportament de l'aplicació utilitzant una configuració determinada i sistemes de predicció que ens poden dir quin seria el rendiment després de modificar la configuració del sistema. Tot i això, aquest procés no és senzill i és específic per a cada aplicació paral·lela.

Així, doncs, per a poder comparar sistemes i establir una referència quant al rendiment que poden oferir els sistemes d'altes prestacions, s'acostumen a utilitzar proves de rendiment[37], tal com veurem a continuació.

(37)En anglès, *benchmarks*.

### 4.5.1. Proves de rendiment

Tal com hem comentat, les proves de rendiment o *benchmarks* ens permeten mesurar el rendiment d'un sistema o d'un component en concret (CPU, memòria, disc, etc.). Mitjançant els *benchmarks* podem establir referències i tenir criteris per a comparar diferents sistemes o components. En concret, els *benchmarks* es tracten d'un conjunt de programes representatius de la càrrega del sistema que es vol analitzar.

El *benchmark* de referència per a sistemes d'altes prestacions és el Linpack, que va desenvolupar Jack Dongarra el 1976 a l'Argone National Laboratory. El Linpack és un conjunt de subrutines que analitzen i resolen equacions lineals d'alta densitat, les quals s'acostumen a utilitzar en les aplicacions d'altes prestacions. Els sistemes d'equacions lineals que es té en compte utilitzen diferents formes de matrius: generals, simètriques, triangulars, etc.

En general, el *benchmark* fa la resolució d'un sistema d'equacions generat aleatòriament, expressat com una matriu de coeficients que es representen amb nombre de punt flotant, normalment de precisió de 64 bits. El pas crucial d'aquesta solució és la descomposició LU amb pivot parcial de la matriu de coeficients. El resultat del *benchmark* és un valor de rendiment expressat en Mflops, la unitat tradicionalment utilitzada per a fer comparacions entre diferents sistemes.

Originalment, Linpack es va implementar en Fortran per a màquines uniprocessador, vectorials i de memòria compartida. A mesura que els sistemes de memòria distribuïda es van fer populars, es va fer necessari desenvolupar el *benchmark*HPL[38]. HPL és una implementació de Linpack desenvolupada en llenguatge C que es pot executar en qualsevol equip que disposi d'una implementació d'MPI.

(38)De l'anglès *high performance Linpack*.

HPL és el *benchmark* utilitzat per a mesurar el rendiment dels computadors més ràpids del món, els quals es recopilen cada sis mesos a la llista Top500.

Un altre *benchmark* representatiu és l'HPC Challenge Benchmark. De fet, es tracta d'un conjunt de *benchmarks* que proven diverses característiques de sistemes d'altes prestacions. En concret està compost pels set tests següents:

- HPL: és el *high performance Linpack* que hem comentat més amunt.

- DGEMM: està centrat en la memòria i mesura el rendiment en coma flotant de l'execució de la multiplicació de matrius de nombres reals de doble precisió.

- STREAM: és un *benchmark* basat en un programa sintètic que mesura l'amplada de banda a memòria sostingut (en GB/s).

- PTRANS[39]: està centrat en les comunicacions entre parells de processadors que es comuniquen entre si simultàniament. Aquest *benchmark* està orientat a l'avaluació de la capacitat de la xarxa.

(39) De l'anglès *parallel matrix transpose*.

- Accés aleatori: està centrat a mesurar el rendiment d'accessos aleatoris a memòria (també anomenat *GUPS*). Aquest *benchmark* està orientat a veure quina és la latència que el subsistema de memòria té per lectures. En aquest cas, els accessos a memòria es generen de tal manera que no hi ha mai reús a les memòries cau i sempre s'accedeix a la memòria final.

- FFT: és un *benchmark* que mesura el rendiment en punt flotant de l'execució d'una transformada discreta de Fourier complexa unidimensional de doble precisió.

- Amplada de banda i latència de comunicacions: és un conjunt de proves que mesuren la latència i l'amplada de banda de diversos patrons de comunicacions simultàniament. Està basat en el benchmark b_eff[40].

(40) De l'anglès *effecive bandwidth benchmark*.

### 4.5.2. Llista Top500

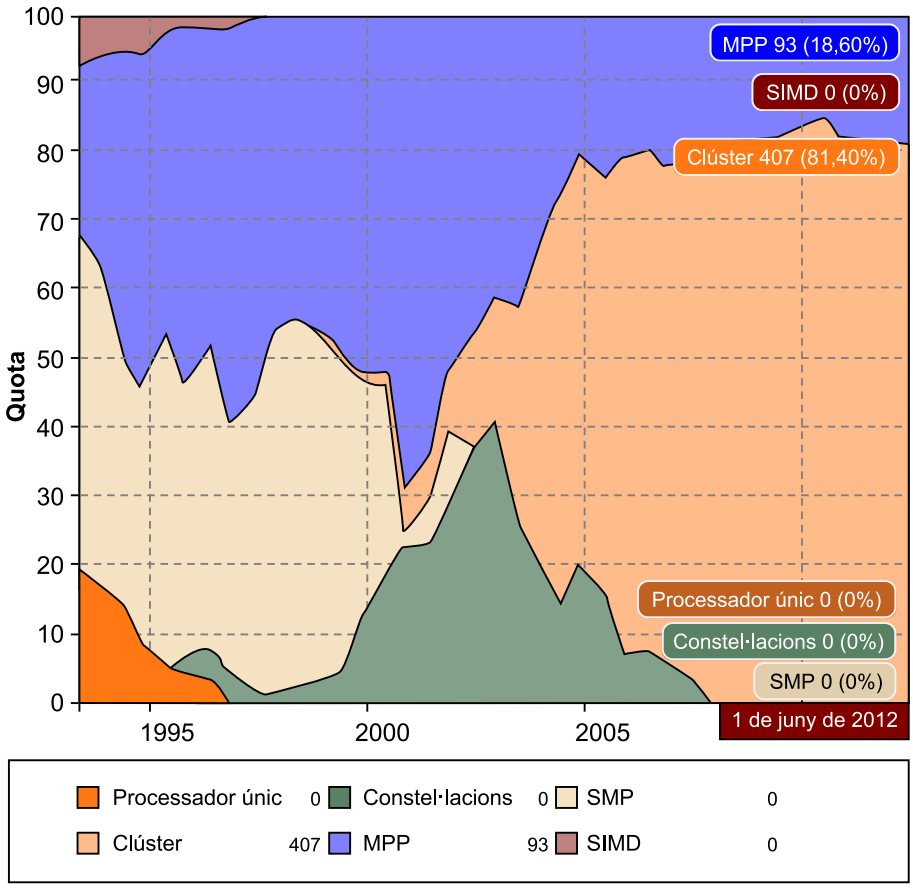
Cada sis mesos, s'avaluen els 500 supercomputadors més ràpids del món executant el *benchmark* Linpack que hem comentat més amunt sobre un conjunt de dades molt gran (entre altres raons, per a evitar limitacions d'escalabilitat a causa d'una mida de problema massa petita). La llista, anomenada *Top500*, varia cada any i es pot veure com una espècie de competició.

A més de fer públics els supercomputadors més ràpids en cada moment i les característiques que tenen, el Top500 publica estadístiques d'altra informació d'interès per entendre l'evolució de la computació d'altes prestacions.

És interessant observar a la figura següent l'evolució en el temps de l'arquitectura dels supercomputadors del Top500. El 1993, hi havia 250 sistemes que eren bàsicament de memòria compartida, i van desaparèixer tots de la llista el juny del 2002. Els sistemes SIMD van desaparèixer el 1997. Els sistemes MPP van ser molt populars cap al 2000 però han tingut una evolució a la baixa tot i que encara se'n poden trobar a la llista. En canvi, els sistemes basats en clúster van aparèixer el 1999 i actualment són els més populars del Top500, i representen la classe d'arquitectura dominant. La diferència princi-

pal entre els clústers i els sistemes MPP és que el clústers utilitzen components de mercat, mentre que els sistemes MPP tenen els propis dissenys específics de nodes, mòduls, xarxa d'interconnexió, etc.

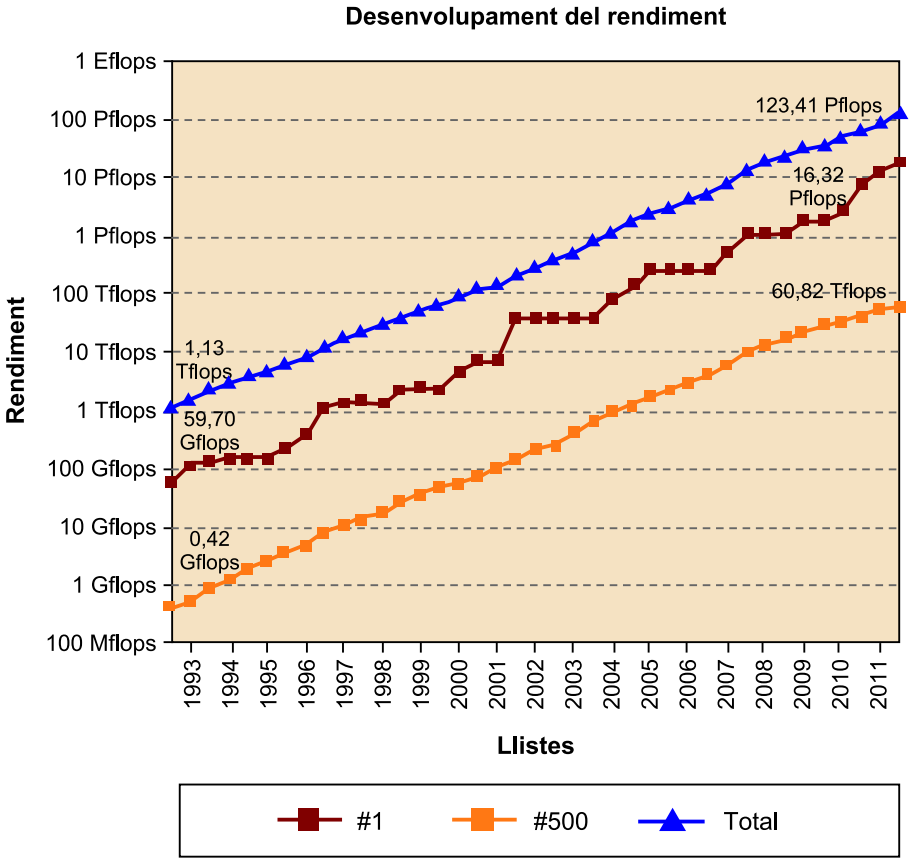
Figura 21. Evolució del tipus d'arquitectures dels sistemes del Top500 des de 1993
Font: http://www.top500.org




La figura següent mostra l'evolució del rendiment dels supercomputadors del Top500 des del 1993 fins al juny del 2012. L'eix de les *Y* està escalat en termes de rendiment sostingut mesurat en Gflops, Tflops i Pflops. La sèrie del mig correspon al rendiment del supercomputador més ràpid del món durant les gairebé dues dècades, el qual s'incrementa de 58,7 Gflops a 16,32 Pflops. La sèrie inferior correspon a la velocitat de l'últim sistema del Top500 (en la posició número 500) i la superior correspon a la suma del rendiment dels 500 supercomputadors d'un mateix període. El juny del 2012 la suma dels 500 supercomputadors suma 123,41 Pflops. És interessant observar que la suma total de rendiment augmenta de manera pràcticament lineal.

Figura 22. Evolució del rendiment dels supercomputadors del Top500 des de 1993

Font: http://www.top500.org




A la taula següent es mostren les característiques dels deu supercomputadors més ràpids en la llista del Top500 del juny del 2012 juntament amb el rendiment que tenen. La velocitat sostinguda, Rmax (en Pflops), es mesura a partir de l'execució del *benchmark* Linpack amb la màxima mida de matriu. Rpeak és la velocitat màxima sostinguda quan tots els elements del sistema són utilitzats completament. La potència elèctrica és en kW, la qual cosa vol dir que el supercomputador de la llista que requereix més potència elèctrica passa els 12 MW. Cal destacar que l'ordre dels sistemes quant al rendiment que tenen no coincideix amb el de nombre de nuclis o potència.

Taula 5. Llista dels 10 supercomputadors més ràpids de la llista del Top500 del juny de 2012

| # | Institució | Computador/Any Fabricant | Nuclis | Rmax/ Rpeak | Potència |
|---|---|---|---|---|---|
| 1 | DOE/NNSA/LLNL<br>Estats Units | Sequoia - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom / 2011<br>IBM | 1572864 | 16324/ 20132 | 7890 |
| 2 | RIKEN Advanced Institute for Computational Science (AICS)<br>Japó | K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect / 2011<br>Fujitsu | 705024 | 10510/ 11280 | 12659 |
| 3 | DOE/SC/Argonne National Laboratory<br>Estats Units | Mira - BlueGene/Q, Power BQC 16C 1.60GHz, Custom / 2012<br>IBM | 786432 | 8162/ 10066 | 3945 |
| 4 | Leibniz Rechenzentrum<br>Alemanya | SuperMUC - iDataPlex DX360M4, Xeon E5-2680 8C 2.70GHz, Infiniband FDR / 2012<br>IBM | 147456 | 2897/ 3185 | 3422 |

| # | Institució | Computador/Any Fabricant | Nuclis | Rmax/ Rpeak | Potència |
|---|---|---|---|---|---|
| 5 | National Supercomputing Center in Tianjin<br>Xina | Tianhe-1A - NUDT YH MPP, Xeon X5670 6C 2.93 GHz, NVIDIA 2050 / 2010<br>NUDT | 186368 | 2566/ 4701 | 4040 |
| 6 | DOE/SC/Oak Ridge National Laboratory<br>Estats Units | Jaguar - Cray XK6, Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA 2090 / 2009<br>Cray Inc. | 298592 | 1941/ 2627 | 5142 |
| 7 | CINECA<br>Itàlia | Fermi - BlueGene/Q, Power BQC 16C 1.60GHz, Custom / 2012<br>IBM | 163840 | 1725/ 2097 | 821 |
| 8 | Forschungszentrum Juelich (FZJ)<br>Alemanya | JuQUEEN - BlueGene/Q, Power BQC 16C 1.60GHz, Custom / 2012<br>IBM | 131072 | 1380/ 1677 | 657 |
| 9 | CEA/TGCC-GENCI<br>França | Curie thin nodes - Bullx B510, Xeon E5-2680 8C 2.700GHz, Infiniband QDR / 2012<br>Bull | 77184 | 1359/ 1667 | 2251 |
| 10 | National Supercomputing Centre in Shenzhen (NSCS)<br>Xina | Nebulae - Dawning TC3600 Blade System, Xeon X5650 6C 2.66GHz, Infiniband QDR, NVIDIA 2050 / 2010<br>Dawning | 120640 | 1271/ 2984 | 2580 |

També cal destacar altres iniciatives que hi estan relacionades, com ara el Graph500 i el Green500. El primer ordena els sistemes d'altes prestacions orientat a aplicacions intensives de dades ja que cada cop estan prenent més protagonisme en la computació d'altes prestacions, mentre que el segon se centra en l'eficiència energètica, tal com veurem en el darrer mòdul de l'assignatura.

**Vegeu també**

De l'eficiència energètica en tractem en el darrer mòdul d'aquesta assignatura.

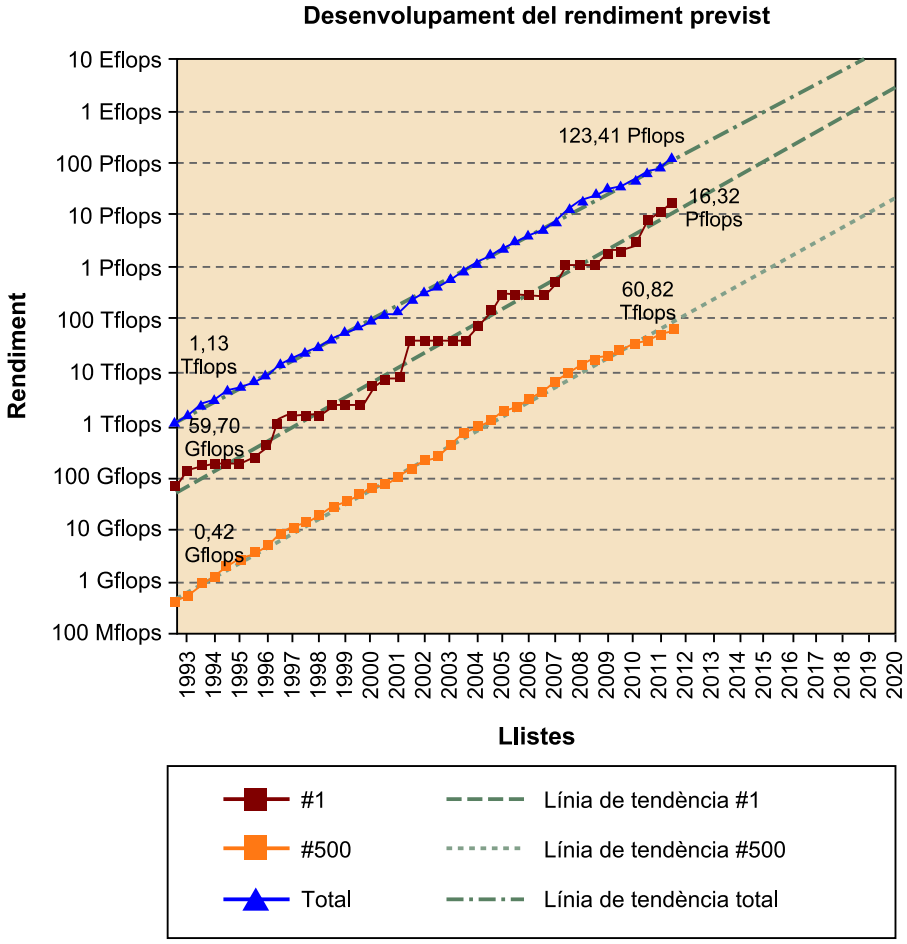
# 5. Reptes de la computació d'altes prestacions

Amb els avenços tecnològics el rendiment dels computadors d'altes prestacions millora any rere any, tal com hem vist a la figura 21. Tot i això, molts d'aquests avenços tecnològics (per exemple, l'augment de densitat de transistors en els microprocessadors) estan arribant al límit per diversos motius, com ara els límits físics o energètics, i cal un canvi de paradigma per a poder construir un computador que tingui un rendiment sostingut d'un exaflop.

L'objectiu actual de la comunitat científica és poder desenvolupar sistemes de l'escala de l'exaflop aproximadament el 2018. Aquest objectiu coincideix amb la projecció del rendiment dels supercomputadors amb vista al 2020, tal com mostra la figura següent, però tal com veurem a continuació hi ha nous reptes que caldrà prendre molt seriosament per a assolir l'exaflop. De fet, dins de les múltiples iniciatives en aquesta direcció cal destacar el projecte The International Exascale Software, en el qual científics d'arreu del món han definit un full de ruta per aconseguir aquesta escala de rendiment de manera sostenible. El projecte té el suport d'institucions, governs i empreses d'arreu del món, especialment de l'NSF[41] i del Departament d'Energia dels Estats Units, i agències a Europa i Àsia.

[41] Sigles de la National Science Foundation.

Figura 23. Projecció del rendiment dels supercomputadors del Top500 fins al 2020




Font: http://www.top500.org

## 5.1. Concurrència extrema

Per a aconseguir un rendiment d'exaflop caldrà augmentar el nivell de concurrència fins a 1.000 vegades més per cada treball individual. Això vol dir que els treballs estaran compostos possiblement de centenars de milers o milions d'elements que cal sincronitzar i gestionar adequadament i de manera eficient.

D'aquesta manera, caldrà disposar de nous models de programació que donin resposta a la necessitat de tant nivell de paral·lelisme i que, a més, facilitin que grups d'aplicacions pugin gestionar la concurrència corresponent de manera més natural. Aquesta capacitat segurament haurà d'incloure una escalabilitat forta, ja que l'augment del volum de memòria principal no coincidirà amb el dels processadors. Això també voldrà dir que caldrà reduir al mínim el sobrecost relacionat amb les capes de programari que seran necessàries.

## 5.2. Energia

L'eficiència energètica és una de les prioritats actuals dels professionals que treballen en el disseny dels futurs sistemes d'exaflop que, si es desenvolupés amb tecnologia actual, consumiria uns quants centenars de MW de potència

elèctrica, cosa que no és viable ni sostenible des de diverses perspectives. Així, doncs, cal una reducció de l'energia de diverses ordres de magnitud i això ha d'anar acompanyat d'optimitzacions a diferents escales, juntament amb nous algoritmes i dissenys d'aplicacions.

Amb l'eficiència energètica com a prioritat, s'espera que cap al 2017 els computadors arribaran a 200 petaflops amb un límit de consum energètic de 10 MW. Amb vista al 2020 com a màxim, el repte és arribar a 1 exaflop amb un consum de 20 MW. Això comportaria una eficiència energètica vint vegades superior a la de les màquines que consumeixen menys actualment –per exemple, BlueGene/Q (vegeu la taula 5).

Primer cal tenir en compte que no tota l'energia es destina a alimentar els nuclis. En els sistemes actuals, els processadors necessiten un gran volum d'energia, sovint un 40% o fins i tot més. L'energia restant s'utilitza per a les memòries, la xarxa d'interconnexió i el sistema d'emmagatzemament. Aquestes proporcions estan canviant i la memòria principal cada cop consumeix una proporció més gran d'energia.

També cal tenir en compte que l'energia que es requereix no solament és per a còmput, sinó que també és per a transportar per la xarxa i analitzar les dades que es poden anar generant durant simulacions a escales extremes. Com que una part important de l'energia requerida per a un sistema a escala d'exaflop s'haurà de destinar a moure les dades entre processadors i memòria i de manera més global, caldrà tenir la infraestructura de programari que ho pugui gestionar eficaçment.

**Vegeu també**

Recordeu que d'aquestes qüestions de l'eficiència energètica en tractarem en el darrer mòdul d'aquesta assignatura.

### 5.3. Tolerància a fallades

Els futurs sistemes d'exaflop estaran construïts amb dispositius VLSI que no seran tan fiables com els que s'utilitzen actualment. Tot el programari, i per tant totes les aplicacions, hauran de tenir en compte la tolerància a fallades com un factor més en el disseny. Així, doncs, quan l'escala del sistema augmenti fins a l'exaflop haurem de ser compatibles amb el reconeixement i adaptació a errors de manera contínua i proporcionar els mecanismes necessaris perquè les aplicacions facin el mateix. De fet, el paral·lelisme massiu i l'elevat nombre de components electrònics d'aquests sistemes farà que les fallades siguin inevitables i també que hi hagi una certa incertesa que s'haurà de tenir en compte com a element en el disseny d'aquestes futures aplicacions.

## 5.4. Heterogeneïtat

Els sistemes heterogenis ofereixen l'oportunitat d'explotar el rendiment de dispositius, com ara GPU per a computació de propòsit general. Un exemple d'això és el computador Nebulae, el qual utilitza CPU Intel Xeon i acceleradors Nvidia. De la mateixa manera, les aplicacions científiques s'estan tornant més heterogènies.

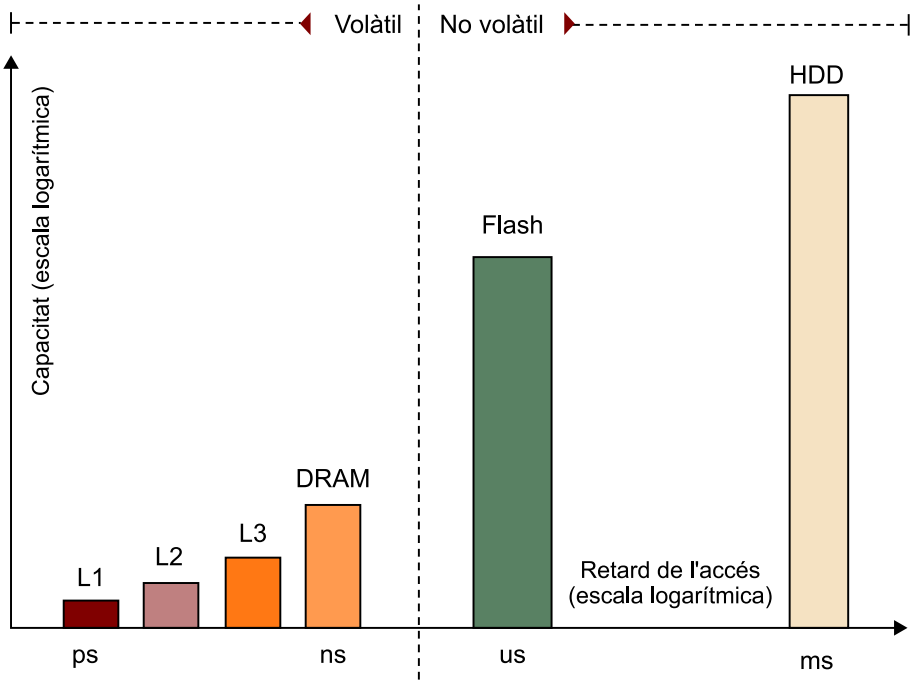
## 5.5. Entrada/sortida i memòria

No disposar de prou capacitat d'entrada/sortida avui en dia és un coll d'ampolla. A més, caldrà gestionar i analitzar grans volums de dades que es generaran molt ràpidament, cosa que augmentarà vertiginosament durant la pròxima dècada.

La jerarquia de memòria haurà de canviar, ja que hi haurà noves tecnologies de memòria que caldrà tenir en compte. Aquest canvi en la jerarquia de memòria acabarà afectant tant els models de programació com les optimitzacions. La figura següent mostra la diferència d'accés als diversos nivells de la jerarquia de memòria com a motivació per a la incorporació de noves tecnologies, com ara la memòria no volàtil o NVRAM[42].

(42)De l'anglès *non-volatile random-access memory*.

Figura 24. Diferències en temps d'accés als diversos nivells de la jerarquia de memòria
Font: Fusion-IO




Cal tenir en compte que els dos eixos de la figura anterior estan en escala logarítmica. Així, doncs, podem veure clarament que hi ha una diferència molt elevada tant de rendiment com de temps d'accés entre la memòria DRAM i el disc dur. A la taula 6 es presenta una analogia força il·lustrativa.

Taula 6. Analogia dels temps d'accés al diferents nivells de la jerarquia de memòria

| Nivell | Menjar | Temps d'accés relatiu |
|---|---|---|
| Cau nivell 1 | Menjar de la boca | Fraccions de segon |
| Cau nivell 2 | Agafar menjar del plat | 1 segon |
| Cau nivell 3 | Agafar menjar de la taula | Uns pocs segons |
| DRAM | Agafar menjar de la cuina | Uns pocs minuts |
| FLASH | Agafar el menjar d'una botiga propera | Unes poques hores |
| Disc dur | Agafar el menjar de Mart | 3-5 anys |

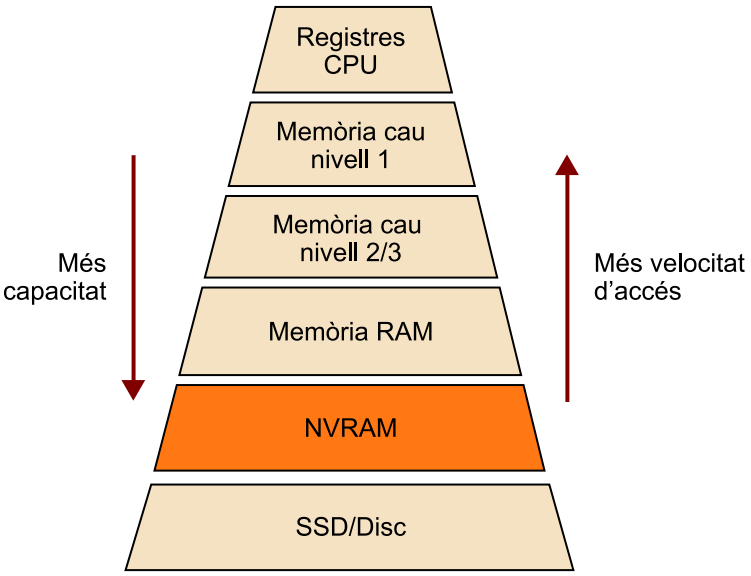
Tenint en compte el que hem exposat, una de les solucions que s'estan adoptant i que passaran a tenir un paper destacat en el desenvolupament de sistemes d'exaflop és estendre la jerarquia de memòria amb un nou nivell que ocuparà la memòria no volàtil, com mostra la figura següent, la qual estén els conceptes que han sortit a la figura 12. També s'espera la utilització d'altres tecnologies que proporcionen rendiments intermedis com ara la utilització de tecnologia SSD[43].

[43] De l'anglès *solid state drive*.

**Vegeu també**

Trobareu la figura 12 al subapartat 2.2.2, en traçar els sistemes de memòria compartida.

Figura 25. Jerarquia de memòria estesa amb memòria RAM no volàtil



# Bibliografia

**Buyya, R.** (1999). *High Performance Cluster Computing: Architectures and Systems* (vol. 1). Englewood Cliffs, NJ: Prentice Hall.

**Grama, A. i altres** (2003). *Introduction to Parallel Computing*. Reading, Massachusetts: Addison-Wesley.

**Hwang, K.; Fox, G. C.; Dongarra, J. J.** (2012). *Distributed and Cloud Computing: From Parallel Processing to the Internet of Things*. Burlington, Massachusetts: Morgan Kaufmann.

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

**Kirk, D. B.; Hwu, W. W.** (2010). *Programming Massively Parallel Processors: A Hands-on Approach*. Burlington, Massachusetts: Morgan Kaufmann.

**Lin, C.; Snyder, L.** (2008). *Principles of Parallel Programming*. Reading, Massachusetts: Addison-Wesley.

**Munshi, A. i altres** (2011). *OpenCL Programming Guide*. Reading, Massachusetts: Addison-Wesley Professional.

**Pacheco, P.** (2011). *An Introduction to Parallel Programming*. Burlington, Massachusetts: Morgan Kaufmann.