

PAR



Paral·lelisme • 2025-26 QP • RogeR Bitlloch

Teoria Pre-parcial

T1. Introducció al Paral·lelisme

T2. Mesurar el paral·lelisme

T3. Descomposició de tasques

Basat en les diapositives de la FIB

Índex

Avaluació Paral·lelisme	4
Calendari del curs	4
T1. Introducció al Paral·lelisme	5
1.1 Inici Paral·lelisme	5
1.2 Models d'execució	5
1.3 Elements de l'execució paral·lela	6
1.4 Execució seqüencial vs execució paral·lela	6
1.5 Problemes del paral·lelisme	6
1.5.1 Dependències	6
1.5.2 Concurrencia	7
1.6 Abstracció del codi en tasques	8
1.7 Consolidació T1 - Introducció al Paral·lelisme	8
T2. Mesurar el paral·lelisme	9
2.1 Construcció d'un Task Dependence Graph (TDG)	9
2.2 Fracció paral·lelitzable i llei d'Amdahl	10
2.3 Scalability	11
2.4 Overheads	12
2.5 Granularitat	13
2.6 Memòria distribuïda i cost de compartir dades	14
2.7 Consolidació T2 — Mesurar el paral·lelisme	15
T3. Descomposició de tasques	16
3.1 Regió paral·lela (<code>parallel</code>)	16
3.2 Models d'execució	17
3.2.1 Tasques implícites — paral·lelisme per threads	17
3.2.2 Tasques explícites — paral·lelisme per tasques	17
3.3 Instruccions bàsiques d'OpenMP	18
3.4 Dependències en OpenMP (<code>depend</code>)	18
3.5 Privacitat de variables en OpenMP	19
3.6 Protegir problemes de concurrència (Data Race)	20
3.7 Descomposició iterativa de bucles	22
3.7.1 Amb tasques implícites	22
3.7.2 Amb tasques explícites	23
3.8 Recursive Task Decomposition	24
3.8.1 Leaf Strategy	24
3.8.2 Leaf amb Cut-off	25
3.8.3 Tree Strategy	26
3.8.4 Tree amb Cut-off	27
3.8.5 Control de cut-off amb <code>final</code> i <code>omp_in_final</code>	27
3.9 Consolidació T3 — Descomposició de tasques	28

Avaluació Paral·lelisme

- **Teoria:** 65% – Parcial (P), Final Part 1 (FP1) i Final Part 2 (FP2)
 - **Part 1** (T1, T2 i T3): 32,5% – $\max(P, FP1)$ si $P \geq 5$, sinó $FP1$
 1. Task Dependence
 - (a) Dibuixar TDG
 - (b) Calcular T_1 , T_∞ i $P_{\min}$
 - (c) Calcular T_P + diagrama amb datasharing
 2. Paral·lelitzar bucle amb tasques explícites o implícites
 3. Paral·lelitzar programa recursiu amb cutoff
 - **Part 2** (T4 i T5): 32,5% – FP2
 4. Numa Nodes (nombre de bits / taula) i Aplicar Padding
 5. Paral·lelitzar programa Geometric Data Decomposition
- **Laboratori:** 35% = $\max(SL, FL)$ si $FL > 4$, sinó $0.35 \cdot SL + 0.65 \cdot FL$.
 - **Informes Lab** (SL): OBLIGATORIS ($0.3 \cdot Lab3 + 0.5 \cdot Lab4 + 0.2 \cdot Lab5$)
 - **Final Lab** (FL): Mateix dia del final (pots portar chuletari fet a mà)

Si aprobeu l'assignatura i FP2, us pugem fins a un 10% de la nota fent qüestionaris d'Atenea!!!

Calendari del curs

Dilluns 16h15 - 17h45		Contingut
S1	16 Feb	Introducció + T1 (Introducció al Paral·lelisme)
S2	23 Feb	T2 (Mesurar el paral·lelisme)
S3	2 Març	T3.1 #pragma OMP implícites vs explícites
S4	9 Març	T3.2 Iterative Task Decomposition
S5	16 Març	T3.3 Recursive Task Decomposition
S6	23 Març	LAB + T3.4 Barriers i Depends
	30 Març	NO HI HA CLASSE - Setmana Santa
I1		T1 i T2 + Exercicis TDG
I2		T3.1 (Iterative i Reductions)
I3		T3.2 (Recursive Leaf i Tree)
I4		Examen Pràctica 2024-25Q1
I5		Examen Pràctica 2024-25Q2
13 Abril: PARCIAL (32,5%)		

1 Introducció al Parallelisme

1.1 Inici Parallelisme

Després de molts anys seguint la llei de Moore, va començar a arribar un punt on, malgrat augmentar els esforços per seguir millorant el hardware, no ens donaven un tant bon benefici. Per tant, cal buscar alternatives en el **model d'execució** per millorar l'eficiència.

1.2 Models d'execució

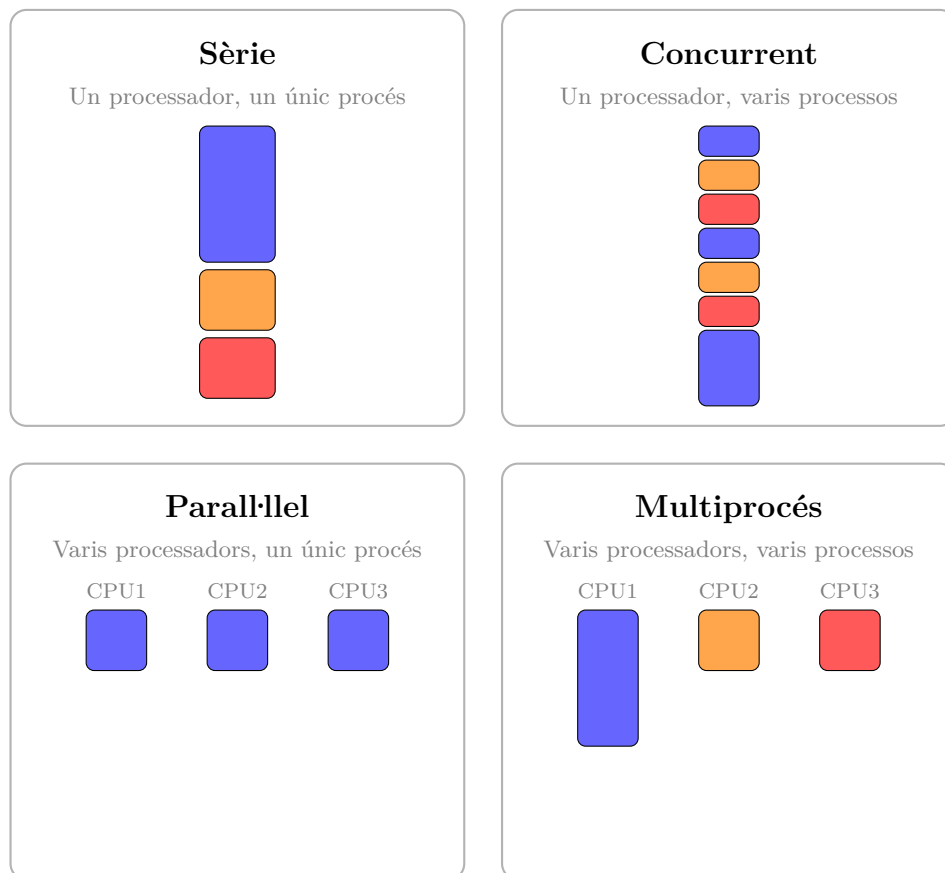
Execució seqüencial: Una sola CPU executa un **únic programa** de principi a fi, seguint estrictament l'ordre de les seves instruccions.

Execució en sèrie: Una sola CPU executa **diversos programes** un rere l'altre, sense solapament en el temps.

Execució concurrent: Una sola CPU alterna l'execució de **diferents processos** mitjançant *time-slicing* (commutació de context). En cada instant només s'executa una tasca, però totes progressen de manera intercalada, donant així una sensació de rendiment (*throughput*).

Execució paral·lela: Un **únic procés** es divideix en tasques que es poden executar simultàniament en diferents nuclis o CPUs. Això ens permet reduir el temps d'execució d'un sol procés. Tot dividint la càrrega de treball entre diferents CPUs, aconseguint més rendiment per a un únic programa.

Execució multiprocés: Diversos **processos independents** s'executen en paral·lel utilitzant múltiples CPUs, millorant el rendiment global del sistema.



Cal garantir que el resultat SEMPRE sigui el MATEIX independentment del model d'execució.

1.3 Elements de l'execució paral·lela

Programa / aplicació: és el codi escrit i emmagatzemat en memòria. Quan el sistema operatiu el posi en marxa, es convertirà en un **procés**.

Procés: programa en execució.

Cada procés té el seu propi **espai de memòria** i s'executa sobre la CPU.

Thread (hilo/fil d'execució): és una **línia d'execució** dins d'un procés.

Els threads d'un mateix procés **comparteixen el mateix codi i la mateixa memòria**, però cada thread manté la seva **pila i registres propis**.

Processador / CPU / core: unitat física capaç d'executar instruccions.

Per aquesta assignatura, quan parlem de *processador*, *CPU* o *core* ens referirem a una **unitat d'execució independent**.

1.4 Execució seqüencial vs execució paral·lela

En **execució seqüencial**, un procés avança amb un únic **thread**: hi ha un sol flux d'instruccions i, en cada instant, la feina recau sobre una única CPU.

En canvi, en l'**execució paral·lela**, el **mateix únic procés** es divideix en diversos **threads** que poden executar-se alhora en diferents CPUs. Com que tots els threads formen part del mateix procés, **comparteixen el mateix codi i el mateix espai de memòria**. Però cada thread conserva **pila i registres propis**. Al tenir varis threads podem usar múltiples processadors alhora, per executar el codi de forma simultània.

Això ens permet dividir el temps d'execució fins a: $T_P (\text{Ideal}) = T_1 / n_{\text{processadors}}$

Aquest cas és idíl·lic: Pressuposa que som capaços de fer un repartiment perfecte de càrrega de treball entre tots els processadors, **sense dependències** ni **sense overheads**.

1.5 Problemes del paral·lelisme

La majoria de programes estan pensats per executar-se de forma **seqüencial**. Per tant, quan els paral·lelitzem ens hem d'assegurar que el resultat es mantingui igual!

1.5.1 Dependències

Quan executem varies parts d'un mateix programa alhora no podem conèixer al 100% quin serà l'ordre d'execució de les seves parts. Això ens pot causar problemes si una part del codi **necessita** el resultat d'una altra.

Per exemple, si una instrucció llegeix **X** abans que una altra l'hagi inicialitzat o actualitzat, podem obtenir un valor incorrecte.

CPU 1	CPU 2
X = 10;	
	X = X + 2;
print(X);	

Si no imposem cap ordre, **X = X + 2** podria executar-se abans de **X = 10**, o el **print(X)** podria passar abans de l'increment, i per tant el nostre programa al paral·lelitzar-lo no tindria el funcionament esperat i seria incorrecte.

Per evitar-ho, caldrà **establir dependències** i obligar que certes instruccions s'executin **abans** que d'altres, segons les dades que accedeixin o modifiquin.

1.5.2 Concurrencia

Data race condition: Dos threads comparteixen una mateixa adreça de memòria (per exemple una variable) i la intenten **modificar/utilitzar alhora**. Això pot fer que el resultat variï entre execucions.

CPU 1	CPU 2
<code>tmp = x;</code>	<code>tmp = x;</code>
<code>tmp = tmp + 1;</code>	<code>tmp = tmp + 1;</code>
<code>x = tmp;</code>	<code>x = tmp;</code>

Si x comença valguent 0, l'esperat seria acabar amb $x = 2$. Però si tots dos threads llegeixen $x = 0$ abans que cap escriu, tots dos acabaran escrivint 1 i el resultat final serà $x = 1$.

- Bloquejar la dada mentre l'estiguem utilitzant. (Semàfor)

Starvation: Un thread no pot accedir a un recurs compartit, perquè un altre thread el té reservat molta estona, perdent parallelisme potencial.

CPU 1	CPU 2
<code>for (...) {</code>	<code>for (...) {</code>
<code> lock(m);</code>	<code> lock(m);</code>
<code> feina_llarga();</code>	<code> // sovint queda esperant</code>
<code> unlock(m);</code>	<code> feina_curta();</code>
<code>}</code>	<code> unlock(m);</code>
	<code>}</code>

- Repartir millor la càrrega de treball entre threads (no hi ha una “solució” única i general).

Deadlock: Dues tasques esperen mútuament que l'altra alliberi l'element que ella té i es crea un **bloqueig cíclic**; el programa queda en espera infinita.

CPU 1	CPU 2
<code>lock(A);</code>	<code>lock(B);</code>
<code>lock(B); // espera B</code>	<code>lock(A); // espera A</code>
<code>// NO avança</code>	<code>// NO avança</code>

- Imposar un **ordre** en l'adquisició de bloquejos per evitar esperes circulars.

Livelock: Variant del deadlock: les tasques no queden “aturades” esperant, sinó que intenten evitar el bloqueig canviant d'estat, però continuen bloquejant-se mútuament.

CPU 1	CPU 2
<code>lock(A);</code>	<code>lock(B);</code>
<code>trylock(B); // falla</code>	<code>trylock(A); // falla</code>
<code>unlock(A); // cedeix</code>	<code>unlock(B); // cedeix</code>
<code>lock(A); // reintenta</code>	<code>lock(B); // reintenta</code>
<code>...</code>	<code>...</code>

- Imposar ordre també als **desbloquejos** i/o aplicar **backoff** per evitar que en reintentar-ho es repeteixi el patró.

1.6 Abstracció del codi en tasques

Per evitar aquests problemes a l'hora de paral·lelitzar, és clau entendre el codi i identificar quines parts es poden executar en paral·lel i quines han d'executar-se abans que d'altres.

Per fer-ho, abstraurem el programa en **tasques**:

Una **tasca** és una unitat lògica de treball dins del programa. Pot correspondre a una funció, un bucle o un fragment de codi que es pugui executar de manera independent sense modificar dades que no li pertanyen.

Aquesta abstracció en tasques definirà l'estratègia de paral·lelització i ens permetrà separar la lògica global del programa en parts manejables per analitzar:

- el temps d'execució de cada tasca,
- les dependències,
- i els possibles conflictes (accés concurrent a dades compartides).

1.7 Consolidació T1 - Introducció al Paral·lelisme

Això és tot el que hauries de saber del Tema 1:

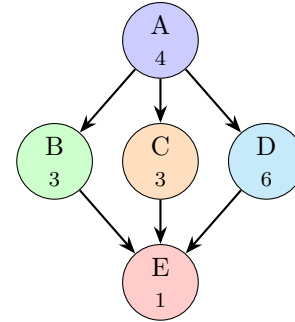
- D'on sorgeix el paral·lelisme.
- Diferència entre execucions: **sèrie**, **concurrent**, **paral·lel** i **multiprocés**.
- Principals problemes del paral·lelisme (dependències i conflictes).
- Diferenciar entre:
 - **Procés**: aïllat, espai de memòria propi.
 - **Thread**: comparteix codi i espai de memòria, però no registres i pila.
 - **Processador/CPU/core**: unitat de hardware física que executa threads.
 - **Tasques**: unitat de treball/computació que es pot executar de manera simultània dins d'un programa.

2 Mesurar el parallelisme

2.1 Construcció d'un Task Dependence Graph (TDG)

Quan representem les tasques del nostre programa com a **nodes** i les dependències de dades entre tasques com a **arestes**, obtenim un **Task Dependence Graph (TDG)**.

- Cada **node** representa una tasca.
- Les tasques indiquen el **temps d'execució**.
El temps de cada node ens ajudarà a detectar **colls d'ampolla** i a fer una millor repartició de la càrrega de treball.
- Les **arestes** indiquen **dependències de dades**.
Una tasca no pot començar fins que les tasques predecessores han acabat.
- Cada CPU només pot executar **una tasca** alhora.



Exemple d'un TDG

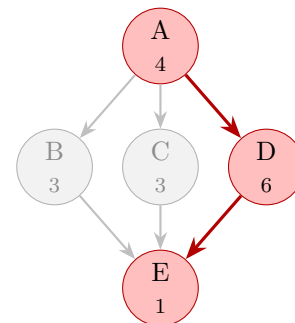
Un cop construït el TDG, el podem utilitzar per analitzar fàcilment l'estratègia de paral·lelització:

T_P : temps d'execució si usem P processadors.
(respectant dependències)

T_1 : temps d'execució seqüencial.

Per calcular-lo sumem els temps d'execució de **totes** les tasques.

T_∞ : temps d'execució mínim que podem aconseguir amb aquesta estratègia. Per calcular-lo mirarem la durada del **camí crític** del TDG.



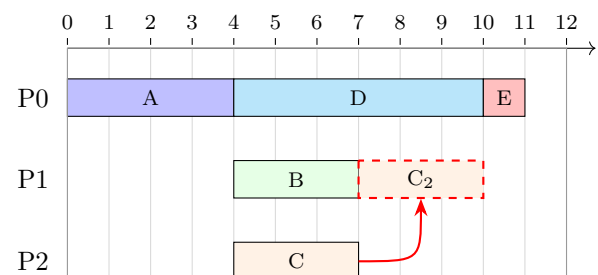
Camí crític

Paral·lelisme ideal/màxim: $\frac{T_1}{T_\infty}$ Major speedUp que podem obtenir amb aquesta estratègia.

$P_{\min}$: És el nombre mínim de processadors necessaris per assolir el **paral·lelisme ideal** (és a dir, executar el programa en T_∞)

Per trobar $P_{\min}$, distribueix les tasques en una línia de temps entre varis processadors:

1. Assigna el **camí crític** a un processador (fixa el temps mínim possible, T_∞).
2. Reparteix la resta de tasques en altres processadors, respectant dependències i sense excedir T_∞ .
3. Si no hi caben, incrementa P i repeteix. Si en canvi pots compactar tasques i alliberar algun processador decrementa P .



Planificador de 3 Processadors

Quan totes les tasques caben dins T_∞ , el valor de P és $P_{\min}$.

2.2 Fracció paral·lelitzable i llei d'Amdahl

Separarem el temps d'execució en dues parts:

T_{seq} : temps no paral·lelitzable (seqüencial)

T_{par} : temps paral·lelitzable

$$T = T_{\text{seq}} + T_{\text{par}}.$$

Definim la fracció paral·lelitzable del programa com:

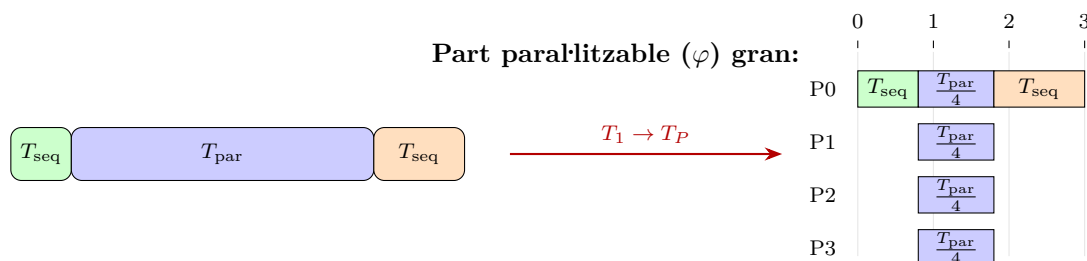
$$\varphi = \frac{T_{\text{par}}}{T_1} \quad \Rightarrow \quad 1 - \varphi = \frac{T_{\text{seq}}}{T_1}.$$

Intuïtivament, φ mesura quina part del codi *podria* beneficiar-se del paral·lelisme.

$1 - \varphi$ en canvi ens expressa quina fracció del programa ens frena d'obtenir un millor rendiment.

Model ideal amb P processadors: Si suposem un repartiment perfecte de la part paral·lelitzable (sense desbalanceig ni costos addicionals):

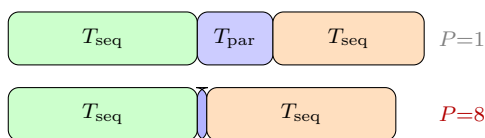
$$T_P = T_{\text{seq}} + \frac{T_{\text{par}}}{P} = (1 - \varphi) T_1 + \varphi \frac{T_1}{P}.$$



Llei d'Amdahl: El guany del paral·lelisme està limitat per la part **seqüencial** del programa:

- Si φ és gran (quasi tot és paral·lelitzable), podem obtenir speedups elevats, però sempre acotats per $1 - \varphi$.
- Si φ és petita (part seqüencial gran), encara que augmentem molt P , el speedup quedarà limitat.

φ **petit** $\rightarrow T_{\text{seq}}$ **domina**

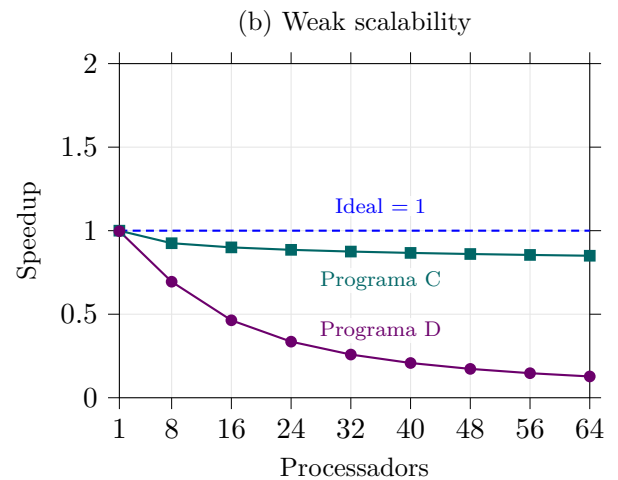
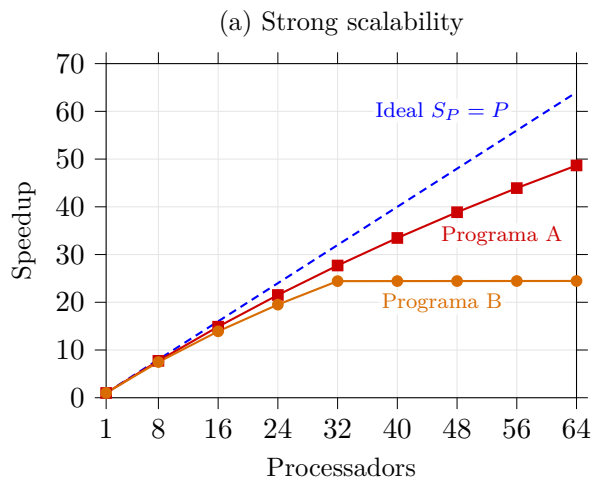


Idea clau: per millorar el rendiment, sovint és més rendible reduir T_{seq} (o augmentar φ fent més gran T_{par}) que no pas afegir processadors indefinidament. Ja que per molts processadors que tinguem si no tenen tasques a executar no serviran de res.

2.3 Scalability

L'**escalabilitat** (o *scalability*) és la capacitat d'un programa de seguir millorant el seu rendiment en augmentar els recursos. Mesurarem com varia l'*speedup* $S_P = \frac{T_1}{T_P}$ quan augmentem el nombre de processadors.

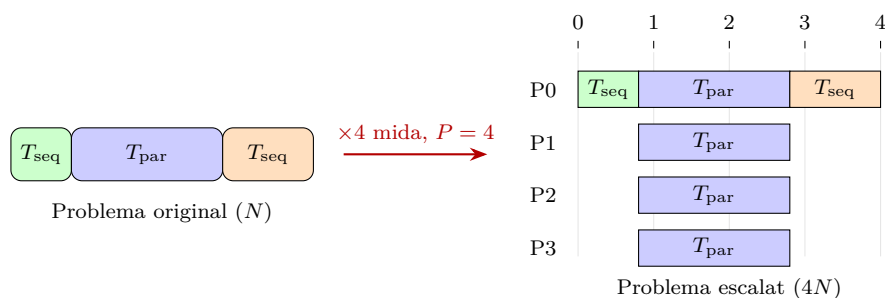
- Strong scalability:** en aquest model mantenim **fixa** la mida del problema i busquem disminuir T_P incrementant P . Com més proper estigui el speedup a l'ideal $S_P = P$, millor **strong scalability** tindrà la nostra estratègia.
- Weak scalability:** aquí no busquem reduir el temps d'execució, sinó mantenir-lo mentre augmentem la mida del problema proporcionalment a P . En el gràfic, l'ideal és mantenir-se a 1. Com més a prop d'1 es mantingui la corba, millor **weak scalability**.



En **strong scaling**, el programa A manté un comportament proper a l'ideal, mentre que el programa B es veu que es comença a estancar a partir de 32 processadors (limitacions per part seqüencial i/o overheads), reduint el guany per afegir més processadors.

En **weak scaling**, s'observa com l'overhead de comunicació i sincronització augmenta amb P , fent que el rendiment es degradi progressivament en el programa D mentre el programa C es manté més a prop de l'ideal.

Llei de Gustafson: Encara que la strong scalability sigui dolenta (i, per tant, una part important no sigui paral·lelitzable), podem aplicar la llei de Gustafson, que contradiu la llei d'Amdahl. Augmentar la mida del problema, per fer créixer la part paral·lelitzable, i així poder treure profit de la weak scalability.



Al quadruplicar la mida del problema i usar $P = 4$ processadors, cada processador realitza la mateixa feina que l'únic processador original, mantenint $T_P \approx T_1$.

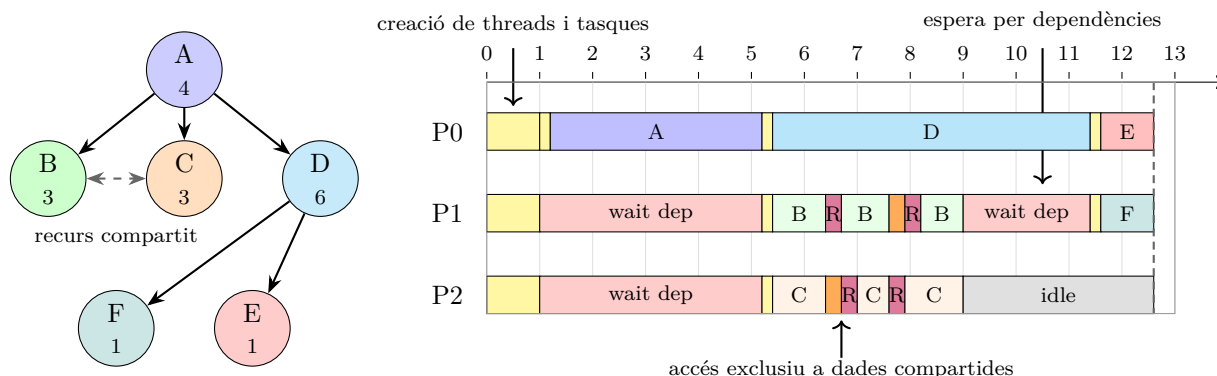
2.4 Overheads

Parallelitzar un programa no és perfecte. Fins i tot si un codi és totalment parallelitzable, en executar-lo sobre P processadors no obtindrem T_1/P :

$$T_P \neq \frac{T_1}{P} \quad \Rightarrow \quad T_P = \frac{T_1}{P} + T_{\text{overheads}}$$

Els **overheads** són costos addicionals que apareixen per coordinar l'execució en paral·lel.

- **Gestió de threads i tasques:** El simple fet de parallelitzar un programa implica crear i mantenir threads i gestionar tasques (encuar, planificar, repartir i sincronitzar...). Aquesta feina es fa en temps d'execució i consumeix temps de CPU, de manera que es dedica a fer gestions i no a càlcul útil del programa.
- **Accés a recursos compartits:** com hem vist abans, en parallelitzar poden aparèixer problemes de concurrència (secció 1.5.2). Per evitar-los, sovint cal protegir les dades compartides amb mecanismes de sincronització (veure secció 3.6), fent que un thread hagi d'esperar que un altre acabi d'utilitzar la dada, perdent temps de comput.
- **Load imbalance:** Sovint alguns processadors acaben les tasques que tenen assignades abans que d'altres. Quan volen continuar, poden quedar-se inactius perquè no hi ha cap tasca disponible: o bé perquè la resta de feina encara està en execució en altres CPUs, o bé perquè cal esperar que es resolguin dependències abans de poder iniciar la següent tasca.
- **Altres pèrdues:** jerarquia de memòria, replicació de càlculs, enviaments de dades, etc.



Podem veure diversos **overheads** que fan créixer T_P respecte l'ideal:

- **Gestió** (groc): petits trams al crear cada thread i també abans de crear cada tasca.
- **Espera per dependències** (*wait dep*, rosa): al principi, P1 i P2 esperen fins que acaba A; més endavant, tornen a esperar fins que D es resol (*dep resolta*) per poder executar E i F.
- **Accés a recurs compartit** entre B i C (trams R): les barres de B i C apareixen fragmentades perquè alternen càlcul amb seccions d'accés exclusiu, i això fa que s'esperin quan el recurs és ocupat per l'altra tasca.
- **Idleness** (gris): després d'acabar C, P2 queda inactiu fins al final perquè no té cap tasca disponible a executar, és un exemple de temps perdut per desbalanceig de càrrega.

Cal anar amb molt de compte: com més augmentem el nombre de tasques i el nombre de processadors, més grans poden esdevenir els costos d'overhead. Pot arribar a passar que el programa dediqui més temps a crear i gestionar tasques que a executar-les. Amb certes estratègies, utilitzar més processadors i crear més tasques, **no sempre** equival a millor rendiment.

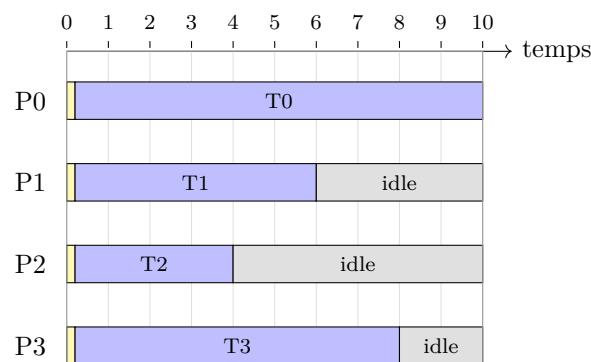
2.5 Granularitat

Un cop hem identificat una part del programa que podem paral·lelitzar, el pas següent és decidir en **quantes tasques** la dividim (granularitat). Aquesta decisió afecta directament el rendiment, perquè l'eficiència depèn del speedup i del nombre de processadors:

$$E_P = \frac{S_P}{P}.$$

- **Granularitat gruixuda → tasques grans.**

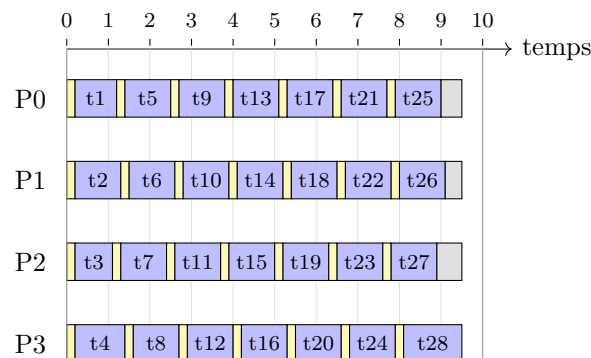
Si les tasques són molt grans, n'hi haurà poques i cadascuna tindrà molta càrrega de treball. Això redueix el cost de gestió (menys tasques a crear i planificar), però incrementa el risc de **load imbalance**: si una tasca és significativament més llarga, fixa el T_P i la resta de processadors poden quedar-se inactius (*idleness*) fins que acabi.



T_P queda determinat per la tasca més llarga: els processadors que acaben abans no poden contribuir i es queden esperant (*idleness*).

- **Granularitat fina → tasques petites.**

Si fem les tasques molt petites, n'hi haurà moltes i per tant la feina quedarà molt més repartida. Quan un thread acaba, pot continuar amb una altra tasca disponible, reduint el desbalanceig i l'*idleness*, però pagant un cost addicional que en crear i planificar moltes tasques, creixen els **overheads de gestió**.



Repartiment més homogeni i amb menys *idleness*, però amb un cost de gestió més gran, degut als trans d'overhead (groc) per la creació de tasques.

Idea clau: Cal trobar un compromís: les tasques han de ser prou petites perquè tots els threads es puguin repartir la càrrega de treball, però amb tasques prou grans per mantenir els costos d'overheads sota control.

2.6 Memòria distribuïda i cost de compartir dades

En un mateix procés, tots els *threads* comparteixen el mateix espai d'adreces (memòria compartida). Tot i això, a nivell de maquinari cada processador P_i té la seva pròpia memòria/caches i les dades no sempre són **locals** al nucli on s'executa el thread. Quan cal accedir a dades que no estan disponibles localment, s'han de transferir i això té un cost.

$$t_{\text{exe}} = t_{\text{comput}} + t_{\text{transferencies}}$$

Model simplificat:

Considerem un model senzill on cada processador P_i té la seva pròpia memòria i tots estan interconnectats mitjançant una xarxa d'interconnexió:

- **Accés local:** els processadors accedeixen a dades locals (a la seva pròpia memòria) amb instruccions normals de *load/store*. En aquest model idealitzat, aquests accessos locals tenen cost 0.
- **Accés remot:** si un processador necessita dades que estan a la memòria d'un altre, utilitza un model de *message passing*: el processador **demana** les dades i l'altre processador les **serveix**. En aquesta assignatura assumirem un temps d'accés remot que seguirà un model lineal:

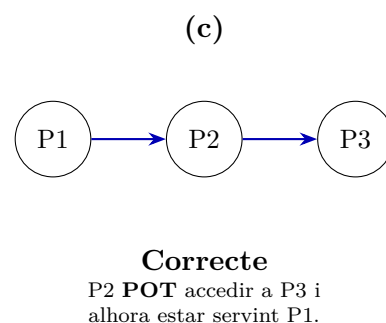
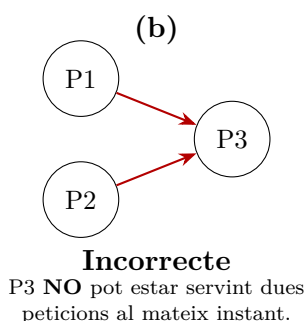
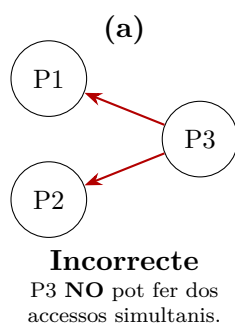
$$t_{\text{accés}} = t_s + m \cdot t_w$$

t_s és el cost fix de preparar la comunicació (*startup*)

$m \cdot t_w$ és el cost de transferència (enviar m bytes amb cost temporal t_w per byte).

Simplificacions del model: Per fer un model més tractable, també assumirem que:

- En un instant, un processador P_i només pot fer **un** accés remot a un altre P_j ,
- En un instant, un processador P_i només pot **servir un** accés remot d'un altre P_k ,
- Ambdues accions poden passar simultàniament: $P_i \rightarrow P_j$ i $P_i \leftarrow P_k$.



Aquestes hipòtesis no descriuen tots els detalls del maquinari real, però són suficients per raonar sobre com el cost de compartir dades i la sincronització poden limitar l'escalabilitat.

2.7 Consolidació T2 — Mesurar el paral·lelisme

Del tema 2 hauries de saber:

- Construir un TDG a partir d'un codi.
- Calcular T_1 , T_∞ , T_1/T_∞ (paral·lelisme ideal), $P_{\min}$ i T_P .
- Conèixer la **lleï d'Amdahl i Gustafson** i el paral·lelisme potencial.
- Conèixer **strong** i **weak** scalability a partir de l'speedup.
- Conèixer els diferents **overheads** que pot provocar el programa.
- **Granularitat** de tasques vs nombre de tasques:
 - Granularitat gruixuda $\rightarrow$ load imbalance.
 - Granularitat fina $\rightarrow$ overheads.
- Saber calcular el temps de comunicació de dades entre processadors.

3 Descomposició de tasques

Fins ara ens hem centrat en *com ha de ser* una bona descomposició en tasques: mida, repartiment i estructura. Ara passem a la pràctica i veurem **com es creen i s'executen** aquestes tasques en codi. Ho farem amb **OpenMP**, una API estàndard per a programació paral·lela en memòria compartida (principalment en C/C++ i Fortran).

En OpenMP, el punt de partida és la **regió paral·lela**: un bloc que crea el conjunt de threads que compartiran la càrrega de treball i dins del qual es generen i s'executen les unitats de treball.

3.1 Regió paral·lela (`parallel`)

La directiva `#pragma omp parallel` delimita una regió que s'executa amb un **team** de N threads. Un d'ells és el *master* (id0) i la resta actuen com a *workers*. Tots els threads **comparteixen el mateix espai de memòria**, i el runtime d'OpenMP s'encarrega de crear el team i gestionar-ne l'execució.

El nombre de threads el decideix el runtime per defecte (normalment, un per core disponible), però es pot especificar explícitament amb la clàusula `num_threads`:

```
// codi seqüencial
#pragma omp parallel [num_threads(N)]
{
    ... // codi executat dins la regió paral·lela
} // barrera implícita: espera tots els threads
// codi seqüencial
```

La regió `parallel` incorpora una **barrera implícita al final**: cap thread pot sortir-ne fins que tots han acabat. Això garanteix que l'execució seqüencial no es reprèn mentre algun thread encara treballa dins del bloc.

3.2 Models d'execució

Un cop creat el team, OpenMP ofereix dos enfocaments per explotar el paral·lelisme: **paral·lelisme per threads** (tasques implícites) i **paral·lelisme per tasques** (tasques explícites).

3.2.1 Tasques implícites — paral·lelisme per threads

En crear el team, OpenMP assigna a **cada thread una tasca implícita** corresponent al cos del bloc `parallel`: tots els threads executen el mateix fragment de codi simultàniament. Per repartir la feina, podem consultar el nombre total de threads i l'identificador de cadascun, i així fer que cada thread processi una part diferent de les dades:

```
int nt = omp_get_num_threads(); // nombre total de threads del team
int id = omp_get_thread_num();  // id del thread actual (0 .. nt-1)

#pragma omp parallel num_threads(N)
{
    // cada thread executa aquest bloc com a tasca implícita
}
```

3.2.2 Tasques explícites — paral·lelisme per tasques

Les tasques explícites es creen **manualment** amb la directiva `#pragma omp task`. En lloc d'assignar feina directament a cada thread, un thread s'encarrega de **crear les tasques** i dipositar-les en una **pool de tasques**, d'on els threads les agafen i executen de forma dinàmica.

Per evitar que **tots els threads creïn les mateixes tasques**, s'utilitza la directiva `#pragma omp single`, que garanteix que només **un thread** executa el bloc de creació, mentre la resta esperen a que hi hagi tasques a la pool:

```
#pragma omp parallel
#pragma omp single
{
    #pragma omp task
    {
        // codi de la tasca
    }
}
```

La clau d'aquest model és la separació entre *creació* i *execució*: una tasca pot executar-se **més tard** i per un **thread diferent** del que l'ha creada. Això permet una distribució de càrrega molt més flexible que el model de tasques implícites.

3.3 Instruccions bàsiques d'OpenMP

Creació de tasques:

- `#pragma omp parallel [num_threads(N)]`: crea tasques implícites (una per thread). Inclou una barrera implícita al final. Opcionalment podem usar `num_threads(N)` per especificar el nombre de threads.
- `#pragma omp single`: només un thread executa el bloc; la resta esperen tasques a la pool. Inclou una barrera implícita al final (eliminable amb `nowait`).
- `#pragma omp task`: crea una tasca explícita que s'afegeix a la pool i s'executarà quan un thread estigui disponible. Opcionalment podem usar `depend(in/out: var)` per establir dependències entre tasques.
- `#pragma omp taskloop`: crea tasques explícites a partir de les iteracions d'un **for comptable**. Inclou un `taskgroup` implícit al final. Es pot controlar la granularitat amb:
 - `grainsize(m)`: entre m i $2m$ iteracions per tasca.
 - `num_tasks(m)`: aproximadament m tasques creades.

Barreres de tasques:

- `#pragma omp taskwait`: suspèn la tasca actual fins que totes les seves tasques **filles directes** hagin acabat. No espera descendents de nivells més profunds.
- `#pragma omp taskgroup`: suspèn la tasca actual al final del bloc, esperant totes les tasques **filles i tots els seus descendents**.

3.4 Dependències en OpenMP (depend)

La clàusula `depend(...)` s'afegeix a `#pragma omp task` per indicar dependències de dades entre tasques **germanes** (del mateix pare), evitant haver de gestionar barreres manualment.

Especificadors de dependència:

- `depend(in: var)`: la tasca espera totes les germanes anteriors amb `out` o `inout` sobre la mateixa variable. Garanteix dependències **RaW** (Read after Write).
- `depend(out: var)`: la tasca espera les germanes anteriors amb `in`, `out` o `inout` sobre la mateixa variable. Garanteix dependències **WaR** (Write after Read) i **WaW** (Write after Write).
- `depend(inout: var)`: combina `in` i `out`. Espera totes les germanes que llegeixin o escriguin la mateixa variable.

3.5 Privacitat de variables en OpenMP

En paralelitzar, cal decidir si les variables són **compartides** entre tots els threads o si cada thread té una còpia **privada**. Aquesta decisió és **crítica** per evitar *data races* i comportaments inesperats.

Aquestes són les clàusules que podem col·locar al costat d'una directiva `#pragma omp parallel, task` o `taskloop`, per controlar l'àmbit de cada variable:

- `private(var)`: cada thread/tasca rep una còpia privada **no inicialitzada**. El valor original **no es conserva ni es restaura** en acabar.
- `firstprivate(var)`: cada thread/tasca rep una còpia privada **inicialitzada** amb el valor actual. El valor original **no es conserva ni es restaura** en acabar.
- `shared(var)`: la variable és compartida entre tots els threads de l'equip. Caldrà vigilar la concurrència! És l'única manera de fer **persistent** una dada entre tasques sense recórrer a memòria global.

Tant amb `private` com amb `firstprivate`, el valor de la còpia local es perd en acabar la tasca.

Regles per defecte:

Context	Àmbit de la variable	Atribut per defecte
<code>parallel</code>	Declarada fora la regió paral·lela	<code>shared</code>
<code>parallel</code>	Declarada dins la regió paral·lela	<code>private</code>
<code>task</code>	Declarada fora la tasca (dins la regió paral·lela)	<code>firstprivate</code>

En tasques explícites, les variables externes són `firstprivate` per defecte. Això vol dir que **capturen el valor en el moment de creació**, no d'execució. Si necessitem que la tasca modifiqui una variable visible fora d'ella, cal marcar-la com a `shared` explícitament.

3.6 Protegir problemes de concurrència (Data Race)

En càlcul paral·lel és habitual acumular resultats parcials en una variable compartida (per exemple, un producte escalar). Múltiples threads accedint i modificant simultàniament la mateixa variable provoquen **data races**. OpenMP ofereix diversos mecanismes per gestionar-ho, ordenats de **menor a major eficiència**:

- **Secció crítica (critical)**: garanteix que només un thread executa el bloc cada vegada. És la solució més general però la menys eficient, ja que **serialitza tots els accessos**, fins i tot quan no hi ha conflicte real.

```
#pragma omp parallel
{
    int val = compute_local(...);
    #pragma omp critical
    {
        insert_sorted(shared_list, val); // operacio arbitraria
    }
}
```

- **Locks**: ofereixen un control més fi que **critical**. En lloc de bloquejar globalment un bloc de codi, cada lock s'associa a un **recurs concret** (per exemple, una posició d'una taula hash), permetent que threads treballin en paral·lel sobre recursos independents. Si tots accedeixen al mateix recurs, el rendiment serà similar a **critical**.

Per utilitzar-los cal declarar variables de tipus `omp_lock_t`, inicialitzar-les i destruir-les:

- `omp_init_lock(&lock)`: inicialitza el lock.
- `omp_set_lock(&lock)`: si el lock està lliure, el bloqueja i continua; si està ocupat, el thread queda bloquejat esperant.
- `omp_unset_lock(&lock)`: allibera el lock perquè un altre thread el pugui adquirir.
- `omp_destroy_lock(&lock)`: destrueix el lock.

```
omp_lock_t v_locks[M]; // un lock per cada recurs
for (int i = 0; i < M; i++)
    omp_init_lock(&v_locks[i]);

#pragma omp parallel
{
    while (...) {
        int index = ...;
        omp_set_lock(&v_locks[index]); // bloqueja index
        function(v[index], ...);       // acces protegit
        omp_unset_lock(&v_locks[index]); // desbloqueja index
    }
}

for (int i = 0; i < M; i++)
    omp_destroy_lock(&v_locks[i]);
```

- **Operació atòmica (`atomic`):** garanteix que una operació d'actualització simple sigui indivisible. Només funciona per a operacions bàsiques (+, -, *, /, &, |, ^, <<, >>). Menys sobrecost que `critical`, però genera una operació atòmica **per cada actualització**, de manera que amb molts threads sobre la mateixa variable la contenció pot ser elevada.

```
int count = 0;
#pragma omp parallel
{
    // ... cada thread fa feina ...
    if (found_match(...)) {
        #pragma omp atomic
        count++; // actualitzacio atomica puntual
    }
}
```

- **Reducció manual:** cada thread acumula en una variable **local** (privada) i fa una **única operació atòmica** al final per combinar el resultat parcial amb el global. Redueix les operacions atòmiques de N iteracions al **nombre de threads**.

```
int total = 0;
#pragma omp parallel {
    int id = omp_get_thread_num();
    int nt = omp_get_num_threads();
    int BS = N / nt;
    int total_local = 0;
    for (int i = id*BS; i < (id+1)*BS; i++) {
        total_local += v[i] * w[i];
    }
    #pragma omp atomic
    total += total_local; // una unica atomica per thread
}
```

- **Reducció automàtica (`reduction`):** la clàusula `reduction(op:var)` és la solució més neta i eficient. OpenMP gestiona automàticament les còpies privades i les combina al final. Operadors suportats: +, -, *, /, &, |, ^, max, min. Per a tasques explícites amb `task` (no `taskloop`), cal usar `task_reduction` (al pare) + `in_reduction` (a cada tasca filla).

```
int total = 0;
#pragma omp parallel
#pragma omp single {
    #pragma omp taskloop reduction(+:total)
    for (int i = 0; i < N; i++)
        total += v[i] * w[i]; // OpenMP gestiona les copies privades
}
```

Mètode	Sobrecost	Escalabilitat	Ús típic
<code>critical</code>	Molt alt	Baixa	Accés genèric a recursos
Locks	Alt	Mitjana	Recursos independents (taula hash)
<code>atomic</code>	Mitjà	Baixa	Operacions simples sobre una variable
Reducció manual	Baix	Alta	Càlculs acumulatius
Reducció automàtica	Baix	Alta	Sumes, màxims, mínims. ...

3.7 Descomposició iterativa de bucles

En paralelitzar un bucle, repartim les iteracions entre diversos threads perquè s'executin de manera concurrent. L'objectiu és distribuir la càrrega de treball tot minimitzant la sincronització i els *overheads*.

3.7.1 Amb tasques implícites

Quan utilitzem `#pragma omp parallel`, OpenMP crea una tasca implícita per cada thread del *team*. Tots executen el mateix codi, i cada thread calcula quin rang d'iteracions li correspon a partir del seu identificador i del nombre total de threads.

Versió bàsica: una operació atòmica per iteració

La forma més directa de protegir una variable compartida és aplicar `atomic` a cada actualització. Això evita *data races*, però introdueix un cost de sincronització per iteració.

```
int sum = 0;
#pragma omp parallel shared(sum)
{
    int id = omp_get_thread_num();
    int nt = omp_get_num_threads();
    int BS = N / nt;
    int start = id * BS;
    int end = start + BS; // o (id + 1)*BS;

    for (int i = start; i < end; i++) {
        #pragma omp atomic // o critical en cas de no poder fer atomic
        sum += v[i]; // una operacio atomica per CADA iteracio
    }
}
```

Versió eficient: reducció manual

Podem reduir dràsticament el nombre d'operacions atòmiques acumulant el resultat primer en una variable local i combinant el resultat parcial al final. Així, el bucle és independent entre threads i només es sincronitza un cop per thread:

```
int sum = 0;
#pragma omp parallel shared(sum)
{
    ...

    int local_sum = 0;
    for (int i = start; i < end; i++)
        local_sum += v[i];

    #pragma omp atomic // o critical en cas de no poder fer atomic
    sum += local_sum; // reduccio manual: una operacio atomica per thread
}
```

Això passa de N operacions atòmiques a P (una per thread).

Aquesta optimització només la podem aplicar quan les iteracions són independents entre si i l'operació és associativa! Si existeixen dependències entre iteracions, no podem acumular resultats locals sense alterar el funcionament del programa.

3.7.2 Amb tasques explícites

En el model de tasques explícites, un thread (el *master*) crea tasques que s'insereixen en una *pool* de tasques. La resta de threads del *team* van agafant tasques de la pool i les van executant a mesura que queden lliures. Un cop el thread creador acaba de generar totes les tasques, també passa a executar-ne de la pool.

Creació manual (#pragma omp task): En aquesta versió, un thread (dins d'un **single**) crea explícitament una tasca per cada bloc d'iteracions. Cada tasca acumula el seu resultat parcial en una variable local i el combina amb un **atomic**, de manera anàloga a la reducció manual vista anteriorment:

```
int sum = 0;
#pragma omp parallel
#pragma omp single
{
    int nt = omp_get_num_threads();
    int BS = N / nt;
    for (int t = 0; t < nt; t++) {
        int start = t * BS;
        int end = start + BS; // o (t + 1)*BS;
        #pragma omp task firstprivate(start, end) shared(sum)
        {
            int local_sum = 0;
            for (int i = start; i < end; i++)
                local_sum += v[i];
            #pragma omp atomic
            sum += local_sum;
        }
    }
} // barrera implicita del single, espera que finalitzin totes les tasques
```

Versió amb reducció automàtica (taskloop): Una altra forma més neta és delegar tant el repartiment d'iteracions com la gestió de la reducció a l'entorn d'execució d'OpenMP. Quan el bucle és comptable, podem usar la instrucció **taskloop**:

```
int sum = 0;
#pragma omp parallel
#pragma omp single
{
    int nt = omp_get_num_threads();
    #pragma omp taskloop reduction(+:sum) num_tasks(nt)
    for (int i = 0; i < N; i++) {
        sum += v[i];
    } // taskgroup implicit: espera totes les tasques del taskloop
}
```

- **num_tasks(nt)** indica quantes tasques es crearan.
També podríem usar **grainsize(m)** per indicar la mida de cada tasca, i que se'n creïn tantes com calgui.
- **reduction(+:sum)** manté còpies privades de la variable i les combina al final, sense que el programador hagi de gestionar cap operació atòmica.

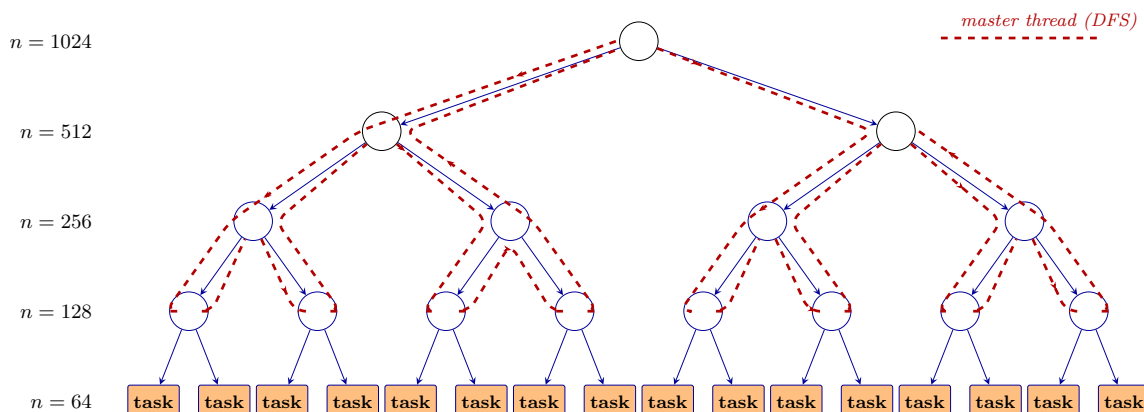
Aquesta és la forma **recomanada** per a bucles regulars amb operacions associatives simples, perquè és més llegible. Ara bé, **taskloop** només es pot aplicar a bucles comptables amb operacions de reducció estàndard (+, *, max, min...). Quan el bucle no és comptable o la combinació de resultats requereix una lògica més complexa, caldrà recórrer a la reducció manual amb **atomic**.

3.8 Recursive Task Decomposition

Quan treballem amb algoritmes *divide-and-conquer*, també podem paral·lelitzar la generació i execució de tasques de forma recursiva. Hi ha dues estratègies principals: **Leaf** i **Tree**.

3.8.1 Leaf Strategy

En l'estratègia **Leaf**, un únic thread (master) recorre seqüencialment l'arbre de recursió. Quan arriba a un cas base (fulla), crea una tasca explícita perquè sigui executada per qualsevol thread lliure del *team*. Per tant, les tasques es creen **només a les fulles**!



Passos per paral·lelitzar:

1. **Regió paral·lela i creació de tasques:** Obrim una regió paral·lela amb `single` perquè un sol thread recorri l'arbre.

```
#pragma omp parallel
#pragma omp single
crida_recursiva(...)
```

2. **Creem tasques** en el cas base de la recursió usant `#pragma omp task`.

```
crida_recursiva() {
    if (cond) {
        #pragma omp task {
            cas_base();
        }
    }
    else crida_recursiva();
}
```

3. **Data race conditions:** La variable global de resultat és compartida i modificada per totes les tasques $\Rightarrow$ cal protegir-la.
4. **Sincronització:** Afegim `#pragma omp taskwait` per assegurar que totes les tasques han acabat abans de consumir el resultat.

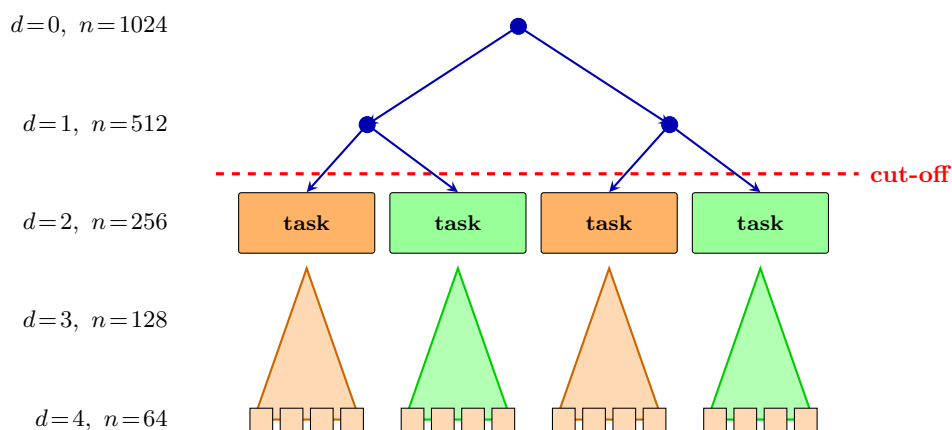
```
#pragma omp parallel
#pragma omp single
crida_recursiva(...)
#pragma omp taskwait
// Valor de resultat correcte
```

Problemes de l'estratègia Leaf.

- Si baixem fins a molta profunditat podem arribar a tenir **moltes tasques** (una per cada fulla $\Rightarrow 2^{\text{profunditat}}$ tasques, de mida $N/2^{\text{profunditat}}$ cadascuna).
- Un sol thread s'encarrega de crear totes les tasques $\Rightarrow$ molt **overhead de creació** concentrat en un únic thread.
- Si la granularitat de les tasques és molt petita $\Rightarrow$ **s'executen les tasques molt més ràpid del que es creen**, provocant que els threads quedin inactius esperant.
- Si recórrer l'arbre és car, es concentrarà molta càrrega de feina en un únic thread mentre els altres estaran esperant que els arribi la feina $\Rightarrow$ Load Imbalance.

3.8.2 Leaf amb Cut-off

Per reduir l'*overhead* de creació de tasques i el temps que un únic thread dedica a recórrer l'arbre, introduïm un *cut-off* per profunditat. A partir d'un cert nivell (CUTOFF), el thread creador **deixa de baixar seqüencialment** i crea una tasca que **assumeix la continuació de la recursió** de manera independent. Això reparteix el recorregut (i la creació de tasques) entre múltiples threads.



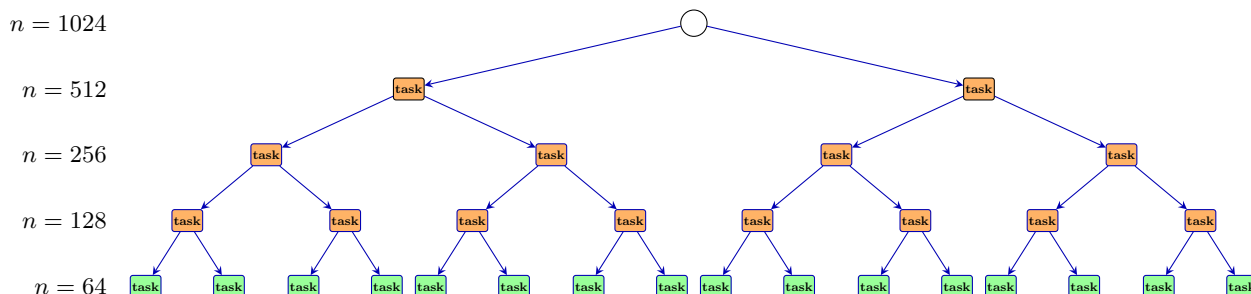
Estratègia *Leaf* amb *cut-off*: a partir de la profunditat de tall ($d = 2$), el thread creador deixa de recórrer seqüencialment i genera noves tasques que continuen la recursió de manera independent.

Quan `depth == CUTOFF`, es crea una tasca que encapsula la recursió del subarbre a partir d'aquell node. Dins d'aquesta tasca, la recursió continua seqüencialment fins arribar al cas base. A partir d'aquest punt, el cas base **s'executa dins la mateixa tasca** (no cal crear cap altra tasca "de continuació" perquè ja som dins del subarbre delegat).

Cal tenir en compte que, si s'arriba al cas base amb `depth < CUTOFF`, encara s'ha de crear la tasca explícita del cas base (tal com feiem en l'estratègia Leaf original).

3.8.3 Tree Strategy

En l'estratègia *Tree*, cada crida recursiva crea una nova tasca, que al seu torn generarà més crides i, per tant, més tasques. Hi ha tasques que **només creen tasques** (nodes interns) i d'altres que **només executen el cas base** (fulles).

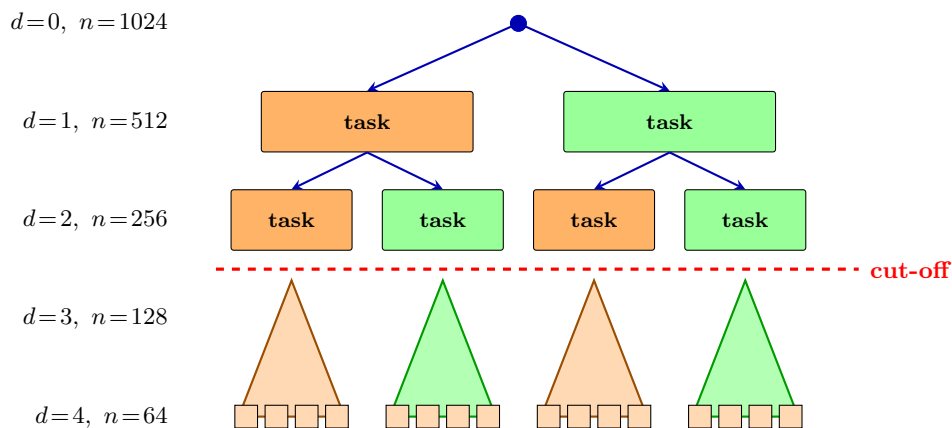


Problemes de l'estratègia Tree

- Es creen tasques a **tots els nivells** $\Rightarrow$ el nombre total pot ser enorme $2^{d+1} - 1$
- Per tant el cost dels *overheads* de creació, gestió i sincronització pot arribar a ser enorme.
- Amb granularitat massa fina, el programa pot dedicar més temps a gestionar tasques que a executar-les.

3.8.4 Tree amb Cut-off

A partir d'un cert nivell (CUTOFF), es deixa de crear tasques i la branca restant s'executa seqüencialment (tant casos recursius com base).



Estratègia *Tree* amb *cut-off*: es creen tasques a cada nivell per sobre del tall. Per sota, cada tasca continua la recursió de manera seqüencial sense generar noves tasques.

3.8.5 Control de cut-off amb final i omp_in_final

OpenMP proporciona un mecanisme integrat per controlar el *cut-off* sense necessitat de gestionar manualment la profunditat amb un *if-else*:

- `#pragma omp task final(condició)`: si la condició és certa, la tasca es marca com a **final**. Una tasca final s'executa immediatament pel thread que la crea (sense afegir-la a la *pool*), i totes les tasques filles generades dins d'una tasca final també s'executen immediatament (és a dir, de forma seqüencial).
- `omp_in_final()`: retorna `true` si la tasca actual s'està executant en un context final. Permet decidir, en temps d'execució, si cal crear més tasques o continuar seqüencialment.

La clàusula `final(depth >= CUTOFF)` marca la tasca com a final quan s'assoleix la profunditat de tall. A partir d'aquell punt, la crida a `omp_in_final()` retorna cert i s'executa la branca seqüencial, evitant la creació de noves tasques.

3.9 Consolidació T3 — Descomposició de tasques

Del tema 3 hauries de saber:

- Entendre la diferència entre **tasques implícites** (parallelisme per threads) i **tasques explícites** (parallelisme per tasques), i saber quan convé cada model.
- Conèixer les directives bàsiques d'OpenMP: `parallel`, `single`, `task`, `taskloop`, `taskwait` i `taskgroup`, i saber quines inclouen barreres implícites.
- Saber establir **dependències entre tasques** amb la clàusula `depend(in/out/inout)` i identificar els tipus de dependència (RaW, WaR, WaW).
- Decidir correctament l'**àmbit de cada variable** (`private`, `firstprivate`, `shared`) i conèixer les regles per defecte segons el context (`parallel` vs `task`).
- Identificar **data races** i saber protegir-les amb el mecanisme adequat (`critical`, locks, `atomic`, reducció manual o `reduction`), entenent els compromisos de sobrecost i escalabilitat de cadascun.
- Parallelitzar **bucles iteratius** tant amb tasques implícites com explícites, aplicant reducció manual o automàtica (`taskloop` + `reduction`) segons el cas.
- Parallelitzar **codi recursiu** amb les estratègies **Leaf** i **Tree**, identificant els problemes de cada una (overhead de creació, granularitat, load imbalance) i aplicant **cut-off** per mitigar-los (manualment o amb `final/omp_in_final`).



ASSES NORD

González Tablas, 7
08034 Barcelona



93.125.70.02 / 605.411.687



@fiberases
@fib_primer_ases



www.asesacademia.com

