

PAR



Parallelisme • 2025-26 QP • RogeR Bitlloch

LABORATORIS

L3. Anàlisi d'estratègies de paral·lelisme

L4. Parallel task decomposition implementation and analysis

L5. Geometric data decomposition implementation and analysis

Basat en les diapositives de la FIB

Índex

Lab1: Introducció a l'entorn (boada)	4
1.1 Connexió i setup inicial	4
1.2 Transferència de fitxers (scp)	4
1.3 Compilació i execució	5
1.4 Strong vs. Weak scalability	5
Lab3: Tareador (anàlisi de tasques)	6
3.1 Què és Tareador?	6
3.2 API de Tareador	6
3.3 Compilar i executar	6
3.4 Analitzar dependències	7
3.5 Desactivar dependències	7
3.6 Simular execució paral·lela	7
3.7 Estratègies a analitzar	8
3.7.1 Iteratives	8
3.7.2 Recursives	9
Lab4: Modelfactors i Paraver	10
4.1 Pipeline d'anàlisi	10
4.2 Execució dels scripts	10
4.3 Com llegir les taules de Modelfactors	11
4.4 Paraver: anàlisi de traces	11
4.5 Estratègies de paral·lelització	12
4.5.1 Iterativa: Tiled (coarse grain)	12
4.5.2 Iterativa: Fine grain	12
4.5.3 Recursiva: Leaf	12
4.5.4 Recursiva: Tree	12
4.6 Comparació final	12
Lab5: Geometric Data Decomposition	13
5.1 Setup i workflow	13
5.2 Les 3 descomposicions	13
5.2.1 1D Block per columnes	13
5.2.2 1D Block-Cyclic per columnes	14
5.2.3 1D Cyclic per files	14
5.3 Comparativa	14
5.4 Anàlisi Paraver (16 threads)	15
5.5 Conceptes clau d'aquest lab	15

Lab1: Introducció a l'entorn (boada)

1.1 Connexió i setup inicial

```
ssh -X par????@boada.ac.upc.edu      # connexió amb X11
ssh -t par????@boada.ac.upc.edu passwd # canvi contrasenya
```

Des de fora la UPC cal la [VPN de la UPC](#).

En windows cal instal·lar [putty](#) per la secure shell i [xming](#) per X11.
També pots usar [MobaXterm](#) que integra les dues coses.

Espai de treball

El nostre espai de treball amb les nostres dades serà:

```
cd /scratch/nas/1/par????          # També podem fer cd ~/ (home)
```

Podem copiar el material dels labs al nostre espai de treball fent:

```
cp ~par0/sessions/labX.tar.gz ~/    # copiar lab X
tar -zxvf labX.tar.gz               # descomprimir
```

Environment

```
source ~/environment.bash
# Per automatitzar-ho (sinó caldrà fer-ho cada cop que fem login):
echo "source ~/environment.bash" >> ~/.bashrc
```

1.2 Transferència de fitxers (scp)

```
# boada -> PC local
scp -O par????@boada.ac.upc.edu:labX/fitxer.c .
# PC local -> boada
scp -O fitxer.c par????@boada.ac.upc.edu:labX/
# També podem copiar un directori sencer
scp -O -r par????@boada.ac.upc.edu:labX/ .
```

1.3 Compilació i execució

Compilar sempre amb `make` (mirar el `Makefile` per entendre targets).

Interactiu (nodes 11–13, 1 sol core)

S'executa als nodes interactius (boada-11 a 13), limitats a **1 core**.

Els temps **no són fiables** per mesurar rendiment, però serveix per comprovar que l'execució és correcta abans d'enviar als nodes d'execució.

```
./run-seq.sh <exe> <iters>          # execució seqüencial
./run-omp.sh <exe> <iters> <threads> # execució OpenMP
```

La diferència és que `run-omp.sh` configura el nombre de threads OpenMP (`OMP_NUM_THREADS`) i llença el binari compilat amb `-qopenmp`, mentre que `run-seq.sh` simplement executa el binari seqüencial sense cap runtime OpenMP.

Via cua SLURM (nodes 18–20, multicore)

S'envien als nodes d'execució (boada-18 a 20), que són **multicore** i executen en **aïllament** — per tant els temps sí que són representatius.

```
sbatch submit-seq.sh <exe> <iters>
sbatch submit-omp.sh <exe> <iters> <threads>
squeue -u par????      # estat dels jobs
scancel <jobid>         # cancel·lar job
```

Cada `sbatch` genera fitxers `.o` (stdout) i `.e` (stderr) amb el `jobid` al nom.

1.4 Strong vs. Weak scalability

Strong: mida fixa, augmentem threads $\rightarrow$ volem reduir temps.

Weak: mida $\propto$ threads $\rightarrow$ volem mantenir temps constant.

```
sbatch submit-strong-omp.sh  # strong scaling
sbatch submit-weak-omp.sh   # weak scaling
# Visualitzar:
ps2pdf pi_omp-????.ps      # generar pdf
xpdf pi_omp-????.pdf        # obrir pdf
```

Lab3: Tareador (anàlisi de tasques)

3.1 Què és Tareador?

Tareador és una eina d'anàlisi, **NO paral·lelitza**. Ens mostra de forma visual les tasques i les seves dependències per en un futur usar `#pragma omp` per paral·lelitzar.

3.2 API de Tareador

Abans de res cal activar el Tareador:

```
#include <tareador.h>

// Activar/desactivar l'anàlisi
tareador_ON();
/* Main program */
tareador_OFF();
```

Un cop el tenim activat podem crear tasques mitjançant:

```
// Definir una tasca potencial
tareador_start_task("TaskName");
/* Codi que sera una tasca */
tareador_end_task("TaskName");
```

3.3 Compilar i executar

Iteratiu

```
make mandel-seq-iter-tar          # compilar amb la lib de Tareador
./run-tareador.sh mandel-seq-iter-tar      # sense args
./run-tareador.sh mandel-seq-iter-tar -d    # amb display
./run-tareador.sh mandel-seq-iter-tar -h    # amb histogram
```

Recursiu

```
make mandel-seq-rec-tar          # compilar versio recursiva
./run-tareador.sh mandel-seq-rec-tar      # executar
```

L'execució és **molt lenta** degut a la instrumentació. Al acabar s'obre una finestra amb el **Task Dependency Graph (TDG)**: cada node és una tasca (com més gran, més instruccions executa) i cada aresta és una dependència (blau = dades, verd/groc = control).

Consell pràctic: Treballa sempre sobre el codi base (`mandel-seq-iter-tar.c` per l'iteratiu, `mandel-seq-rec-tar.c` per el recursiu). Quan acabis una estratègia, fes una còpia (`cp`) amb un nom descriptiu per l'entrega abans de modificar el codi per el següent apartat.

Veure el Mandelbrot (opcional)

Per veure com funciona el programa i la imatge que genera:

```
# Iteratiu:
sbatch submit-seq.sh ./mandel-seq-iter -h -o -i 10000
./mandel-seq-iter -d -i 10000      # o interactiu amb display
# Recursiu:
sbatch submit-seq.sh ./mandel-seq-rec -h -o -i 10000
./mandel-seq-rec -d -i 10000
```

Això no és necessari per la pràctica.

3.4 Analitzar dependències

Al TDG pots veure quines variables causen dependències:

- **Botó dret sobre aresta** → Dataview → edge: mostra la variable que causa la dependència.
- **Botó dret sobre tasca** → Dataview → Edges-in / Edges-out: totes les dependències d'entrada/sortida.
- **Botó dret sobre tasca** → Dataview → node: variables llegides (verd), escrites (blau), ambdues (taronja). Tipus: G=global, H=heap, S=stack.

Per cada dependència cal raonar: **força l'ordre** de les tasques (dependència real), o es pot resoldre amb **sincronització de data sharing** (*atomic*, *critical*, etc.)?

3.5 Desactivar dependències

Si una variable causa dependències que volem ignorar (per simular que les estem tractant amb sincronització i veure com quedaria el paral·lelisme):

```
tareador_disable_object(&VariableName);
/* Codi on ignorem VariableName */
tareador_enable_object(&VariableName);
```

Això elimina totes les dependències causades per `VariableName` al TDG. Només s'han de desactivar les dependències que es podrien resoldre amb data sharing (p.ex. histograma, display), **mai** les dependències reals sobre la matriu de resultats.

3.6 Simular execució paral·lela

Clicar View Simulation a la finestra de Tareador → s'obre Paraver amb una simulació ideal:

- Cada línia horitzontal = un processador.
- Colors = tasques (coincideixen amb el TDG).
- Número a baix-dreta = temps simulat (en unitats d'instrucció).
- Línies grogues = dependències entre tasques.

3.7 Estratègies a analitzar

Per cada estratègia cal:

1. Instrumentar el codi.
2. Identificar dependències i desactivar les de data sharing.
3. Simular amb 1 i 128 processadors per obtenir T_1 i T_∞ .
4. Calcular $\text{Parallelisme} = \frac{T_1}{T_\infty}$ i l'Eficiència(P) = $\frac{\text{Par}}{P}$.

3.7.1 Iteratives

Original tile

Cada iteració del bucle x és una tasca (16 tasques en total).

- **Dependències amb -d:** la variable `X11_COLOR_fake` serialitza l'execució.
Cal desactivar-la amb `tareador_disable_object`.
- **Dependències amb -h:** el vector `histogram` serialitza l'execució.
Cal desactivar-lo de la mateixa manera.
- Ambdues variables requeriran tractar la sincronització quan es paral·lelitzi per tal d'evitar data race condition. Aquestes conclusions s'apliquen a **totes** les estratègies següents, per això no cal repetir -d/-h.
- **Resultat:** paral·lelisme baix malgrat ser embarrassingly paral·lel, a causa del **load unbalance**: algunes tasques només omplen el tile (ràpid) mentre que d'altres computen tots els punts (lent).

Fine grain

Es descomposa cada tile en 4 tipus de tasques: (1) check borders verticals, (2) check borders horitzontals, (3) cada fila py de fill tile, (4) cada fila py de compute tile.

- **Dependència per equal:** es resol creant dues variables separades, `equal1` i `equal2` (una per cada check), i comprovant `if (equal1 & equal2)` abans del condicional.
- **Dependència per M[y][x] al fill:** es resol guardant el valor a una variable auxiliar fora de la tasca.
- **Resultat:** el paral·lelisme més alt de totes les estratègies, però amb load unbalance (les fill tasks són molt curtes vs. les compute llargues) i un nombre elevat de tasques que pot introduir overhead.

Column of tiles

S'intercanvien els bucles $x \leftrightarrow y$ i cada iteració x (una columna sencera de tiles) és una tasca.

- **Sense dependències** $\rightarrow$ embarrassingly paral·lel.
- **Resultat:** paral·lelisme molt baix amb només 4 tasques. Les columnes centrals del fractal fan molta més feina que les exteriors, provocant un load imbalance molt gran.

3.7.2 Recursives

Leaf

Només els casos base de la recursió (fill tile i compute tile) es defineixen com a tasques.

- **Sense dependències** entre tasques.
- **Resultat:** paral·lisme baix malgrat crear 16 tasques. El motiu és que els checks de borders es fan seqüencialment durant el recorregut recursiu, inflant T_∞ . A més, hi ha load unbalance entre **fill** i **compute**.

Tree

Es creen tasques a cada nivell de la recursió: les 4 crides recursives són tasques, i també els checks de borders verticals i horitzontals.

- **Dependència RaW per equal:** mateixa solució que a fine grain (equal1/equal2). El condicional **if** (equal1 && equal2) és una dependència **real** que no es pot eliminar.
- **Resultat:** millor que leaf perquè els checks de borders es fan en paral·lel, però les dependències reals entre nivells de la recursió limiten l'speedup.

Millor iterativa: Fine grain, tot i que l'elevat nombre de tasques pot introduir overhead al paral·lelitzar.

Millor recursiva: Tree, ja que els checks de borders es fan de forma simultània en lloc de seqüencialment.

Lab4: Modelfactors i Paraver

En aquest lab agafem les estratègies de paral·lelització del lab passat i les fem realitat amb OpenMP.

4.1 Pipeline d'anàlisi

codi font $\xrightarrow{\text{make}}$ executable $\xrightarrow{\text{Extrac}}$ traces (.prv, .pcf, .row) $\xrightarrow{\text{Paraver}}$ visualització.

4.2 Execució dels scripts

```
# Compile (revisa el Makefile per veure els targets)
make mandel-omp-iter # en els recursius: mandel-omp-rec
```

```
# Executar codi seqüencial (genera outputs de referència)
sbatch submit-seq.sh mandel-seq-iter -h -o -i 10000
# Renombra els outputs abans d'executar el paral·lel!
mv mandel_image.jpg mandel_image_seq.jpg
mv mandel_histogram.out mandel_histogram_seq.out
```

```
# Executar codi paral·lel i comprovar correctesa
sbatch submit-omp.sh <exe> 20
cmp mandel_image.jpg mandel_image_seq.jpg
cmp mandel_histogram.out mandel_histogram_seq.out
```

```
# Modelfactors (traces + taules)
sbatch submit-strong-extrac.sh <exe>
# Genera directori: <exe>-strong-extrac/
# -> modelfactors.out, modelfactor-tables.pdf
# -> traces .prv per cada #threads
```

```
# Strong scaling (genera speedup-*.txt i elapsed-*.txt)
sbatch submit-strong-omp.sh
```

```
# Monitoritzar tasca
squeue -u par????
watch squeue # Ctrl-C per sortir
```

Important: Renombra el directori *-strong-extrac i posa-li el nom de l'estratègia abans de tornar a executar els Modelfactors, sinó se sobreescriu!

4.3 Com llegir les taules de Modelfactors

1. **Taula 1 — Mètriques globals:** elapsed time, speedup, efficiency. Visió general.
2. **Taula 2 — Desglossament de l'eficiència:**
 - φ (fracció paral·lela): si és baix $\rightarrow$ massa codi seqüencial.
 - **Load balancing (LB):** si és baix $\rightarrow$ càrrega mal repartida.
 - **In execution efficiency:** si és baix $\rightarrow$ massa overheads al camí crític.
 - **Computation scalability:** si és baix $\rightarrow$ hardware no escala.
3. **Taula 3 — Detall de tasques explícites:**
 - **# explicit tasks:** tasques creades en total.
 - **LB (# tasks)** Distribució entre threads del nombre de tasques.
 - **LB (time tasks):** Distribució entre threads del temps d'execució entre tasques (més rellevant).
 - **Time per task (avg μ s):** granularitat.
 - **Overhead synch/sched %:** temps perdut en sync i scheduling per tasca.
 - **# taskwait/taskgroup:** punts de sincronització.

4.4 Paraver: anàlisi de traces

```
wxparaver                                # obrir Paraver
# File > Load Trace > navegar fins .prv dins *-strong-extrae/
# Clic "New single timeline window" (icona dalt-esquerra)
```

Colors del timeline (Thread State)

Blau fosc = Running (codi útil), blau clar = Idle, vermell = Synchronization, blanc = Not created, groc = Scheduling / Fork-join. Un píxel pot representar molts canvis d'estat: fes zoom abans de treure conclusions.

Navegació

- Botó esquerre arrossegat $\rightarrow$ zoom in.
- Botó dret $\rightarrow$ Undo Zoom / Fit time scale.
- Doble clic esquerre $\rightarrow$ info del punt (thread, estat, durada).
- Botó dret $\rightarrow$ Info Panel $\rightarrow$ Colors tab per la llegenda.
- Botó dret $\rightarrow$ View $\rightarrow$ Event Flags per events OpenMP.

Sincronitzar finestres

Botó dret a finestra A $\rightarrow$ Copy, botó dret a finestra B $\rightarrow$ Paste $\rightarrow$ Default. O bé Synchronise $\rightarrow$ 1 a ambdues finestres.

Hints principals (menú Hints)

- Useful/Instantaneous parallelism: # threads executant codi útil a cada instant.
- tasking/explicit tasks created&exec: creació (groc) i execució (blau) de tasques.
- tasking/explicit tasks duration: granularitat (gradient de colors).
- Thread state profile: histograma % temps per estat i thread.

4.5 Estratègies de paral·lelització

4.5.1 Iterativa: Tiled (coarse grain)

Cada tile és una tasca. Un sol thread (el del **single**) les crea totes; poques tasques en total.

- `#pragma omp task firstprivate(x, y)private(equal)` dins el doble bucle de tiles.
- `#pragma omp atomic` per histograma, `#pragma omp critical` per display.
- **Resultat:** speedup s'estanca aviat. LB molt dolent (tasques de mida desigual), overhead synch creixent per la barrera final. A Paraver: barres blaves desiguals + grans zones vermelles.

4.5.2 Iterativa: Fine grain

Refina la tiled: dins cada tile, sub-tasques per comprovar vores en paral·lel + sub-tasques per fila per fill/compute. No eliminis la tasca pare del tile.

- `task shared(equal_vertical)` i `shared(equal_horitzontal)` per les vores.
- `taskwait` entre comprovació de vores i fill/compute.
- Tasca per fila: `task firstprivate(x, py)`.
- **Resultat:** speedup gairebé lineal. LB(time) ≈ 1.0 , overheads baixos. La millor eficiència de totes. A Paraver: paral·lelisme instantani alt i sostingut.

4.5.3 Recursiva: Leaf

Divisió recursiva en 4 quadrants fins a TILE. Tasques **només a les fulles** (fill i compute); la recursió i comprovació de vores són seqüencials.

- `task firstprivate(y,x)` només a fill i compute.
- **Resultat:** la pitjor de totes. Speedup ~ 1.3 i s'estanca. Un sol thread fa tota la recursió, els altres esperen. Overhead synch per tasca explota (milers de %). A Paraver: thread 1 treballant, la resta gairebé tot vermell.

4.5.4 Recursiva: Tree

Millora la leaf: tant les vores com les crides recursives són tasques. Múltiples threads recorren l'arbre en paral·lel. `final(condition)` limita la creació a mides petites (cutoff).

- Vores en paral·lel: dues tasques `shared(equal1/equal2) + taskwait`.
- Crides recursives: `task final(NRows*NCols <= CUTOFF_SIZE)`.
- Dins `omp_in_final()`: tot seqüencial (sense crear tasques).
- **Resultat:** temps absolut més baix. Speedup creixent fins a 16-20. LB(time) alt, overhead synch baix. A Paraver: blau dominant, paral·lelisme alt i sostingut.

4.6 Comparació final

- **Iterativa Tiled:** s'estanca aviat, la pitjor iterativa.
- **Iterativa Fine Grain:** la més escalable (millor eficiència a molts threads).
- **Recursiva Leaf:** la pitjor de totes, gairebé no escala.
- **Recursiva Tree:** temps absolut més baix, bona escalabilitat.

Tree recursiva $\rightarrow$ més ràpida

Fine grain iterativa $\rightarrow$ més escalable.

Lab5: Geometric Data Decomposition

Versió simplificada del Mandelbrot (sense tiles ni comprovació de vores): es calcula tota la matriu. Implementem tres repartiments geomètrics de les dades entre threads. Novetat respecte labs anteriors: anàlisi de memòria (L2 cache misses).

5.1 Setup i workflow

```
cd && tar -zxvf ~par0/sessions/lab5.tar.gz && cd lab5/
make
```

1. Seqüencial (referència)

```
sbatch submit-seq.sh mandel-seq-iter-simple -h -o -i 256
mv mandel_image.jpg image_seq.jpg
mv mandel_histogram.out hist_seq.out
```

2. Paral·lel + correctesa

```
sbatch submit-omp.sh mandel-omp-iter-simple 20
cmp mandel_image.jpg image_seq.jpg
cmp mandel_histogram.out hist_seq.out
```

3. Model factors

```
sbatch submit-strong-extrae.sh mandel-omp-iter-simple
# Renombra *-strong-extrae/ abans de la següent!
```

4. Memòria (L2 cache misses) -- NOU en aquest lab

```
sbatch submit-strong-memory.sh mandel-omp-iter-simple
# Genera: *-strong-memory/
# -> traces .prv + l2_misses_....txt
# -> El .txt té: total L2 misses + promig per thread
# Paraver: obrir traça 16 threads i carregar l2d_implicit.cfg
```

5. Strong scaling

```
sbatch submit-strong-omp.sh
```

5.2 Les 3 descomposicions

Totes parteixen de $M[\text{ROWS}][\text{COLS}]$. La matriu és row-major (C): elements consecutius d'una fila són contigus en memòria.

5.2.1 1D Block per columnes

Cada thread rep un bloc contigu de columnes. Si $\text{COLS} \% \text{nt} \neq 0$, els primers rem threads agafen una columna extra.

```
int bs = COLS / nt;
int rem = COLS % nt;
int start = bs*id + (id < rem ? id : rem);
int end = start + bs + (id < rem ? 1 : 0);
for (int py = 0; py < ROWS; py++)
    for (int px = start; px < end; px++)
        M[py][px] = mandelbrot_point(...);
```

- **LB:** molt dolent. Les zones del Mandelbrot amb més iteracions cauen totes al mateix thread.
- **Localitat:** mitjana. Accés per columnes dins una fila = salts en memòria (row-major).
- **False sharing:** baix (blocs contigus grans).
- **L2 misses:** totals creixen amb #threads (blocs més estrets → més pàgines tocades per fila).
- **Resultat:** la pitjor de les tres. Speedup es queda molt per sota de l'ideal. A Paraver: barres molt desiguals, paral·lelisme instantani cau ràpidament.

5.2.2 1D Block-Cyclic per columnes

Columnes repartides cíclicament en blocs de mida BS (= línia de cache / `sizeof(int)` = 64/4 = 16).

```
int BS = 64 / sizeof(int); // = 16
for (int py = 0; py < ROWS; py++)
    for (int px = id*BS; px < COLS; px += BS*nt)
        for (int pxx = px; pxx < min(px+BS, COLS); pxx++)
            M[py][pxx] = mandelbrot_point(...);
```

- **LB:** bo. El repartiment cíclic barreja zones pesades i lleugeres.
- **Localitat:** dolenta. Cada thread salta entre blocs no contigus dins una fila.
- **False sharing:** eliminat si BS = mida de línia de cache (cada bloc cau exactament en línies de cache pròpies).
- **L2 misses:** totals estables ($\approx$ seqüencial) per a ≥ 8 threads gràcies a l'alineament. Anòmals a 2-4 threads per efectes d'arquitectura.
- **Resultat:** molt millor que block. Speedup gairebé lineal. A Paraver: barres gairebé iguals, paral·lelisme alt.

5.2.3 1D Cyclic per files

Cada thread agafa files cíclicament (una fila per torn, round-robin).

```
for (int py = id; py < ROWS; py += nt)
    for (int px = 0; px < COLS; px++)
        M[py][px] = mandelbrot_point(...);
```

- **LB:** excel·lent. Cada thread barreja files de totes les zones.
- **Localitat:** excel·lent. Accés row-major = elements consecutius en memòria.
- **False sharing:** zero. Cada thread treballa sobre files senceres (mai comparteix línia de cache).
- **L2 misses:** totals pràcticament constants ($\approx$ seqüencial) independentment del #threads.
- **Resultat:** la millor de les tres. Speedup el més alt, eficiència la més alta. A Paraver: barres perfectament iguals, paral·lelisme instantani ple durant tota l'execució.

5.3 Comparativa

	Block col.	Block-cyclic col.	Cyclic files
Load balance	Dolent	Bo	Excel·lent
Localitat espacial	Mitjana	Dolenta	Excel·lent
False sharing	Baix	Nul (si BS=cache line)	Nul
L2 misses totals	Creixen	Estables	Estables
Speedup	Molt baix	Gairebé lineal	El més alt

Cyclic per files guanya: accés row-major → menys L2 misses, repartiment cíclic de files → bon balanceig, files senceres per thread → zero false sharing.

5.4 Anàlisi Paraver (16 threads)

Per cada implementació, obrir la traça de 16 threads amb `wxparaver`:

- Useful/Instantaneous parallelism: paral·lelisme al llarg del temps.
- OpenMP/Implicit tasks in `parallel` constructs: durada de cada implicit task per thread.
- Per memòria: carregar `l2d_implicit.cfg` → L2 cache misses per thread.

Important: Ajustar **totes** les traces a la mateixa duració i mida per comparar. Des de la traça més llarga: botó dret → Copy, a les altres: Paste → Time.

5.5 Conceptes clau d'aquest lab

- **Row-major (C):** `M[i][j]` i `M[i][j+1]` són contigus en memòria. Accedir per files = bo per cache. Accedir per columnes = salts.
- **False sharing:** dos threads escriuen a posicions diferents de la mateixa línia de cache → invalidacions constants → overhead. S'evita si cada thread treballa sobre files senceres o blocs alineats a cache line.
- **L2 cache misses:** mètrica de rendiment de memòria. Menys misses = millor ús de la jerarquia de cache. Es mesuren amb `submit-strong-memory.sh` i es visualitzen a Paraver amb `l2d_implicit.cfg`.
- **Implicit tasks:** en aquest lab no hi ha tasques explícites (`task`). Cada thread executa una "implicit task" dins el `#pragma omp parallel`. Modelfactors reporta overheads de fork/join en lloc de synch/sched.



ASSES NORD

González Tablas, 7
08034 Barcelona



93.125.70.02 / 605.411.687



@fiberases
@fib_primer_ases



www.asesacademia.com

