Computadors

Paral·lelisme (1/2)

Grau en Ciència i Enginyeria de Dades

Facultat d'Informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya (UPC)

Creative Commons License

Aquest document utilitza Creative Commons Attribution 3.0 Unported License

Els detalls d'aquesta licencia es poden trovar a
https://creativecommons.org/licenses/by-nc-nd/4.0

Index

- **Introducció**
- **Models de Programació**
  - OpenMP
- Altres mètodes de paral·lelisme a nivell de thread/procés

Introducció al Paral·lelisme

- Paral·lelisme és la capacitate de fer varies accions alhora

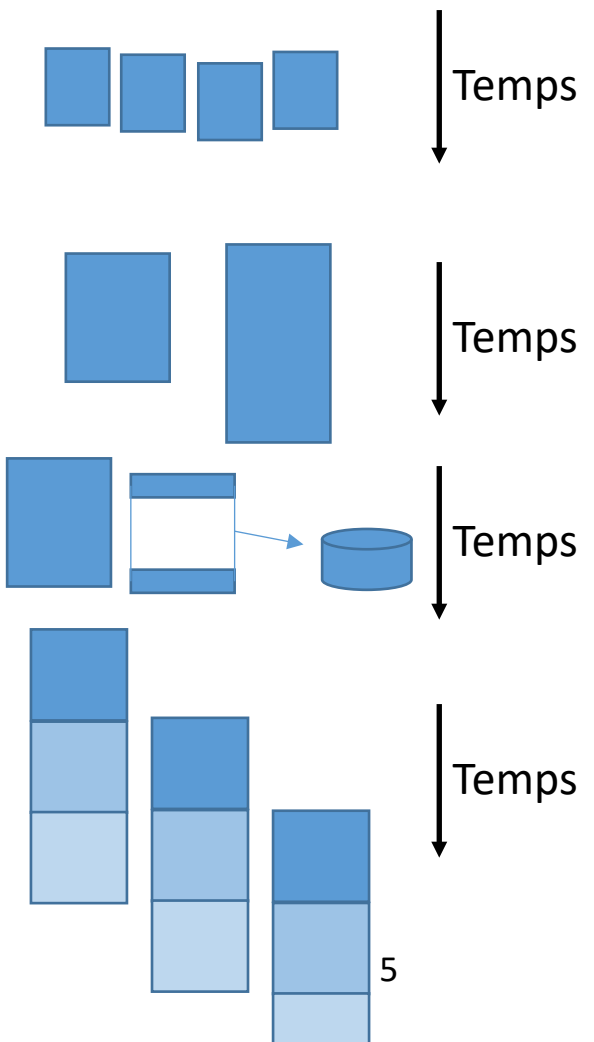
- Computació paral·lela: múltiples computacions executades alhora…
  - …però quin tipus de computacions?

- Resum de “nivells de paral·lelisme”
  - Bit-Level: nivell més petit (e.g. un càlcul de 64-bit vs dos càlculs de 32-bit)
  - Instruction-Level: multiples instruccions per cicle (e.g. 1 instr. ALU i 1 instr. FPU alhora)
  - Data-Level: mateixa operació a múltiples dades (e.g. operacions vectorials)
  - Task-Level: diferentes computacions al mateix o diferents conjunts de dades (e.g. apps complexes)

- **Aquest tema es centra en el paral·lelisme Data-Level i Task-Level**

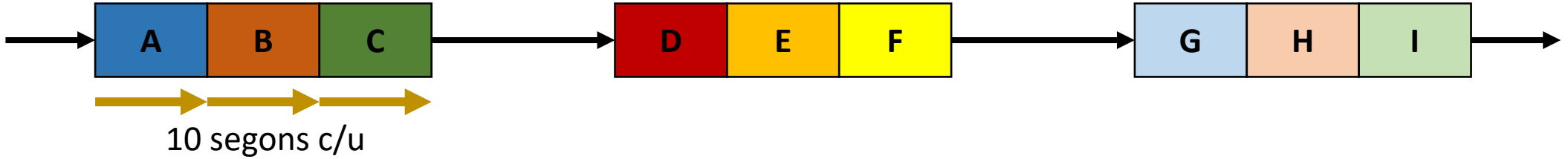
Casos d'ús de Programació Paral·lela

- Explotació del Paral·lelisme
  - La càrrega de feina es pot dividir en múltiples subconjunts de càrregues de feina que es poden processor en paral·lel
    - Sense o amb poques dependències
  - Varies entrades independents es poden processor en paral·lel
    - Sense o amb poques dependencies
- Encapsulament de tasques
  - Mòduls de tasques específiques d'una part de l'applicació, per millorar el rendiment
    - e.g. motor d'IA i física en videojocs
- Eficiència d'E/S
  - Desacoplar la gestió d'E/S de l'execució de la càrrega de treball principal
- Processament en pipelining (mantenir la Qualitat-de-Servei necessari)
  - Millorar la distribució de la càrrega de treball entre les etapes del pipeline per millorar el rendiment

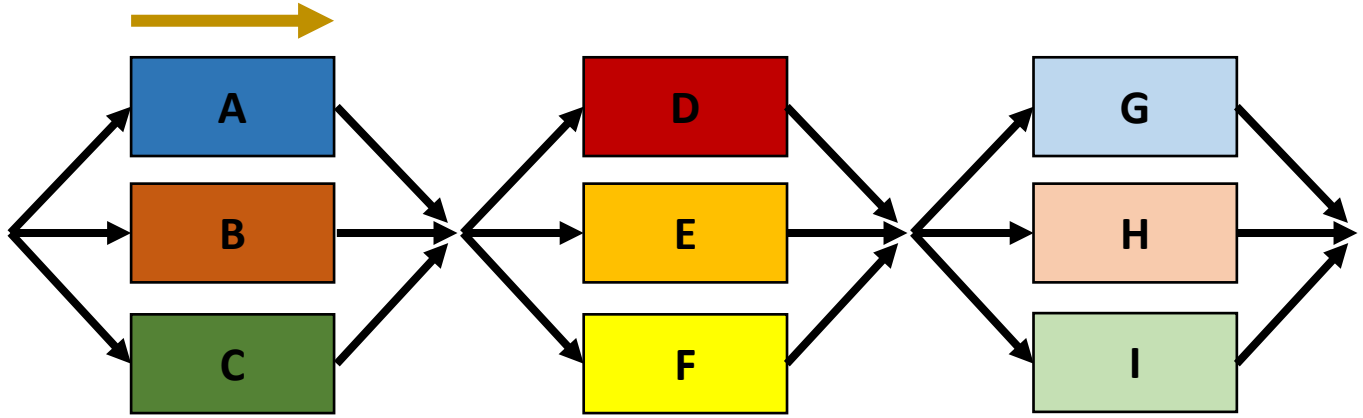


Execució Seqüencial vs Paral·lela

- **Execució Seqüencial:** el codi només pot ser executat per un únic Sw Thread
  És independent de la disponibilitat dels recursos Hw

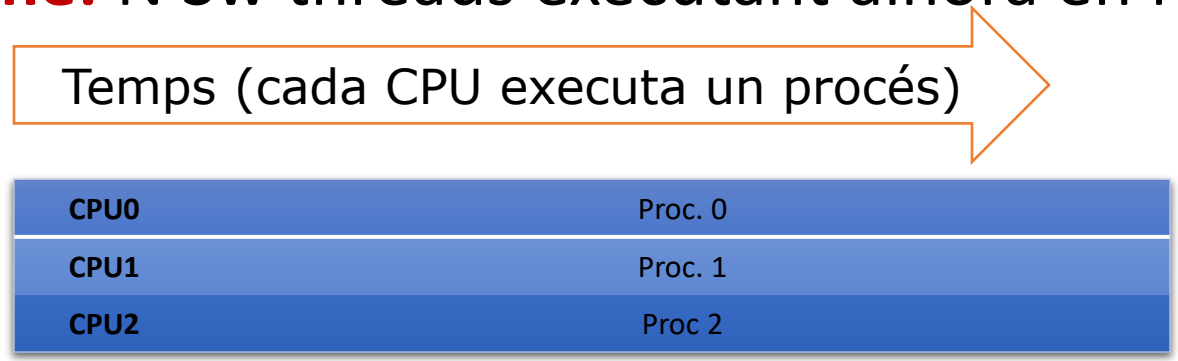


- **Execució Paral·lela:** varis Sw Threads poden executar el codi a la vegada El codi pot mostrar dependències ocasionals o sincronitzacions puntuals

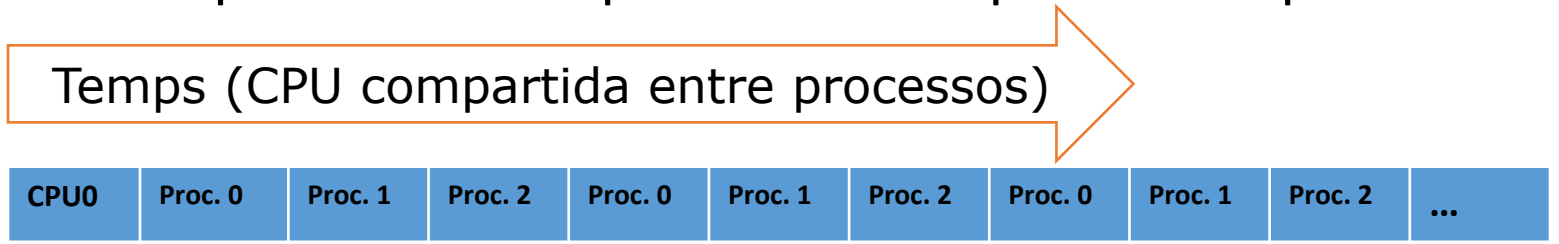


Paral·lelisme vs Concurrència

- **Paral·lelisme:** N Sw threads executant alhora en N Hw threads

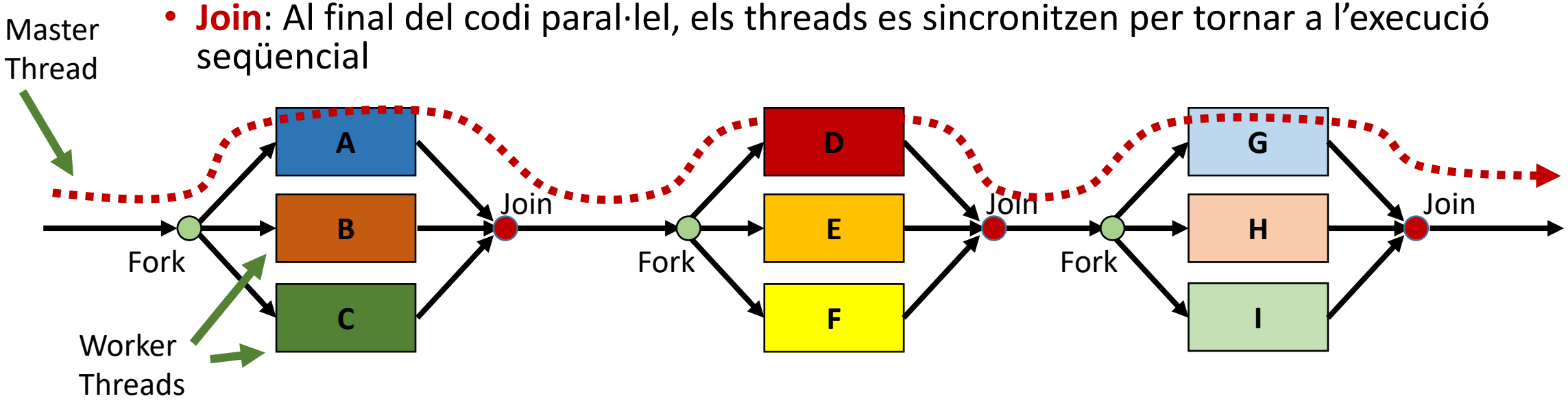


- **Concurrència:** N Sw threads podrien executar en paral·lel, però no hi han recursos Hw suficients
  - El SO selecciona quin Sw thread pot executar i quin ha d'esperar



Model Fork-Join

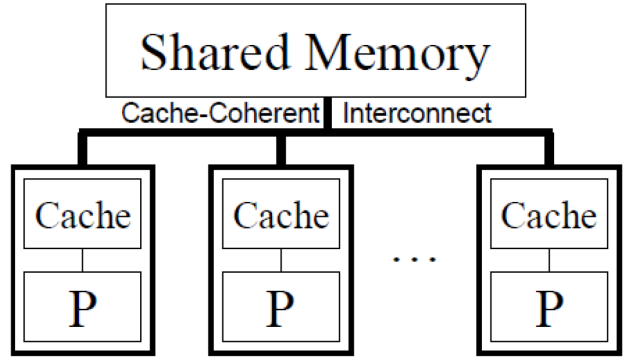
- Programes paral·lels tenen regions de codi que poden ser paral·lelitzades
  - **Fork**: Un troç de codi que es bifurca per ser executat per múltiples threads
  - **Join**: Al final del codi paral·lel, els threads es sincronitzen per tornar a l'execució seqüencial



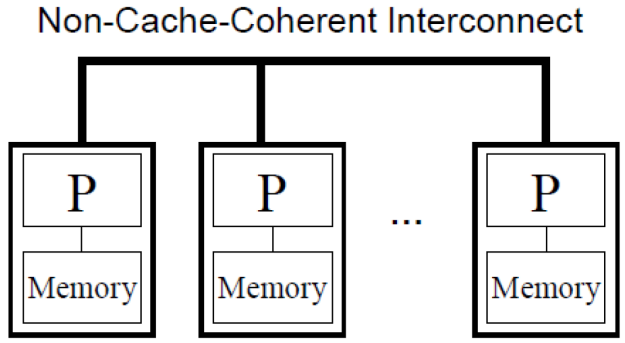
- El master thread genera varis worker threads en regions paral·leles
  - El master thread participa com a worker thread

Models de Programació Paral·lela

- En aquest tema ens centrem en:
  - **OpenMP** (Open specification for Multi Processing)
    - http://www.openmp.org
    - API per escriure programes paral·lels de memòria compartida
      - (C, C++ and Fortran)
    - Memòria compartida; un node d'un cluster



- Hi ha altres:
  - OpenMPI (Message Passing Interface)
    - https://www.open-mpi.org
    - Es veu en PSD (Q4)
    - Memòria distribuida; varis nodes d'un cluster



Figures agafades de "*Using OpenMP: Portable Shared Memory Parallel Programming*"

Dificultats al Compartir Dades

- El correcte comportament del programa NO POT dependre de la seqüència d'execució dels processos o dels threads (**race condition**)
  - Està fora del nostre control

- Les dades compartides HAN D'ESTAR protregits correctamente per garantir la consistència de les dades
  - **Critical sections** (codi que accedeix a dades compartides) ha d'estar correctamente protegit i accedit
  - Realment la secció crítica necessita ser protegida???

- Hem d'evitar els **deadlocks:** dos o més threads esperen per altres threads per sempre

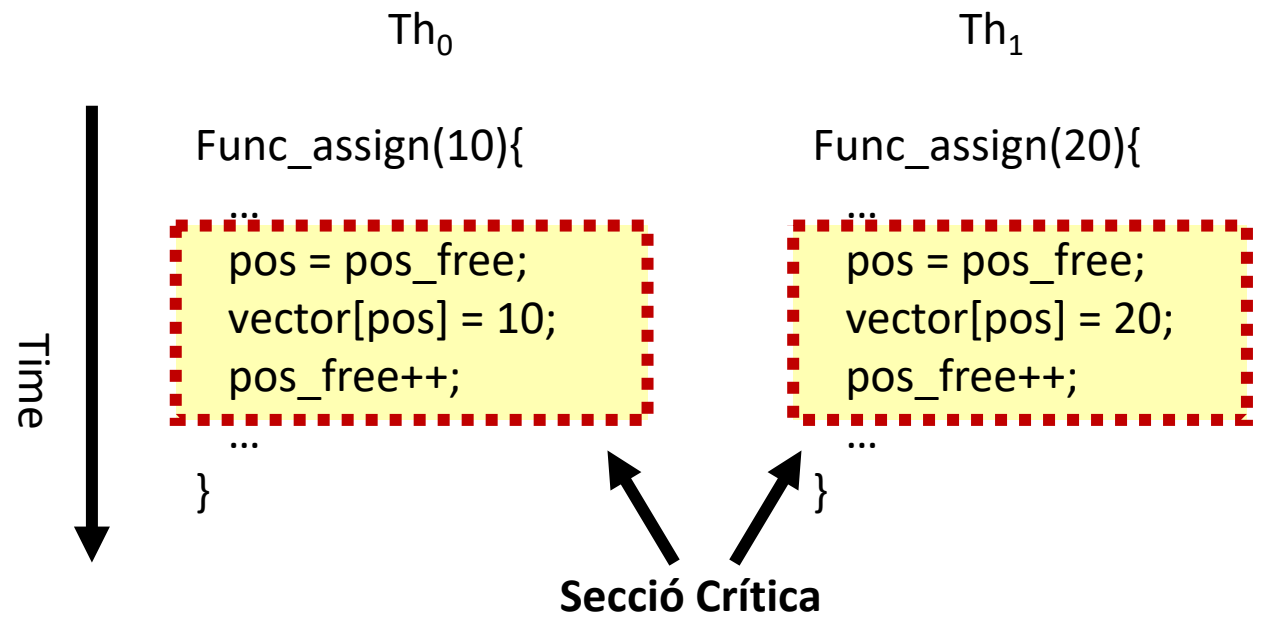
- Sobre protecció de codi paral·lel degrada el rendiment
  - Codi innecessariament protegit

- És difícil garantir totalment el correcte comportament de codis multithreaded
  - Possibles colisions entre threads depèn de la seqüencia d'execució dels processos/threads
    - E.g. El debugger altera la seqüència d'execució (de vegades fins i tot és determinística)

El Gran Problema: Dades Compartides

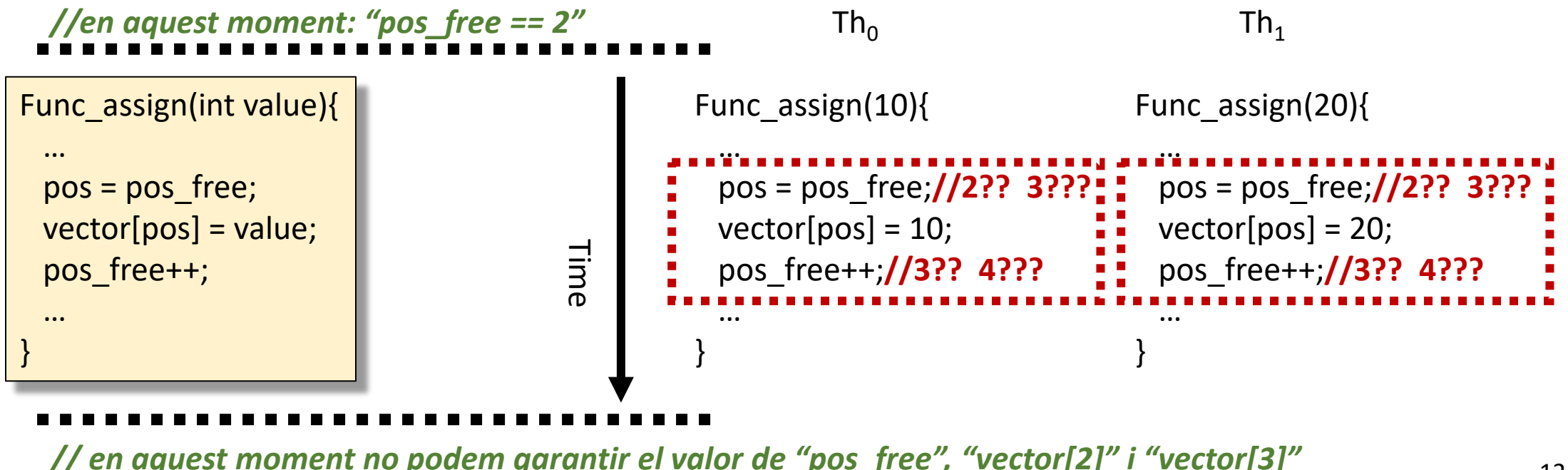
- Els Threads poden accedir a qualsevol dada compartida
- Secció Crítica
  - Codi que accedeix a dades compartides

```
Func_assign(int value){
  …
  pos = pos_free;
  vector[pos] = value;
  pos_free++;
  …
}
```



El Gran Problema: Dades Compartides

- No podem controlar quan els diferents threads poden coincidir en l'accés a les dades compartides
- Les dades compartides HAN DE SER degudament protegides per garantir la consistència de dades



Mecanismes de Sincronització

- **Mutex** (MUTual EXclusion)
  - El thread que executa espera fins que el candau (mutex) está disponible per tenir la seva possessió
- **Barriers**
  - Un thread espera per altres threads fins que arribin al mateix punt de control
- Semaphores
  - Un recurs comú pot ser accedit per múltiples threads
- Spinlocks
  - Comprovació activa d'un candau
- Suport Hardware
  - Atomicament poder llegir i modificar el contingut d'una adreça de memòria

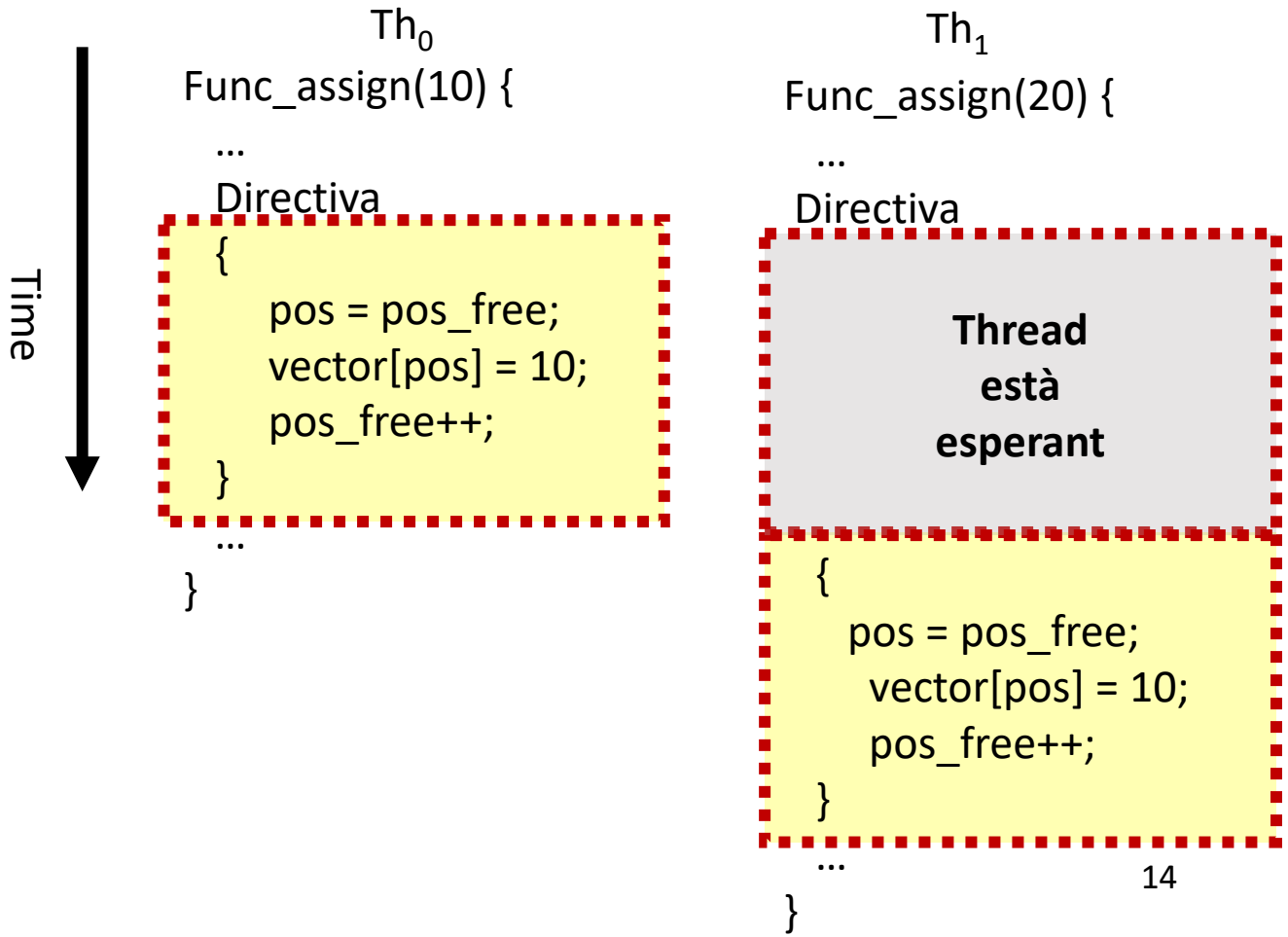
Sincronització: Exclusió Mutua

- Només un únic thread pot estar dins d'una mateixa secció crítica

- Utilització

  Directive [lock name]

- Limitacions
  - Pot produïr excepcions
  - No es pot sortir sense desbloquejar
  - Seccions crítiques sense nom es bloquejan entre sí
    - Lock_name ajuda a distingir

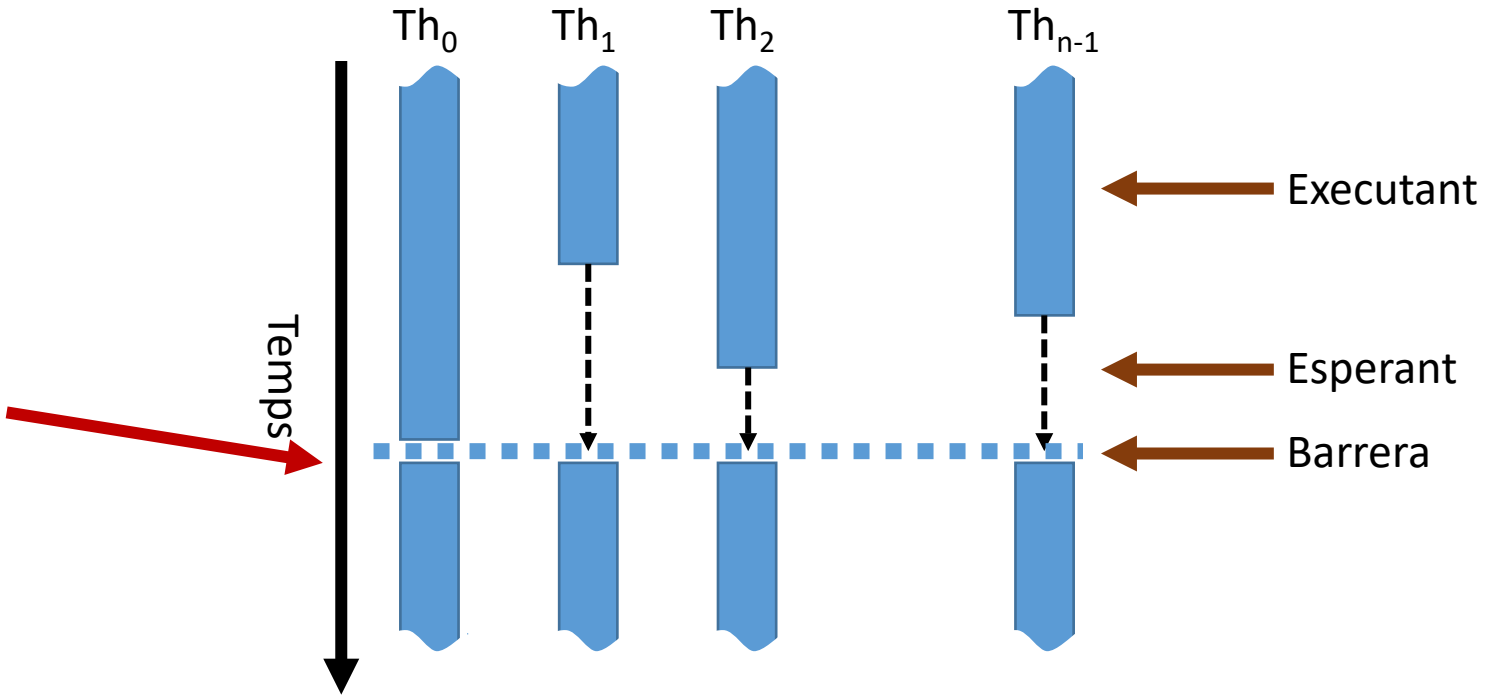


Sincronització: Barreres

- Tots els threads esperan pels altres threads fins que arribin a un punt determinat d'execució
- Hi ha barreres implícites al final de les regions paral·leles

- Utilització

```
{
    …
    Directiva Barrier
    …
}
```



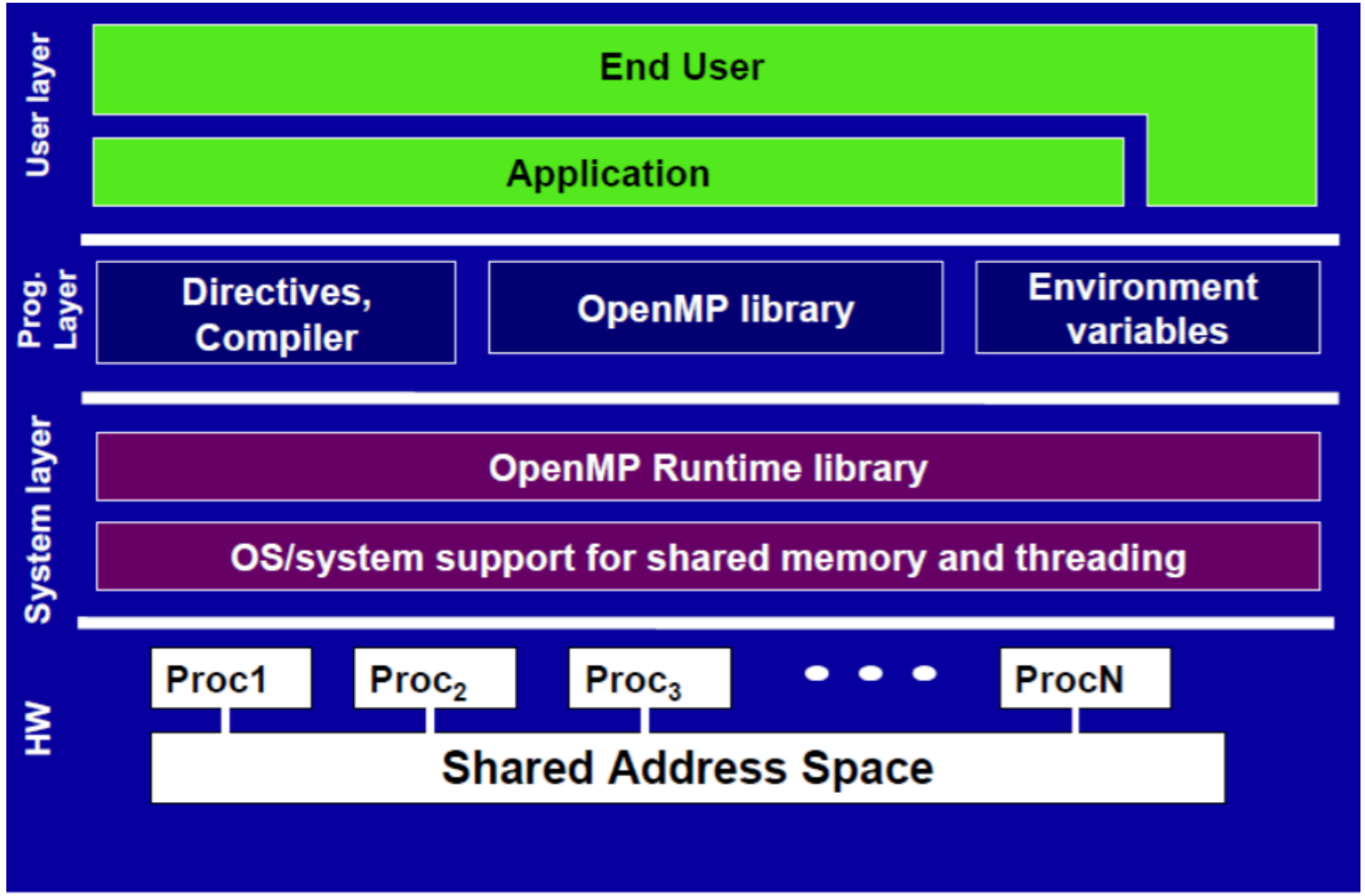
Index

- Introducció
- Models de Programació
  - **OpenMP**
- Altres mètodes de paral·lelisme a nivell de thread/procés

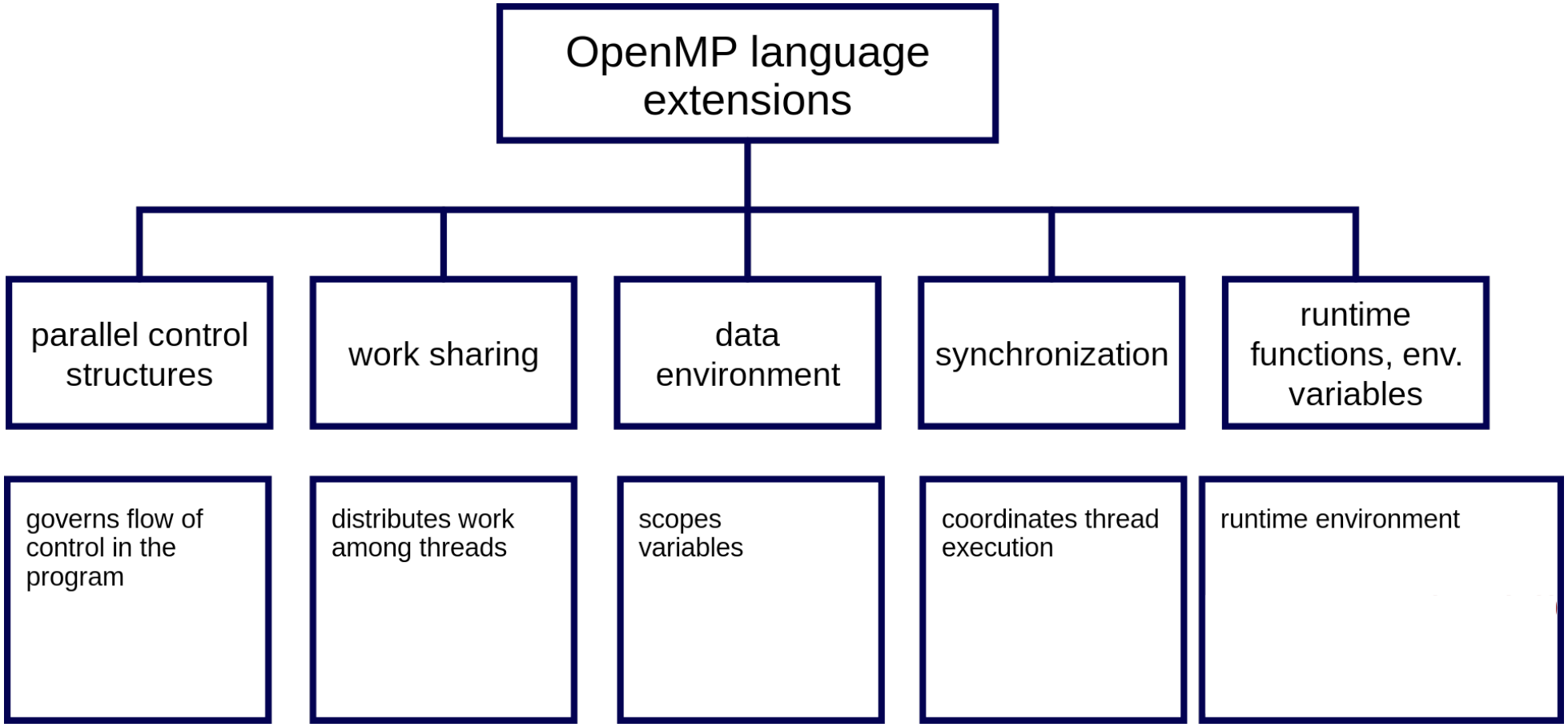
OpenMP

- OpenMP: és una API de codi obert que dona suport a models de programación de memòria compartida
  - Des de la versió OpenMP-1 (finals dels ’90s) a la versió 5.2 (de 2021)
- Directives
  - Informació addicional pel compilador:
  - **#pragma omp** directive [clause [clause …]]
  - Exemple de clause: àmbit d’una variable
    - Private: cada thread té la seva propia copia de la variable
    - Shared: els threads comparteixen una única copia de la variable especificada
    - …
- Format de les rutines & variables d’entorn
  - omp_... & OMP_...

Components OpenMP

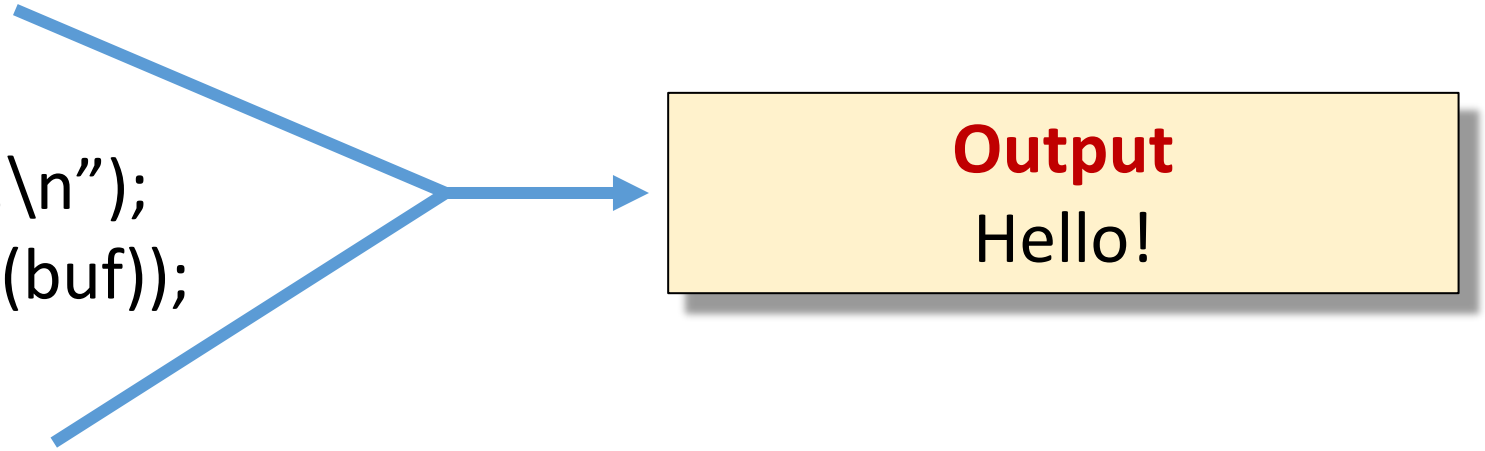


Extensions Clau OpenMP



Exemple: Hello World

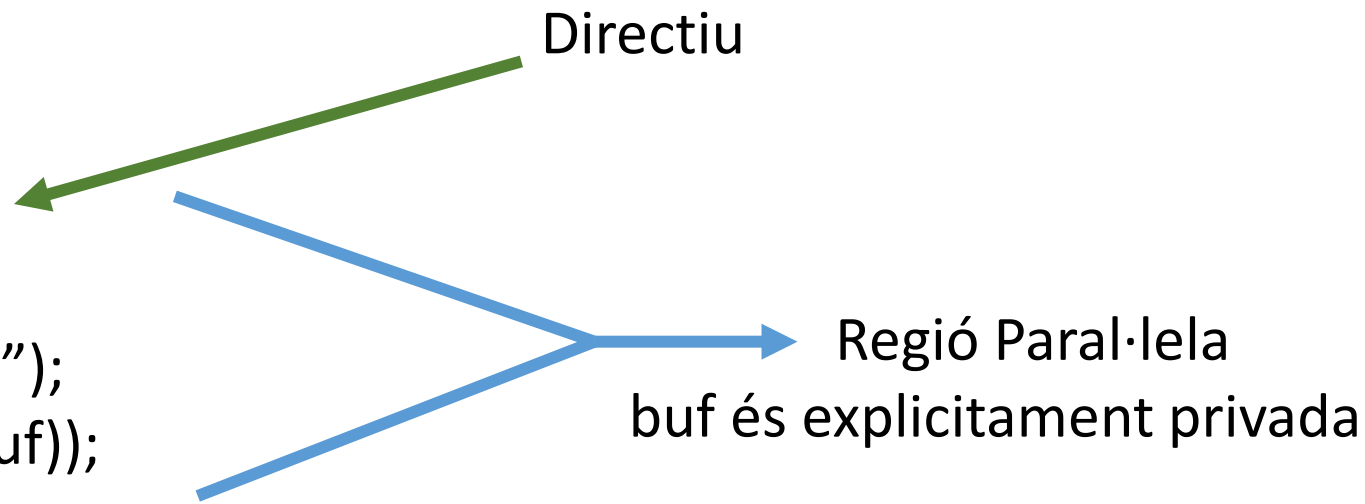
```
int main() {
    …
    sprintf(buf, “Hello!\n”);
    write(1, buf, strlen(buf));
    …
}
```



Estructures de Control: Parallel

- GCC flag: **“-fopenmp”** → Habilita el tractament de directives OpenMP
  - gcc –fopenmp -o hello hello.c

```
#include <omp.h>
int main() {
    …
    #pragma omp parallel
    {
        char buf[200];
        sprintf(buf, “Hello!\n”);
        write(1, buf, strlen(buf));
    }
    …
}
```



Entorn d'Execució

- Variables d'entorn
  - OMP_NUM_THREADS
    - E.g.: export OMP_NUM_THREADS=4

```
#include <omp.h>
int main() {
    …
    #pragma omp parallel
    {
        char buf[200];
        sprintf(buf, "Hello!\n");
        write(1, buf, strlen(buf));
    }
    …
}
```

Output

Hello!
Hello!
Hello!
Hello!

Funcions

- Rutines de la Llibreria
  - omp_get_thread_num()
    - Retorna el ID del thread actual

```
#include <omp.h>
int main() {
    …
    #pragma omp parallel num_threads(4)
    {
        int thid = omp_get_thread_num();
        char buf[200];
        sprintf(buf, “Hello %d!\n”, thid);
        write(1, buf, strlen(buf));
    }
    …
}
```

Output

Hello 0!
Hello 3!
Hello 1!
Hello 2!

Constructors per paral·lelitzar codi

- Paral·lelitzar regions de codi per a què s'executin per threads que s'agafen d'una cua
  - **#pragma omp for**
    For(i=0; i<n; i++){
    …
    }
  - **#pragma omp task**
    Func();
  - **#pragma omp section**
    {
    Func();
    }
- Cas particular: forçar a que només s'executi per un únic thread

  #pragma omp single
  {
  …
  }
- Aquests constructors han d'estar dins d'un codi paral·lelitzat

  #pragma omp parallel
  {
  …
  }

Compartir Càrrega de treball

- Exemple amb for

```
#include <omp.h>
int main() {
    …
    #pragma omp parallel
    {
        #pragma omp single
        {
            numthreads = omp_get_num_threads();
        }
        int thid = omp_get_thread_num();
        char buf[400];
        #pragma omp for
        for (i=0; i<10; i++) {
            sprintf(buf, “Hello i %d th %d!\n”, i, thid);
            write(1, buf, strlen(buf));
        }
    }
    …
}
```

Output

```
Hello i 0 th 0!
Hello i 1 th 0!
Hello i 2 th 1!
Hello i 3 th 1!
Hello i 4 th 1!
Hello i 5 th 2!
Hello i 6 th 2!
Hello i 7 th 2!
Hello i 8 th 3!
Hello i 9 th 3!
```

Àmbit de les Dades

- **private**: Cada thread té la seva còpia de la dada
  - Altres threads no poden accedir a aquesta dada
  - Els canvis només són visibles al thread propietari de la dada
  - Per defecte, els iteradors de bucles (e.g. for) són private
- **shared**: Els threads comparteixen una única copia de la dada
  - Altres threads poden accedir a aquesta dada
  - Els threads poden **llegir i escriure** sobre aquesta dada **simultàneament**
  - Per defecte, totes les variables en una àrea de treball compartida són shared excepte els iteradors de bucle
- L'àmbit per defecte és … depèn 🙄

- **default (shared|none)**: explícitament determina l'àmbit per defecte de les variables d'una regió paral·lela

- Exemple d'utilització
  - **#pragma omp parallel shared(x,y) private(thid)**

Mecanismes de Sincronització

- Mutex (MUTual EXclusion)
  - El thread que executa espera fins que el candau (mutex) está disponible per tenir la seva possessió
- Barriers
  - Un thread espera per altres threads fins que arribin al mateix punt de control
- Semaphores
  - Un recurs comú pot ser accedit per múltiples threads
- Spinlocks
  - Comprovació activa d'un candau
- Suport Hardware
  - Atomicament poder llegir i modificar el contingut d'una adreça de memòria

Algunes clàusules de sincronització

- **Critical**
  - Restringeix l'execució del block associate a un únic thread alhora

    #pragma omp critical [(name) [hint (hint-expression)]]

    structured-block
- **Barrier**
  - Especifica un barrier explícit en un punt concret del codi

    #pragma omp barrier
  - Per tasks existeix la variant "#pragma omp taskwait"
- **Atomic**
  - Garanteix que una variable compartida s'accedeix de manera atòmica

    #pragma omp atomic

    expression

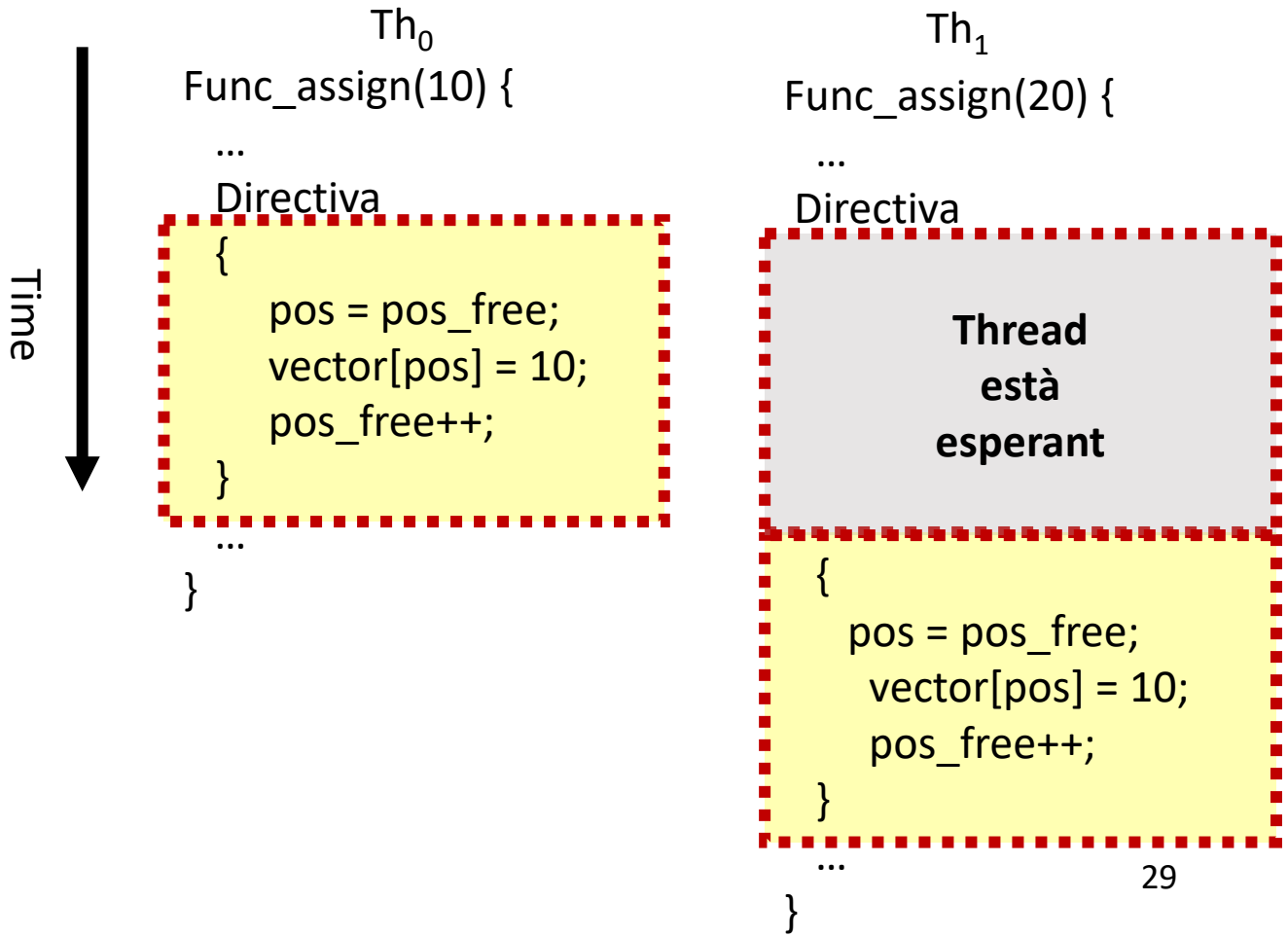
Sincronització: Exclusió Mutua

- Només un únic thread pot estar dins d'una mateixa secció crítica

- Utilització

  Directive [lock name]

- Limitacions
  - Pot produïr excepcions
  - No es pot sortir sense desbloquejar
  - Seccions crítiques sense nom es bloquejan entre sí
    - Lock_name ajuda a distingir



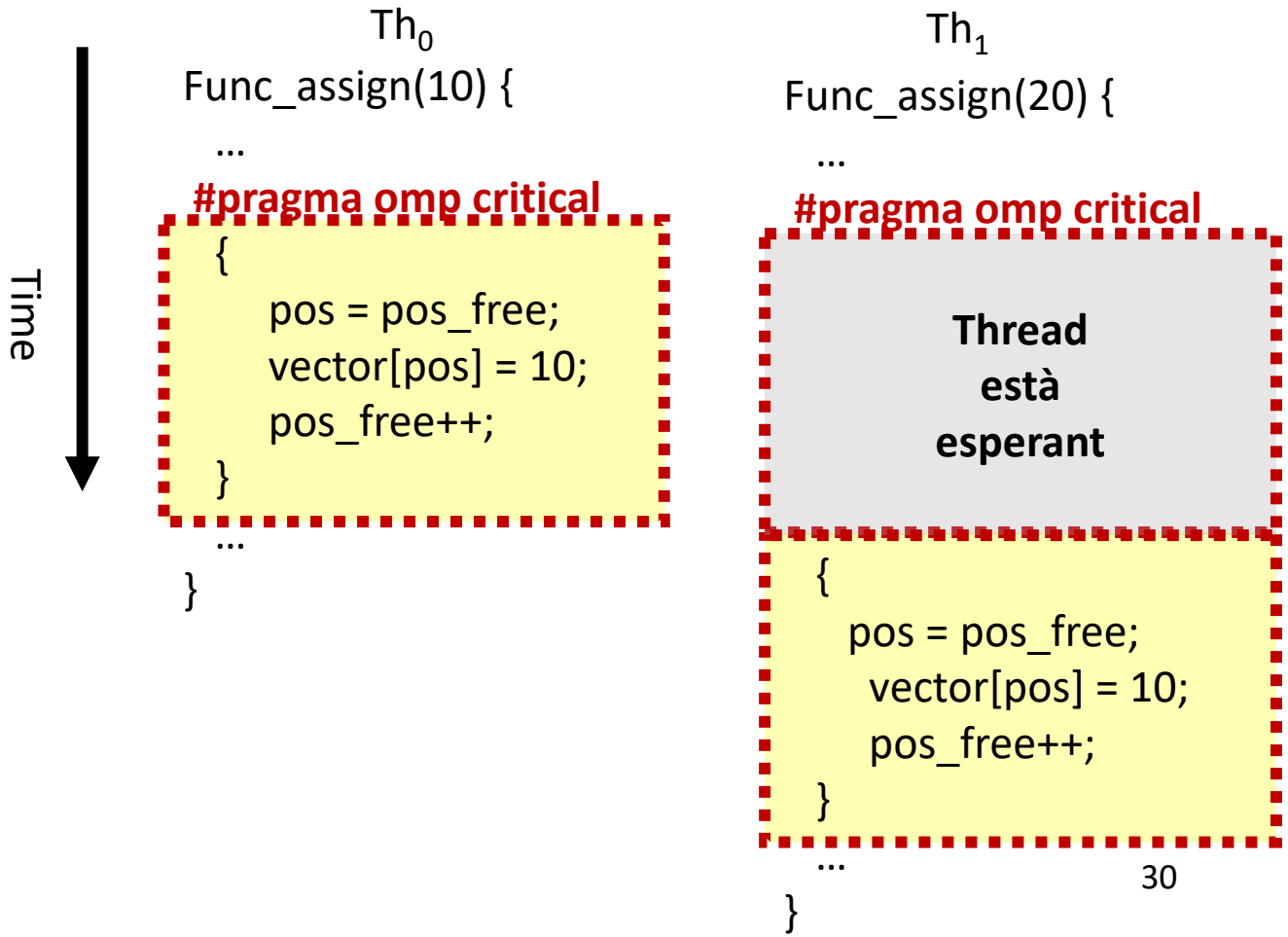
Sincronització: Exclusió Mutua

- Només un únic thread pot estar dins d'una mateixa secció crítica

- Utilització

  Directive [lock name]

- Limitacions
  - Pot produïr excepcions
  - No es pot sortir sense desbloquejar
  - Seccions crítiques sense nom es bloquejan entre sí
    - Lock_name ajuda a distingir



Bibliografia

- OpenMPI
  - https://www.open-mpi.org/

- Using OpenMP: Portable Shared Memory Parallel Programming

- Ho veureu en profunditat en l'assignatura PSD (Q4, GCED)