CPA – Computació Paral·lela

Grau en Enginyeria Informàtica

S2. Programació amb OpenMP

J. M. Alonso, P. Alonso, F. Alvarruiz, I. Blanquer,
J. Ibáñez, E. Ramos, J. E. Román

Departament de Sistemes Informàtics i Computació
Universitat Politècnica de València

Curs 2024/25

etsinf

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Contingut

Apartat 1

Conceptes Bàsics

- Model de Programació
- Exemple Simple

L'Especificació OpenMP

Estàndard de facto per a programació en memòria compartida

`http://www.openmp.org`

Especificacions:

- Fortran: 1.0 (1997), 2.0 (2000)
- C/C++: 1.0 (1998), 2.0 (2002)
- Fortran/C/C++: 2.5 (2005), 3.0 (2008), 3.1 (2011), 4.0 (2013), 4.5 (2015), 5.0 (2018)

Antecedents:

- Estàndard ANSI X3H5 (1994)
- HPF, CMFortran

Model de Programació

La programació en OpenMP es basa principalment en directives del compilador

Exemple

```
void daxpy(int n, double a, double *x, double *y, double *z)
{
    int i;
    #pragma omp parallel for
    for (i=0; i<n; i++)
        z[i] = a*x[i] + y[i];
}
```

Avantatges

- Facilita la migració (el compilador ignora els `pragma`)
- Permet la paralel·lització incremental
- Permet l'optimització per part del compilador

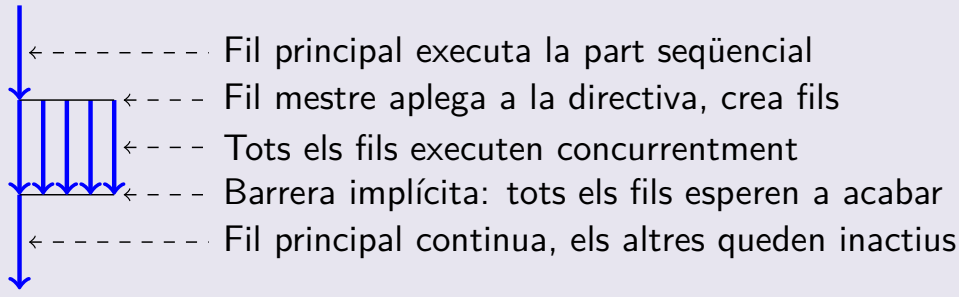
A més: funcions (veure `omp.h`) i variables d'entorn

Model d'Execució

El model d'execució d'OpenMP segueix un esquema *fork-join*

Hi ha directives per a crear fils i dividir el treball

Esquema



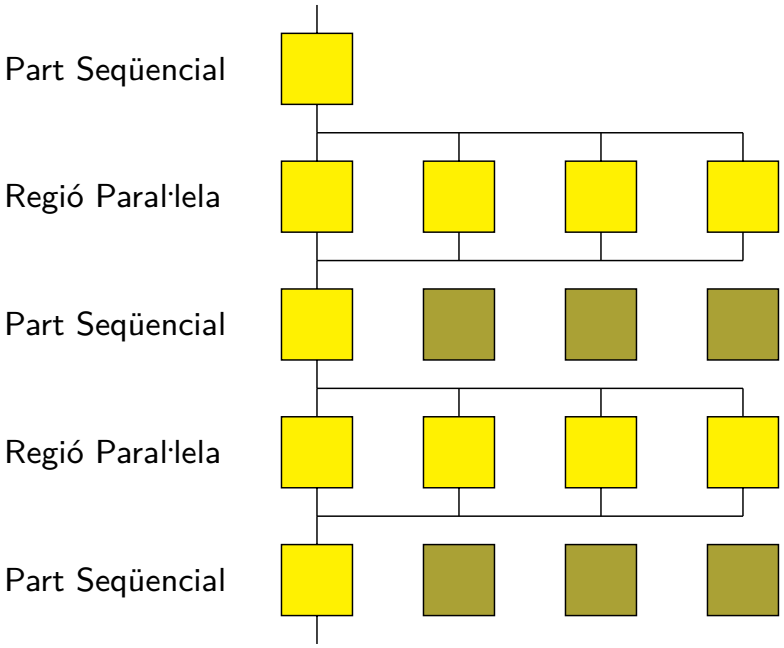
Les directives defineixen regions paral·leles

Altres directives/clàusules:

- Indicar tipus de variable: `private`, `shared`, `reduction`
- Sincronització: `critical`, `barrier`

Model d'Execució - Fils

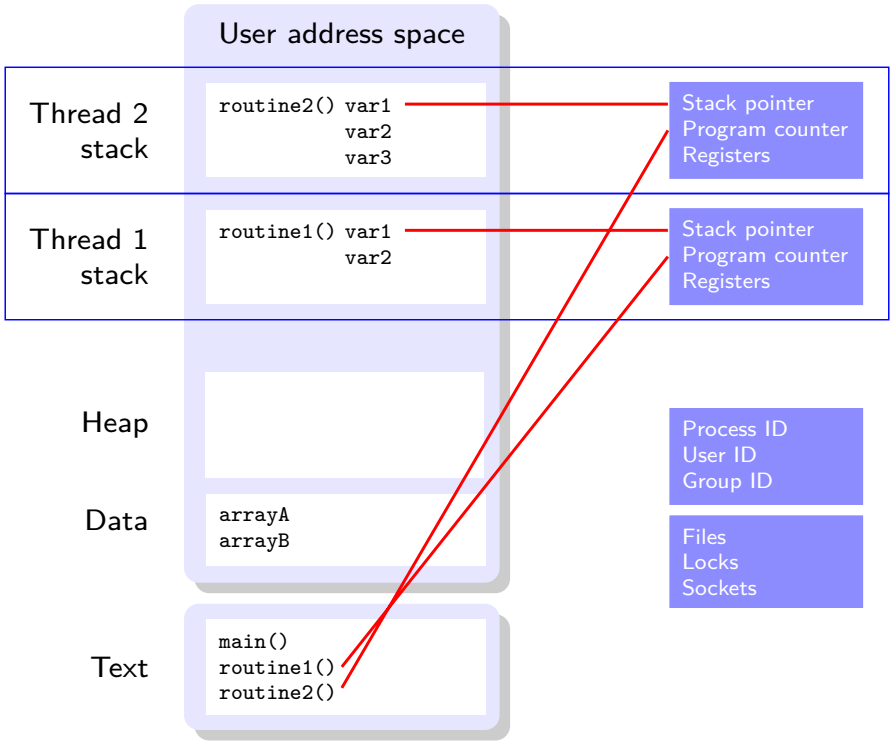
En OpenMP els fils inactius no es destrueixen, queden a l'espera de la següent regió paral·lela



Els fils creats per una directiva s'anomenen equip (*team*)

Model d'Execució - Memòria

Cada fil té el seu propi context d'execució (incloent la pila)



Sintaxi

Directives:

```
#pragma omp <directiva> [clausula [...]]
```

Ús de funcions:

```
#include <omp.h>
...
iam = omp_get_thread_num();
```

Compilació condicional: la macro `_OPENMP` conté la data de la versió d'OpenMP suportada, p.e. 201107

Compilació:

```
gcc> gcc -fopenmp prg-omp.c
sun> cc -xopenmp -x03 prg-omp.c
intel> icc -qopenmp prg-omp.c
```


Exemple Simple

Exemple

```
void daxpy(int n, double a, double *x, double *y, double *z)
{
    int i;
    #pragma omp parallel for
    for (i=0; i<n; i++)
        z[i] = a*x[i] + y[i];
}
```

- En arribar a la directiva `parallel` es creen els fils (si no s'han creat abans)
- Les iteracions del bucle es reparteixen entre els fils
- Per defecte, totes les variables són compartides, excepte la variable del bucle (`i`) que és privada
- En finalitzar se sincronitzen tots els fils

Nombre i Identificador de Fil

El nombre de fils es pot especificar:

- Amb la clàusula `num_threads`
- Amb la funció `omp_set_num_threads()` *abans* de la regió paral·lela
- En executar, amb `OMP_NUM_THREADS`

Funcions útils:

- `omp_get_num_threads()`: retorna el nombre de fils
- `omp_get_thread_num()`: retorna l'identificador de fil (comencen en 0, el fil principal és sempre 0)

```
omp_set_num_threads(3);
printf("fils abans = %d\n",omp_get_num_threads());
#pragma omp parallel for
for (i=0; i<n; i++) {
  printf("fils = %d\n",omp_get_num_threads());
  printf("jo soc %d\n",omp_get_thread_num());
}
```


Apartat 2

Paral·lelització de Bucles

- Directiva `parallel for`
- Tipus de Variables
- Millora de Prestacions

Directiva `parallel for`

Es paral·lelitza el bucle que va a continuació

C/C++

```
#pragma omp parallel for [clausula [...]]
for (index=first; test_expr; increment_expr) {
    // cos del bucle
}
```

OpenMP imposa restriccions al tipus de bucle, per exemple:

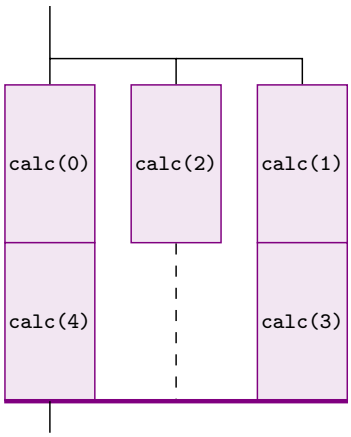
```
for (i=0; i<n && !trobat; i++)
  if (x[i]==elem) trobat=1;
```


Exemple de `parallel for`

Una possible execució amb 3 fils

Bucle senzill

```
#pragma omp parallel for
for (i=0; i<5; i++) {
  a[i] = calc(i);
}
```



Barrera implícita al finalitzar la construcció `parallel for`

Variables:

- a: accés concurrent, però no hi ha més d'un fill accedint a la mateixa posició
- i: distint valor en cada fil → necessiten una còpia privada

Tipus de Variables

Es classifiquen les variables segons el seu abast (*scope*)

- Privades: cada fil té una rèplica diferent
- Compartides: tots els fils poden llegir i escriure

Font comuna d'errors: no triar correctament l'abast

L'abast es pot modificar amb clàusules afegides a les directives:

- `private`, `shared`
- `reduction`
- `firstprivate`, `lastprivate`

`private`, `shared`, `default`

Si no s'especifica l'abast d'una variable, per defecte és `shared`

Excepcions (`private`):

- Índex del bucle que es paral·lelitza
- En subrutines invocades, les variables locals (excepte si es declaren `static`)
- Variables automàtiques declarades dins del bucle

Clàusula `default`

- `default(none)` obliga a especificar l'abast de totes

private, shared

private

```
suma = 0;
#pragma omp parallel for private(suma)
for (i=0; i<n; i++) {
    suma = suma + x[i]*x[i];
}
```

Incorrecte: després del bucle només existeix la suma del fil principal (amb valor 0) - a més, les còpies de cada fil no s'inicialitzen

shared

```
suma = 0;
#pragma omp parallel for shared(suma)
for (i=0; i<n; i++) {
    suma = suma + x[i]*x[i];
}
```

Incorrecte: condició de carrera en llegir/escriure `suma`

reduction

Per a realitzar reduccions amb operadors commutatius i associatius (`+`, `*`, `-`, `&`, `|`, `^`, `&&`, `||`, `max`, `min`)

reduction(redn_oper: var_list)

```
suma = 0;
#pragma omp parallel for reduction(+:suma)
for (i=0; i<n; i++) {
    suma = suma + x[i]*x[i];
}
```

Cada fil realitza una porció de la suma, al final es combinen en la suma total

És com una variable privada, però:

- Al final, els valors privats es combinen
- S'inicialitza correctament (a l'element neutre de l'operació)

`firstprivate`, `lastprivate`

Les variables privades es creen sense un valor inicial i després del bloc `parallel` queden indefinides

- `firstprivate`: inicialitza al valor del fil principal
- `lastprivate`: es queda amb el valor de l'"última" iteració

Exemple

```
alpha = 5.0;
#pragma omp parallel for firstprivate(alpha) lastprivate(i)
for (i=0; i<n; i++) {
    z[i] = alpha*x[i];
}
k = i;    /* i té el valor n */
```

El comportament per defecte intenta evitar còpies innecessàries

Garantir Suficient Treball

La paral·lelització de bucles suposa un *overhead*: activació i desactivació de fils, sincronització

En bucles molt senzills, l'overhead pot ser major que el temps de càlcul

Clàusula `if`

```
#pragma omp parallel for if(n>5000)
for (i=0; i<n; i++)
    z[i] = a*x[i] + i[i];
```

Si l'expressió és falsa, el bucle s'executa seqüencialment

Aquesta clàusula es podria usar també per a evitar dependències de dades detectades en temps d'execució

Bucles Niats

Cal posar la directiva abans del bucle a paral·lelitzar

Cas 1

```
#pragma omp parallel for \
        private(j)
for (i=0; i<n; i++) {
    for (j=0; j<m; j++) {
        // cos del bucle
    }
}
```

Cas 2

```
for (i=0; i<n; i++) {
    #pragma omp parallel for
    for (j=0; j<m; j++) {
        // cos del bucle
    }
}
```

- En el primer cas, les iteracions de `i` es reparteixen; el mateix fil executa el bucle `j` complet
- En el segon cas, en cada iteració de `i` s'activen i desactiven els fils; hi ha `n` sincronitzacions

Bucles Niats - Intercanvi

Habitualment es recomana paral·lelitzar el més extern

- En casos en què les dependències de dades impedeixen açò, es pot intentar intercanviar els bucles

Codi seqüencial

```
for (j=1; j<n; j++)
    for (i=0; i<n; i++)
        a[i][j] = a[i][j] + a[i][j-1];
```

Codi paral·lel amb bucles intercanviats

```
#pragma omp parallel for private(j)
for (i=0; i<n; i++)
    for (j=1; j<n; j++)
        a[i][j] = a[i][j] + a[i][j-1];
```

Aquestes modificacions poden tenir impacte en l'ús de la cache

Planificació

Idealment, totes les iteracions costen el mateix i a cada fil se li assigna aproximadament el mateix nombre d'iteracions

En la realitat, es pot produir desequilibri de la càrrega amb la conseqüent pèrdua de prestacions

---

En OpenMP és possible especificar la planificació

Pot ser de dos tipus:

- Estàtica: les iteracions s'assignen a fils a priori
- Dinàmica: l'assignació s'adapta a l'execució actual

La planificació es realitza a nivell de rangs contigus d'iteracions (*chunks*)

Planificació - Clàusula `schedule`

Sintaxi de la clàusula de planificació:

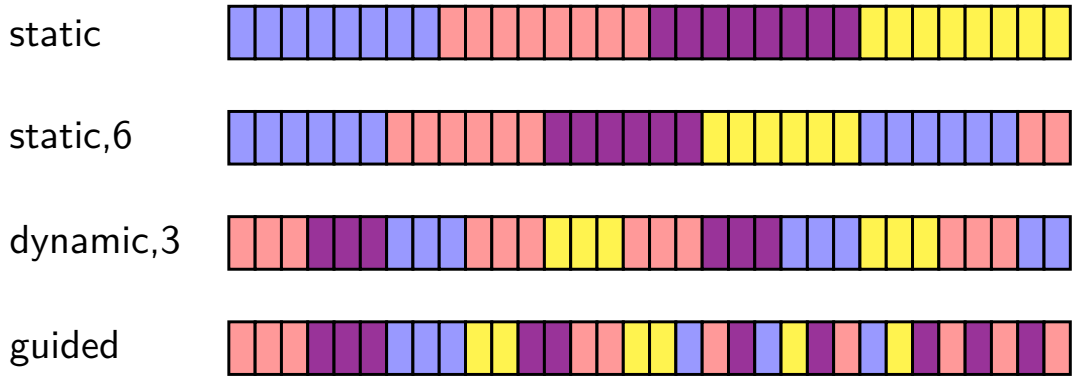
```
schedule(type[,chunk])
```

- `static` (sense `chunk`): a cada fil se li assigna estàticament un rang aproximadament igual
- `static` (amb `chunk`): assignació cíclica (*round-robin*) de rangs de grandària `chunk`
- `dynamic` (`chunk` opcional, per defecte 1): es van assignant segons es demanen (*first-come, first-served*)
- `guided` (`chunk` mínim opcional): com `dynamic` però la grandària del rang va decreixent exponencialment ($\propto n_{rest}/n_{fils}$)
- `runtime`: s'especifica en temps d'execució amb la variable d'entorn `OMP_SCHEDULE`

Planificació - Exemple

Exemple: bucle de 32 iteracions executat amb 4 fils

```
$ OMP_NUM_THREADS=4 OMP_SCHEDULE=guided ./prog
```



Apartat 3

Regions Paral·leles

- Directiva `parallel`
- Repartiment del Treball

Directiva `parallel`

S'executa de forma replicada el bloc que va a continuació

C/C++

```
#pragma omp parallel [clause [clause ...]]
{
    // bloc
}
```

Algunes clàusules permeses són: `private`, `shared`, `default`, `reduction`, `if`

Exemple - s'escriuen tantes línies com fils

```
#pragma omp parallel private(myid)
{
  myid = omp_get_thread_num();
  printf("soc el fil %d\n",myid);
}
```


Repartiment del Treball

A més de l'execució replicada, sol ser necessari repartir el treball entre els fils

- Cada fil opera sobre una part d'una estructura de dades, o bé
- Cada fil realitza una operació diferent

Possibles formes de realitzar el repartiment:

- Segons l'identificador de fil
- Cua de tasques paral·leles
- Mitjançant construccions OpenMP específiques

Repartiment segons Identificador de Fil

S'utilitzen les funcions

- `omp_get_num_threads()`: retorna el nombre de fils
- `omp_get_thread_num()`: retorna l'identificador de fil

per a determinar què part del treball realitza cada fil

Exemple - identificadors de fil

```
#pragma omp parallel private(myid)
{
    nthreads = omp_get_num_threads();
    myid = omp_get_thread_num();
    dowork(myid, nthreads);
}
```


Repartiment mitjançant Cua de Tasques Paral·leles

Una cua de tasques paral·leles és una estructura de dades compartida que conté una llista de "tasques" a realitzar

- Les tasques es poden processar concurrentment
- Qualsevol tasca pot realitzar-se per qualsevol fil

```
int get_next_task() {
    static int index = 0;
    int result;
    #pragma omp critical
    {   if (index==MAXIDX) result=-1;
        else { index++; result=index; }
    }
    return result;
}
...
    int myindex;
    #pragma omp parallel private(myindex)
    {   myindex = get_next_task();
        while (myindex>-1) {
            process_task(myindex);
            myindex = get_next_task();
        }
    }
```

Construccions de Repartiment del Treball

Les solucions anteriors són bastant primitives

- El programador s'encarrega de dividir el treball
- Codi fosc i complicat en programes llargs

OpenMP disposa de construccions específiques (*work-sharing constructs*)

Hi ha de tres tipus:

- Construcció `for` per a repartir iteracions de bucles
- Seccions per a distingir porcions del codi
- Codi a executar per un sol fil

Hi ha una barrera implícita al final del bloc

Construcció `for`

Reparteix de forma automàtica les iteracions del bucle

Exemple de bucle compartit

```
#pragma omp parallel
{
    ...
    #pragma omp for
    for (i=1; i<n; i++)
        b[i] = (a[i] + a[i-1]) / 2.0;
}
```

El bucle es *comparteix* entre els fils, en lloc de fer-lo replicat

Les directives `parallel` i `for` es poden combinar en una

Construcció de Bucle - Clàusula `nowait`

Quan hi ha diversos bucles independents dins d'una regió paral·lela, `nowait` evita la barrera implícita

Bucles sense barrera

```
void a8(int n, int m, float *a, float *b, float *y, float *z)
{
    int i;
    #pragma omp parallel
    {
        #pragma omp for nowait
        for (i=1; i<n; i++)
            b[i] = (a[i] + a[i-1]) / 2.0;

        #pragma omp for
        for (i=0; i<m; i++)
            y[i] = sqrt(z[i]);
    }
}
```


Construcció `sections`

Per a trossos de codi independents difícils de paral·lelitzar

- Individualment suposen molt poc treball, o bé
- Cada fragment és inherentment seqüencial

Pot també combinar-se amb `parallel`

Exemple de seccions

```
#pragma omp parallel sections
{
    #pragma omp section
      Xaxis();
    #pragma omp section
      Yaxis();
    #pragma omp section
      Zaxis();
}
```

Un fil pot executar més d'una secció

Clàusules: `private`, `first/lastprivate`, `reduction`, `nowait`

Construcció `single`

Fragments de codi que han d'executar-se per un sol fil

Exemple `single`

```
#pragma omp parallel
{
    #pragma omp single nowait
      printf("Comença work1\n");
    work1();

    #pragma omp single
      printf("Finalitzant work1\n");

    #pragma omp single nowait
      printf("Acabat work1, comença work2\n");
    work2();
}
```

Algunes clàusules permeses: `private`, `firstprivate`, `nowait`

Apartat 4

Sincronització

- Exclusió Mútua
- Altre Tipus de Sincronització

Condició de Carrera (1)

El següent exemple il·lustra una condició de carrera

Cerca de màxim

```
cur_max = -100000;
#pragma omp parallel for
for (i=0; i<n; i++) {
    if (a[i] > cur_max) {
        cur_max = a[i];
    }
}
```

Seqüència amb resultat incorrecte:

Fil 0: llig a[i]=20, llig cur_max=15
Fil 1: llig a[i]=16, llig cur_max=15
Fil 0: comprova a[i]>cur_max, escriu cur_max=20
Fil 1: comprova a[i]>cur_max, escriu cur_max=16

Condició de Carrera (2)

Hi ha casos en què l'accés concurrent no produeix condició de carrera

Exemple d'accés concurrent sense condició de carrera

```
trobat = 0;
#pragma omp parallel for
for (i=0; i<n; i++) {
    if (a[i] == valor) {
        trobat = 1;
    }
}
```

Encara que diversos fils escriguen alhora, el resultat és correcte

---

En general, es necessiten mecanismes de sincronització:

- Exclusió mútua
- Altre tipus de sincronització

Exclusió Mútua

L'exclusió *mútua* en l'accés a les variables compartides evita qualsevol condició de carrera

OpenMP proporciona tres construccions diferents:

- Seccions crítiques: directiva `critical`
- Operacions atòmiques: directiva `atomic`
- Panys: rutines `*_lock`

Directiva `critical` (1)

En l'exemple anterior, l'accés en exclusió mútua a la variable `cur_max` evita la condició de carrera

Cerca de màxim, sense condició de carrera

```
cur_max = -100000;
#pragma omp parallel for
for (i=0; i<n; i++) {
    #pragma omp critical
    if (a[i] > cur_max) {
        cur_max = a[i];
    }
}
```

Quan un fil arriba al bloc `if` (la secció crítica), espera fins que no hi ha un altre fil executant-ho al mateix temps

OpenMP garanteix progrés (almenys un fil dels quals espera entra en la secció crítica) però no espera limitada

Directiva `critical` (2)

En la pràctica, l'exemple anterior resulta ser seqüencial

Tenint en compte que `cur_max` mai es decrementa, es pot plantejar la següent millora

Cerca de màxim, millorat

```
cur_max = -100000;
#pragma omp parallel for
for (i=0; i<n; i++) {
    if (a[i] > cur_max) {
        #pragma omp critical
        if (a[i] > cur_max)
            cur_max = a[i];
    }
}
```

El segon `if` és necessari perquè s'ha llegit `cur_max` fora de la secció crítica

Esta solució entra en la secció crítica amb menor freqüència

Directiva `critical` amb Nom

En afegir un nom, es permet tenir diverses seccions crítiques sense relació entre elles

Cerca de màxim i mínim

```
cur_max = -100000;
cur_min = 100000;
#pragma omp parallel for
for (i=0; i<n; i++) {
    if (a[i] > cur_max) {
        #pragma omp critical (maxim)
        if (a[i] > cur_max)
            cur_max = a[i];
    }
    if (a[i] < cur_min) {
        #pragma omp critical (minim)
        if (a[i] < cur_min)
            cur_min = a[i];
    }
}
```

Directiva atomic

Operacions atòmiques de lectura-modificació-escritura

```
#pragma omp atomic
x <binop>= expr
```

```
#pragma omp atomic
x++, ++x, x--, --x
```

on <binop> pot ser +, *, -, /, %, &, |, ^, <<, >>

Exemple atomic

```
#pragma omp parallel for shared(x, index, n)
for (i=0; i<n; i++) {
    #pragma omp atomic
    x[index[i]] += work1(i);
}
```

El codi és molt més eficient que amb `critical` i permet actualitzar elements de `x` en paral·lel

Directiva barrier

En arribar a una barrera, els fils esperen a que arriben tots

```
#pragma omp parallel private(index)
{
    index = generate_next_index();
    while (index>0) {
        add_index(index);
        index = generate_next_index();
    }
    #pragma omp barrier
    index = get_next_index();
    while (index>0) {
        process_index(index);
        index = get_next_index();
    }
}
```

Se sol usar per a assegurar que una fase l'han acabada tots abans de passar a la següent fase

Directiva `ordered`

Per a permetre que una porció del codi de les iteracions s'execute en l'ordre seqüencial original

Exemple `ordered`

```
#pragma omp parallel for ordered
for (i=0; i<n; i++) {
    a[i] = ...   /* calcul complex */
    #pragma omp ordered
    fprintf(fd, "%d %g\n", i, a[i]);
}
```

Restriccions:

- Si un bucle paral·lel conté una directiva `ordered`, cal afegir la clàusula `ordered` també al bucle
- Només es permet una única secció `ordered` per iteració