CPA – Computación Paralela

Grado en Ingeniería Informática

S2. Programación con OpenMP

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

Departament de Sistemes Informàtics i Computació
Universitat Politècnica de València

Curso 2020/21

etsinf

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Contenido

Apartado 1

Conceptos Básicos

- Modelo de Programación
- Ejemplo Simple

La Especificación OpenMP

Estándar de facto para programación en memoria compartida

`http://www.openmp.org`

Especificaciones:

- 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)

Antecedentes:

- Estándar ANSI X3H5 (1994)
- HPF, CMFortran

Modelo de Programación

La programación en OpenMP se basa principalmente en directivas del compilador

Ejemplo

```
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];
}
```

Ventajas

- Facilita la migración (el compilador ignora los `#pragma`)
- Permite la paralelización incremental
- Permite la optimización por parte del compilador

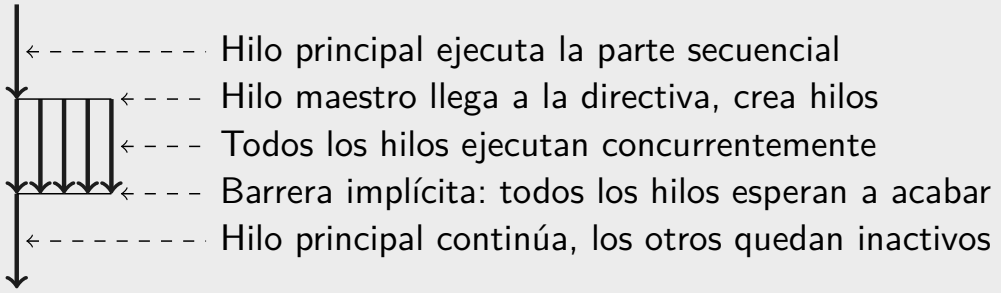
Además: funciones (ver `omp.h`) y variables de entorno

Modelo de Ejecución

El modelo de ejecución de OpenMP sigue un esquema *fork-join*

Hay directivas para crear hilos y dividir el trabajo

Esquema



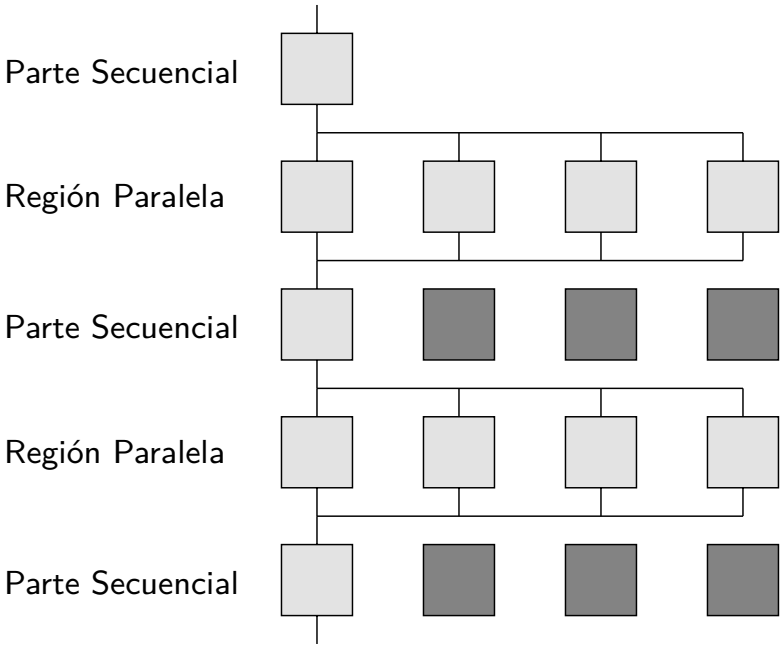
Las directivas definen regiones paralelas

Otras directivas/cláusulas:

- Indicar tipo de variable: `private`, `shared`, `reduction`
- Sincronización: `critical`, `barrier`

Modelo de Ejecución - Hilos

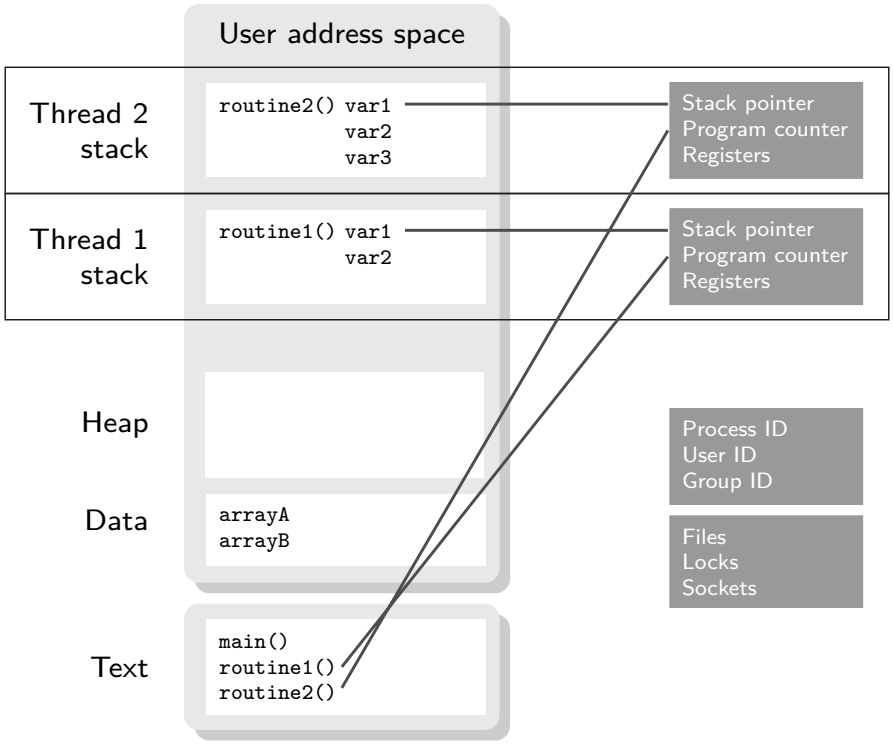
En OpenMP los hilos inactivos no se destruyen, quedan a la espera de la siguiente región paralela



Los hilos creados por una directiva se llaman equipo (*team*)

Modelo de Ejecución - Memoria

Cada hilo tiene su propio contexto de ejecución (incluyendo la pila)



Sintaxis

Directivas:

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

Uso de funciones:

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

Compilación condicional: la macro `_OPENMP` contiene la fecha de la versión de OpenMP soportada, p.e. 201107

Compilación:

```
gcc-4.2> gcc -fopenmp prg-omp.c
sun> cc -xopenmp -x03 prg-omp.c
intel> icc -openmp prg-omp.c
```


Ejemplo Simple

Ejemplo

```
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];
}
```

- Al llegar a la directiva `parallel` se crean los hilos (si no se han creado antes)
- Las iteraciones del bucle se reparten entre los hilos
- Por defecto, todas las variables son compartidas, excepto la variable del bucle (`i`) que es privada
- Al finalizar se sincronizan todos los hilos

Número e Identificador de Hilo

El número de hilos se puede especificar:

- Con la cláusula `num_threads`
- Con la función `omp_set_num_threads()` *antes* de la región paralela
- Al ejecutar, con `OMP_NUM_THREADS`

Funciones útiles:

- `omp_get_num_threads()`: devuelve el número de hilos
- `omp_get_thread_num()`: devuelve el identificador de hilo (empiezan en 0, el hilo principal es siempre 0)

```
omp_set_num_threads(3);
printf("hilos antes = %d\n",omp_get_num_threads());
#pragma omp parallel for
for (i=0; i<n; i++) {
  printf("hilos = %d\n",omp_get_num_threads());
  printf("yo soy %d\n",omp_get_thread_num());
}
```


Apartado 2

Paralelización de Bucles

- Directiva `parallel for`
- Tipos de Variables
- Mejora de Prestaciones

Directiva `parallel for`

Se paraleliza el bucle que va a continuación

C/C++

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

OpenMP impone restricciones al tipo de bucle, por ejemplo:

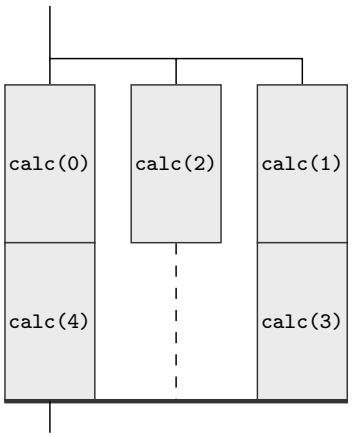
```
for (i=0; i<n && !encontrado; i++)
  if (x[i]==elem) encontrado=1;
```


Ejemplo de `parallel for`

Veamos una posible ejecución con 3 hilos

Bucle sencillo

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



Barrera implícita al finalizar la construcción `parallel for`

Variables:

- `a`: acceso concurrente, pero no hay más de un hilo accediendo a la misma posición
- `i`: distinto valor en cada hilo $\rightarrow$ necesitan una copia privada

Tipos de Variables

Se clasifican las variables según su alcance (*scope*)

- Privadas: cada hilo tiene una réplica distinta
- Compartidas: todos los hilos pueden leer y escribir

Fuente común de errores: no elegir correctamente el alcance

El alcance se puede modificar con cláusulas añadidas a las directivas:

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

`private`, `shared`, `default`

Si no se especifica el alcance de una variable, por defecto es `shared`

Excepciones (`private`):

- Índice del bucle que se paraleliza
- En subrutinas invocadas, las variables locales (excepto si se declaran `static`)
- Variables automáticas declaradas dentro del bucle

Cláusula `default`

- `default(none)` obliga a especificar el alcance de todas

private, shared

private

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

Incorrecto: tras el bucle sólo existe la suma del hilo principal (con valor 0) - además, las copias de cada hilo no se inicializan

shared

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

Incorrecto: condición de carrera al leer/escribir `suma`

reduction

Para realizar reducciones con operadores conmutativos y asociativos (`+`, `*`, `-`, `&`, `|`, `^`, `&&`, `||`, `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 hilo realiza una porción de la suma, al final se combinan en la suma total

Es como una variable privada, pero:

- Al final, los valores privados se combinan
- Se inicializa correctamente (al elemento neutro de la operación)

`firstprivate`, `lastprivate`

Las variables privadas se crean sin un valor inicial y tras el bloque `parallel` quedan indefinidas

- `firstprivate`: inicializa al valor del hilo principal
- `lastprivate`: se queda con el valor de la "última" iteración

Ejemplo

```
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 tiene el valor n */
```

El comportamiento por defecto intenta evitar copias innecesarias

Garantizar Suficiente Trabajo

La paralelización de bucles supone un *overhead*: activación y desactivación de hilos, sincronización

En bucles muy sencillos, el overhead puede ser mayor que el tiempo de cálculo

Cláusula `if`

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

Si la expresión es falsa, el bucle se ejecuta secuencialmente

Esta cláusula se podría usar también para evitar dependencias de datos detectadas en tiempo de ejecución

Bucles Anidados

Hay que poner la directiva antes del bucle a paralelizar

Caso 1

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

Caso 2

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

- En el primer caso, las iteraciones de `i` se reparten; el mismo hilo ejecuta el bucle `j` completo
- En el segundo caso, en cada iteración de `i` se activan y desactivan los hilos; hay `n` sincronizaciones

Bucles Anidados - Intercambio

Habitualmente se recomienda paralelizar el más externo

- En casos en que las dependencias de datos impiden esto, se puede intentar intercambiar los bucles

Código secuencial

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

Código paralelo con bucles intercambiados

```
#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];
```

Estas modificaciones pueden tener impacto en el uso de cache

Planificación

Idealmente, todas las iteraciones cuestan lo mismo y a cada hilo se le asigna aproximadamente el mismo número de iteraciones

En la realidad, se puede producir desequilibrio de la carga con la consiguiente pérdida de prestaciones

---

En OpenMP es posible especificar la planificación

Puede ser de dos tipos:

- Estática: las iteraciones se asignan a hilos a priori
- Dinámica: la asignación se adapta a la ejecución actual

La planificación se realiza a nivel de rangos contiguos de iteraciones (*chunks*)

Planificación - Cláusula `schedule`

Sintaxis de la cláusula de planificación:

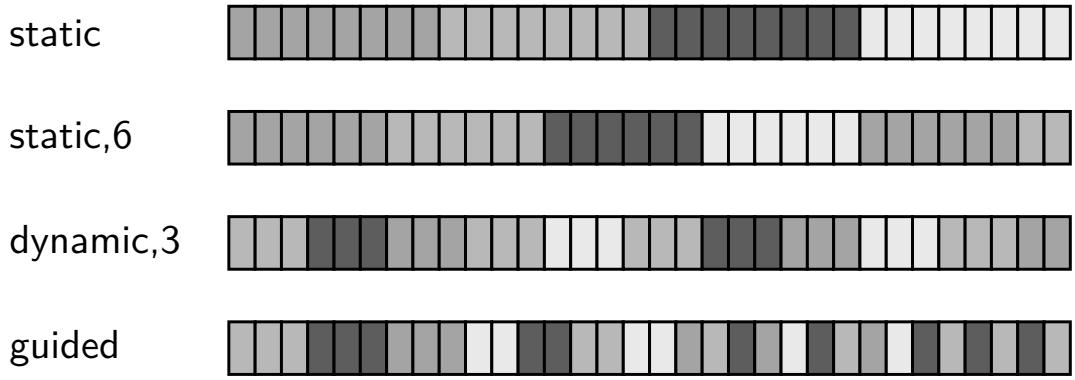
```
schedule(type[,chunk])
```

- `static` (sin `chunk`): a cada hilo se le asigna estáticamente un rango aproximadamente igual
- `static` (con `chunk`): asignación cíclica (*round-robin*) de rangos de tamaño `chunk`
- `dynamic` (`chunk` opcional, por defecto 1): se van asignando según se piden (*first-come, first-served*)
- `guided` (`chunk` mínimo opcional): como `dynamic` pero el tamaño del rango va decreciendo exponencialmente ($\propto n_{rest}/n_{hilos}$)
- `runtime`: se especifica en tiempo de ejecución con la variable de entorno `OMP_SCHEDULE`

Planificación - Ejemplo

Ejemplo: bucle de 32 iteraciones ejecutado con 4 hilos

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



Apartado 3

Regiones Paralelas

- Directiva `parallel`
- Reparto del Trabajo

Directiva `parallel`

Se ejecuta de forma replicada el bloque que va a continuación

C/C++

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

Algunas cláusulas permitidas son: `private`, `shared`, `default`, `reduction`, `if`

Ejemplo - se imprimen tantas líneas como hilos

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


Reparto del Trabajo

Además de la ejecución replicada, suele ser necesario repartir el trabajo entre los hilos

- Cada hilo opera sobre una parte de una estructura de datos, o bien
- Cada hilo realiza una operación distinta

Posibles formas de realizar el reparto:

- Según el identificador de hilo
- Cola de tareas paralelas
- Mediante construcciones OpenMP específicas

Reparto según Identificador de Hilo

Se utilizan las funciones

- `omp_get_num_threads()`: devuelve el número de hilos
- `omp_get_thread_num()`: devuelve el identificador de hilo

para determinar qué parte del trabajo realiza cada hilo

Ejemplo - identificadores de hilo

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


Reparto mediante Cola de Tareas Paralelas

Una cola de tareas paralelas es una estructura de datos compartida que contiene una lista de "tareas" a realizar

- Las tareas se pueden procesar concurrentemente
- Cualquier tarea puede realizarse por cualquier hilo

```
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();
        }
    }
```

Construcciones de Reparto del Trabajo

Las soluciones anteriores son bastante primitivas

- El programador se encarga de dividir el trabajo
- Código oscuro y complicado en programas largos

OpenMP dispone de construcciones específicas (*work-sharing constructs*)

Hay de tres tipos:

- Construcción `for` para repartir iteraciones de bucles
- Secciones para distinguir porciones del código
- Código a ejecutar por un solo hilo

Hay una barrera implícita al final del bloque

Construcción `for`

Reparte de forma automática las iteraciones del bucle

Ejemplo de bucle compartido

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

El bucle se *comparte* entre los hilos, en vez de hacerlo replicado

Las directivas `parallel` y `for` se pueden combinar en una

Construcción de Bucle - Cláusula `nowait`

Cuando hay varios bucles independientes dentro de una región paralela, `nowait` evita la barrera implícita

Bucles sin 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]);
    }
}
```


Construcción `sections`

Para trozos de código independientes difíciles de paralelizar

- Individualmente suponen muy poco trabajo, o bien
- Cada fragmento es inherentemente secuencial

Puede también combinarse con `parallel`

Ejemplo de secciones

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

Un hilo puede ejecutar más de una sección
Cláusulas: `private`, `first/lastprivate`, `reduction`, `nowait`

Construcción `single`

Fragmentos de código que deben ejecutarse por un solo hilo

Ejemplo `single`

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

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

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

Algunas cláusulas permitidas: `private`, `firstprivate`, `nowait`

Directiva `master`

Identifica un bloque de código dentro de una región paralela que se ha de ejecutar solo por el hilo principal (*master*)

Ejemplo `master`

```
#pragma omp parallel for
for (i=0; i<n; i++) {
    calc1();     /* realizar cálculos */
    #pragma omp master
        printf("Resultados intermedios: ...\n", ...);
    calc2();     /* realizar más cálculos */
}
```

Diferencias con `single`:

- No se requiere que todos los hilos alcancen esta construcción
- No hay barrera implícita (los otros hilos simplemente se saltan ese código)

Apartado 4

Sincronización

- Exclusión Mutua
- Otro Tipo de Sincronización

Condición de Carrera (1)

El siguiente ejemplo ilustra una condición de carrera

Búsqueda de máximo

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

Secuencia con resultado incorrecto:

Hilo 0: lee `a[i]`=20, lee `cur_max`=15
Hilo 1: lee `a[i]`=16, lee `cur_max`=15
Hilo 0: comprueba `a[i]>cur_max`, escribe `cur_max`=20
Hilo 1: comprueba `a[i]>cur_max`, escribe `cur_max`=16

Condición de Carrera (2)

Hay casos en que el acceso concurrente no produce condición de carrera

Ejemplo de acceso concurrente sin condición de carrera

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

Aunque varios hilos escriban a la vez, el resultado es correcto

---

En general, se necesitan mecanismos de sincronización:

- Exclusión mutua
- Otro tipo de sincronización

Exclusión Mutua

La *exclusión mutua* en el acceso a las variables compartidas evita cualquier condición de carrera

OpenMP proporciona tres construcciones diferentes:

- Secciones críticas: directiva `critical`
- Operaciones atómicas: directiva `atomic`
- Cerrojos: rutinas `*_lock`

Directiva `critical` (1)

En el ejemplo anterior, el acceso en exclusión mutua a la variable `cur_max` evita la condición de carrera

Búsqueda de máximo, sin condición 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];
    }
}
```

Cuando un hilo llega al bloque `if` (la sección crítica), espera hasta que no hay otro hilo ejecutándolo al mismo tiempo

OpenMP garantiza progreso (al menos un hilo de los que espera entra en la sección crítica) pero no espera limitada

Directiva `critical` (2)

En la práctica, el ejemplo anterior resulta ser secuencial

Teniendo en cuenta que `cur_max` nunca se decrementa, se puede plantear la siguiente mejora

Búsqueda de máximo, mejorado

```
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 segundo `if` es necesario porque se ha leído `cur_max` fuera de la sección crítica

Esta solución entra en la sección crítica con menor frecuencia

Directiva `critical` con Nombre

Al añadir un nombre, se permite tener varias secciones críticas sin relación entre ellas

Búsqueda de máximo y mínimo

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


Directiva `atomic`

Operaciones atómicas de lectura-modificación-escritura

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

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

donde <binop> puede ser +, *, -, /, %, &, |, ^, <<, >>

Ejemplo `atomic`

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

El código es mucho más eficiente que con `critical` y permite actualizar elementos de `x` en paralelo

Directiva barrier

Al llegar a una barrera, los hilos esperan a que lleguen todos

```
#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 suele usar para asegurar que una fase la han terminado todos antes de pasar a la siguiente fase

Directiva ordered

Para permitir que una porción del código de las iteraciones se ejecute en el orden secuencial original

Ejemplo ordered

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

Restricciones:

- Si un bucle paralelo contiene una directiva ordered, hay que añadir la cláusula ordered también al bucle
- Sólo se permite una única sección ordered por iteración