

OpenMP

Metodología de la Programación Paralela

Jesús Sánchez Cuadrado (jesusc@um.es)

Curso 2019/20

Programación en memoria compartida

- El programa se ve como una colección de elementos de proceso (procesos o hilos) accediendo a una zona central de variables compartidas.
- Más de un elemento podría acceder a la misma zona compartida en el mismo instante produciendo resultados impredecibles.
- Los lenguajes proporcionan primitivas para resolver estos problemas de exclusión mutua (secciones críticas, constructores secuenciales, llaves, semáforos...)

OpenMP

- MP = Multi-Processing
- Especificación disponible
 - <https://www.openmp.org/>
- Decisiones de diseño
 - Aumentar un lenguaje estándar con constructores paralelos
 - Declarativo
 - Portable
 - Crear programas paralelos de manera incremental

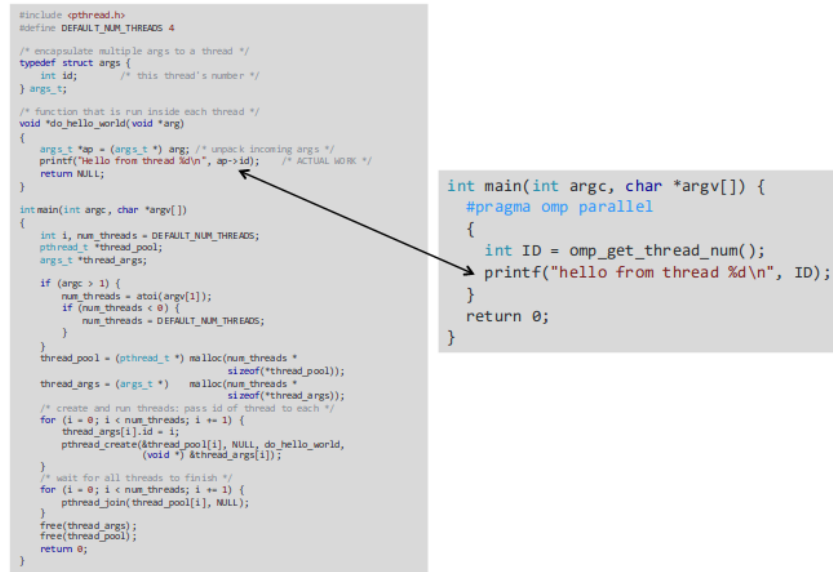
OpenMP vs. Librerías de hilos

OpenMP

- Paralelismo implícito
- Requiere soporte del compilador
- Abstrae de los detalles de bajo nivel

Pthreads

- Paralelismo explícito
- Librería externa
- Detalles de implementación



```
#include <pthread.h>
#define DEFAULT_NUM_THREADS 4

/* encapsulate multiple args to a thread */
typedef struct args {
    int id; /* this thread's number */
} args_t;

/* function that is run inside each thread */
void do_hello_world(void *arg)
{
    args_t *ap = (args_t *) arg; /* unpack incoming args */
    printf("hello from thread %d\n", ap->id); /* ACTUAL WORK */
    return NULL;
}

int main(int argc, char *argv[])
{
    int i, num_threads = DEFAULT_NUM_THREADS;
    pthread_t *thread_pool;
    args_t *thread_args;

    if (argc > 1) {
        num_threads = atoi(argv[1]);
        if (num_threads < 0) {
            num_threads = DEFAULT_NUM_THREADS;
        }
    }

    thread_pool = (pthread_t *) malloc(num_threads *
                                      sizeof(pthread_t));
    thread_args = (args_t *) malloc(num_threads *
                                    sizeof(args_t));

    /* create and run threads: pass id of thread to each */
    for (i = 0; i < num_threads; i++) {
        thread_args[i].id = i;
        pthread_create(&thread_pool[i], NULL, do_hello_world,
                      (void *) &thread_args[i]);
    }

    /* wait for all threads to finish */
    for (i = 0; i < num_threads; i++) {
        pthread_join(thread_pool[i], NULL);
    }

    free(thread_args);
    free(thread_pool);
    return 0;
}
```

```
int main(int argc, char *argv[]) {
    #pragma omp parallel
    {
        int ID = omp_get_thread_num();
        printf("hello from thread %d\n", ID);
    }
    return 0;
}
```

OpenMP

- Basado en directivas del compilador

```
#pragma omp [directiva]
```

Directiva **parallel**

- Se crea un grupo de threads. El que los pone en marcha actúa de maestro.
- Hay barrera implícita al final de la región.
- Con cláusula `if` se evalúa su expresión y si da valor distinto de cero se crean los threads, si es cero se hace en secuencial.
- El número de threads a crear se obtiene por variables de entorno, llamadas a librería o con la cláusula `num threads`.

```
#pragma omp parallel [clausulas]
{
    /* Código paralelo */
}
```

```
int fork = 0;
#pragma omp parallel if (fork)
{ ... }
```

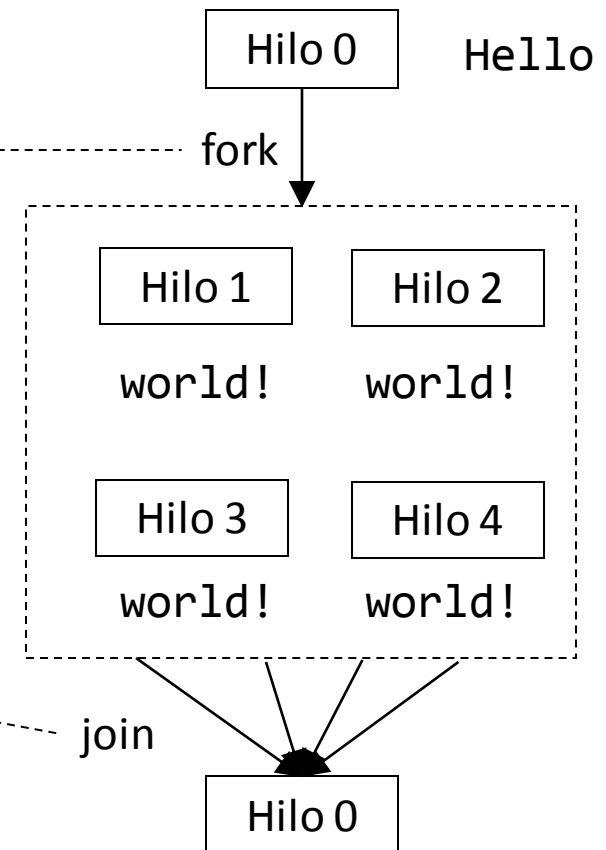
Conceptos

- Team (equipo)
 - Un equipo es un conjunto de hilos que se ejecutan concurrentemente
 - Al principio, el equipo tiene un solo hilo
 - La directiva `parallel` crea un nuevo conjunto de hilos, que están activos hasta que acabe el bloque
 - La directiva `for` divide el trabajo entre los hilos el equipo actual.
 - `parallel for` es una combinación de las dos directivas
 - Siempre hace falta la directiva `parallel` para ejecutar en paralelo

Directiva **parallel**

```
#include <stdio.h>
```

```
int main(void) {  
    printf("Hello\n");  
    #pragma omp parallel  
    {  
        // El bloque se ejecuta en paralelo  
        printf("world!\n");  
    }  
  
    return 0;  
}
```



Directiva **parallel**

- Cuando dentro de una región hay otro constructor paralelo (anidamiento) cada esclavo crea otro grupo de threads esclavos de los que es el maestro.
- Cláusulas (private , firstprivate , default , shared , copyin y reduction) para indicar la forma en que se accede a las variables.
- Con cláusula `if` se evalúa su expresión y si da valor distinto de cero se crean los threads, si es cero se hace en secuencial.
- El número de threads a crear se obtiene por variables de entorno, llamadas a librería o con la cláusula `num threads` .
- Hay barrera implícita al final de la región.

Compilación de programas OpenMP

- Compilación con un compilador C/C++ y la opción de OpenMP:
 - `$ gcc -O3 programa.c -fopenmp`
 - `$ icc -O3 programa.c -openmp`
 - o `-qopenmp` en versiones recientes de icc
- Una vez compilado se ejecuta como cualquier otro programa.
- ¿Cuántos threads se ponen en marcha?
 - Por defecto el número de cores.
 - Se puede cambiar con variable de entorno o función de librería.

Establecer el número de hilos

- Utilizando una variable de entorno

```
$ OMP_NUM_THREADS=4 ./helloworld
```

- Utilizando la función `omp_set_num_threads`
- Utilizando la opción `num_threads` de la directiva `parallel`

Directiva `for`

`ejemplo_for.c`

- Las iteraciones se ejecutan en paralelo por threads que ya existen (creados antes con `parallel`).
- La parte de inicialización del `for` debe ser una asignación.
- La parte de incremento debe ser una suma o resta.
- La parte de evaluación es la comparación de una variable entera sin signo, utilizando un comparador mayor o menor (puede incluir igual).
- Los valores que aparecen en las tres partes del `for` deben ser enteros.
- Hay barrera al final a no ser que se utilice la cláusula `nowait`.
- Cláusulas (`private` , `firstprivate` , `lastprivate` y `reduction`) para indicar la forma en que se accede a las variables.

Bucle **for** paralelo

```
#define SIZE 360
```

```
double sin_table[SIZE];
```

```
#pragma omp parallel for  
for(int i = 0; i < SIZE; i++)  
{  
    sin_table[i] = sin(2 * M_PI * i / SIZE);  
}
```

Bucle **for** paralelo

- Compilación

Thread local



```
int this_thread = omp_get_thread_num();  
int num_threads = omp_get_num_threads();  
int my_start = (this_thread ) * SIZE / num_threads;  
int my_end   = (this_thread+1) * SIZE / num_threads;  
for(int i=my_start; i<my_end; ++n)  
    sin_table[i] = sin(2 * M_PI * i / SIZE);
```

- Limitaciones

- El compilador sólo puede paralelizar bucles en *forma canónica*

¿Es posible imitar un parallel for solo
con una región paralela?

Bucle **for** paralelo

- La variable índice (`index`) debe ser de tipo entero o puntero
- Las expresiones `start`, `end`, e `incr` deben ser del tipo compatible (ej., `index` es un puntero, `incr` debe ser un entero)
- Los valores de `start`, `end` e `incr` no pueden cambiar durante la ejecución del bucle
- La variable `index` sólo puede cambiarse a través de la operación de incremento de la sentencia `for`

```
for ( index = start ; index < end ; index++  
      index <= end ; ++index  
      index >= end ; index--  
      index > end ; --index  
      index += incr  
      index -= incr  
      index = index + incr  
      index = incr + index  
      index = index - incr )
```

Ámbito de las variables

Variable privada

- Cada hilo tiene su propia “versión” de la variable
- Lecturas y escrituras son locales al hilo
- Modificador **private(var)** o declaradas dentro de **parallel**

```
int v_shared = 5;
#pragma omp parallel
{
    int local = ...;
}
```

Variable compartida

- Puede ser accedida por todos los hilos
- Requerirá sección crítica cuando hay escrituras
- Modificador **shared(var)** o declaradas fuera de **parallel**

```
int v = 5;
#pragma omp parallel \
    private(local) shared(v)
{
    int local = ...;
}
```


Bucle **for** paralelo

```
#pragma omp parallel for
for(int i = 0; i < 10; i++)
{
    /* Código paralelo */
}
```

```
-----

int i;
#pragma omp parallel for private(i)
for(i = 0; i < 10; i++)
{
    /* Código paralelo */
}
```

i es local bucle

i no es local al bucle
pero OpenMP la hace
privada por defecto

No es obligatorio **private**
pero es conveniente

Bucle **for** paralelo

```
#pragma omp parallel for private(temp)
for(i=0; i < N; i++){
    for (j=0; j < M; j++){
        temp = b[i] * c[j];
        a[i][j] = temp * temp + d[i];
    }
}
```

¿Por qué es incorrecto?

Manejo de variables

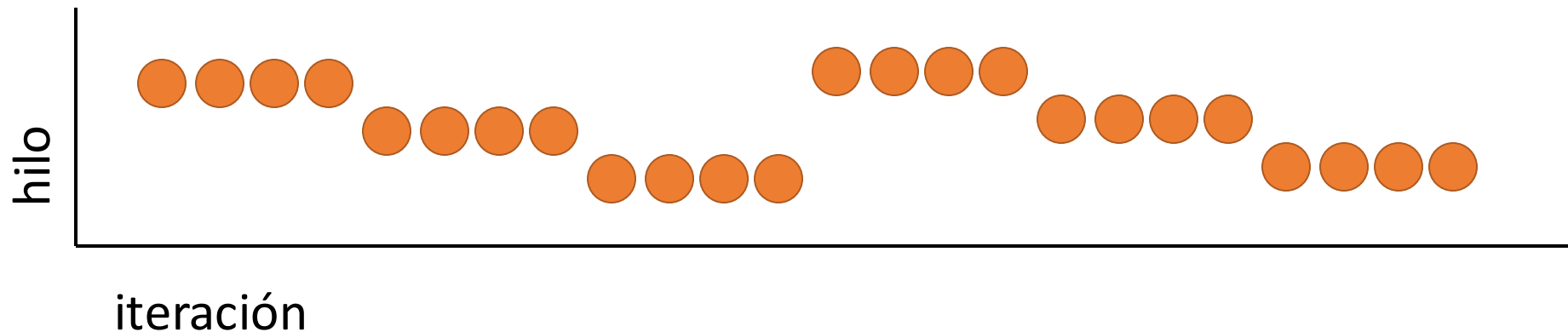
- `private(lista)`
 - privadas a los threads, no se inicializan antes de entrar y no se guarda su valor al salir.
- `firstprivate(lista)`
 - privadas a los threads, se inicializan al entrar con el valor que tuviera la variable correspondiente.
- `lastprivate(lista)`
 - privadas a los threads, al salir quedan con el valor de la última iteración o sección.
- `shared(lista)`
 - compartidas por todos los threads.
- `default(shared|none)`
 - indica cómo serán las variables por defecto.
- `reduction(operador:lista)`
 - se obtienen por la aplicación del operador.
- `copyin(lista)`
 - para asignar el valor de la variable en el master a variables locales privadas a los threads al empezar la región paralela.

Scheduling

- La cláusula **schedule** indica la forma en que se dividen las iteraciones del for entre los threads.
 - `schedule(static, tamaño)` las iteraciones se dividen según el tamaño, y la asignación se hace estáticamente a los threads. Si no se indica el tamaño se divide por igual entre los threads.
 - `schedule(dynamic, tamaño)` las iteraciones se dividen según el tamaño y se asignan a los threads dinámicamente cuando van acabando su trabajo.
 - `schedule(guided, tamaño)` las iteraciones se asignan dinámicamente a los threads pero con tamaños decrecientes.
 - `schedule(runtime)` deja la decisión para el tiempo de ejecución, y se obtienen de la variable de entorno OMP SCHEDULE .

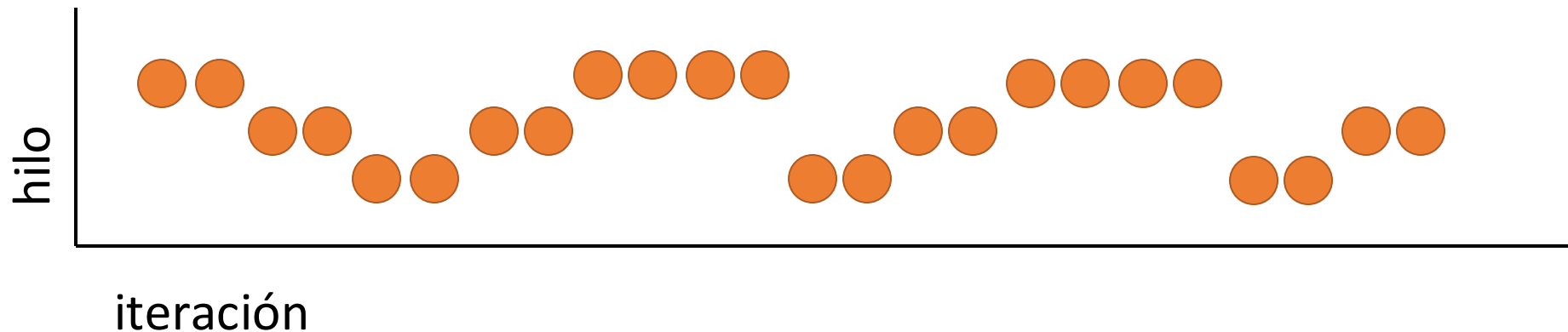
Scheduling

- `schedule(static, 4)`
 - **Ejemplo.** Bucle **for** con 24 iteraciones dividido entre 3 hilos
 - El trabajo (24 iteraciones) se divide antes de comenzar la ejecución entre los 3 hilos
 - Aunque un hilo termine antes no hará trabajo que no le haya sido asignado de antemano



Scheduling

- `schedule(dynamic, 2)`
 - **Ejemplo.** Bucle **for** con 24 iteraciones, 3 hilos
 - Cada hilo va pidiendo trabajo cuando está ocioso.
 - La unidad de trabajo es 2 iteraciones.
 - Apropiado cuando cada iteración puede tener un coste de ejecución diferente
 - Hay mayor “overhead” porque la asignación de trabajo es dinámica



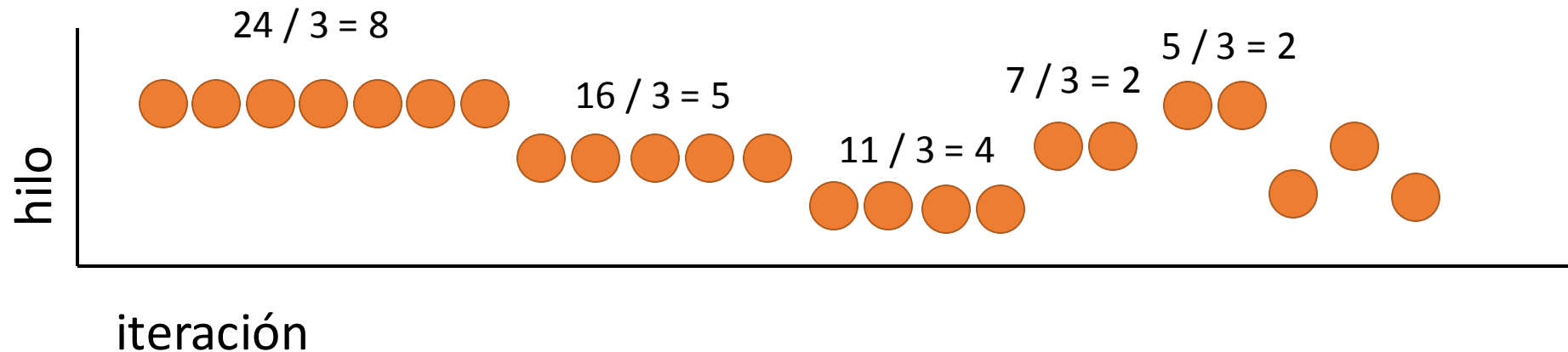
Scheduling

```
int sum = 0.0;
#pragma omp parallel for
for(int i = 0; i <= n; i++) {
    #pragma omp critical
    sum += f(i)
}
```

¿Qué sucede si
f(i) tarda más
cuando i es mayor?

Scheduling

- `schedule(guided, 2)`
 - **Ejemplo.** Bucle **for** con 24 iteraciones, 3 hilos
 - Tamaño de cada trabajo es proporcional al número de iteraciones sin asignar dividido por el número de hilos: iteraciones sin asignar / número de hilos



Scheduling

- Para una buena discusión:
 - <https://stackoverflow.com/questions/10850155/whats-the-difference-between-static-and-dynamic-schedule-in-openmp>
 - <https://stackoverflow.com/questions/42970700/openmp-dynamic-vs-guided-scheduling>
 - <http://jakascorner.com/blog/2016/06/omp-for-scheduling.html>

Operador de reducción

- `#pragma omp parallel private(i) reduction(+:x)`
- Varios hilos calculan un valor que necesita ser combinado con los valores de los otros hilos.
- Se puede utilizar una directiva critical para evitar condiciones de carrera
- La clausula reduction hace el trabajo:
 - OpenMP crea una copia local de la variable reducida en cada hilo
 - Cada hilo va calculando su propio valor local
 - Al final de la ejecución (en el join) los valores se combinan según el operador

Operador de reducción

- Ejemplo, cálculo de PI

```
int  n = 50;
int  i;
double PI25DT = 3.141592653589793238462643;
double pi, h, sum, x;

// Inicializa el número de intervalos a n
h    = 1.0 / (double) n;
sum  = 0.0;
// La variable pi es una variable de reducción mediante suma
#pragma omp parallel for reduction(+:pi) private(x, i)
for (i = 1; i <= n; i++) {
    x = h * ((double)i - 0.5);
    pi += f(x);
}
pi = 4. * h * pi;
```

Sections

codigo3-14.c

ejemplo_sections.c

- Directiva `sections` y `section`
- Cada sección se ejecuta por un thread
- Hay barrera al final a no ser que se utilice la cláusula `nowait`

```
#pragma omp sections [clausulas]
{
    [#pragma omp section]
    bloque1
    [#pragma omp section]
    bloque2
}
```

Sections

- Ejecuta un número fijo de hilos
- Cada sección con un código diferente

```
#define N 1000
void main (){
    int i; float a[N], b[N], c[N];
    for (i=0; i < N; i++) a[i] = b[i] = ...;
    #pragma omp parallel sections shared(a,b,c) private(i)
    {
        #pragma omp section
        { for (i=0; i < N/2; i++) c[i] = a[i] + b[i]; }
        #pragma omp section
        { for (i=N/2; i < N; i++) c[i] = a[i] + b[i]; }
    }
}
```

¿Qué hace este código?

Opción **nowait**

- Admitida en `sections` y `for`
- Evita que los hilos se sincronicen al final de la ejecución de esas cláusulas

```
#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 nowait
    for (i=1; i < n; i++) {
        y[i] = sqrt(z[i]);
    }
}
```

Constructores combinados

- Forma abreviada de directiva `parallel` con una única directiva `for` o `sections`, de las que admite sus cláusulas menos la `nowait`.

```
#pragma omp parallel for [clausulas]  
    bucle for
```

```
#pragma omp parallel sections [clausulas]
```

¿Por qué `nowait` no se permite?

Ejecución secuencial

- `#pragma omp single [clausulas]`

`ejemplo_single.c`

- El bloque se ejecuta por único hilo
- Hay una barrera al final a no ser que se incluya la cláusula `nowait`

- `#pragma omp master`

`ejemplo_master.c`

- El bloque se ejecuta únicamente por el hilo maestro
- No hay sincronización ni al entrar ni al salir

- `#pragma omp ordered`

`ejemplo_ordered.c`

- El bloque se ejecuta en el orden en que se ejecutaría en secuencial

Constructores de sincronización

- `#pragma omp critical [(nombre)]`
- Establece una región de exclusión de mutua
- Sólo un bloque puede estar ejecutándola a la vez
- El nombre es opcional, pero permite que haya más de una región crítica por programa
 - Dos regiones críticas con nombres diferentes se podrán ejecutar a la vez

Constructores de sincronización

- `#pragma omp critical [(nombre)]`

```
int x=0

#pragma omp parallel shared(x)
{
    // Código que calcula “valor”
    #pragma omp critical
        x = x + valor
}
```

single vs. critical

- single
 - El código debe ejecutarlo un único hilo, el resto no lo ejecutan
- critical
 - El código lo ejecuta un único hilo en cierto instante de tiempo, pero todos los hilos lo ejecutarán (si su lógica lo permite)

```
int a=0, b=0;
#pragma omp parallel num_threads(4)
{
    #pragma omp single
    a++;
    #pragma omp critical
    b++;
}
```

¿Cuánto valen a y b al final del bloque paralelo?

Constructores de sincronización

- `#pragma omp atomic`
- Establece una región de exclusión de mutua para actualizar una asignación que tenga la siguiente forma:

`x <op> = <expresión>`

`x++;`

`++x;`

`x--;`

`--x;`

`<op> =`

`+, *, -, /, &, ^, |, <<, >>`

- El compilador utilizará instrucciones CAS para hacerla muy eficiente

Constructores de sincronización

- Candados (lock)
 - Es el mecanismo de sincronización más flexible
- `void omp_init_lock(omp_lock_t *lock);`
 - Para inicializar una llave. Una llave se inicializa como no bloqueada.
- `void omp_init_destroy(omp_lock_t *lock);`
 - Para destruir una llave.
- `void omp_set_lock(omp_lock_t *lock);`
 - Para pedir una llave.
- `void omp_unset_lock(omp_lock_t *lock);`
 - Para soltar una llave.
- `int omp_test_lock(omp_lock_t *lock);`
 - Intenta pedir una llave pero no se bloquea.

Barreras

- Sincronización explícita de los threads
 - Construcción barrier
- Sincronización implícita al final la sección parallel.

```
#pragma omp barrier
// ... código ...

// Barrera según cierta condición
if (x == 0) {
    #pragma omp barrier
}
```

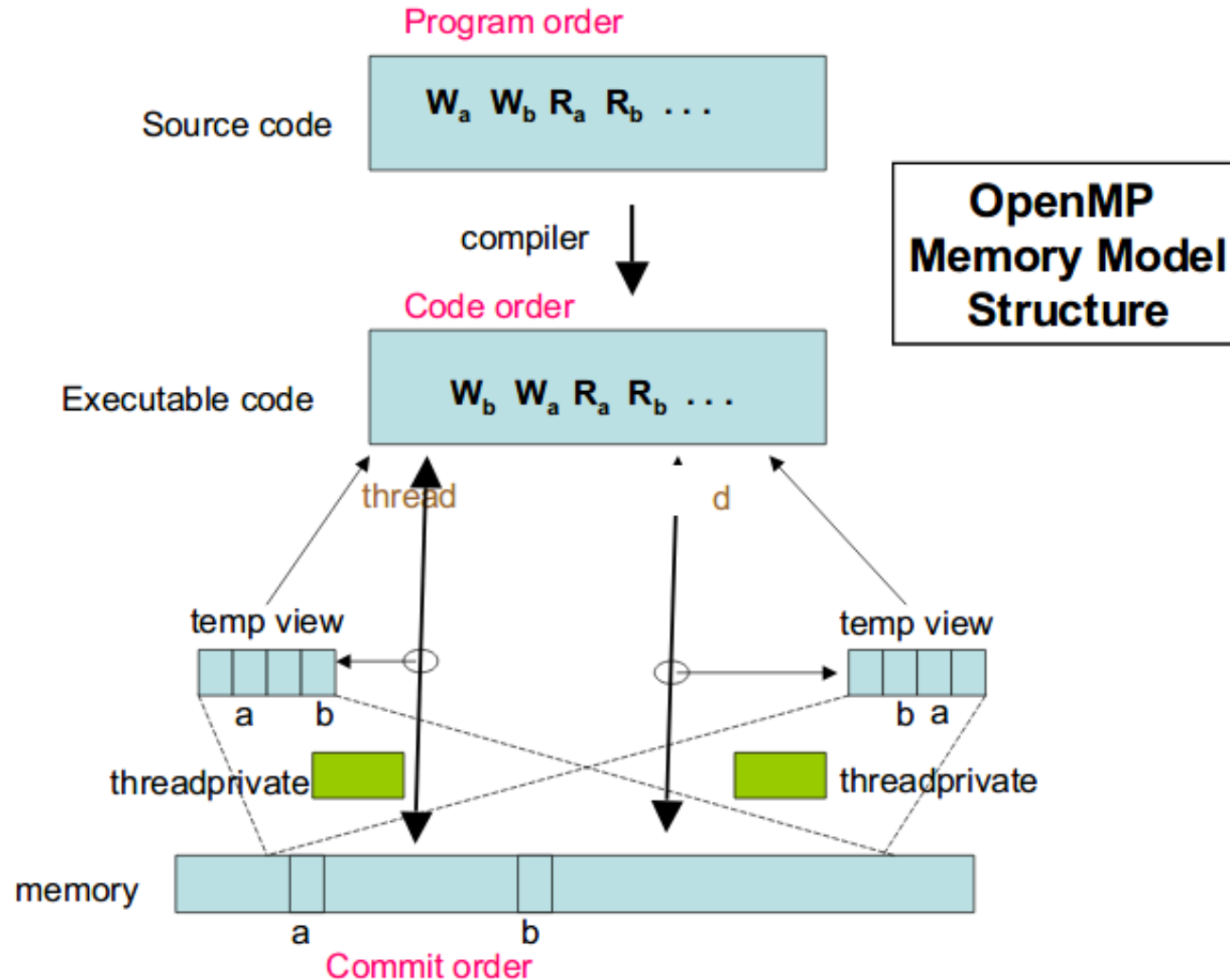
Funciones de librería

- `void omp_set_num_threads(int num_threads);`
 - Establece el número de threads a usar en la siguiente región paralela.
- `int omp_get_num_threads(void);`
 - Obtiene el número de threads que se están usando en una región paralela.
- `int omp_get_max_threads(void);`
 - Obtiene la máxima cantidad posible de threads.
- `int omp_get_thread_num(void);`
 - Devuelve el número del thread.
- `int omp_get_num_procs(void);`
 - Devuelve el máximo número de procesadores que se pueden asignar al programa.
- `int omp_in_parallel(void);`
 - Devuelve valor distinto de cero si se ejecuta dentro de una región paralela.

Funciones de librería

- `int omp_set_dynamic(void);`
 - Permite poner o quitar el que el número de threads se pueda ajustar dinámicamente en las regiones paralelas.
- `int omp_get_dynamic(void);`
 - Devuelve un valor distinto de cero si está permitido el ajuste dinámico del número de threads.
- `int omp_set_nested(int);`
 - Para permitir o desautorizar el paralelismo anidado
- `int omp_get_nested(void);`
 - Devuelve un valor distinto de cero si está permitido el paralelismo anidado

Modelo de memoria de OpenMP



Consistencia de memoria

ejemplo flush.c

- El compilador puede reordenar las instrucciones
- Un hilo ejecutándose en un core puede mantener las variables en registros (no hacer commit a memoria)
- El resto de hilos leerán de su cache local o de memoria principal y no verán el valor
- `#pragma omp flush [lista variables]`
 - Asegura que las variables que aparecen en la lista quedan actualizadas para todos los threads.
- `#pragma omp threadprivate(lista variables)`
 - Para declarar globales variables privadas a los threads.

Consistencia de memoria

- En OpenMP se hace flush implícito en ciertas situaciones:
- Barreras – `barrier`
- A la entrada y la salida de:
 - `parallel`, `parallel worksharing`, `critical`, regiones `ordered`
- A la entrada ya salida de `atomic`
- A la salida de regiones `worksharing` (a no ser que se indique `nowait`)
- En `omp_set_lock`, `omp_set_nest_lock`, `omp_set_nest_lock`, `omp_unset_nest_lock`
- En `omp_test_lock`, `omp_test_nest_lock`, si se adquiere el candado

Consistencia de memoria

- Directivas en las que aparece implícitamente una directiva `flush`

Directiva	Momento de flush
<code>barrier</code>	Al ejecutarse
<code>critical</code>	Al entrar y al salir
<code>ordered</code>	Al entrar y al salir
<code>parallel</code>	Al salir
<code>for</code>	Al salir
<code>sections</code>	Al salir
<code>single</code>	Al salir

Consistencia de memoria

- Para comunicar el valor de una variable compartida de un hilo a otro, hay que realizar los siguientes pasos, en este orden:
 1. Escribir en la variable en el hilo A
 2. Hacer flush (implícito o explícito) de la variable en el hilo A
 3. Hacer flush (implícito o explícito) de la variable en el hilo B
 4. Leer la variable en el hilo B

Consistencia de memoria

- Modificador `volatile`
 - Indica que una variable siempre debe escribirse en memoria principal
 - Hay un `flush()` implícito antes de cada lectura y después de cada escritura
 - Impide optimizaciones

Búsqueda en array

```
#pragma omp parallel private(i, id, p, load, begin, end)
{
    p = omp_get_num_threads();
    id = omp_get_thread_num();
    load = N/p; begin = id*load; end = begin+load;
    for (i = begin; ((i<end) && keepon); i++) {
        if (a[i] == x) {
            keepon = 0;
            position = i;
        }
    }
}
```

¿Hay algún problema?

Modelos de memoria

- <https://stackoverflow.com/questions/12429818/does-explicit-lock-automatically-provide-memory-visibility>
- <http://tutorials.jenkov.com/java-concurrency/java-memory-model.html>
- <https://medium.com/@pablocastelnovo/variables-vol%C3%A1tiles-en-java-f5ae078bf8b9>
- <https://stackoverflow.com/questions/41612143/is-openmp-atomic-write-needed-if-other-threads-read-only-the-shared-data>

False sharing

- Memorias cache
 - Cache hit: cuando la CPU necesita un valor de memoria y el valor está en la cache, lo puede recuperar rápidamente
 - Cache miss: Cuando la CPU necesita un valor pero no está en la cache y debe ir a buscarlo a memoria
 - Los memoria cache se organiza en líneas o bloques, típicamente de 64 bytes

False sharing

- Coherencia espacial
 - Si el programa usa una dirección de memoria ahora, posiblemente necesita pronto los valores de las direcciones de memoria cercanas
- Coherencia temporal
 - Si el programa está usando una dirección de memoria ahora, posiblemente la volverá a utilizar pronto

Si se cumplen estas reglas => Cache hit ++
Si no => Cache miss ++

False sharing

cache-miss-row.c

cache-miss-col.c

- ¿Cuál va más rápido?
 - Complejidad igual – $O(n^2)$

```
double array[NUM];

for( int i = 0; i < NUM; i++ ){
    for( int j = 0; j < NUM; j++ ){
        sum += array[ i ][ j ];
        // acceso por filas
    }
}
```

```
double array[NUM];

for( int i = 0; i < NUM; i++ ){
    for( int j = 0; j < NUM; j++ ){
        sum += array[ j ][ i ];
        // acceso por columnas
    }
}
```

False sharing

- Herramientas para analizar fallos de cache

```
$ valgrind --tool=cachegrind ./mi_programa
```

```
$ sudo perf stat -d ./mi_programa
```

* <http://www.brendangregg.com/perf.html>

False sharing

- Cachegrind
 - Simula cómo interacciona el programa con la memoria cache y con el predictor de saltos
 - Asume una arquitectura con dos niveles de cache, L1 y LL.
 - Si la arquitectura tiene más niveles sólo se analiza el último
 - Por desgracia no funciona bien con programas multi-hilo
 - Usar perf
- I => Instruction
- D => Data

<http://valgrind.org/docs/manual/cg-manual.html>

cache-miss-row.c

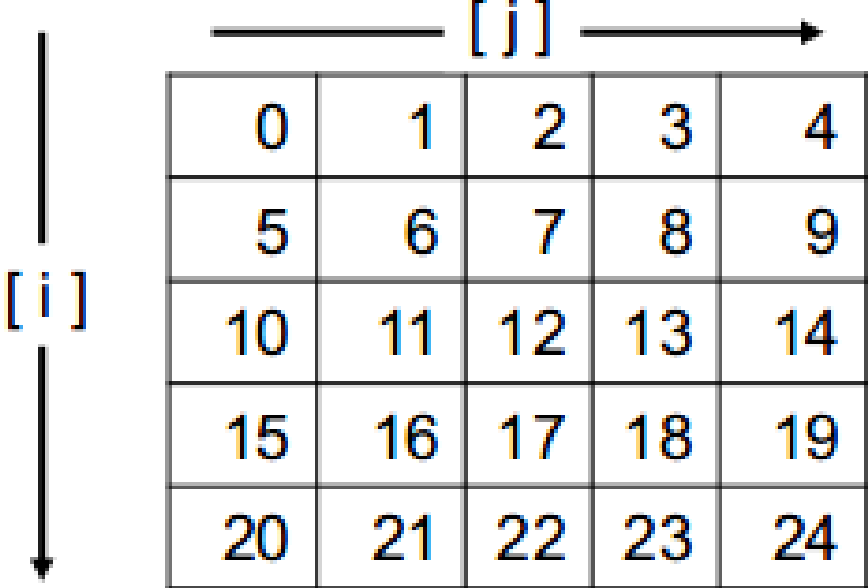
```
==23156== D    refs:          700,133,994 (600,101,603 rd    + 100,032,391 wr)
==23156== D1  misses:         6,254,292 ( 6,253,488 rd    +           804 wr)
==23156== LLd misses:        6,253,340 ( 6,252,605 rd    +           735 wr)
==23156== D1  miss rate:           0.9% (           1.0%    +           0.0% )
==23156== LLd miss rate:           0.9% (           1.0%    +           0.0% )
```

cache-miss-col.c

```
==26047== D    refs:          700,133,998 (600,101,605 rd    + 100,032,393 wr)
==26047== D1  misses:       100,004,290 (100,003,482 rd    +           808 wr)
==26047== LLd misses:        6,263,341 ( 6,262,604 rd    +           737 wr)
==26047== D1  miss rate:        14.3% (          16.7%    +           0.0% )
==26047== LLd miss rate:           0.9% (           1.0%    +           0.0% )
```

False sharing

- Al recorrer por columnas estamos “sacando” de la memoria cache líneas que vamos a necesitar en otras iteraciones
- Es mejor recorrer en la dirección de las líneas de cache

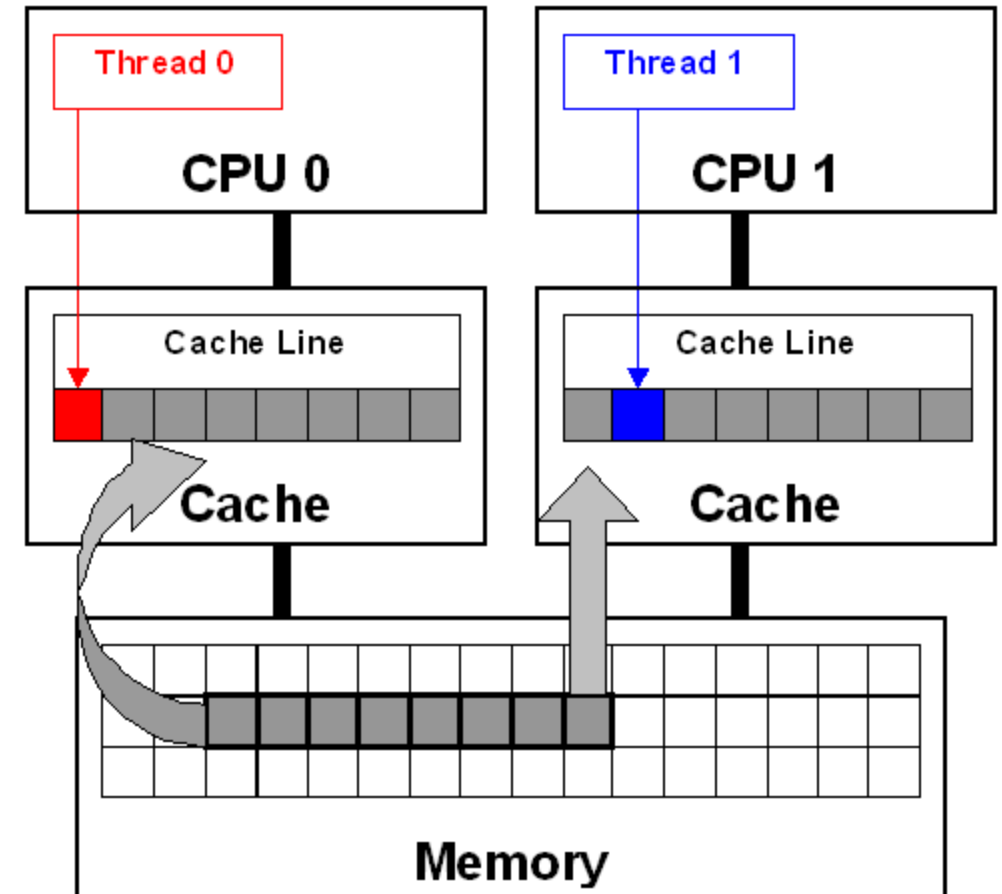


A 5x5 grid representing a memory array. The columns are indexed by $[j]$ (horizontal arrow) and the rows by $[i]$ (vertical arrow). The values in the cells are as follows:

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

False sharing

- El problema es aún peor cuando tenemos varios hilos
- Al escribir en una posición de memoria se invalida la línea entera
- Se puede producir un efecto ping-pong



Errores comunes

- Ver presentación adicional

Ejercicio

- Buscar en un array
 - Versión ineficiente, ¿por qué?

```
#pragma omp parallel for
for ( i = 0 ; i < N; ++i ) {
    #pragma omp critical
    {
        if (arr[i] > max )
            max = arr[i] ;
    }
}
```

Versión algo mejor, ¿por qué?

```
#pragma omp parallel for
for ( i = 0 ; i < N; ++i ) {
    #pragma omp flush (max)
    if (arr[i] > max) {
        #pragma omp critical
        {
            if (arr[i] > max )
                max = arr[i] ;
        }
    }
}
```

Ejercicio

- Búsqueda en array
 - Versión propuesta para emular reduce
 - ¿Es mejor?
 - ¿Es correcta?

```
#pragma omp parallel
{
    int priv_max;
    #pragma omp for
    for ( i = 0 ; i < N; ++i ) {
        if (arr[i] > priv_max) {
            priv_max = arr[i];
        }
        #pragma omp flush(max)
        if (privmax > max) {
            #pragma omp critical
            if (priv_max > max )
                max = priv_max;
        }
    }
}
```

Tareas

- Se introducen en OpenMP 3.0
- OpenMP cambia de “thread-centric” a “task-centric”.
- Para expresar paralelismo irregular y no estructurado.
- Para paralelizar bucles while . Por ejemplo, recorrido de una lista de punteros de longitud no conocida.

```
p = list_head();  
while (!list_end(p)) {  
    procesar( p );  
    p = list_next(p);  
}
```

Tareas

- Ejemplo
 - Recorrido de una lista
 - La directiva single asegura que un solo hilo se encarga de generar las tareas.

```
#pragma omp parallel
#pragma omp single
{
    while (!list_tail(p)) {
        p = list_next(p);
        #pragma omp task
        procesar(p);
    }
    #pragma omp taskwait
}
```

Tareas

- Hay una cola por equipo (privada, manejada por OpenMP)
- Un hilo (ej., master) comienza a generar tareas
- Las tareas pueden generar tareas recursivamente
- El sistema organiza pone las tareas en hilos automáticamente
- Dos tipos de tareas:
 - **Tied task**: ligada a un thread fijo que la ejecuta. Puede tener pausas o hacer otra cosa, pero ese thread acaba la tarea.
 - **Untied task**: a cualquier thread del team el scheduler le puede asignar una untied task que esté suspendida en ese momento

Tareas

- `#pragma omp task [clausulas]`
 bloque
 - `if (scalar expression)`
 - `untied`
 - `default(shared|none)` , `private` , `firstprivate` y `shared`
- `#pragma omp taskwait`
 - La tarea actual se suspende hasta la finalización de sus tareas hijas.
 - Un conjunto de tareas será forzado a completarse:
 - En una barrera implícita de threads.
 - En una barrera explícita de threads (`#pragma omp barrier`).
 - En una barrera de tareas (`#pragma omp taskwait`).

Tareas

- Secuencia de Fibonacci

```
int fib(int n) {  
    if (n < 2) return n;  
    int x = fib(n - 1);  
    int y = fib(n - 2);  
    return x + y;  
}
```

* <https://www.youtube.com/watch?v=Wx4eQQihP6I>

```
#pragma omp parallel  
#pragma omp single  
{  
    fib(10000);  
}
```

```
int fib(int n) {  
    if (n < 2) return n;  
    #pragma omp task shared(x)  
    int x = fib(n - 1);  
    #pragma omp task shared(y)  
    int y = fib(n - 2);  
    #pragma omp taskwait  
    return x + y;  
}
```


Medir tiempos

```
double inicio, fin;  
inicio = omp_get_wtime();  
/* código */  
fin = omp_get_wtime();  
  
printf("Tiempo: %2.f", fin - inicio);
```

+ Preciso

```
time ./programa
```

+ Fácil

```
Mooshak (calisto.inf.um.es, quad-core)
```

+ Speed-up

Código portable en OpenMP

```
#if defined (_OPENMP)
#include <omp.h>
#endif
int main() {
    int iam = 0, np = 1;
    #pragma omp parallel private(iam,np)
    {
        #if defined (_OPENMP)
            np = omp_get_num_threads();
            iam = omp_get_thread_num();
        #endif
        printf("Hello from thread %d out of %d \n", iam, np);
    }
}
```

Experimento rápido

```
for t in 1 2 4 8 12 16; do  
    OMP_NUM_THREADS=$t ./programa  
done
```

Variables de entorno

- `OMP_WAIT_POLICY=active`
 - Los hilos hacen esperar active (spin) en lugar de dormir (sleep)
- `OMP_DYNAMIC=false`
 - Obliga al runtime a usar exactamente tantos hilos como los que se indiquen
- `OMP_PROC_BIND=true`
 - Evita que los hilos puedan moverse entre los cores (thread/core affinity)

Bibliografía

- Diapositivas basadas en:
 - Curso de Domingo Giménez Cánovas
 - <http://dis.um.es/~domingo/mpp.html>
 - An Introduction to Parallel Programming, Peter Pacheco
 - Disponible en el Aula Virtual
 - Introducción a la Programación Paralela, Domingo Giménez Cánovas
 - Disponible en la biblioteca
 - Ejemplos y foros de Internet
 - Probar búsquedas como “OpenMP examples”