

OPEN MP BASICS

Constructs: elementos más importantes de OMP.

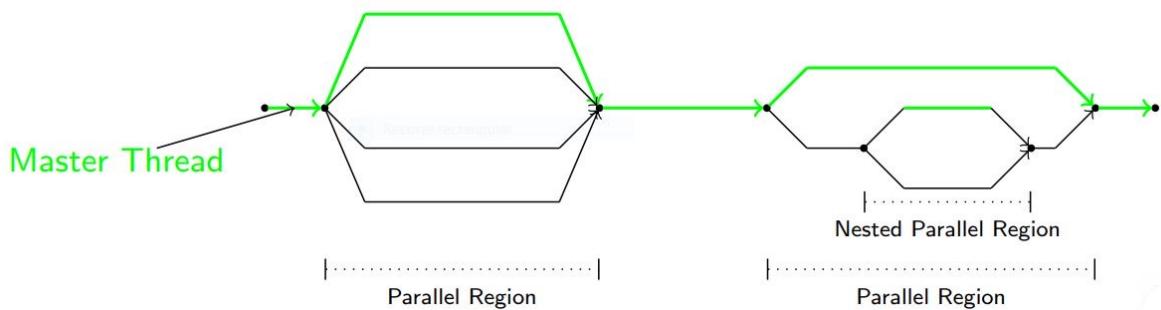
- ↳ Crean threads y tareas
- ↳ Comparten el trabajo entre threads.
- ↳ Sincronizan los threads y la memoria + esperan la terminación de tareas

Sintaxis: ~ directivas de compilador

`#pragma omp construct [clauses]`

- Se aplican sobre uno o varios statements.
- Tienen un punto de entrada y uno de salida → no branching

OPEN MP MODELS



Fork-join model:

- ↳ Thread master invoca un grupo de threads que se unen al final de la región paralela.

pueden colaborar para trabajar

Memory Model:

- Los threads pueden ver diferentes valores para la misma variable.
- Cada thread puede compartir variables o tenerlas privadas.

CREATING THREADS & ACCESSING DATA

Parallel construct

```
#pragma omp parallel [clauses]
```

structured block

Pueden ser:

Specifying #threads:

ICV (internal control variable)

`nthreads-var` se puede definir desde el CLI:

↳ `setenv OMP_NUM_THREADS X,y...`

- num_threads (expression)
- if (expression)
- shared (var-list)
- private(var-list)
- firstprivate(var-list)
- reduction(operator:var-list)

¿Cómo modificamos #threads?

- 1) `omp_set_num_threads` que modifica la "X" de `OMP_NUM_THREADS`.
- 2) `num_threads` provocando que la implementación ignore `OMP_NUM_THREADS` durante la ejecución de la región.

```
void main () {  
    #pragma omp parallel  
    ...  
    omp_set_num_threads(2);  
    #pragma omp parallel  
    ...  
    #pragma omp parallel num_threads(random()%4+1) if(0)  
    ...  
}
```

CLAUSES

if

sirve para especificar una expresión

→ false: sólo usa 1 thread (lo cual no significa que se ignore la región paralela)

shared

la variable marcada como shared, es la misma tanto dentro como fuera del construct.

→ todos los threads ven la misma variable pero no necesariamente el mismo valor.

→ Para actualizar los valores correctamente, normalmente se necesita algún tipo de sincronización.



Default: las variables son shared.

private

la variable marcada como private, dentro del constructo es una nueva variable del mismo tipo con un valor indefinido para cada thread.

→ Todos los threads tienen una variable diferente.

→ Puede accederse sin sincronización.

firstprivate

la variable marcada como firstprivate, dentro del constructo es una nueva variable del mismo tipo pero inicializada con el valor de la variable original

→ Todos los threads tienen una variable diferente con el mismo valor inicial.

→ Puede accederse sin sincronización.

SOME API CALLS:

Routines:

int

omp_get_num_threads

Devuelve #threads en el grupo
(0 si no paralelo)

int

omp_get_thread_num

Dev. id del thread en el grupo. $0 < id < \#th$.

void

omp_set_num_threads

Establece el #th a usar en las regiones paralelas

int

omp_get_max_threads

Dev. #threads que se pueden usar para la siguiente región paralela.

double

omp_get_wtime

Dev. #segundas desde un punto arbitrario.

THE SINGLE CONSTRUCT:

Dentro del grupo de threads sólo hay uno haciendo el trabajo.

#pragma omp single [clauses]

structured block

Pueden ser:

- private
- firstprivate
- nowait

* Default: después del structured block hay una barrera implícita.

Nowait quita la barrera implícita

→ permite solapar la ejecución de trabajos NO dependientes (no datarace)

dentro de un mismo single.

THREAD SYNCHRONIZATION:

shared memory model

- Datarace }
- Los threads se comunican compartiendo variables.
 - Necesitan sincronizarse para imponer orden a su secuenciamento.

Mecanismos de sincronización:

Barrier los threads NO pueden traspasar la barrera hasta que   TODOS lleguen a la barrera  TODO el trabajo previamente generado se haya completado



Algunos constructs tienen una barra implícita al final:

→ parallel

Critical genera una zona de exclusión mutua donde solo un thread puede trabajar un momento dado.
por defecto todas las regiones críticas son iguales.
Con un nombre se pueden generar múltiples zonas de exclusión mutua → sólo aquellas con nombre se sincronizan.

Atomic se asegura de que una posición concreta de memoria se acceda atómicamente, evitando la posibilidad de threads escribiendo y leyendo simultáneamente.
Sólo se pueden realizar operaciones sencillas: sumas, restas, igualaciones... Se comprenden en:

1 Atomic updates: $x = 1$, $x = x - \text{foo}()$, $x[\text{index}] +=$

2 Atomic reads: $\text{value} = *p$

3 Atomic writes: $*p = \text{value}$

Atomic > Critical en eficiencia.

Reduction: Todos los threads acumulan valores en una única variable.

reduction(operator : list)

↳ $+, -, *, |, ||, \&, \&&, ^, \min, \max$.

El compilador crea copias privadas de cada variable dentro de list y se inicializan correctamente

→ Al final de la región, el compilador se asegura de actualizar correctamente las variables.

Locks :

Para sincronización de bajo nivel

omp_lock()

Variables especiales que viven en memoria con 2 operadores básicos:

- **Acquire**: cuando un thread tiene el lock, nadie más lo coge, el thread hace el trabajo en privado
- **Release**: permite a otros threads adquirir el lock.

Primitivas:

→ **omp_init_lock**: inicializa el lock.

→ **omp_set_lock**: adquiere el lock.

→ **omp_unset_lock**: libera el lock

→ **omp_test_lock**: intenta adquirir el lock.

→ **omp_destroy_lock**: libera los recursos del lock.

```
#include <omp.h>
void foo ()
{
    omp_lock_t lock;

    omp_init_lock(&lock); ← Lock must be initialized before being used
    #pragma omp parallel
    {
        omp_set_lock(&lock);
        // mutual exclusion region ← Only one thread at a time here
        omp_unset_lock(&lock);
    }
    omp_destroy_lock(&lock);
}
```



jiji !!

SCREENSHOT^{SUB}

OPEN MP TASKS

TASK: Unidad de trabajo cuya ejecución se puede **delegar**

↳ los threads de un mismo **grupo** cooperan para ejecutarlos.

• **Implicita:** una región **paralela** crea tareas **implícitas** que se asignan a cada tarea.

• **Explícita:** un **task construct** la crea y empaqueta el código y los datos.

Creación:

Creando tareas explícitas

* `#pragma omp task [clauses]`

structured block

Las cláusulas pueden ser las vistas anteriormente.

||

`#pragma omp parallel` ← genera una región paralela

`#pragma omp single` ← hace que un thread entre en el bloque

{

:
`#pragma omp task` ← el thread que ha entrado en el bloque GENERA las tareas.

}

El resto, una vez se acaba la creación, cooperan para ejecutarlos



* Variables globales → compartidas

* Variables dentro tarea → privadas

→ El resto → Firstprivate (excepto si shared)

Final clause

Si la expresión del Final es True:

- La tarea generada y todas sus tareas hijas pasan a ser Final
- La ejecución de la tarea final se incluye inmediatamente en la tarea generadora

Sincronización de tareas

• Taskwait

(Padre)

Suspende la tarea actual esperando la consecución de sus tareas hijas

• Taskgroup

(Padre)

Suspende la tarea actual al final del bloque esperando a la consecución de las tareas hijas y de sus descendientes.

Compartir datos dentro de tareas

Uno puede usar critical y atómico para sincronizar los accesos a datos compartidos dentro de la tarea

Dependencias entre tareas

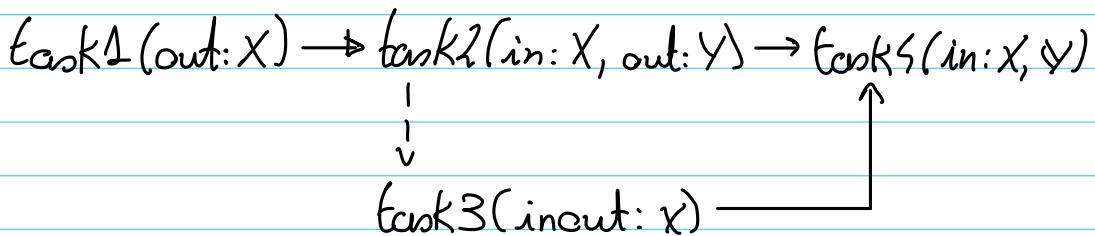
Permite definir dependencias entre tareas relacionadas
(del mismo paralelo)

```
#pragma open task [depend (in: var-list)]  
[depend (out: var-list)]  
[depend (inout: var-list)]
```

In: la cláusula `depend(in: X)` generará una tarea dependiente de todas las tareas generadas por `depend(out: X)` y `depend(inout: X)`

Out & Inout: La cláusula `depend(out: X)` y `depend(inout: X)` generan una tarea dependiente de todas las tareas mencionando X

Ejemplo



Tipos de dependencias

- 1) read-after-write: out → in
- 2) write-after-read: in → out
- 3) write-after-write: out → out.

Taskloop

```
#pragma omp taskloop [clauses]
    - for( init; test; increment)
```

Especifica que las iteraciones de los bucles controlados asociados serán ejecutadas en paralelo usando OpenMP task

⚠ Se asocia un taskgroup implícito para evitar ponerlos nogroup

Clause:

- Para compartición de datos
- shared (list)
 - private (list)
 - firstprivate (list)

- Para generación de tareas
- grainsize (n)
 - num-tasks (n)
 - if (expression)
 - final (expression)

Reduction

Cláusulas:

```
#pragma omp taskloop [{ reduction | in-reduction } (op: list) ]
```

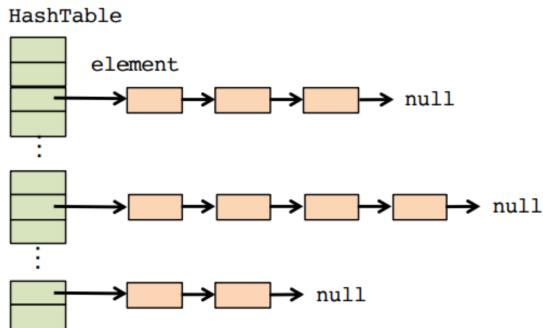
```
#pragma omp task [in-reduction (op: list) ]
```

```
#pragma omp taskgroup [task-reduction (id: list) ]
```

⚠ Las reducciones con tareas explicitas siempre ocurren en el entorno de taskgroup

Serialización:

Es fácil paralelizar usando `taskloop`, sin embargo, hay que proteger diferentes `regiones (slots)`



```
omp_lock_t hash_lock[SIZE_HASH];  
  
#pragma omp parallel  
#pragma omp single  
{  
    for (i = 0; i < SIZE_HASH; i++) omp_init_lock(&hash_lock[i]);  
  
    #pragma omp taskloop  
    for (i = 0; i < SIZE_TABLE; i++) {  
        int index = hash_function (dataTable[i], SIZE_HASH);  
        omp_set_lock (&hash_lock[index]);  
        insert_element (dataTable[i], index, HashTable);  
        omp_unset_lock (&hash_lock[index]);  
    }  
  
    for (i = 0; i < SIZE_HASH; i++) omp_destroy_lock(&hash_lock[i]);  
}
```

- Con un critical sería suficiente para evitar data races pero causaría una **serialización importante**.

→ se asocia un lock a cada índice de la HashTable.

Los threads pueden insertar elementos en paralelo mientras no los inserten en el mismo índice.

?? Dudas:

Que significa que dos criticals con el mismo nombre se si sincronicen (o dos con diferente nombre NO se sincronicen)

* Parallel dentro de Parallel

* Preguntar diapo 68

* Preguntar ejemplo in out

* Preguntar diapo 78: diferencia.

xg critical atomic?

critical & atomic generan barrera?