COMPUTACIÓN DE ALTA PERFORMANCE

Curso 2024

Sergio Nesmachnow (sergion@fing.edu.uy)

Centro de Cálculo

cecal

FACULTAD DE INGENIERIA

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

TEMA 7: OpenMP

CONTENIDO

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- Desarrollo de aplicaciones paralelas con OpenMP
- Conceptos y sintaxis
- Modelo de ejecución y memoria
- La API de OpenMP
  - Crear y distribuir trabajos
  - Paralelización de ciclos
  - Master, secciones y variables
  - Asignación de trabajo
  - Tareas

Paralelismo de ciclos

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- Distribuir un ciclo entre threads

```
#pragma omp for [cláusula[[,] cláusula] ...]
    {bloque estructurado: loop}
```

- Cláusula puede ser:
  - private(lista), firstprivate(lista), lastprivate(lista)
  - reduction(operador:lista)
  - schedule(tipo), permite definir la política de asignación de iteraciones a threads
  - nowait
  - collapse(n)
  - ordered

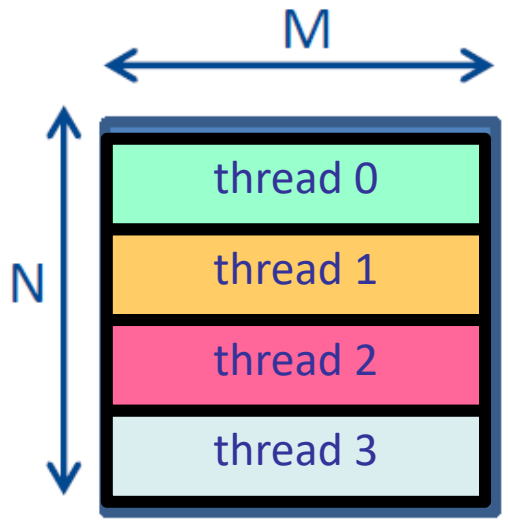
Paralelismo de ciclos

- Requisitos:
  - Iteraciones independientes (responsabilidad del programador)
  - Las variables de la iteración pueden ser enteros, punteros e iteradores.
  - El loop se debe expresar de manera de poder calcular el número de iteraciones

```
#pragma omp for [cláusula[[,] cláusula] ...]
    for(expresión inicial; condición; expresión de incremento)
```

- La variable de iteración se privatiza automáticamente
  - Otras variables deben privatizarse explícitamente

```
void foo(int *m, int N, int M){
int i, j;
#pragma omp parallel for private(j)
    for (i=0;i<N;i++){
            for (j=0;j<M;j++);
                    m[i*N+j] = 0;
}
```



Ciclos y lastprivate

```
void main(void){
int i=10;
int C=3, N=8;
int a[50];
#pragma omp parallel{
   #pragma omp for lastprivate(i)
   for (i=C;i<N-1;i++){
      a[i]=2*i;
   }
}
printf("i: %d\n",i);
}
```

```
void main(void){
int i=10;
int C=3, N=8;
int a[50];
#pragma omp parallel{
   #pragma omp for
   for (i=C;i<N-1;i++){
      a[i]=2*i;
   }
}
printf("i: %d\n",i);
}
```

- Cuál es el valor de i que se imprime en cada caso ?

Ciclos paralelos y asignación de trabajo

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- schedule determina qué iteraciones (chunks) son realizadas por cada thread
  - Cuándo no se indica, se utiliza el defindo por defecto por la implementación

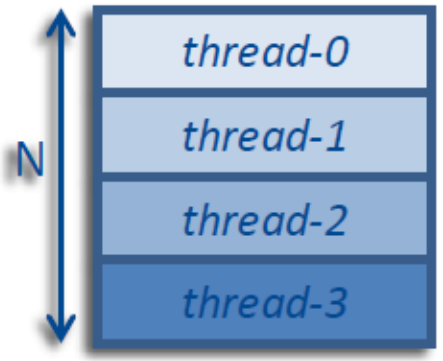
```
#pragma omp for schedule(tipo [,chunk size])
    for(expresión inicial; condición; expresión de incremento)
```

- Cinco tipos, tres con sus variantes
  - `static [,chunk size]`: bloques únicos o intercalados
  - `dynamic [,chunk size]`: asignación dinámica
  - `guided [,chunk size]`: asignación dinámica, chunks de tamaño decreciente
  - `auto`: asignación automática definida por la implementación
  - `runtime`: la asignación se retrasa hasta el momento de ejecución

Asignación de trabajo en ciclos: static

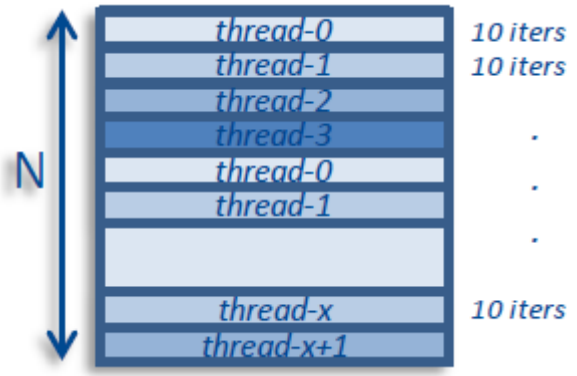
```
#pragma omp parallel for private(j) schedule(static)
    for (i=0;i<N;i++){
            for (j=0;j<M;j++);
                    m[i*N+j] = 0;
}
```

- static sin parámetros
  - El espacio de iteraciones se divide en partes (chunks) de aproximadamente el mismo tamaño.
  - Los chunks se asignan a los threads disponibles siguiendo una estrategia Round Robin




```
#pragma omp parallel for private(j) schedule(static,10)
```

- static con parámetro chunk size
  - El espacio de iteraciones se divide en partes (chunks) del tamaño indicado por el parámetro.
  - Los chunks se asignan a los threads disponibles siguiendo una estrategia Round Robin, intercalados

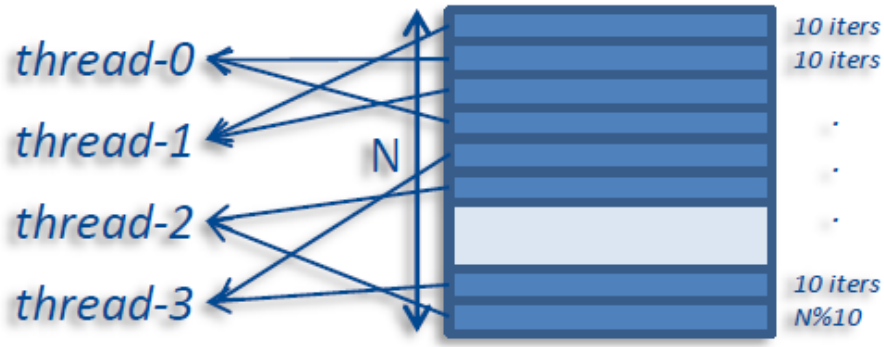


Asignación de trabajo en ciclos: dynamic

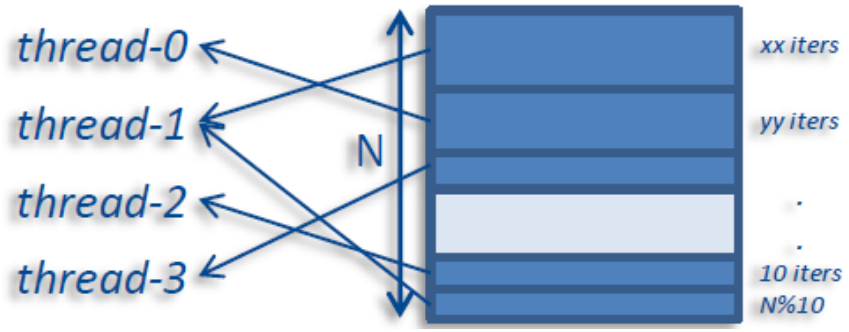
UNIVERSIDAD DE LA REPÚBLICA URUGUAY

```
#pragma omp parallel for private(j) schedule(dynamic,10)
    for (i=0;i<N;i++){
            for (j=0;j<M;j++);
                    m[i*N+j] = 0;
}
```

- dynamic
  - Los threads toman chunks dinámicamente
  - El chunk size por defecto es 1



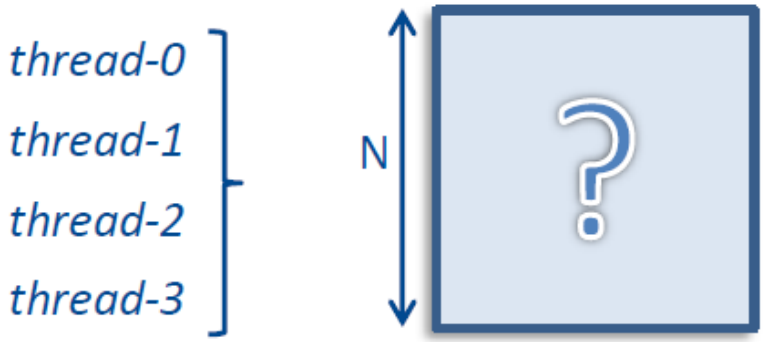
- guided (variante de dynamic, balance)
  - Los chunks decrecen en tamaño a medida que son tomados por los threads
  - El tamaño inicial es (aproximadamente) N/#threads
  - El tamaño final es el parámetro indicado



Asignación de trabajo en ciclos: auto/runtime

```
#pragma omp parallel for private(j) schedule(auto/runtime)
    for (i=0;i<N;i++){
            for (j=0;j<M;j++);
                    m[i*N+j] = 0;
}
```

- auto
  - La implementación decide cómo realizar la asignación
  - Posiblemente aplique static ☺



- runtime
  - Difiere la asignación a tiempo de ejecución
  - Permite utilizar la variable OMP_SCHEDULE
  - Se puede implementar en el código mediante la función omp_set_schedule()

```
void omp_set_schedule(omp_sched_t tipo,
int chunk_size);
```

```
$ export OMP_SCHEDULE=static,1024
$ ./myMatrixMultiply
Computing matrix multiplication...
```

Asignación de trabajo estática vs. dinámica

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

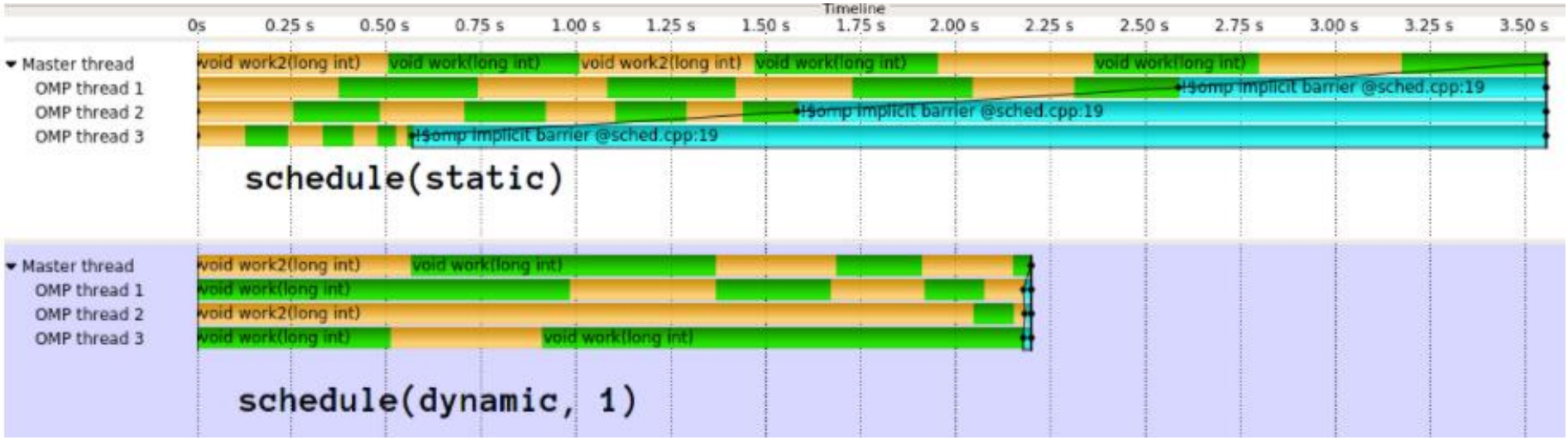
Asignación estática	Asignación dinámica
Bajo overhead (pocas comunicaciones)	Mayor overhead (requiere comunicaciones)
Buena localidad (e.g., para loops consecutivos)	Localidad no tan buena
Carga no balanceada	Carga balanceada

- Asignación estática
  - Si los threads alcanzan el loop al mismo tiempo
  - Si todas las iteraciones tienen la misma carga de trabajo
  - Si loops consecutivos utilizan los mismos datos (por ejemplo, en el procesamiento matricial, permite aprovechar el cache)
- Asignación dinámica o guiada
  - Si los threads alcanzan el loop en diferentes momentos
  - Si las iteraciones tienen diferentes cargas de trabajo
  - guided para mejorar el balance cuando las iteraciones tienen cargas de trabajo muy diferentes

Caso de estudio: carga de trabajo variable

```
#include <omp.h>
void work(long ww) {
    volatile long sum = 0;
    for (long w = 0; w < ww; w++) sum += w;
}
int main() {
    const long max = 32, factor = 100000001;
    #pragma omp parallel for schedule(guided, 1)
    for (int i=0;i<max;i++) {
        work((max-i)*factor);
    }
}
```

La carga de trabajo decrece con las iteraciones



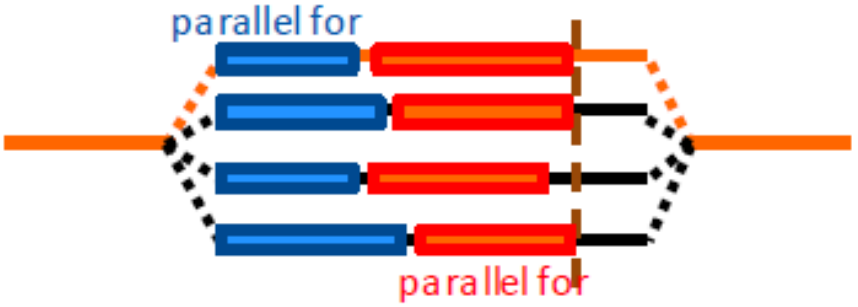
Evitar la barrera implícita al final de un ciclo

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

```
#pragma omp for nowait
```

- Permite solapar la ejecución de bucles independientes

```
#define N 1000
void main (void) {
   int i, a[N],b[N];
   #pragma omp parallel{
        #pragma omp for nowait
        for(i=0;i<N;i++)
                a[i] = 0;
        #pragma omp for
        for(i=0;i<N;i++)
                b[i] = 0;
   }
}
```

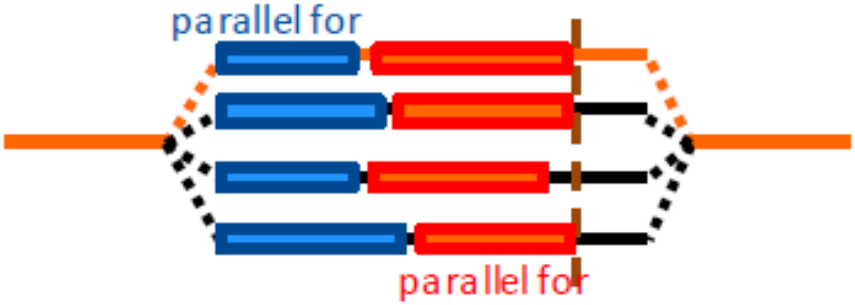


- Una mejor solución si los bucles trabajan sobre el mismo espacio de iteración, es combinarlos manualmente

Evitar la barrera implícita al final de un ciclo

- `nowait` también permite solapar la ejecución de algunos bucles independientes

```
#define N 1000
void main (void) {
   int i, a[N],b[N];
   #pragma omp parallel{
        #pragma omp for schedule(static) nowait
        for(i=0;i<N;i++)
                a[i] = 0;
        #pragma omp for schedule(static)
        for(i=0;i<N;i++)
                a[i] = a[i] + foo(i);
   }
}
```



  - Asignación estática sobre el mismo espacio de iteración y las iteraciones dependen de los índices
  - Una mejor opcion es combinarlos manualmente

Evitar la barrera implícita al final de un ciclo

- La solución no es genérica, NO FUNCIONA en otros casos

```
void main (void) {
   int i, a[N],b[N];
   #pragma omp parallel{
        #pragma omp for schedule(dynamic) nowait
        for(i=0;i<N;i++)
                a[i] = 0;
        #pragma omp for schedule(dynamic)
        for(i=0;i<N;i++)
                a[i] = a[i] + foo(i);
   }
}
```

  - Sin asignación estática y con iteraciones que dependen de los índices
    - La única solución correcta es combinarlos manualmente
  - Si el espacio de iteraciones no es el mismo
  - Si hay dependencias

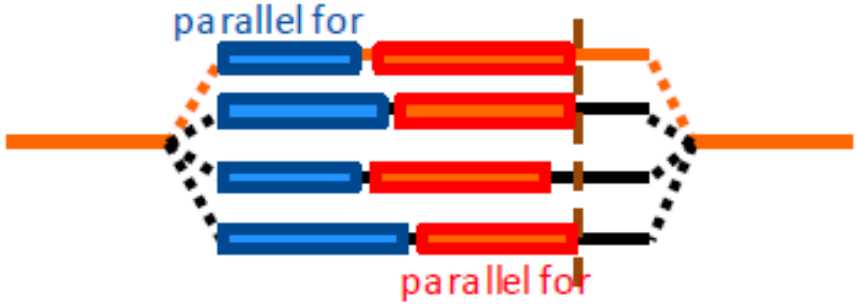
Combinar dos ciclos anidados

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

`#pragma omp for collapse(n)`

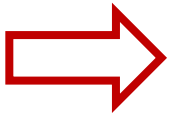
- Permite solapar la ejecución de bucles independientes

```
void main(void) {
int i, j;
#pragma omp parallel {
#pragma omp for collapse(2)
   for(i=0;i<N;i++)
       for(j=0;j<M;j++)
              foo(i,j);
}
}
```



- Útil cuando el primer loop (o ambos) tienen pocas iteraciones.
- Los ciclos deben estar perfectamente anidados (sin instrucciones entre ellos)

```
#pragma omp for collapse(2)
    for(i=0;i<N;i++)
       for(j=0;j<M;j++)
           foo(i,j);
```



```
#pragma omp for
    for(idx=0;idx<(N*M);idx++)
        foo(fi(idx),fj(idx));
```

Sincronizaciones

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- Las sincronizaciones son necesarias en muchos casos:
  - Para imponer un orden para acceder a ciertas regiones
  - Para asegurar la exclusion mutua en el acceso
  - Para aguardar a otros thread en un punto del flujo de ejecución
  - Para esperar que una cierta condición se cumpla
- OpenMP prové diferentes mecanismos de sincronización
  - Los constructores master y critical (ya presentados)
  - La directiva barrier
  - El constructor atomic
  - taskwaiting, taskgroup y depend (para tareas)

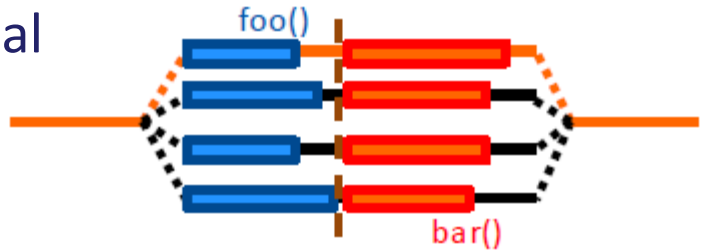
Barreras

```
#pragma omp barrier
```

- Los threads esperan en la barrera
  - Hasta que todos los threads del equipo alcanzan la barrera
  - Algunos constructores tienen una barrera implícita al final (por ejemplo, el constructor parallel)

```
#pragma omp parallel{
    foo();
    #pragma omp barrier
    bar();
}
```

  - Fuerza a que todas las invocaciones a foo() finalicen antes de ejecutar la primera invocación a bar()
  - El constructor agrega una barrera implícita al final



Exclusión mutua con atomic

```
#pragma omp atomic
```

- Mecanismo especial de exclusion mutua para operaciones de lectura y escritura
  - Solo soporta expresiones simples de lectura y modificación

```
int x = 1
#pragma omp parallel num_threads (2)
#pragma omp atomic
    x++;
printf (“x: %d\n”,x);
```

  - Siempre imprime x: 3
  - atomic protege solamente a la variable que se lee o modifica
    - `x++` protege a x
    - `x = x + foo()` protege a x, pero no protege a foo()
  - atomic es más eficiente que critical, pero no son compatibles entre si
    - No se garantiza un correcto funcionamiento si una misma variable o expresión se declara atómica y luego crítica

OpenMP: tareas

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- Las tareas son unidades de trabajo que pueden ejecutarse inmediatamente o diferirse
  - Están disponibles a partir de OpenMP 3.0
  - Proveen soporte para cómputos no estructurados
- Las tareas se componen de:
  - Un código a ejecutar (conjunto de instrucciones, funciones, etc.)
  - Un entorno de datos, que se inicializa al momento de crear la tarea
  - Un conjunto de variables de control internas
- En OpenMP las tareas se crean:
  - Cuando se encuentra un constructor task (creación explícita de una tarea)
  - Cuando se encuentra un constructor taskloop (creación explícita de una tarea por chunk en un loop)
  - Cuando se encuentra un constructor target (creación explícita de una tarea destino)

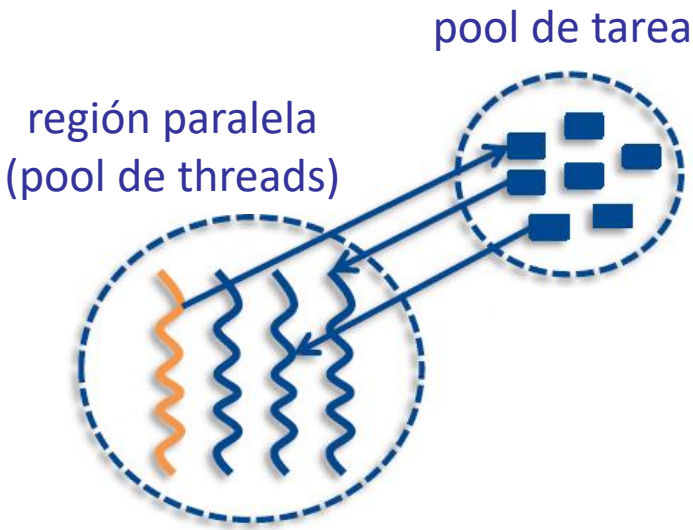
Modelo de ejecución de tareas

- Provee soporte para paralelismo no estructurado
  - Ciclos sin límite preestablecido

```
while ( <expr> ) {
    ...;
}
```

  - Invocaciones a funciones recursivas

```
void myCode( <args> ) {
    ...;
    myCode( <args> );
    ...;
}
```



- Se pueden configurar tareas en varios escenarios:
  - Un único creador vs. multiples creadores
  - Múltiples ejecutores: todos los threads del equipo son candidatos a ejecutar tareas

Constructor de tareas

```
#pragma omp task [cláusula [[,] cláusula]...]
{bloque estructurado}
```

- Define (y difiere) una unidad de trabajo
  - Ejecutable por cualquier thread del equipo
- Cláusulas
  - private(lista), firstprivate(lista), shared(lista)
  - untied
  - if (expression escalar)
  - mergeable
  - final (expression escalar)
  - priority (valor)
  - depend (tipo: lista)

Entorno de datos para tareas

- Reglas implícitas para compartir datos en el entorno de tareas
  - Se hereda lexicamente el atributo shared (no siempre son compartidas)
    - Una variable será compartida en una tarea si al momento de crear la tarea está definida como shared.
    - Todas las demás variables son firstprivate

```
int a ;
void foo (int b){
int c,d;
#pragma omp parallel private(c){
    int e;
    #pragma omp task{
        int g;
        a = <expr>;    // a es shared (está definida como global)
        b = <expr>;    // b es shared (es parámetro, local a void foo)
        c = <expr>;    // c es firstprivate (se define como private)
        d = <expr>;    // d es shared (por defecto)
        e = <expr>;    // e es firstprivate (se define dentro de parallel)
        g = <expr>;    // g es firstprivate (es local a la tarea)
    }
}}
```

Ejemplo de tareas

- Recorrer una estructura tipo arbol

```
struct node {
    struct node *left;
    struct node *right;
};
extern void process (struct node *);
void traverse (struct node *p) {
    if (p->left)
        #pragma omp task
            traverse(p->left);
    if (p->right)
        #pragma omp task
            traverse(p->right);
    process(p);
}
```

p es firstprivate por defecto

  - Las tareas deben ejecutarse dentro de una region paralela para que ejecuten simultáneamente
  - No se incluyen sincronizaciones, por lo cual las tareas pueden ejecutar en cualquier orden (no debe asumirse posorden, como en un código secuencial)

Ejemplo de tareas

- Recorrer una estructura tipo arbol posorden (izquierdo, derecho, raíz)

```
struct node {
    struct node *left;
    struct node *right;
};
extern void process (struct node *);
void traverse (struct node *p) {
    if (p->left)
        #pragma omp task
            traverse(p->left);
    if (p->right)
        #pragma omp task
            traverse(p->right);
    #pragma omp taskwait
    process(p);
}
```

p es firstprivate por defecto

  - Se incluye una sincronización con taskwait, que garantiza el procesamiento en posorden

Tareas ligadas y no ligadas

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- Por defecto las tareas están ligadas a un único thread
  - Siempre ejecutan en el mismo thread
  - Esta decision puede condicionar el desempeño de la aplicación
- Para desligar las tareas se utiliza la cláusula `untied`

```
#pragma omp task untied
{bloque estructurado}
```

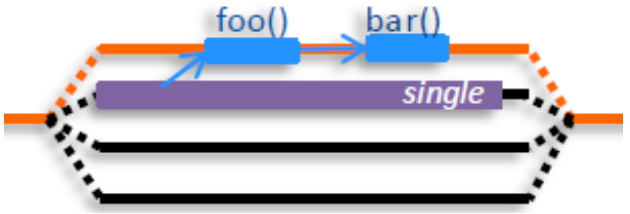
  - Una tarea no ligada potencialmente puede ejecutarse en [y cambiarse a] cualquier thread del equipo
  - Puede ocasionar inconvenientes con thread id, threadprivate, critical, y otras construcciones que referencian o se relacionan a un thread
  - Por otra parte, proporciona más flexibilidad al entorno de ejecución para la asignación de threads/recursos

Tareas: suspender y resumir

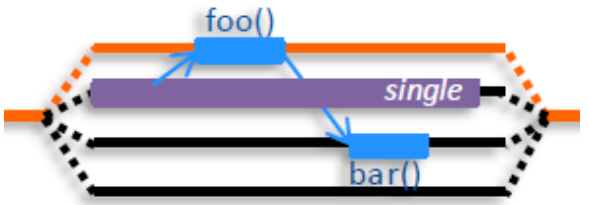
- Los scheduling points son puntos especiales en los cuales las tareas pueden ser suspendidas/resumidas (checkpoints).
  - Se imponen algunas restricciones para evitar deadlocks
  - La creación de tareas y los puntos de sincronización son scheduling points implícitos (el thread que crea tareas puede participar en su ejecución)
  - Los scheduling points se pueden definir explícitamente con la directiva `#pragma omp taskyield`

```
#pragma omp parallel
#pragma omp single{
    #pragma omp task [untied]{
        foo();
    #pragma omp taskyield
        bar();
    }
}
```

tied



untied



  - Thread esperando en un lock puede suspender la tarea y trabajar en otra

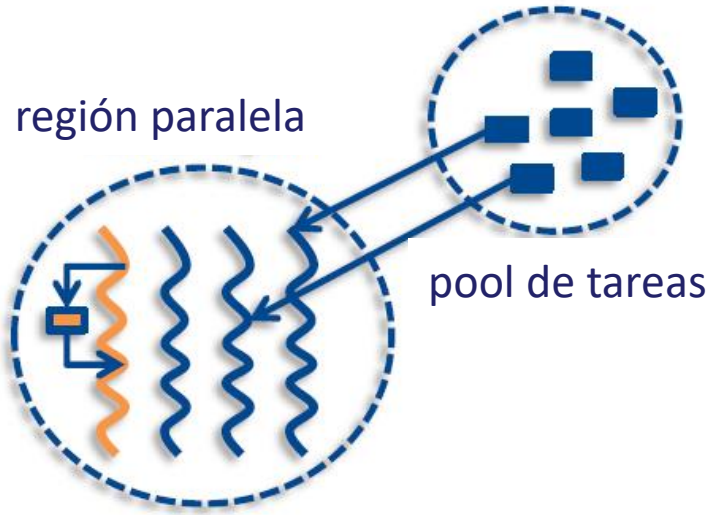
```
while (!omp_test_lock(lock))
    #pragma omp taskyield
```

Control de la asignación de tareas

```
#pragma omp task if(expresión)
```

- La tarea siempre se crea, el if indica si se define como no diferible
- Si la expresión evalúa a true: tarea normal (va al pool de tareas)
- Si la expresión evalúa a falso:
  - La tarea creadora se suspende
  - La nueva tarea se ejecuta lo antes posible
  - La tarea padre resume su ejecución cuando la nueva tarea finaliza

- Permite optimizar la creación y ejecución de tareas
  - Reduce el paralelismo, pero también reduce el impacto del overhead (no usa cola de ejecución, etc.)



Tareas incluidas y finales

- Una tarea es incluida cuando su ejecución se haya incluida secuencialmente en otra tarea (generadora)
  - Una tarea incluida es no diferible y se ejecuta de manera inmediata por el thread en ejecución
- Dos maneras de generar una tarea incluida: tarea hija o tarea final

```
#pragma omp task final(expresión)
{bloque estructurado}
```

- Permite definir una tarea final
  - La tarea final se crea y se ejecuta normalmente
  - En el contexto de la tarea final, todas las tareas se generarán como incluidas

```
#pragma omp task if(0){ // No diferida
    #pragma omp task    // Tarea regular
        for(i=0;i<3;i++){
            #pragma omp task // Tarea regular
            bar();
        }
}
```

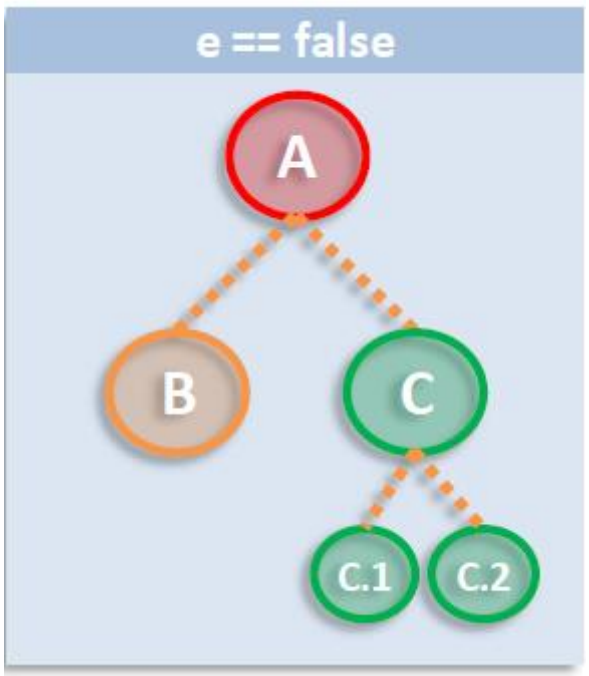
```
#pragma omp task if(1){ // Tarea regular
   …
}
```

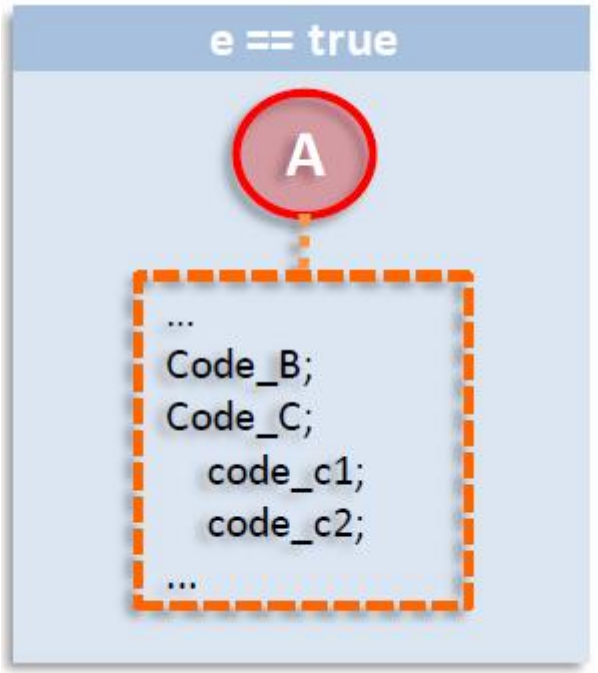
Tareas incluidas y finales

- Creación de un grafo de tareas

```
#pragma omp parallel
#pragma omp single // Solo un thread realiza la creación de las tareas
{
    #pragma omp task final(e)
    {
    #pragma omp task
        { code B; }
    #pragma omp task
        { code C; }
    }
    #pragma omp taskwait
}
```

– El código C puede crear tareas internamente (e.g., C.1 y C.2)
– Para esperar por la finalización de las tareas se utiliza taskwait





Tareas fusionables

```
#pragma omp task mergeable [if(expresión)]
```

- Permite la ejecución de una tarea en el mismo entorno de datos de la tarea generadora, reduciendo el overhead (no hay asignación ni creación de un nuevo entorno de datos)
  - La tarea generadora debe ser no diferida o incluida
- Tareas no diferidas y fusionables pueden ejecutar como un llamado a función, pero sin variables privadas

```
#include <stdio.h>
void foo(){
   int x = 2;
   #pragma omp task shared(x) mergeable
   {
        x++;
   }
   #pragma omp taskwait
   printf("%d\n",x);      // Imprime 3
}
```

Sin definir la variable x como compartida, la definición no es correcta !!

Prioridad de tareas

```
#pragma omp task priority(valor)
```

- Permite especificar (sugerir) una prioridad para la tarea
  - Valores más altos indican mayor prioridad y la tarea será asginada antes
- Todas las tareas prontas para ejecución se insertan en una cola y cuando un thread queda disponible, ejecuta (una de las) tarea(s) de mayor valor de prioridad.

```
#pragma omp parallel
#pragma omp single
{
   for(i=0;i<SIZE;i++){
       #pragma omp task priority(1)
       { code_A ;}
   }
   #pragma omp task priority(100)
       { code_B ;}
}
```

- La tarea del código B tiene prioridad para la ejecución
  - Por ejemplo, puede ser el envío de un mensaje importante
  - Podría ejecutarse entre las tareas del for

Sincronización de tareas

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

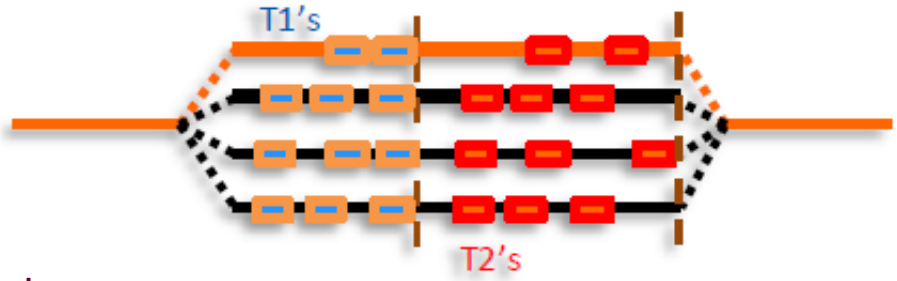
- OpenMP prové diferentes mecanismos para sincronización
- Para threads (ya vistos):
  - master
  - critical
  - atomic
- Barrier (agrega semántica para tareas)
- Específicos para tareas:
  - taskwait
  - taskgroup
  - depend

Sincronización de tareas: barrier

```
#pragma omp barrier
```

- Al comportamiento ya explicado (los threads no pueden continuar hasta que todos los threads hayan alcanzado la barrera) se agrega una nueva condición: todas las tareas previamente generadas deben completarse
  - Recordemos que algunos constructores tienen una barrera implícita al final (parallel, single, sections, for, etc.)

```
#pragma omp parallel
{
   #pragma omp master
   generate_task_T1();
   #pragma omp barrier
   #pragma omp master
   generate_task_T2();
}
```



→ Fuerza a que todas las tareas T1 ejecuten y finalicen

→ Barrera implícita, fuerza a que todas las tareas T2 finalicen

Sincronización de tareas: taskwait

```
#pragma omp taskwait
```

- Permite esperar por la finalización de tareas hijas
  - Solo hijas directas, no otras tareas descendientes (sincronización “plana”)
  - Incluye un punto implícito de asignación/checkpoint

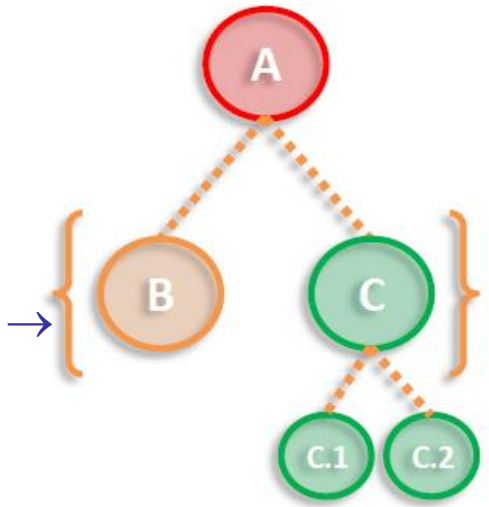
```
#pragma omp parallel
#pragma omp single
{
   #pragma omp task
   {
      #pragma omp task
      { ... B }

       #pragma omp task
        { ... C }

   #pragma omp taskwait
   }
}
```

→ La tarea A espera por la finalización de sus hijas

La tarea C puede crear nuevas tareas



  - Se espera por C1 y C2 al final del constructor single/parallel

Sincronización de tareas: taskgroup

```
#pragma omp taskgroup
{bloque estructurado}
```

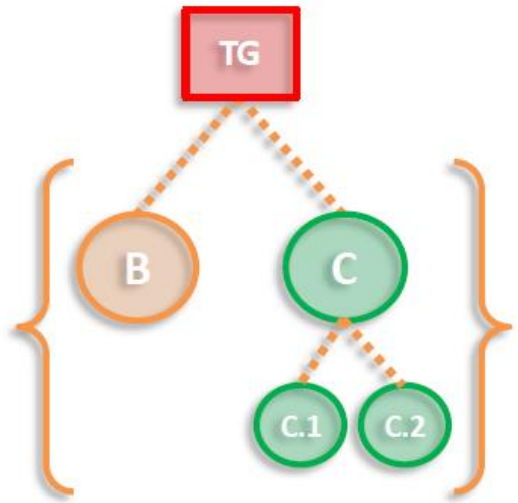
- No crea tareas, crea un contexto que permite esperar por la finalización de todas las tareas descendientes
  - Todas las tareas asociadas al bloque estructurado (sincronización “profunda”)
  - Incluye un punto implícito de asignación/checkpoint al final del constructor

```
#pragma omp parallel
#pragma omp single{
   #pragma omp taskgroup
   {
       #pragma omp task
       { ... B }

       #pragma omp task
       { ... C }
   }
}
```

La tarea C puede crear nuevas tareas

→ Se espera por la finalización de todas las tareas descendientes en el grupo

→ 

Dependencias entre tareas

```
#pragma omp task depend (tipo:lista)
```

- La clausula no indica una dependencia de control, sino una dependencia de datos
  - Especifica la direccionalidad de una lista de variables
- El tipo de dependencia puede ser
  - in: la tarea solamente lee de los datos especificados
  - out: la tarea solamente escribe a los datos especificados
  - inout: la tarea lee y escribe los datos especificados
- En la lista pueden indicarse
  - Variables, un bloque de memoria con nombre
  - Secciones de arreglos (un subconjunto del arreglo: A[inicio:largo])
- Tipos de dependencia
  - Read after write, write after read, write after write, read after read

Dependencias entre tareas

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- Las dependencias indican cómo dos tareas usan datos que las relacionan
- Read after Write (RaW)
  - La segunda tarea lee un dato que escribe la primera tarea
  - La segunda tarea debe ejecutarse después de la primera

```
#pragma omp task depend(out:x)
   foo(x)
#pragma omp task depend(in:x)
   foo(x)
```

- Write after Read (WaR), también llamada “antidependencia”
  - La primera tarea lee un dato y la segunda tarea lo sobrescribe.
  - La segunda tarea debe ejecutarse luego de la primera, para evitar sobrescribir el valor inicial

```
#pragma omp task depend(in:x)
   foo(x)
#pragma omp task depend(out:x)
   foo(x)
```

Dependencias entre tareas

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

- Las dependencias indican cómo dos tareas usan datos que las relacionan
- Write after Write (WaW)
  - Ambas tareas escriben la misma variable
  - Dado que la variable puede ser utilizada por una tarea intermedia, las dos escrituras deben ejecutarse en el orden correcto [indicado por el código].

```
#pragma omp task depend(out:x)
   foo(x)
#pragma omp task depend(out :x)
   foo(x)
```

- Read after Read (RaR)
  - Ambas tareas leen una variable
  - Dado que ninguna de las tareas tiene una declaración "out", pueden ejecutarse en cualquier orden.

```
#pragma omp task depend(in:x)
   foo(x)
#pragma omp task depend(in:x)
   foo(x)
```

Dependencias entre tareas

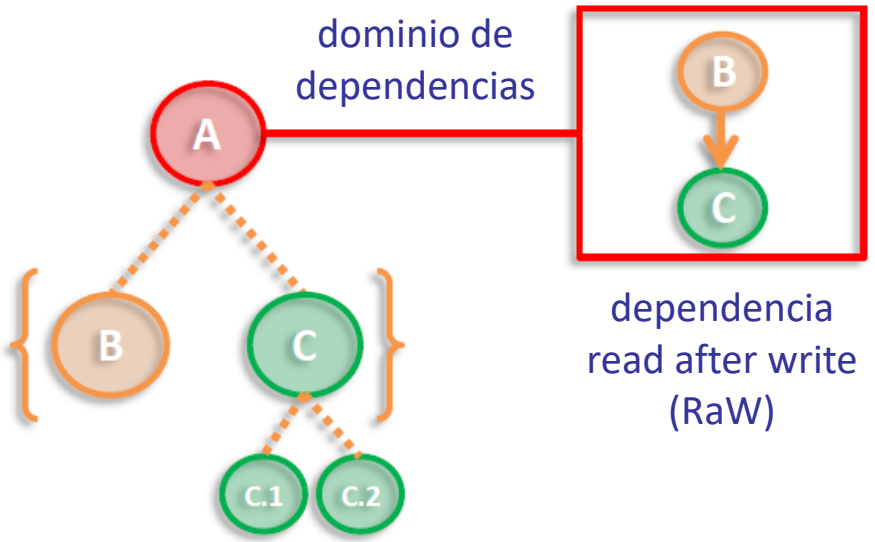
```
#pragma omp task depend (in:lista)
```

- La tarea depende de todas las tareas hermanas que referencian al menos a uno de los elementos en la lista en una dependencia out o inout
  - Esos elementos serán utilizados como entrada

```
#pragma omp task depend (out/inout:lista)
```

- La tarea depende de todas las tareas hermanas que referencian al menos a uno de los elementos en la lista en una dependencia in, out o inout

```
#pragma omp parallel
#pragma omp single
{
   #pragma omp task
   {
      #pragma omp task depend(out:a)
      { ... B }
      #pragma omp task depend(in:a)
      { ... C }
}
```



Dependencias entre tareas

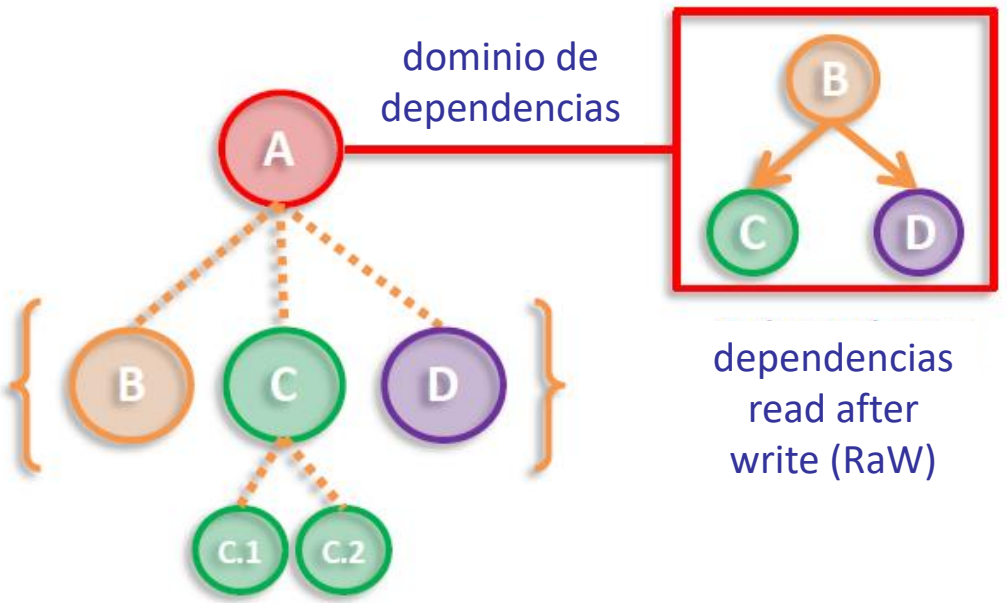
- Dependencias entre un escritor y n lectores

```
#pragma omp parallel
#pragma omp single
{
   #pragma omp task
   {
      #pragma omp task depend(out:a)
      { ... B }

      #pragma omp task depend(in:a)
      { ... C }

      #pragma omp task depend(in:a)
      { ... D }

   #pragma omp taskwait
   }
}
```



Dependencias entre tareas

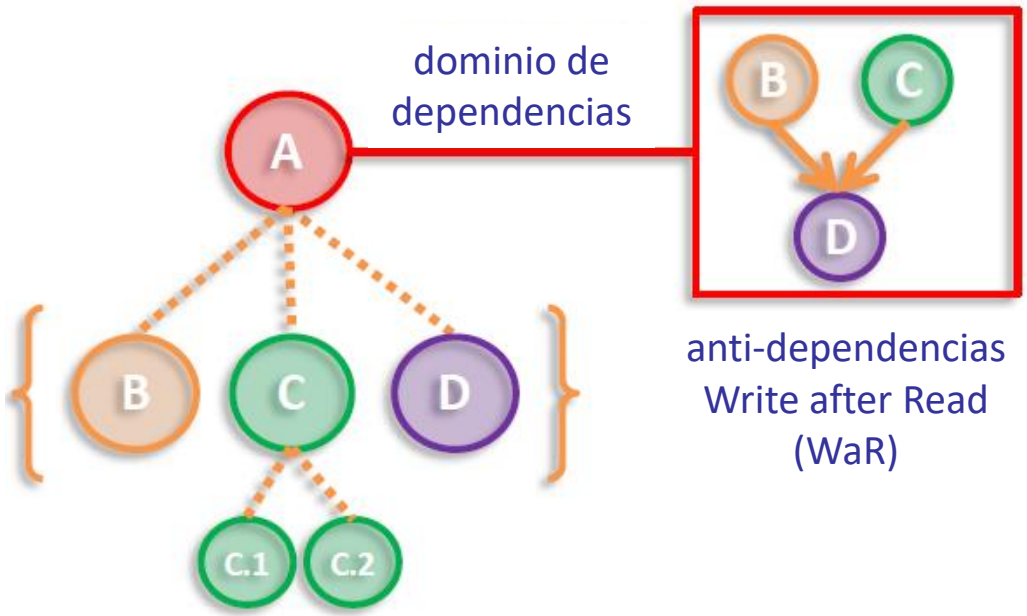
- Dependencias entre n lectores y un escritor

```
#pragma omp parallel
#pragma omp single
{
   #pragma omp task
   {
      #pragma omp task depend(in:a)
      { ... B }

      #pragma omp task depend(in:a)
      { ... C }

      #pragma omp task depend(out:a)
      { ... D }

   #pragma omp taskwait
   }
}
```



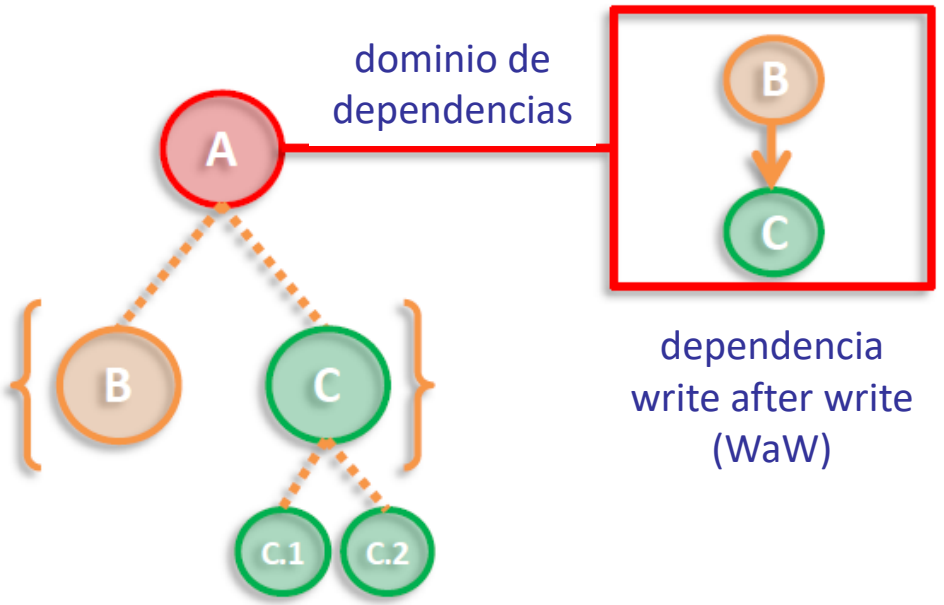
Dependencias entre tareas

- Dependencia entre dos escritores

```
#pragma omp parallel
#pragma omp single
{
   #pragma omp task
   {
      #pragma omp task depend(out:a)
      { ... B }

      #pragma omp task depend(out:a)
      { ... C }

   #pragma omp taskwait
   }
}
```



Dependencias entre tareas

```
#pragma omp task depend(tipo:lista)
```

- Restricciones en los elementos de la lista:
  - Los elementos en la lista de dependencias de una tarea o de tareas hermanas deben tener almacenamiento idéntico o disjunto
  - No se puede incluir una sección de arreglo de largo nulo
  - No se puede incluir una variable que sea parte de otra variable (por ejemplo, un campo de una estructura), salvo que sea una sección de arreglo

```
#define N 100
...
#pragma omp task depend(out:a[0..N])
{ ... B }

#pragma omp task depend(out:a[25:50])
{ ... C }
```

Dependencias entre tareas

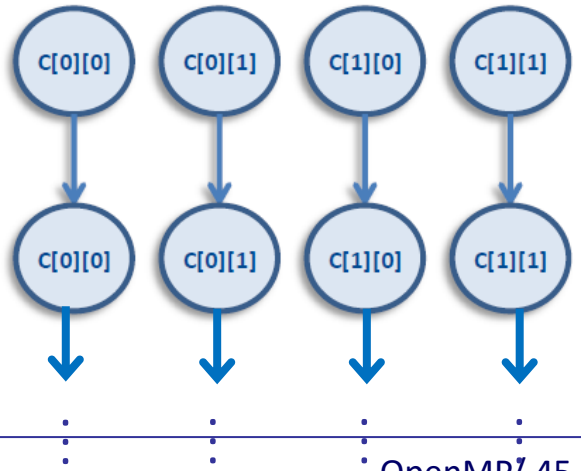
- Ejemplo: multiplicación de matrices (se asume que BS divide a N)

```
voidmatmul_block(intN,intBS,float*A,float*B,float*C){
#pragma omp parallel
#pragma omp single{
    int i,j,k;
    for(i=0;i<N;i+=BS){
        for(j=0;j<N;j+=BS){
            for(k=0;k<N;k+=BS){
                #pragma omp task depend (in:A[i:BS][k:BS],B[k:BS][j:BS])\
                                 depend (inout:C[i:BS][j:BS])
                    matmul_block(N,BS,&A[i][k],&B[k][j],&C[i][j]);
                }
        }
    }
}
}
```

Entrada (se lee de memoria)

A[i][k] B[k][j]

A[0][0]	A[0][1]	×	B[0][0]	B[0][1]	=	C[0][0]	C[0][1]
A[1][0]	A[1][1]		B[1][0]	B[1][1]		C[1][0]	C[1][1]



- Dependencia real: inout (para las acumulaciones en C[i][j])

Taskloop

- Permite evitar dos restricciones impuestas para el constructor loop/for
  - Todos los threads en el equipo deben alcanzar el constructor
  - No se pueden anidar constructores paralelos

```
void main (void){
#pragma omp parallel
#pragma omp sections
{
    #pragma omp section
        synthetic_phase1();
    #pragma omp section
        synthetic_phase2();
    #pragma omp section
        synthetic_phase3();
}
}
```

```
void synthetic_phase2(){
#pragma omp for
    for(i=0;i<N;i++)
        { ... }
}
```

```
void synthetic_phase2(){
#pragma omp taskloop
    for(i=0;i<N;i++)
        { ... }
}
```

  - El constructor está específicamente diseñado para paralelismo anidado en el modelo de tareas

Taskloop

```
#pragma omp taskloop [clausula[[,] clausula]...]
{bloque estructurado:ciclo}
```

- Permite diferir/paralelizar la ejecución de un ciclo con tareas
  - Particiona el espacio de iteraciones del loop asociado en tareas explícitas para su ejecución en paralelo usando el thread pool
  - Cláusulas: if, shared(lista), private(lista), lastprivate(lista), reduction(op:lista) num_tasks(n), collapse(n), final(expr), priority(valor), untied, mergeable, allocate(op:lista)
  - grainsize(size): valor mínimo de iteraciones asignadas a cada tarea, el valor máximo es 2×size.
  - nogroup: no se crea un entorno taskgroup

Taskloop grainsize

```
#pragma omp taskloop grainsize(size)
{bloque estructurado:ciclo}
```

- Permite definir la granularidad de los subbloques (tareas) generados
  - La granularidad será mayor o igual a min(size,#iteraciones)
  - La granularidad será menor a 2×size
  - Busca un compromiso entre el número de tareas y la cantidad de trabajo que realiza cada tarea
- No puede combinarse con la cláusula num_tasks

```
void synthetic_phase2() {
#pragma omp taskloop grainsize(10)
    for(i=0;i<N;i++)
       { ... }
}
```

Taskloop num_tasks

```
#pragma omp taskloop num_tasks(num)
{bloque estructurado:ciclo}
```

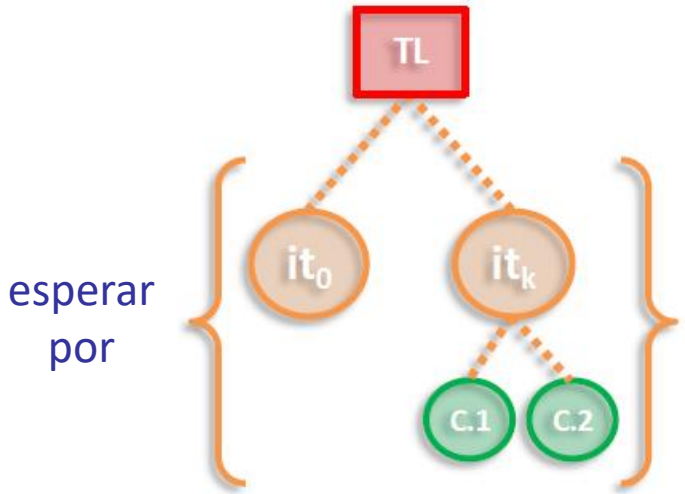
- Permite definir el número de tareas generadas
  - Debe ser mayor o igual a min(num-tasks,#iteraciones)
  - Cada tarea puede tener como mínimo una iteración del ciclo
  - Busca definir el grado de paralelismo deseado para la ejecución
- No puede combinarse con la cláusula grainsize

```
void synthetic_phase2() {
#pragma omp taskloop num_tasks(5)
    for(i=0;i<N;i++)
       { ... }
}
```

Taskgroup asociado con taskloop

- Taskloop puede asociarse con un taskgroup (contexto)

```
void synthetic_phase2() {
#pragma omp taskgroup
{
    #pragma omp taskloop grainsize(10)
        for(i=0;i<N;i++)
        { ... }
}
foo();
bar();
}
```




```
#pragma omp taskloop nogroup
```

- La cláusula nogroup permite continuar la ejecución de las tareas sin esperar por las restantes tareas creadas

Ejemplos

1. Cálculo de la secuencia de Fibonacci

```
#include <stdio.h>
#include <omp.h>
#define THRESHOLD 5
int fib(int n){
  int i, j;
  if (n<2)                    // paso base de la recursión
      return n;               // el paso recursivo genera dos tareas para las invocaciones recursivas
  #pragma omp task shared(i) firstprivate(n) final(n <= THRESHOLD)
      i=fib(n-1);             // calcula F(n-1), la variable n es privada
  #pragma omp task shared(j) firstprivate(n) final(n <= THRESHOLD)
      j=fib(n-2);             // calcula F(n-2)
  #pragma omp taskwait
  return i+j;
}
```

Si n <= THRESHOLD las tareas generadas serán finales. En ese caso se generarán tareas incluidas, que serán ejecutadas inmediatamente por el thread en ejecución, reduciendo el overhead de colocarlas en el pool de tareas y realizar una posterior asignación

Ejemplos

1. Cálculo de la secuencia de Fibonacci

```
int main(){
  int n = 30;
  omp_set_dynamic(0);
  omp_set_num_threads(4);          // pool de 4 threads
  #pragma omp parallel shared(n) // variable n compartida
  {
     #pragma omp single            // solo un thread realiza la invocación inicial e imprime resultado
         printf ("fib(%d) = %d\n", n, fib(n));
  }
}

% gcc -fopenmp -O3 –o Fibonacci task_example.cc

% ./fibonacci
fib(30) = 832040
```

Ejemplos

2. Dependencia entre tareas

```
#include <omp.h>
int main(){
int a,b,c;
#pragma omp parallel {
   #pragma omp master {
      #pragma omp task depend(out:a) {
         #pragma omp critical
            printf ("Task 1\n");
        }
      #pragma omp task depend(out:b) {
        #pragma omp critical
            printf ("Task 2\n");
      }
      #pragma omp task depend(in:a,b) depend(out:c) {
        printf ("Task 3\n");
      }
      #pragma omp task depend(in:c){
        printf ("Task 4\n");
      }
}
```

Las cuatro tareas generadas son hermanas y ejecutan en la misma región paralela.
La tarea 3 define dependencias en la variables a y b, por lo cual depende de las tareas 1 y 2.
Del mismo modo, la tarea 4 depende de la tarea 3.
La tarea 3 nunca puede ejecutarse antes de que finalicen 1 y 2. La tarea 4 siempre se ejecuta al final.

```
% cc -xopenmp -O3
task_depend.c
% a.out
Task 2
Task 1
Task 3
Task 4

% a.out
Task 1
Task 2
Task 3
Task 4
```

Ejemplos

3. No dependencia entre tareas no hermanas

```
#include <omp.h>
int main(){
int a,b,c;
#pragma omp parallel {
   #pragma omp master {
      #pragma omp task depend(out:a) {
         #pragma omp critical
            printf ("Task 1\n");
         }
      #pragma omp task depend(out:b) {
         #pragma omp critical
            printf ("Task 2\n");
            #pragma omp task depend(out:a,b,c) {
               #pragma omp critical
               printf ("Task 5\n");
            }
      }
      #pragma omp task depend(in:a,b) depend(out:c) {
         printf ("Task 3\n");
      }
      #pragma omp task depend(in:c){
         printf ("Task 4\n");
      }
}
```

La tarea 5 es hija de la tarea 2; no es hermana de las tareas 1, 3 y 4.

A pesar de las dependencias definidas en las variables a, b y c, no hay dependencia.

```
% cc -xopenmp -O3
task_nodepend.c
% a.out
Task 2
Task 5
Task 1
Task 3
Task 4

% a.out
Task 1
Task 2
Task 3
Task 5
Task 4
```

Recursos

1. Ejemplos de todas las directivas y cláusulas: https://www.openmp.org/wp-content/uploads/OpenMP4.0.0_Examples.pdf
2. Ejemplos de tareas https://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-task.html
3. Ejemplos de dependencias https://docs.oracle.com/cd/E77782_01/html/E77801/gozsa.html