

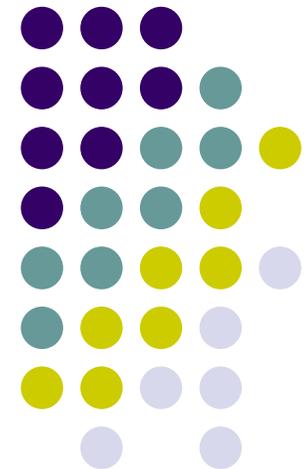
Curso de Computación Científica en Clusters

Programación de Plataformas Paralelas I: OpenMP

Javier Cuenca



Universidad de Murcia



Nociones básicas



- Modelo de programación fork-join, con generación de múltiples threads.
- Inicialmente se ejecuta un thread hasta que aparece el primer constructor paralelo, se crean threads esclavos y el que los pone en marcha es el maestro.
- Al final del constructor se sincronizan los threads y continúa la ejecución el maestro.

Ejemplo inicial: ejemplo_hello.c



```
#include <omp.h>
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);
    }
}
```

Directivas



```
#pragma omp nombre-directiva [cláusulas]
```

Constructores: Constructor **parallel**



```
#pragma omp parallel [cláusulas]
        bloque
```

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

Constructores: Constructor **for**



```
#pragma omp for [cláusulas]
      bucle for
```

- Las iteraciones se ejecutan en paralelo por threads que ya existen
- 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 con un valor, 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.
- Hay una serie de cláusulas (private, firstprivate, lastprivate y reduction) para indicar la forma en que se accede a las variables.

Constructores:

Constructor **for**: ejemplo_for.c



```
#include <omp.h>
int main()
{
    int iam =0, np = 1, i=0;
    #pragma omp parallel private(iam, np,i)
    {
        #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);

        #pragma omp for
        for(i=0;i<(np*2);i++)
        {
            printf("Thread %d, contador %d \n",iam,i);
        }
    }
}
```

Constructores: Constructor **for**



Una cláusula **schedule** indica la forma como 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, empezando en tamaño $\text{numiter}/\text{np}$ y acabando en "tamaño".
- **schedule(runtime)** deja la decisión para el tiempo de ejecución, y se obtienen de la variable de entorno **OMP_SCHEDULE**.
- → *ver modificaciones ejemplo_for.c*

Constructores: Constructor **sections**



```
#pragma omp sections [cláusulas]
{
    [#pragma omp section]
    bloque
    [#pragma omp section]
    bloque
    ...
}
```

- Cada sección se ejecuta por un thread.
- Hay barrera al final de “sections” a no ser que se utilice la cláusula **nowait**.
- Hay una serie de cláusulas (**private**, **firstprivate**, **lastprivate** y **reduction**) para indicar la forma en que se accede a las variables.

Constructores:

Constructor **sections**: ejemplo_sections.c



```
#pragma omp parallel private(iam, np,i)
{
  #pragma omp sections
  {
    #pragma omp section
    printf("Soy el thread %d, en solitario en la seccion 1ª \n",iam);
    #pragma omp section
    printf("Soy el thread %d, en solitario en la sección 2ª \n",iam);
    #pragma omp section
    printf("Soy el thread %d, en solitario en la seccion 3ª \n",iam);
  }//sections
}//parallel
```

Constructores: Constructores combinados



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

Es forma abreviada de directiva **parallel** que tiene una única directiva **for**, y admite sus cláusulas menos la **nowait**.

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

Es forma abreviada de directiva **parallel** que tiene una única directiva **sections**, y admite sus cláusulas menos la **nowait**.



Constructores: Constructores de ejecución secuencial

```
#pragma omp single [cláusulas]  
bloque
```

- El bloque se ejecuta por un único thread. No tiene por qué ser el maestro.
- Hay barrera al final a no ser que se utilice la cláusula **nowait**.

```
#pragma omp master  
bloque
```

- El bloque lo ejecuta el thread maestro.
- No hay sincronización al entrar ni salir.

```
#pragma omp ordered  
bloque
```

- Todo dentro de un for, el bloque se ejecuta en el orden en que se ejecutaría en secuencial.



Constructores:

Constructores de ejecución secuencial: ejemplos

- `ejemplo_single.c`
 - Barreras al final de cada single
- `ejemplo_master.c`
 - Ejecución solamente por thread maestro (el 0)
 - No hay barreras
- `ejemplo_ordered.c`
 - Se ordena la ejecución por iteraciones del bucle

Constructores:

Constructores de ejecución secuencial: ejemplo_single.c



```
#pragma omp parallel private(iam, np,i)
{
#pragma omp single
    printf("Soy el thread %d, actuando en solitario dentro del primer
    bloque\n",iam);
    sleep(1);
#pragma omp single
    printf("Soy el thread %d, actuando en solitario dentro ddel
    segundo bloque \n",iam);
    sleep(1);
#pragma omp single
    printf("Soy el thread %d, actuando en solitario dentro ddel tercer
    bloque \n",iam);
    sleep(1);

printf("Soy el thread %d, fuera de los singles\n",iam);
} //parallel
```

Constructores:

Constructores de ejecución secuencial: ejemplo_master.c



```
#pragma omp parallel private(iam, np,i)
{
#pragma omp master
    {
        printf("Soy el thread %d, actuando en solitario dentro del primer
        bloque\n",iam);
        sleep(1);
    }
#pragma omp master
    {
        printf("Soy el thread %d, actuando en solitario dentro ddel
        segundo bloque \n",iam);
        sleep(1);
    }
#pragma omp master
    {
        printf("Soy el thread %d, actuando en solitario dentro ddel tercer
        bloque \n",iam);
        sleep(1);
    }

    printf("Soy el thread %d, fuera de los singles\n",iam);
} //parallel
```

Constructores:

Constructores de ejecución secuencial: ejemplo_ordered.c



```
#pragma omp parallel private(iam, np, i)
{
#pragma omp for ordered
for(i=0; i<5; i++)
{
    printf("\t\tSoy el thread %d, antes del ordered en la iteracion
    %d\n", iam, i);

    #pragma omp ordered
    {
        printf("Soy el thread %d, actuando en la iteracion
        %d\n", iam, i);
        sleep(1);
    }
}
} //parallel
```



Constructores: Constructores de sincronización

```
#pragma omp critical [nombre]  
bloque
```

- Asegura exclusión mutua en la ejecución del bloque.
- El nombre se puede usar para identificar secciones críticas distintas.

```
#pragma omp barrier
```

- Sincroniza todos los threads en el equipo.

```
#pragma omp atomic  
expresión
```

- La expresión debe ser $x \text{ binop} = \text{exp}$, $x++$, $++x$, $x--$ o $--x$, donde x es una expresión con valor escalar, y binop es un operador binario.
- Asegura la ejecución de la expresión de forma atómica.



Constructores:

Constructores de sincronización: ejemplo_critical.c

```
#pragma omp parallel private(iam, np,i)
{

#pragma omp critical
{
    printf("Soy el thread %d, al inicio de la seccion critica
    \n",iam);
    sleep(1);
    printf("\t\tSoy el thread %d, al final de la seccion critica
    \n",iam);
}

} //parallel
```



Constructores:

Constructores de sincronización: ejemplo_barrier.c

```
#pragma omp parallel private(iam, np,i)
{

    printf("Soy el thread %d, antes del barrier \n",iam);
    #pragma omp barrier
    printf("\t\tSoy el thread %d, despues del barrier \n",iam);

}//parallel
```



Constructores:

Constructores de manejo de variables

```
#pragma omp flush [lista]
```

- Asegura que las variables que aparecen en la lista quedan actualizadas para todos los threads.
- Si no hay lista se actualizan todos los objetos compartidos.
- Se hace flush implícito al acabar **barrier**, al entrar o salir de **critical** u **ordered**, al salir de **parallel**, **for**, **sections** o **single**.
- →ejemplo_flush.c

```
#pragma omp threadprivate (lista)
```

- Usado para declarar variables privadas a los threads.



Constructores:

Constructores de manejo de variables: ejemplo_flush.c

```
#pragma omp parallel private(iam, np,i)
{
    #pragma omp master
    {
        x=999;
        #pragma omp flush(x)
        printf("Soy el thread %d, actualizando x=999 \n\n",iam);
    }

    printf("Soy el thread %d, antes del flush, con x=%d
\n",iam,x);
    while (x==0)
    { #pragma omp flush(x) }

    printf("\t\tSoy el thread %d, despues del flush, con x=%d
\n",iam,x);
}
```



Claúsulas de alcance de datos

- `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.

Claúsulas de alcance de datos: Ejemplos: ejemplo_private.c



```
int x=1234;
#pragma omp parallel private(iam, np,i,x)
{
    printf("Soy el thread %d, antes de actualizar, con x=%d
\n",iam,x);
    x=iam*1111;
    printf("\t\tSoy el thread %d, despues de actualizar, con x=%d
\n",iam,x);
}
printf("\n Despues de pragma parallel x=%d \n\n",x);
```

Claúsulas de alcance de datos: Ejemplos: ejemplo_firstprivate.c



```
int x=1234;
#pragma omp parallel firstprivate(iam, np,i,x)
{
    printf("Soy el thread %d, antes de actualizar, con x=%d
\n",iam,x);
    x=iam*1111;
    printf("\t\tSoy el thread %d, despues de actualizar, con x=%d
\n",iam,x);
}
printf("\n Despues de pragma parallel x=%d \n\n",x);
```

Claúsulas de alcance de datos: Ejemplos: ejemplo_lastprivate.c



```
int x=1234;
#pragma omp parallel private(iam, np,i,x)
{
    printf("Soy el thread %d, antes de actualizar, con x=%d
    \n",iam,x);
    #pragma omp for lastprivate(x) schedule(dynamic)
    for(i=0;i<11;i++)
    {
        x=iam*i;
        printf("\tSoy el thread %d, actualizando en for, i=%d
        x=%d\n",iam,i,x);
    }
    printf("\t\tSoy el thread %d, despues de actualizar, con x=%d
    \n",iam,x);
}
```



Claúsulas de alcance de datos: Ejemplos: ejemplo_reduction.c

```
int x=1000;int iam =0, np = 1, i=0,j=0;
printf("\n Antes de pragma parallel x=%d \n\n",x);

#pragma omp parallel private(iam, np,i) reduction(+:x)
{ printf("Soy el thread %d, antes de actualizar, con x=%d
  \n",iam,x);
  x=iam*10;
  printf("\t\tSoy el thread %d, despues de actualizar, con x=%d
  \n",iam,x);
} //parallel

printf("\n Despues de pragma parallel x=%d \n\n",x);
```

Claúsusulas de alcance de datos: Ejemplos: ejemplo_copyin.c



```
int x;
#pragma omp threadprivate(x)
int main() {
    int iam =0, np = 1, i=0,j=0;
    x=9999; // lo ponemos en el master
    #pragma omp parallel private(iam, np,i,x) copyin(x)
    {
        printf("Soy el thread %d, antes de actualizar, con x=%d
        \n",iam,x);
        x=1000+iam;
        printf("\t\tSoy el thread %d, despues de que el master
        actualice, con x=%d \n",iam,x);
    }//parallel

    printf("\n Despues de pragma parallel x=%d \n\n",x);
} //main
```



Funciones de librería

Poner al principio: `#include <omp.h>`

- `void omp_set_num_threads(int num_threads);`
 - Pone 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.

→ `ejemplo_nested.c`

- `int omp_get_nested(void);`
 - Devuelve un valor distinto de cero si está permitido el paralelismo anidado.

Funciones de librería

ejemplo_nested.c



```
#pragma omp parallel private(iam, np,i)
{
    #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);
    omp_set_nested(1);
    #pragma omp parallel private(iam, np,i)
    {
        #if defined (_OPENMP)
            np = omp_get_num_threads();
            iam = omp_get_thread_num();
        #endif
        printf("\t\tHello from thread %d out of %d \n",iam,np);
    }//parallel
}//parallel
```



Funciones de librería

- `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.

→ `ejemplo_lock.c`



Funciones de librería: ejemplo_lock.c

```
omp_init_lock(&lck);
#pragma omp parallel shared(lck) private(id)
{
    id=omp_get_thread_num();
    omp_set_lock(&lck);
    printf("My thread id is %d.\n",id);
    omp_unset_lock(&lck);

    while (! omp_test_lock(&lck))    {
        skip(id); }//while

    work(id);    //ahora tengo el lock, entonces hago el trabajo
    omp_unset_lock(&lck);
}//parallel
omp_destroy_lock(&lck);
```



Variables de entorno

OMP_SCHEDULE

- indica el tipo de scheduling para **for** y **parallel for**.

OMP_NUM_THREADS

- Pone el número de threads a usar, aunque se puede cambiar con la función de librería.

OMP_DYNAMIC

- Autoriza o desautoriza el ajuste dinámico del número de threads.

OMP_NESTED

- Autoriza o desautoriza el anidamiento. Por defecto no está autorizado.

Ejemplos: Busqueda 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;
        }
        #pragma omp flush(keepon)
    }
}
```

Ejemplos:

Multiplicación matriz-vector



```
void mv(double *a, int fa, int ca, int lda, double *b, int fb, double
    *c, int fc)
{
    int i, j; double s;
    #pragma omp parallel
    {
        #pragma omp for private(i,j,s) schedule(static,BLOQUE)
        for (i = 0; i < fa; i++)
        {
            s=0.;
            for(j=0;j<ca;j++)
                s+=a[i*lda+j]*b[j];
            c[i]=s;
        }
    }
}
```

Tareas en OpenMP 3.0



- Cambiar OpenMP de “thread-centric” a “task-centric”.
- Para expresar paralelismo irregular y no estructurado
- Para hacer paralelismo anidado de tareas sin que conlleve un estricto paralelismo anidado de threads con sus consecuentes barriers implícitos que puedan ser innecesarios.
- Para paralelizar bucles while. Por ejemplo recorrido de una lista de punteros de longitud no conocida
- Tied task: ligada a un thread fijo que la ejecutará. Puede tener pausas o hacer otra cosa , pero ese thread acabará la tarea
- Untied task: cualquier thread del team el scheduler le puede asignar una untied task que esté suspendida en ese momento

Tareas en OpenMP 3.0:

Constructor task



```
#pragma omp task [clause [[,clause] ...]  
    structured-block
```

Clauses:

```
if (scalar expression)  
untied  
default (shared | none)  
private (list)  
firstprivate (list)  
shared (list)
```

Tareas en OpenMP 3.0: Sincronización y Finalización de tareas



```
#pragma omp taskwait
```

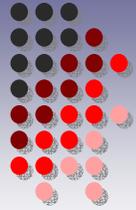
- Tarea actual se suspende hasta finalización de sus tareas hijas
- Util para sincronización de tareas de grano fino

Un conjunto de tareas será forzada a completarse en alguno de estos casos:

- En una barrier implícita de threads
- En una barrier explícita de threads (`#pragma omp barrier`)
- En una barrier de tareas (`#pragma omp taskwait`)

Tareas en OpenMP 3.0:

Ejemplo: recorrido de lista (secuencial)



```
.....  
while(my_pointer)  
{  
    (void) do_independent_work (my_pointer);  
    my_pointer = my_pointer->next ;  
} // End of while loop  
.....
```

Tareas en OpenMP 3.0:

Ejemplo: recorrido de lista (paralelo)



```
my_pointer = listhead;
#pragma omp parallel
{
    #pragma omp single nowait
    {
        while(my_pointer)
        {
            #pragma omp task firstprivate(my_pointer)
            {
                (void) do_independent_work (my_pointer);
            }
            my_pointer = my_pointer->next ;
        }
    } // End of single - no implied barrier (nowait)
} // End of parallel region - implied barrier
```

Tareas en OpenMP 3.0: Ejemplo: Fibonacci



$$F(0) = 1$$

$$F(1) = 1$$

$$F(n) = F(n-1) + F(n-2) \quad (\text{para } n=2,3,\dots)$$

Secuencia valores resultado:

1, 1, 2, 3, 5, 8, 13, ...



Tareas en OpenMP 3.0: Ejemplo: Fibonacci (secuencial recursivo)

```
long comp_fib_numbers(int n)
{
    // Basic algorithm:  $f(n) = f(n-1) + f(n-2)$ 
    long fnm1, fnm2, fn;
    if ( n == 0 || n == 1 )
        return(n);

    fnm1 = comp_fib_numbers(n-1);
    fnm2 = comp_fib_numbers(n-2);
    fn = fnm1 + fnm2;
    return(fn);
}
```

Tareas en OpenMP 3.0:

Ejemplo: Fibonacci (paralelo)



```
long comp_fib_numbers(int n)
{
    long fnm1, fnm2, fn;
    if ( n == 0 || n == 1 )
        return(1);

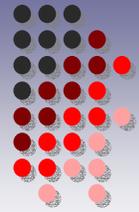
    #pragma omp task shared(fnm1)
        {fnm1 = comp_fib_numbers(n-1);}

    #pragma omp task shared(fnm2)
        {fnm2 = comp_fib_numbers(n-2);}

    #pragma omp taskwait
    fn = fnm1 + fnm2;
    return(fn);
}
```

Tareas en OpenMP 3.0:

Ejemplo: Fibonacci (paralelo)



```
long comp_fib_numbers(int n)
{
    long fnm1,
    if ( n == 0 )
        return 0;
    #pragma omp parallel shared(nthreads)
    {
        #pragma omp single nowait
        {
            result = comp_fib_numbers(n);
        } // End of single
    } // End of parallel region
    #pragma omp taskwait
    fn = fnm1 + fnm2;
    return(fn);
}
```

Tareas en OpenMP 3.0:

Ejemplo: Fibonacci (paralelo, versión 2)



```
long comp_fib_numbers(int n)
{
    long fnm1, fnm2, fn;
    if ( n == 0 || n == 1 ) return(1);
    ← if ( n<20 ) return(comp_fib_numbers(n-1) +comp_fib_numbers(n-
        2));
    #pragma omp task shared(fnm1)
        {fnm1 = comp_fib_numbers(n-1);}
    #pragma omp task shared(fnm2)
        {fnm2 = comp_fib_numbers(n-2);}
    #pragma omp taskwait
    fn = fnm1 + fnm2;
    return(fn);
}
```

Tareas en OpenMP 3.0: Ejemplo: Fibonacci. Probando...



```
. iccStart  
  
icc_compilar ejemplo_fibo  
  
ejemplo_fibo
```

Tareas en OpenMP 3.0:

Ejemplo: Recorrido de árbol en preorden



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

Tareas en OpenMP 3.0:

Ejemplo: Recorrido de árbol en postorden



```
void traverse(binarytree *p)
{
    if (p->left)
    {
        #pragma omp task
        traverse(p->left);
    }
    if (p->right)
    {
        #pragma omp task
        traverse(p->right);
    }

    #pragma omp taskwait
    process(p);
}
```

Tareas en OpenMP 3.0: threads y tareas



Por defecto: tarea t está ligada a un thread h (**tied task**):

- El thread h ejecuta código de t desde principio al final
- La ejecución puede no ser contigua:
 - En los *task scheduling points* el thread h puede dedicarse a realizar otra tarea
 - *Task scheduling points*:
 - Pragma `task` → tareas anidadas
 - Pragma `taskwait`
 - Barreras explícitas o implícitas
 - Finalización de tarea
- Valor de *threadprivate* variables pueden cambiar (por obra de otra tarea ejecutada por el thread h)

Tareas en OpenMP 3.0: threads y tareas



```
int tp;
#pragma omp threadprivate(tp)
int var;
void work()
{
    #pragma omp task          // tarea 1
    {
        /* do work here */
        #pragma omp task      //tarea 1.1. Tarea 1 puede ser suspendida
        {
            tp = 1;
            /* do work here */
            #pragma omp task  //tarea 1.1.1. -> posible cambio a tarea
1
            {
                /* no modification of tp */
            }
            var = tp; //value of tp can be 1 or 2
        }
        tp = 2;
    }
}
```

Tareas en OpenMP 3.0: threads y tareas



Si la tarea t **no** está ligada a un thread h (**untied task**):

- El thread h inicia la ejecución código de t
- Otro thread distinto puede continuar la ejecución de t en cualquier momento
- Mismas restricciones con variables *threadprivates* que en *tied task*
- Evitar cualquier referencia ligada al identificador del thread

Ejercicios



0.- Probar los distintos ejemplos de programas. Razonar su comportamiento.

Conectarse a luna.inf.um.es

```
ssh luna.inf.um.es
```

```
usuario      XXXX
```

```
clave        YYYY
```

```
PATH=$PATH:.
```

Para obtener los programas de ejemplo:

```
cp /home/javiercm/ejemplos_openmp/*.c .
```

Para compilar en luna.inf.um.es el programa ejemplo_hello.c:

```
gcc-4.3 -o ejemplo_hello ejemplo_hello.c -O3 -lm -fopenmp
```

Para ejecutarlo con 4 threads:

```
export OMP_NUM_THREADS=4
```

```
ejemplo_hello
```

Ejercicios



- 1.- En `ejemplo_for.c`: probar distintos schedules: estático vs. dinámico, sin tamaño vs. con tamaño=2. Ejecutar y razonar comportamientos.
- 2.- En `ejemplo_sections.c`: Agrupar las directivas `parallel` y `sections` en una única.
- 3.- Compara y razona el comportamiento de `ejemplo_sections.c` vs. `ejemplo_single.c`.
- 4.- Compara y razona el comportamiento de `ejemplo_sections.c` vs. `ejemplo_critical.c`.
- 5.- A partir del ejemplo de multiplicación matriz-vector `ejemplo_mv.c`: diseña un programa para multiplicar dos matrices $AxB=C$, donde cada thread se encargue de calcular un bloque de filas consecutivas de elementos de C .

Ejercicios



6.- Diseñar, programar y probar un programa de recorrido recursivo de un árbol, usando tareas de OpenMP 3.0. Mostrar qué thread se encarga de procesar cada nodo.

```
. iccStart
```

```
icc_compile ejemplo_recorrido_Arbol
```

```
ejemplo_recorrido_arbol
```