


# Programación paralela con OpenMP

Página oficial: [www.openmp.org](http://www.openmp.org)

- Introducción
- Regiones Paralelas
- Distribución del trabajo
- Gestión de datos (ámbito de las variables)
- Sincronización
- Funciones de librería
- Variables de entorno
- Tareas
- Afinidad
- Soporte para vectorización y coprocesadores

# ¿Qué es OpenMP?



- API estándar “de facto” para la programación paralela *multithread* en sistemas de *memoria compartida* (C/C++ y Fortran)  
Todos los *threads* tienen acceso a la misma memoria global compartida:  
 Se comunican a través de *variables compartidas*
- Diseñado para permitir *paralelizar incrementalmente* programas secuenciales existentes
- El programador no necesita especificar todos los detalles acerca de la *creación* y *destrucción* de *threads*
- Proporciona directivas de *distribución de trabajo* para indicar al compilador dónde se deben crear los *threads* y qué tarea deben realizar

# Introducción



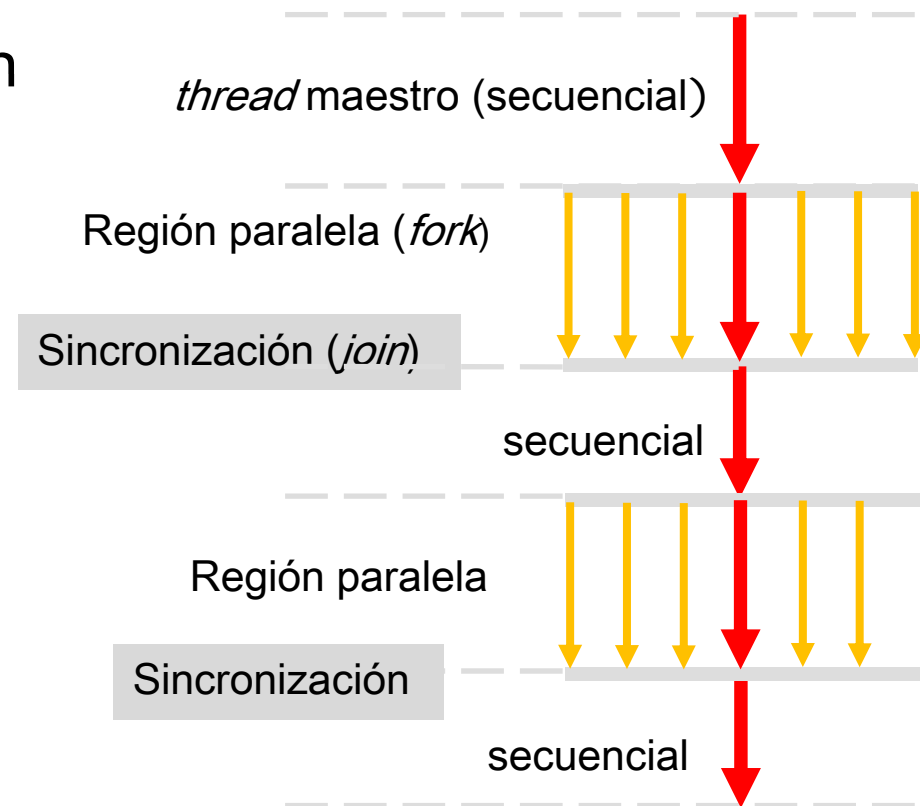
- Consta de:
  - **Directivas** para el compilador
  - **Funciones** de librería
  - **Variables de entorno**
- Ejemplos:
  - #pragma omp parallel**
  - omp\_set\_num\_threads(4)**
  - OMP\_NUM\_THREADS= 4**
- Las directivas se aplican a “bloques estructurados”: una o más sentencias con un punto de entrada y uno de salida
- El código es compilable incluso si el compilador no soporta OpenMP ➡ mismo código para secuencial y paralelo
  - Las directivas se ignoran (depende del compilador) y las funciones de librería se “protegen” con el preprocesador:  

```
#ifdef _OPENMP <omp_...>..... #endif
```
- Los prototipos de funciones y los tipos están en el fichero **<omp.h>**

# Introducción



- Modelo de paralelismo ***fork-join***
- El programa comienza con un solo *thread* (maestro)
- El **maestro** crea *threads* según necesidad
- Requiere mecanismos de **sincronización**



# Ejemplo



```
// Secuencial:
void main()
{ double res[1000];
  int i;

  for(i=0;i<1000;i++)
    huge_comp(res[i]);
  printf("Hecho\n");
}
```

```
// Paralelo:
void main()
{ double res[1000];
  int i;
#pragma omp parallel for
  for(i=0;i<1000;i++)
    huge_comp(res[i]);
  printf("Hecho\n");
}
```

- Similar la versión paralela y la secuencial
  - ↩ Pocos cambios, legible y fácil de mantener
- El pragma hace que se creen *N threads*, que ejecutan el [mismo código](#), cada uno lo aplica a una parte del vector *res*
- Al finalizar el *for* hay una barrera implícita (sincronización)

# Regiones paralelas: Creación de *threads*



- Se declaran con la directiva:

```
#pragma omp parallel [cláusulas]  
  
{// código ...}
```

- Crea N *threads* que ejecutan en paralelo el **mismo código**, solo cambia su ID (0 .. N-1)  
  
“N” depende de variables internas (“num\_threads”, “max\_threads”, “dynamic”) que se pueden consultar y modificar (p.e. mediante la cláusula `num_threads(int)`)
- Se puede utilizar la cláusula **if**, para paralelizar sólo si se cumple una condición: ...`parallel if(n>100)`...
- Al final de la región hay una **barrera implícita** (ningún *thread* puede avanzar hasta que el resto haya llegado)

# Regiones paralelas: Ejemplo



```
#include <omp.h>
```

```
void main()
```

```
{
```

```
    omp_set_num_threads(4);
```

```
    #pragma omp parallel
```

```
    {
```

```
        int id = omp_get_thread_num();
```

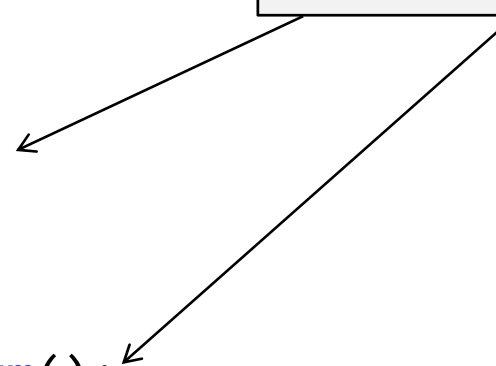
```
        printf("Hola mundo, soy %d\n", id);
```

```
    }    // fin de la region paralela: barrera implicita
```

```
    printf("Fin de Hola mundo\n");
```

```
}
```

Funciones de librería



Cada *thread* ejecuta el primer printf para id = 0 .. 3



# Regiones paralelas: Ejemplo



```
#include <omp.h>
void main()
{
    #pragma omp parallel num_threads(4)
    {
        int id = omp_get_thread_num();
        printf("Hola mundo, soy %d\n", id);
    }    // fin de la region paralela: barrera implicita
    printf("Fin de Hola mundo\n");
}
```

cláusula

# Regiones paralelas: Ejemplo

```
#include <omp.h>
void main() {
#pragma omp parallel
{
    int id = omp_get_thread_num();
    printf("Hola mundo, soy %d\n", id);
}
printf("Fin de Hola mundo\n");
}
```

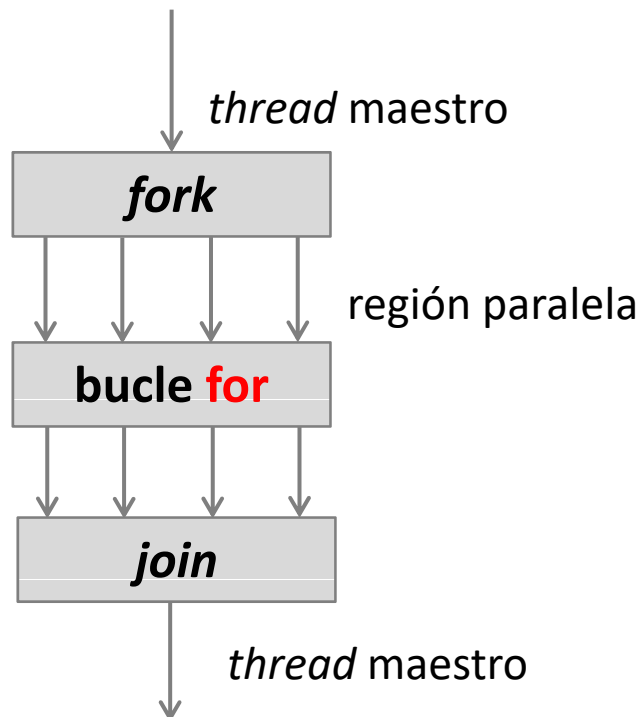
Variable de entorno

```
$ export OMP_NUM_THREADS=4
$ gcc -fopenmp hola.c -o hola
$ ./hola
Hola mundo, soy 3
Hola mundo, soy 0
Hola mundo, soy 2
Hola mundo, soy 1
Fin de Hola mundo
```

# Distribución del trabajo entre los *threads*

## Paralelismo de datos

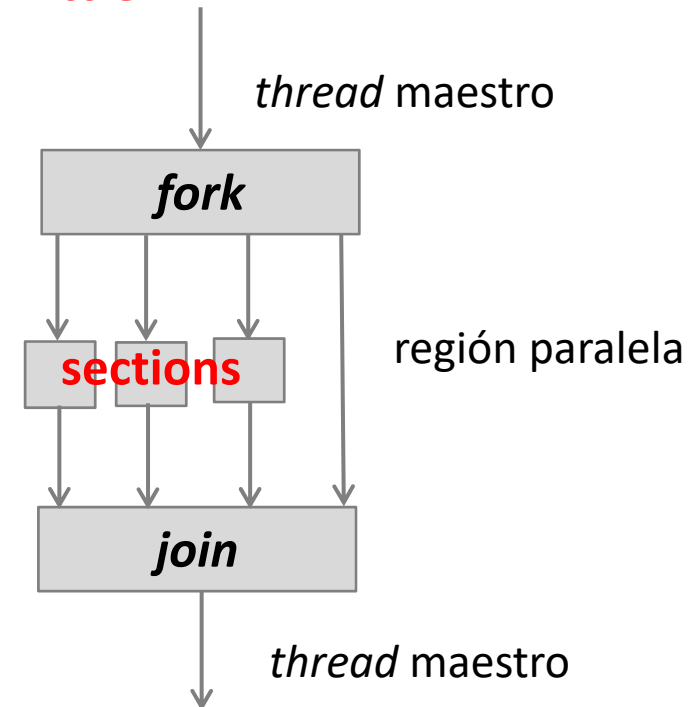
### - **for**



## Paralelismo funcional

### - **sections**

### - **task**



# Distribución del trabajo: *for*



- Se declara con la directiva:

```
#pragma omp for [cláusulas]
for (;;) { // código ... }
```

- Reparte las iteraciones entre los *threads* disponibles. Se puede especificar cómo se hace el reparto mediante la cláusula *schedule*
- Al final hay una barrera implícita, evitable con la cláusula *nowait*
- Se pueden “agrupar” varios bucles en uno, resultando un bucle con TODAS las iteraciones

Ejemplo:

```
#pragma omp for collapse (2)
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        A[i][j] = B[i][j] + C[i][j];
```

# Distribución del trabajo *for*. Ejemplo



## 1. Código secuencial

```
for(i=0;i<N;i++)  
    a[i] = a[i] + b[i];
```

## 2. Región paralela OpenMP y distribución de trabajo manual

(*id, i, nthrds, istart, iend*: son variables privadas, ¿deben serlo?)

```
#pragma omp parallel  
{  
    int id, i, nthrds, istart, iend;  
    id = omp_get_thread_num();  
    nthrds = omp_get_num_threads();  
    istart = id * N / Nthrds;  
    iend = (id+1) * N / Nthrds;  
    for(i=istart; i<iend; i++)  
        a[i] = a[i] + b[i];  
}
```

## 3. Región paralela OpenMP y distribución de trabajo **for**

```
#pragma omp parallel  
#pragma omp for schedule(static)  
for(i=0; i<N; i++)  
    a[i] = a[i] + b[i];
```

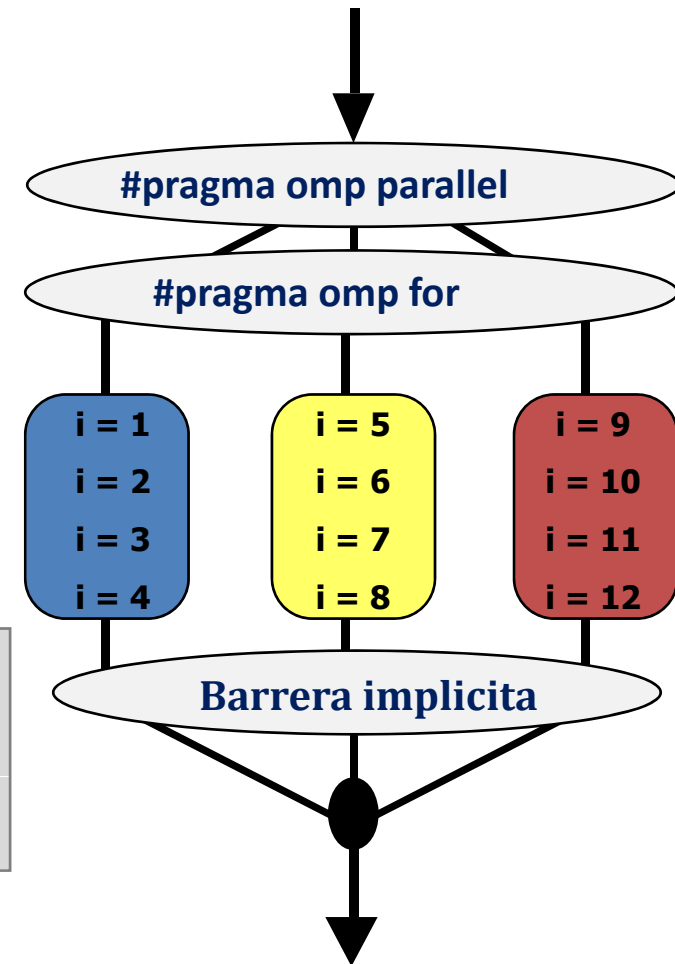
# Distribución del trabajo *for*. Ejemplo

Ejemplo con 3 threads y 12 iteraciones

```
#pragma omp parallel
#pragma omp for schedule(static)
for(i=1; i<N; i++)
    a[i] = a[i] + b[i];
```

Ambos pragmas se pueden combinar:

```
#pragma omp parallel for schedule(static)
for(i=0; i<N; i++)
    a[i] = a[i] + b[i];
```



# Distribución del trabajo *for*



- Bucles que NO se pueden paralelizar:
  - Bucles infinitos o “bloques no estructurados”

```
#pragma omp parallel for
for(i=0; i<N; i++) {
    if (a[i] ... ) break
    ...
}
```

El compilador lo  
notifica

- Bucles que NO se paralelizan correctamente
  - Bucles con **dependencias de datos** *loop-carried*:

*Entre instrucciones correspondientes a distintas iteraciones*

```
#pragma omp parallel for
for(i=2; i<N; i++)
    fib[i] = fib[i-1] + fib[i-2]
```

El compilador NO  
comprueba las  
dependencias

# Distribución *for*. Planificación



- Se declara con la cláusula **schedule**:

```
#pragma omp for schedule(tipo [, chunk])
```

- **Tipos:**

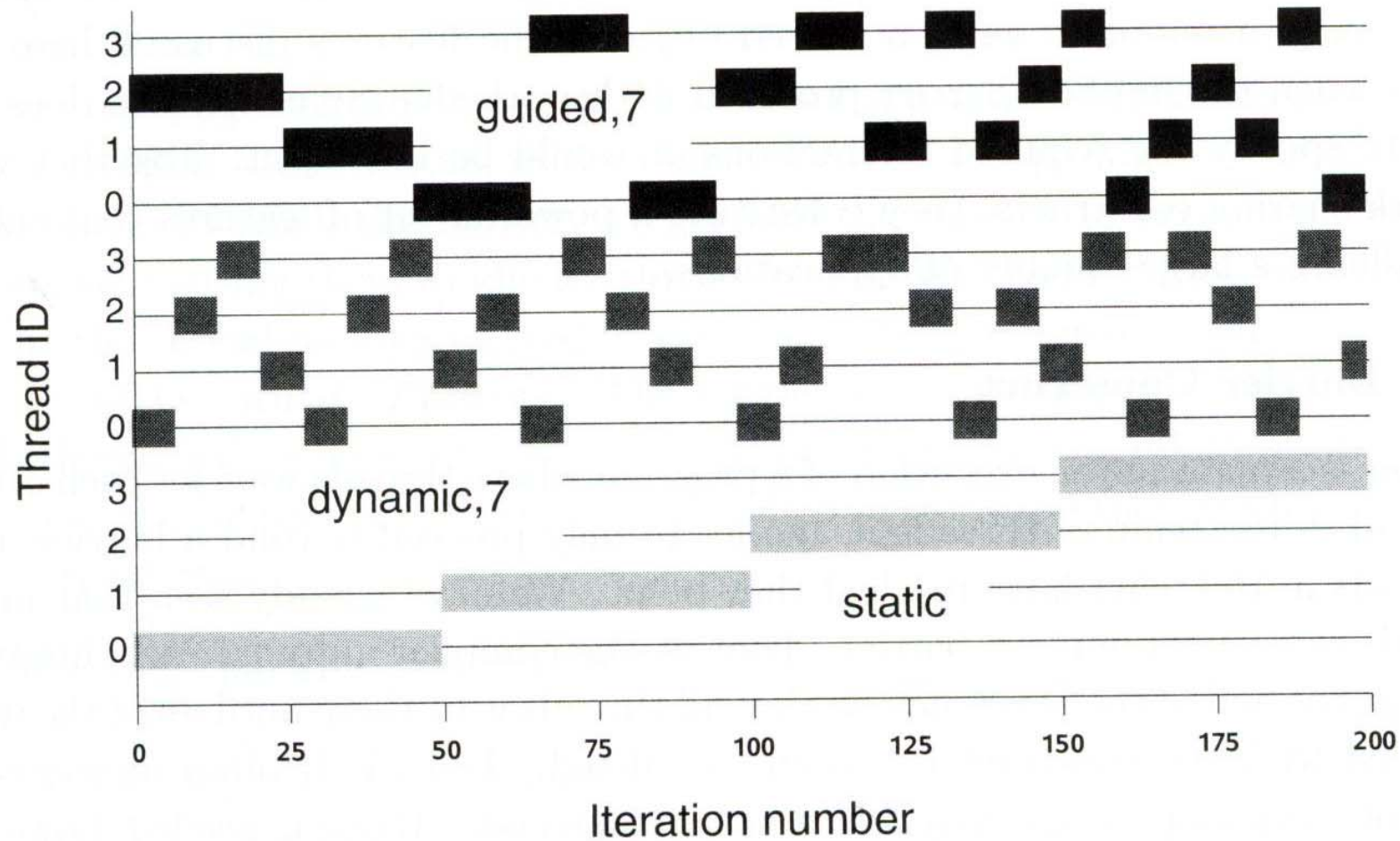
- **static**: bloques fijos de iteraciones de tamaño “chunk” para cada *thread*
- **dynamic**: Cada *thread* ejecuta “chunk” iteraciones, cuando acaba pide otras “chunk” iteraciones, y así hasta terminar con todas
- **guided**: Similar a **dynamic**. La diferencia está en el tamaño del bloque de iteraciones:
  - Inicialmente es grande (proporcional al nº iteraciones sin asignar dividido por nº *threads*) y va bajando hasta tamaño “chunk”
- **runtime**: Planificación y tamaño de bloque determinado por la variable de entorno **OMP\_SCHEDULE**



# Tipos de planificación: Ejemplo



200 iteraciones en 4 *threads*



# Distribución del trabajo: *sections*



- Se declara con la directiva:

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

- Asigna bloques de código independientes a los *threads*.

Cada bloque de código debe estar precedido de la directiva:

```
#pragma omp section
```

- Debe estar dentro de un región paralela.
- Al final hay una barrera implícita, evitable con la cláusula *nowait*

# Distribución del trabajo *sections*: Ejemplo

```
#pragma omp parallel
#pragma omp sections
{
    #pragma omp section
        x_calculation();
    #pragma omp section
        y_calculation();
    #pragma omp section
        z_calculation();
}
```

```
#pragma omp parallel sections
{
    #pragma omp section
        x = x_calculation();
    #pragma omp section
        y = y_calculation();
    #pragma omp section
        z = z_calculation();
}
res = x+y+z;
```

#pragma omp sections **nowait**

Barrera implícita

# Ámbito de las variables



- Modelo de programación de memoria compartida:
  - La mayoría de las variables son **compartidas por defecto**
  - Las variables **globales** son **compartidas**
- Pero **no** todas las variables son compartidas:
  - Algunas deben ser **específicas para cada thread**, p.e.:
    - **tid** = `omp_get_thread_num()`;
    - `#pragma omp parallel for`  
for (**i**= .....)
  - Las variables declaradas dentro de una **región paralela**
  - Las ubicadas en la **pila** de cada *thread* (p.e. parámetros y variables locales en funciones llamadas desde una región paralela)
- Se puede cambiar los ‘atributos’ de las variables heredadas del maestro mediante **cláusulas**.

# Ámbito de las variables: cláusulas



- **shared**: Variable común a todos los *threads*

Cuidado: puede ser necesario utilizar regiones críticas si varios *threads* pueden modificarla

- **private**: Se crea una copia local para cada *thread*.

¡No se inicializa y su valor es indefinido tras la región paralela!

- **firstprivate**: Es “private”, pero se inicializa con el valor de la ‘original’ (variable del maestro)

Ejemplo:

```
int val=0; int z[1000];  
#pragma omp parallel for firstprivate(val)  
for(int i=0;i<10;i++)  
    z[i] = val;
```

# Ámbito de las variables: cláusulas



- **lastprivate**: Es “private”, pero al final de la región paralela se le asigna el valor que toma en la última iteración o sección

Ejemplo:

```
#pragma omp parallel for lastprivate(i)
for (i=0; i<n-1; i++)
    a[i] = b[i] + b[i+1]
a[i]=b[i]; // caso n-1 fuera bucle
```

- **default**: shared, private (FORTRAN) o none

## Recomendación:

Declarar explícitamente el ámbito de todas las variables utilizadas en las regiones paralelas

# Ámbito de las variables: cláusulas



- **threadprivate**: hace una copia “privada” de una variable **global** para cada *thread*, que persiste durante toda la ejecución, en sucesivas regiones paralelas
- **copyin**: inicializa las variables declaradas con **threadprivate** con el valor del maestro

## Ejemplo:

```
int count = 0; // N° tareas hechas por cada hilo
#pragma omp threadprivate(count)
. . . . .
#pragma omp parallel copyin(count)
{ int my_id;
  #pragma omp for schedule(runtime)
  for(i=0; i<N; i++) {
    vec[i] = vec[i]*2; count++; }

  my_id=omp_get_thread_num();
  printf("Tareas del hilo %d = %d\n", my_id, count);
}
```

# Ámbito de las variables: cláusulas



- **copyprivate**: Se utiliza solo con la directiva “single”  
Difunde el valor que toma la variable privada en un *thread* al resto de los *threads* de la región paralela
- **reduction** (*op:var\_list*)
  - Las variables deben ser **shared**
  - Se crea una copia local de cada variable y se inicializa según “*op*”
  - Al final, las copias locales se reducen a una **global**
  - Operadores: + \* - & | ^ && || min max



# Cláusula reduction: Ejemplos



```
double ave=0.0, A[MAX];
int i;
for (i=0; i<MAX; i++)
    ave+= A[i];
ave = ave/MAX;
```

```
double ave=0.0, A[MAX], int i;
#pragma omp parallel for \
    schedule (static,ch)
    reduction(+:ave)
    for (i=0; i<MAX; i++)
        ave+= A[i];
ave = ave/MAX;
```

```
#pragma omp parallel for shared(x, y, n) reduction(+:a)
    reduction(^:b) reduction(min:c) reduction(max:d)
    for (i=0; i<n; i++) {
        a += x[i];
        b ^= y[i];
        if (c > y[i]) c = y[i];
        if (d < x[i]) d = x[i];
    }
```

# Sincronización



## Ejemplo:

```
float dot_prod(float* a, float* b, int N)
{
    float sum = 0.0; int i;
    #pragma omp parallel for shared(sum,a,b)
        for (i=0; i<N; i++)
            sum+= a[i]*b[i];
    return sum;
}
```

### Posible condición de carrera:

Se debe proteger el acceso a la variable compartida sum que es de lectura/escritura.

# Sincronización: Directivas



- **critical** [*name*] {código}: Definición de una **región crítica**.

Los *threads* esperan al comienzo de la región crítica hasta que no haya ningún otro ejecutando una región crítica con el **mismo nombre**.

Ejemplo:

```
#pragma omp parallel shared(sum,a,b){  
    float sLocal = 0; // privada  
    #pragma omp for  
    for (i=0; i<n; i++)  
        sLocal+= a[i]*b[i];  
    #pragma omp critical (update_sum)  
        sum += sLocal;  
}  
return sum;
```

# Sincronización: Directivas



- **atomic**: Asegura la actualización atómica.

Se aplica a la sentencia siguiente

Ejemplo:

```
float dot_prod(float* a, float* b, int N)
{
    float sum = 0.0; int i;
    #pragma omp parallel for shared(sum,a,b)
    for (i=0; i<N; i++)
        #pragma omp atomic
        sum+= a[i]*b[i];
    return sum;
}
```

# Sincronización: Directivas



- **barrier**: Implementa una barrera

Tan pronto como lleguen a ella todos los *threads*, se puede continuar.

Ejemplo:

```
#pragma omp parallel shared(n,a,b)
{
    #pragma omp for
    for(int i=0;i<n;i++)
        a[i] = i;
    #pragma omp barrier
    #pragma omp for
    for (i=0; i<n; i++)
        a[i] += b[i];
}
```

# Sincronización: Directivas



- **ordered**: El código correspondiente, dentro de un bucle paralelo, se ejecuta de forma secuencial

**¡¡evítese en lo posible!!**

- **single**: El código lo ejecuta un solo *thread*
  - Al final del código hay una barrera implícita evitable (**nowait**)
  - Puede hacer visible a los demás *threads* los cálculos hechos (con **copyprivate**)
- **master**: Sólo el maestro ejecuta el código, los demás se lo saltan. Sin barrera.

# Sincronización: Ejemplos



```
#pragma omp parallel for schedule(dynamic) private(a)
for (i=0; i<N; i++){
    a = work(i); // en paralelo
    #pragma omp ordered    // espera que le toque
    printf("%d\n", a); //impresión resultados ordenada
}
```

```
#pragma omp parallel shared(a,b)
{ #pragma omp single
    { a = init();
        printf("Single ejecutada por thread %d\n",
            omp_get_thread_num());
    }
    /* A barrier is automatically inserted here */
    #pragma omp for
    for (i=0; i<n; i++)
        b[i] = a;
}
```

# Sincronización: Ejemplos



```
#pragma omp parallel shared(a,b)
{  #pragma omp master
    { a = init();
      printf("Master ejecutada por thread %d\n",
            omp_get_thread_num());
    }
    #pragma omp barrier
    #pragma omp for
    for (i=0; i<n; i++)
        b[i] = a;
}
```

```
#pragma omp threadprivate(x, y)
void init(float a, float b ) { // a y b privados
#pragma omp single copyprivate(a,b,x,y)
    scanf("%f %f %f %f", &a, &b, &x, &y);
}
```



# Sincronización: Funciones de librería

- **Cerrojos** (tipo `omp_lock_t`)

```
void omp_init_lock(lock)
void omp_destroy_lock(lock)
void omp_set_lock(lock)
void omp_unset_lock(lock)
void omp_test_lock(lock)
```

Ejemplo:

```
omp_lock_t lck;
omp_init_lock(&lck);
#pragma omp parallel private (tmp, id)
{ id = omp_get_thread_num();
  tmp = do_lots_of_work(id);
  omp_set_lock(&lck);
    printf("%d %d", id, tmp);
  omp_unset_lock(&lck);
}
omp_destroy_lock(&lck);
```

# Funciones de librería: Entorno de ejecución



- **Gestión de *threads***

```
void omp_set_num_threads(int)
int omp_get_num_threads(void)
int omp_get_thread_num(void)
int omp_get_max_threads(void)
void omp_set_dynamic(bool)
bool omp_get_dynamic(void)
```

- **Anidamiento del paralelismo**

```
void omp_set_nested(bool)
bool omp_get_nested(void)
```

# Funciones de librería



- ¿En una región paralela?

```
bool omp_in_parallel(void)
```

- Número de procesadores:

```
int omp_num_procs(void)
```

- Tomar tiempos

```
omp_get_wtime(), omp_get_wtick()
```

# Variables del entorno de ejecución



- OMP\_SCHEDULE *"schedule[, chunk\_size]"*
- OMP\_NUM\_THREADS *int\_literal // Máximo*
- OMP\_DYNAMIC *bool //Ajusta el número de threads en cada región paralela*
- OMP\_NESTED *bool*
- OMP\_PROC\_BIND *bool*
- OMP\_STACKSIZE *int [B|K|M|G]*
- OMP\_WAIT\_POLICY *passive // active*
- OMP\_THREAD\_LIMIT *int*

Se verá más adelante

# Variables de entorno (4.0)



- **OMP\_DISPLAY\_ENV** = true | false | verbose

*Ejemplo:*

```
OPENMP DISPLAY ENVIRONMENT BEGIN
OMP_DISPLAY_ENV='TRUE'

_OPENMP='201107'
OMP_DYNAMIC='FALSE'
OMP_MAX_ACTIVE_LEVELS='5'
OMP_NESTED='FALSE'
OMP_NUM_THREADS='96'
OMP_PROC_BIND='FALSE'
OMP_SCHEDULE='STATIC,0'
OMP_STACKSIZE='4194304'
OMP_THREAD_LIMIT='96'
OMP_WAIT_POLICY='PASSIVE'
OPENMP DISPLAY ENVIRONMENT END
```

# Compilación



- Incluir el fichero de cabecera:

```
#include <omp.h>
```

- Compilador de GCC:

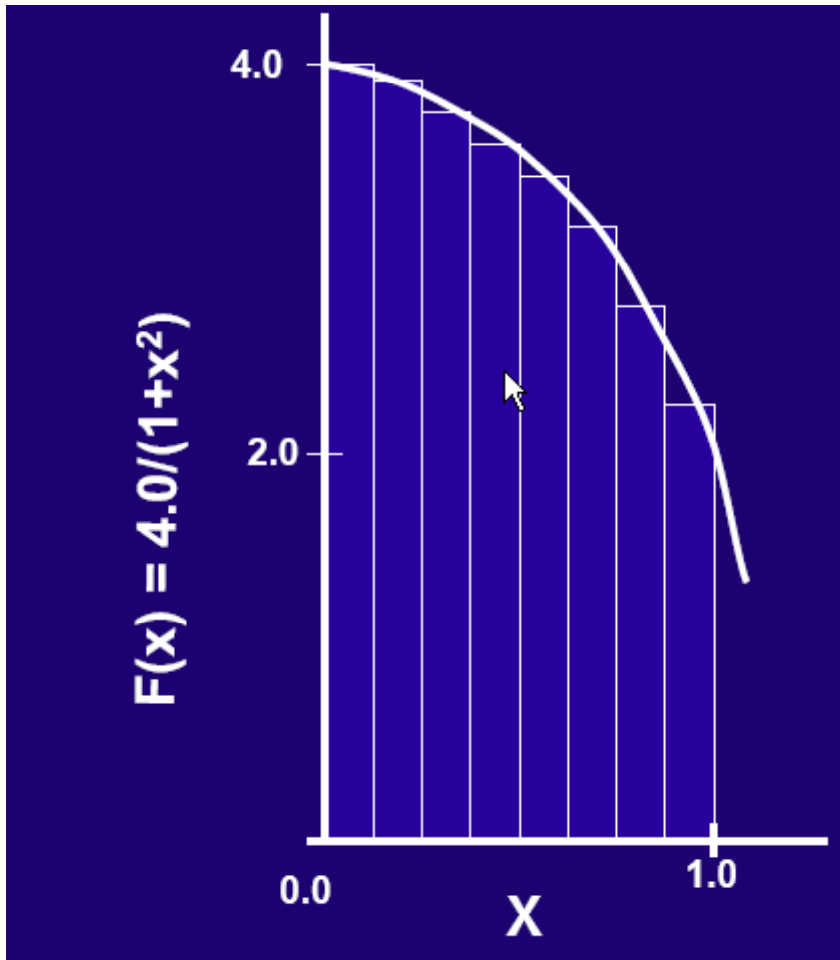
```
gcc -fopenmp  
gfortran -fopenmp
```

Se activa OPENMP

- Compilador de Intel:

```
icc -qopenmp  
ifort -qopenmp
```

## Ejemplo del cálculo de Pi



- Por cálculo numérico:

$$\int_0^1 \frac{4.0}{(1+x^2)} dx = \pi$$

- Se aproxima como:

$$\sum_{i=0}^N F(x_i) \Delta x \approx \pi$$

- con  $F(x_i)$  el alto a mitad del intervalo y  $x$  el ancho del intervalo



## Ejemplo del cálculo de Pi : Secuencial

```
static long num_steps = 100000;
double step;
void main (){
    int i; double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    for (i=0;i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
}
```



# Tareas (a partir de OpenMP 3.0)



- Amplia el rango de aplicaciones a paralelizar, p.e.:
  - Algoritmos recursivos
  - Bucles con un número indefinido de iteraciones (p.e. while)
  - Recorrido de listas ....
- Una **tarea** se compone de:
  - Código a ejecutar
  - Datos: Variables pertenecientes a la tarea (inicializadas cuando se crean)
  - Variables de control internas (num\_threads, max\_threads, schedule, ..)
- Diferencia principal con “*sections*”:
  - Las **tareas** se crean dinámicamente y las **sections** son estáticas
  - Conllevan mayor *overhead*

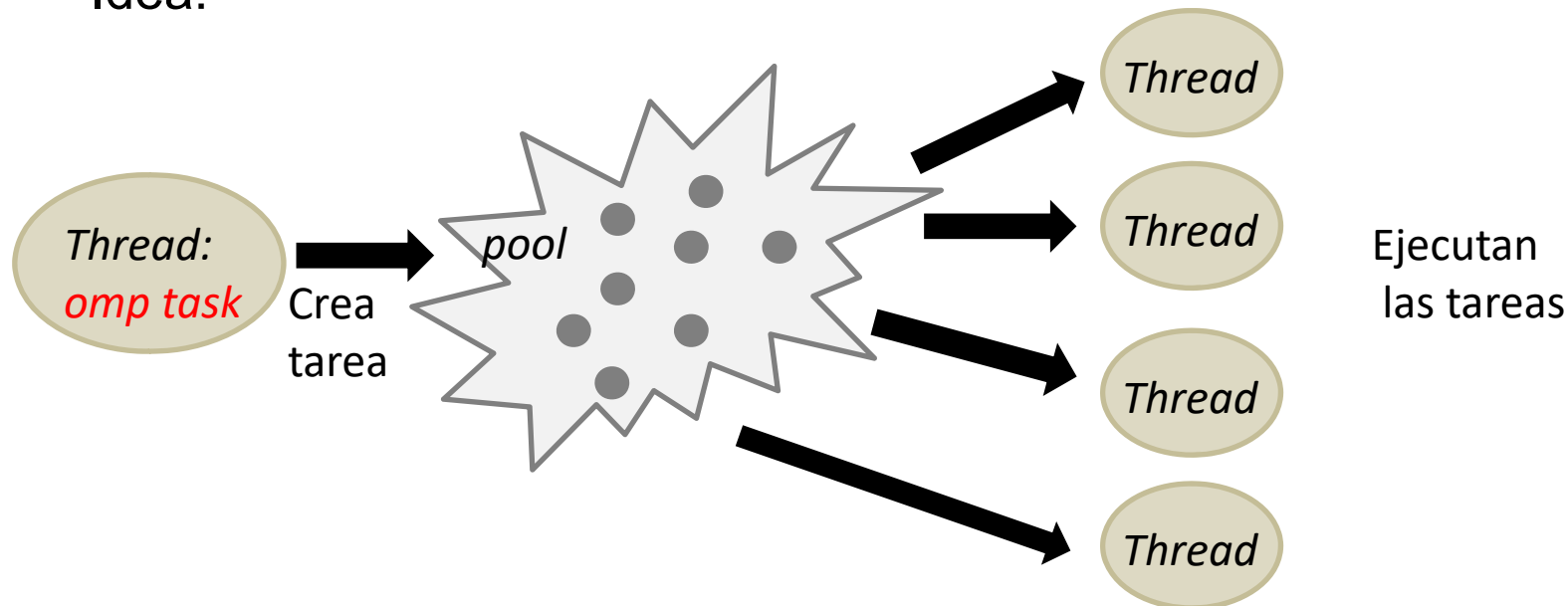
# Tareas



- Se declara con la directiva:

```
#pragma omp task [cláusulas]  
{ código }
```

- Idea:



- Si hay algún *thread* disponible la ejecuta inmediatamente, si no se espera hasta que haya uno libre

# Tareas: Ejemplo



```
pointer = head;
while(pointer) {
    do_independent_work (pointer);
    pointer = pointer->next;
}
```



```
pointer = head;
#pragma omp parallel num_threads(nt)
{
    #pragma omp single nowait
    {
        while(pointer) {
            #pragma omp task firstprivate(pointer)
            do_independent_work (pointer);
            pointer = pointer->next ;
        } // end of single - no implied barrier (nowait)
    } // end of parallel region - implied barrier
```

Se crean “nt” *threads*

Solo un *thread* “Tx” ejecuta el *while*

Cada vez que “Tx” encuentra **task** crea una nueva tarea

Cada tarea es ejecutada por uno de los “nt” *threads*

# Tareas: Ámbito de las variables



- **Se aplican algunas reglas de las regiones paralelas:**
  - Las variables estáticas y globales son compartidas
  - Las variables locales son privadas
- Si el pragma *task* está dentro de un pragma *parallel*:
  - Si es de tipo **shared** se hereda el tipo
  - Si no **firstprivate**
- Las variables de tareas “*orphaned*” son, por defecto, **firstprivate**

# Tareas: Ámbito de las variables



## *Ejemplo:*

```
int a = 1;
void foo() {
  int b=2,c=3;
  #pragma omp parallel private(b)
  {
    int d=4;
    #pragma omp task
    {
      int e=5;
      a = ¿tipo y valor?
      b = ¿tipo y valor?
      c = ¿tipo y valor?
      d = ¿tipo y valor?
      e = ¿tipo y valor?
    }
  }
}
```

# Tareas: Ámbito de las variables



## **Ejemplo:**

```
int a = 1;
void foo() {
  int b=2, c=3;
  #pragma omp parallel private(b)
  {
    int d=4;
    #pragma omp task
    {
      int e=5;
      a = shared = 1
      b = firstprivate = ??
      c = shared = 3
      d = firstprivate = 4
      e = private = 5
    }
  }
}
```

**Recomendación:** Utilizar **default(none)** si no se está muy seguro

# Tareas: Planificación



- **¿Cuántas tareas se pueden generar y cuál su granularidad?**
  - OpenMP “*runtime*” puede suspender la creación de tareas si el “pool” crece demasiado
  - El programador puede evitar la creación de tareas:
    - mediante cláusulas: ***if, final, mergeable***
    - Transformando ***manualmente*** el código
  
- **¿Qué *thread* ejecuta una determinada tarea?**
  - Por defecto: las tareas están ligadas al *thread* que inicia su ejecución (no necesariamente el que la crea)
  - Cláusula ***untied***: si la tarea se suspende, puede ser retomada por cualquier otro *thread*
    - ¡Evitar el uso de variables *threadprivate* así como cualquier referencia ligada al identificador del *thread* !

# Tareas: cláusula “if”



- Cláusula **if** (expresión):

Si *expresión*=false, se crea una tarea “**no diferida**” (*undelayed*):

La tarea que la genera se suspende, hasta que finaliza su ejecución

– Ayuda a evitar generar tareas “con poco trabajo”

```
int main ( ... )
{
    ...
    #pragma omp parallel shared(n)
    {
        #pragma omp single
        res = f(n);
    }
    ...
}
```

```
int f(int n){
    int x; ...
    #pragma omp task shared(x) if (n>30)
        x = f(n-1);
    . . . .
}
```



# Tareas: cláusula “if”



- Cláusula **if** (expresión):

Si *expresión*=*false*, se crea una tarea “**no diferida**”: La tarea que la genera se suspende, hasta que finaliza su ejecución

– Ayuda a evitar tareas “con poco trabajo”

```
int f(int n){  
    #pragma omp task if (n>30)  
    x = f(n-1);  
    . . . . .  
}
```

## Transformación manual:

```
int f(int n){  
    if (n<=30)  
        return f_serie(n);  
  
    int x;  
    #pragma omp task shared(x)  
        x = f(n-1);  
    . . . . .  
}
```

# Tareas: cláusula “final”



- Cláusula **final** (expresión):

Si *expresión*=true, se crea una tarea “**final**”:

Todas las tareas hijas son también tareas **finales** e incluídas

➡ su ejecución se realizará secuencial e inmediatamente  
(*undeferrred execution*)

- En problemas recursivos evita la creación de tareas al llegar a una cierta profundidad:

```
...  
... task shared(x) final (n<30)  
    x = f(n-1);  
...
```

# Tareas: cláusula “taskyield”



- `#pragma omp taskyield`

Para indicar que la tarea puede ser suspendida a favor de la ejecución de otra tarea

```
void foo(omp_lock_t * lock, int n)
{
    for(int i = 0; i < n; i++)
        #pragma omp task
        {
            something_useful();
            while( !omp_test_lock(lock) ) {
                #pragma omp taskyield
            }
            something_critical();
            omp_unset_lock(lock);
        }
}
```

La tarea que espera puede suspenderse aquí, permitiendo que el *thread* realice otro trabajo

# Tareas: Dependencias (4.0)



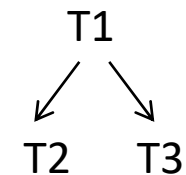
- Se pueden indicar dependencias entre tareas, generadas por los datos que manejan, con la cláusula **depend**:

```
#pragma omp task depend(tipo_dep:lista_vars)
                        tipo_dep: in, out, inout
```

Hasta que no se cumplen todas sus dependencias no puede ejecutar

## Ejemplo:

```
.....
#pragma omp task shared(x, ...) depend(out: x) // T1
    preprocess_some_data(...);
#pragma omp task shared(x, ...) depend(in: x) // T2
    do_something_with_data(...);
#pragma omp task shared(x, ...) depend(in: x) // T3
    do_something_independent_with_data(...);
.....
```



# Tareas: Dependencias (4.0)



- Se pueden indicar dependencias entre tareas, generadas por los datos que manejan, con la cláusula **depend**:

```
#pragma omp task depend(tipo_dep:lista_vars)
                        tipo_dep: in, out, inout
```

Hasta que no se cumplen todas sus dependencias no puede ejecutar

**Ejemplo:**

```
...
char *buffer;
#pragma omp task depend(out:buffer)
{
    buffer = malloc(...);
    stage1 (buffer);
}
#pragma omp task depend(inout:buffer)
    stage2 (buffer);
#pragma omp task depend(input:buffer)
    stage3 (buffer);
```

T1  
↓  
T2  
↓  
T3

# Tareas: Sincronización



- Se espera por su finalización en las **barreras** implícitas y explícitas (**barrier**)
- O utilizando el pragma **taskwait**:

La tarea que lo encuentra se suspende hasta que todas las tareas “hijas” se completen

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

# Afinidad de los *threads*



- *Thread Affinity*: Permite vincular un *thread* a un procesador o conjunto de procesadores.
  - Si no se indica afinidad, los *threads* pueden moverse de un procesador a otro
  - De especial utilidad en nodos “*multisocket*”
- Ventaja:
  - Reutilización de los datos de la caché (menor tasa de fallos)
- Posible inconveniente:
  - Problema de desequilibrio de carga
- OpenMP permite indicar en qué procesadores se ejecutarán los *threads*

# Afinidad: Variables de entorno



- Intel: **KMP\_AFFINITY** [<modifier>,...] <type>

Argumento	Valor defecto	Descripción
modifier	noverbose granularity=core	Opcional. Valores posibles: granularity=<fine, thread, core>, verbose proclist={<proclist>}
type	ninguno	Modelos de afinidad posibles: none, compact, scatter, explicit

## – modifier=**granularity**

- **core**: Cada *thread* es asignado a un core y puede moverse en los contextos de *threads* de ese core (si *hyperthreading*).
- **fine/thread**: Cada *thread* es asignado a un único contexto



# Afinidad: Variable KMP\_AFFINITY



- Intel: **KMP\_AFFINITY** [<modifier>,...] <type>
  - **type**: tipo asignación de *threads* a procesadores
    - **none**: no se usa afinidad  
*Ej*: **KMP\_AFFINITY=verbose,none** (para ver la topología de la máquina)
    - **compact**: Vincula el *thread* <n>+1 al lugar más cercano al *thread* <n>
    - **scatter**: Distribuye los *threads* lo más uniformemente posible entre los procesadores. Lo opuesto a “compact”
    - **explicit**: El usuario especifica explícitamente la asignación  
*Ej*: **KMP\_AFFINITY=verbose, proclist=[{0,2},{4,6},7], explicit**

# KMP\_AFFINITY: Ejemplo



**OMP\_NUM\_THREADS=12 KMP\_AFFINITY=verbose,none ./ejecutable**

OMP: Info #154: KMP\_AFFINITY: Initial OS proc set respected:  
{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23}

Ver en “triqui”

OMP: Info #156: KMP\_AFFINITY: 24 available OS procs

OMP: Info #157: KMP\_AFFINITY: Uniform topology

OMP: Info #179: KMP\_AFFINITY: 2 packages x 6 cores/pkg x 2 threads/core (12 total cores)

OMP: Info #147: KMP\_AFFINITY: Internal thread 0 bound to OS proc set  
{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23}

OMP: Info #147: KMP\_AFFINITY: Internal thread 1 bound to OS proc set  
{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23}

....

OMP: Info #147: KMP\_AFFINITY: Internal thread 11 bound to OS proc set  
{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23}

# Afinidad: ¿Cómo saber la Arquitectura?



```
mgarcia@espino:$ lscpu
```

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                24
On-line CPU(s) list:   0-23
Thread(s) per core:    2
Core(s) per socket:    6
Socket(s):              2
CPU MHz:                600.000
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              12288K
NUMA node0 CPU(s):     0-5,12-17
NUMA node1 CPU(s):     6-11,18-23
```

Ver en “triqui”

# Afinidad: Variables de entorno



- GNU: **GOMP\_CPU\_AFFINITY** (a partir de 3.0).

Alias de:

`KMP_AFFINITY=granularity=fine, proclist=[<proc-list>], explicit`

*Ejemplos:*

`GOMP_CPU_AFFINITY="0-2,4-6,7"`

`GOMP_CPU_AFFINITY=0`

# Afinidad con OpenMP



OpenMP permite indicar dónde se ejecutarán los *threads*, con las variables de entorno:

- **OMP\_PLACES** (a partir de 4.0)

Para indicar los “lugares” posibles en términos del hardware disponible

- **OMP\_PROC\_BIND**

Para indicar cómo se asignan los *threads* a los “lugares”

- **false**: desactiva afinidad (los *threads* se mueven )
  - **true**: la activa (los *threads* no se mueven)
  - **master**, **close** o **spread** (a partir de 4.0)
- } 3.1

# Afinidad: Variables de entorno



- **OMP\_PLACES** (4.0). Los lugares posibles se pueden indicar:
  - Mediante una lista y, para cada lugar: comienzo, longitud e incremento (opcional)

*Ejs:* `OMP_PLACES="{0,1,2,3},{4,5,6,7},{8,9,10,11},{12,13,14,15}"`

`OMP_PLACES="{0:4},{4:4},{8:4},{12:4}" //Equivalente`

`OMP_PLACES="{0,2,4,8},{1,3,5,7}"`

`OMP_PLACES="{0:4:2},{1:4:2}" // Equivalente`

- De modo abstracto con nombres y cuántos hay

*Ejs:* `OMP_PLACES="cores(8)"`

`OMP_PLACES="threads(4)"`

`OMP_PLACES="sockets(2)"`

# Afinidad: Ejemplos

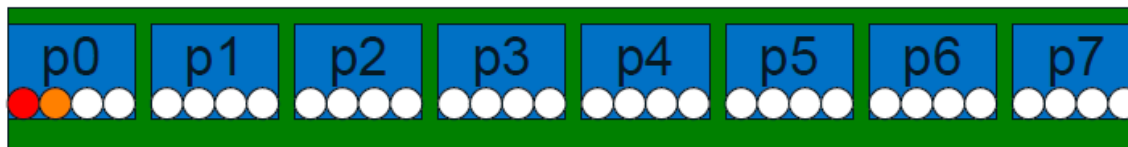


- `OMP_PROC_BIND=master` `OMP_PLACES="cores(8)"`

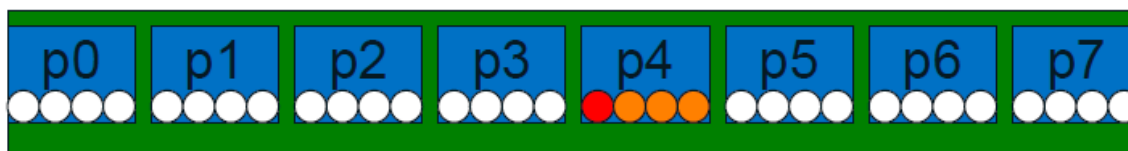
Proximidad de referencias: reutilizar datos de la caché

master 2 \*

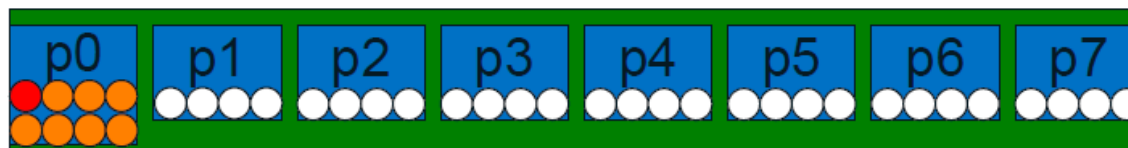
● master    ● worker    ■ partition



master 4



master 8



(\*) N° de *threads*

# Afinidad: Ejemplos

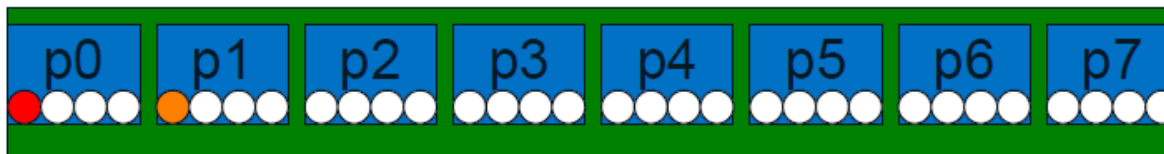


- `OMP_PROC_BIND=close` `OMP_PLACES="cores(8)"`

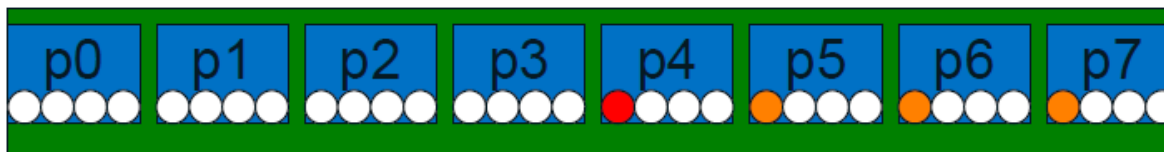
Proximidad de referencias, equilibrio de carga

close 2

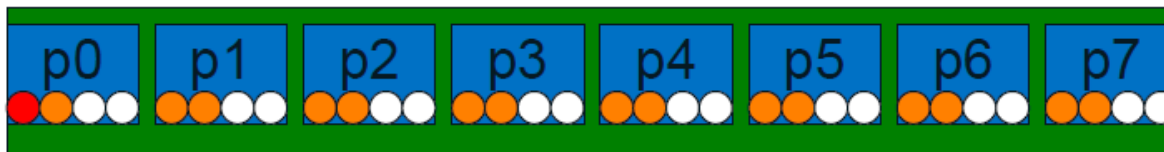
● master ● worker ■ partition



close 4



close 16





# Afinidad: Ejemplos

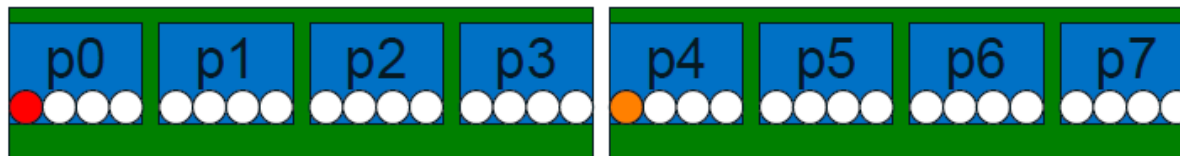


- `OMP_PROC_BIND=spread` `OMP_PLACES="cores(8)"`

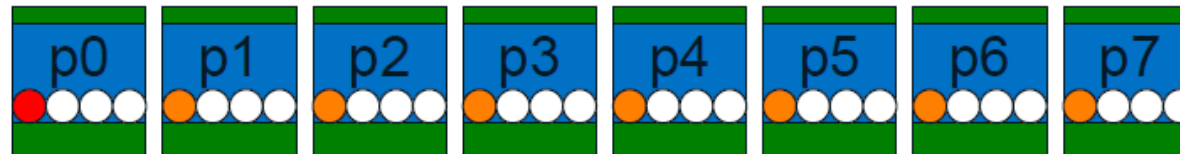
Equilibrio de carga

spread 2

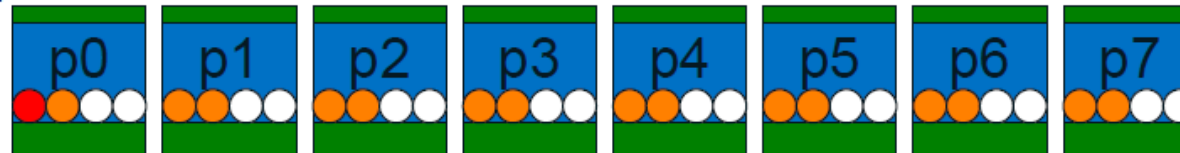
● master ● worker ■ partition



spread 8



spread 16



# Afinidad: Funciones de librería



Para obtener información acerca de la afinidad de los *threads* (a partir de 4.0):

- `omp_get_proc_bind`
- `omp_get_num_places`
- `omp_get_place_num_procs`
- `omp_get_place_proc_ids`
- `omp_get_place_num`

A partir de 4.5

# Vectorización (4.0)



- Directiva:

```
#pragma omp simd [cláusulas]  
for (...)
```

- Permite ejecutar múltiples iteraciones de un bucle concurrentemente mediante instrucciones SIMD



Aprovechar las unidades vectoriales: MMX, SSE, AVX, MIC

- Toma la idea de Intel (icc)
- Se puede combinar con regiones paralelas:

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

- Las iteraciones se distribuyen entre los *threads*
- Las asignadas a cada *thread* se ejecutan como un bucle SIMD

# Soporte para coprocesadores (4.0)

- Directiva:

```
#pragma omp target [cláusulas]
```

- Se busca aprovechar las GPUs y MICs, de gran potencia de cálculo.
  - Se introduce este pragma para indicar el dispositivo sobre el que se quiere ejecutar
  - Toma la idea de PGI, que ya dispone del OpenACC, un OpenMP de pago para GPUs