Programando con OpenMP*

Intel Software College

Objetivos

- Al término de este módulo el estudiante será capaz de
  - Implementar paralelismo de datos
  - Implementar paralelismo de tarea

Agenda

- ¿Qué es OpenMP?
  - Regiones Paralelas
  - Trabajo compartido
  - Ambiente de datos
  - Sincronización
- Tópicos avanzados opcionales

¿Qué es OpenMP?

- API Paralela portable de memoria compartida
  - Fortran, C, y C++
  - Muchos distribuidores soporte para Linux y Windows
- Estand
- Soport
- Combi o fuente
- Estand aralelización dirigid

http://www.openmp.org

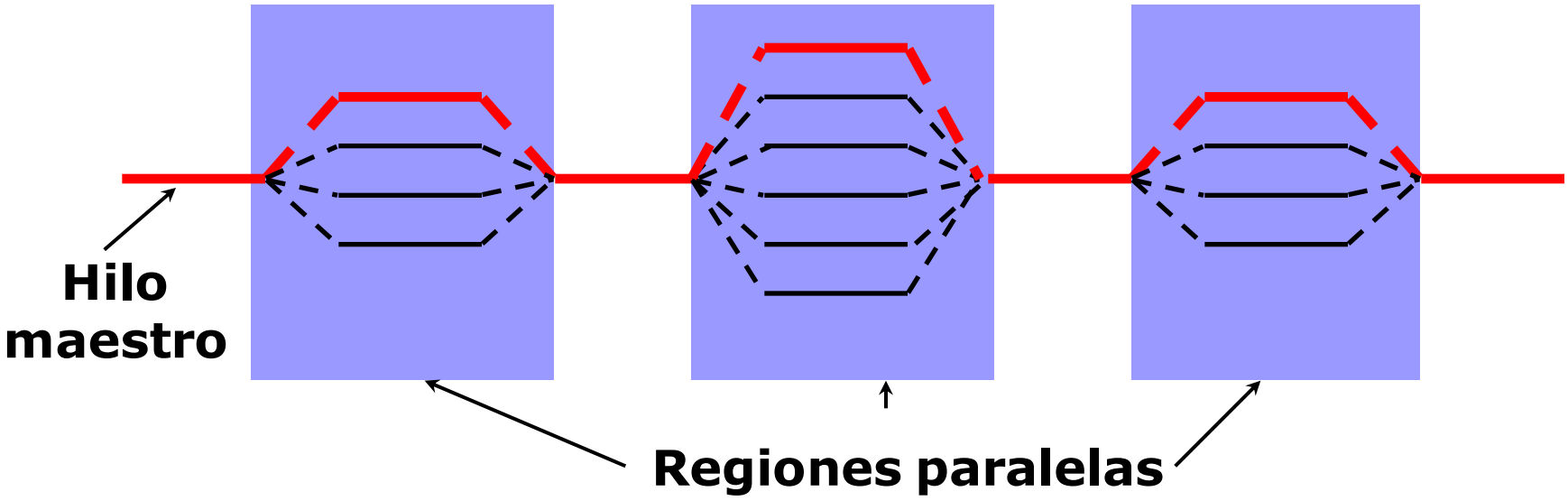
La especificación actual es OpenMP 3.0

318 Páginas

(combina C/C++ y Fortran)

Modelo de programación

- **El hilo maestro se divide en un equipo de hilos como sea necesario**
- **El Paralelismo se añade incrementalmente: el programa secuencial se convierte en un programa paralelo**



Detalles de la sintaxis para comenzar

- Muchas de las construcciones de OpenMP son directivas del compilador o pragmas
  - Para C y C++, los pragmas toman la forma:

```
#pragma omp construct [clause [clause]…]
```

  - Para Fortran, las directivas toman una de las formas:

```
C$OMP construct [clause [clause]…]
!$OMP construct [clause [clause]…]
*$OMP construct [clause [clause]…]
```

- Archivo de cabecera o módulo de Fortran 90

```
#include "omp.h"
use omp_lib
```

Agenda

- ¿Qué es OpenMP?
  - Regiones Paralelas
  - Trabajo compartido
  - Ambiente de datos
  - Sincronización
- Tópicos avanzados opcionales

Región Paralela y Bloques Estructurados (C/C++)

- Muchas de las construcciones de OpenMP se aplican a bloques estructurados
  - Bloque Estructurado: un bloque con un punto de entrada al inicio y un punto de salida al final
  - Los únicos “saltos” permitidos son sentencias de STOP en Fortran y exit() en C/C++

Región Paralela y Bloques Estructurados (C/C++)

```
#pragma omp parallel
{
  int id = omp_get_thread_num();

  more: res[id] = do_big_job (id);

  if (conv (res[id])) goto more;
}
printf ("All done\n");
```

Un bloque estructurado

```
if (go_now()) goto more;
#pragma omp parallel
{
  int id = omp_get_thread_num();
  more:  res[id] = do_big_job(id);
  if (conv (res[id])) goto done;
  goto more;
}
done: if (!really_done()) goto more;
```

Un bloque no estructurado

Actividad 1: Hello Worlds

- Modifica el código serial de “Hello, Worlds” para que se ejecute en paralelo con OpenMP*

Agenda

- ¿Qué es OpenMP?
  - Regiones Paralelas
  - Trabajo compartido
  - Ambiente de datos
  - Sincronización
- Tópicos avanzados opcionales

Trabajo compartido

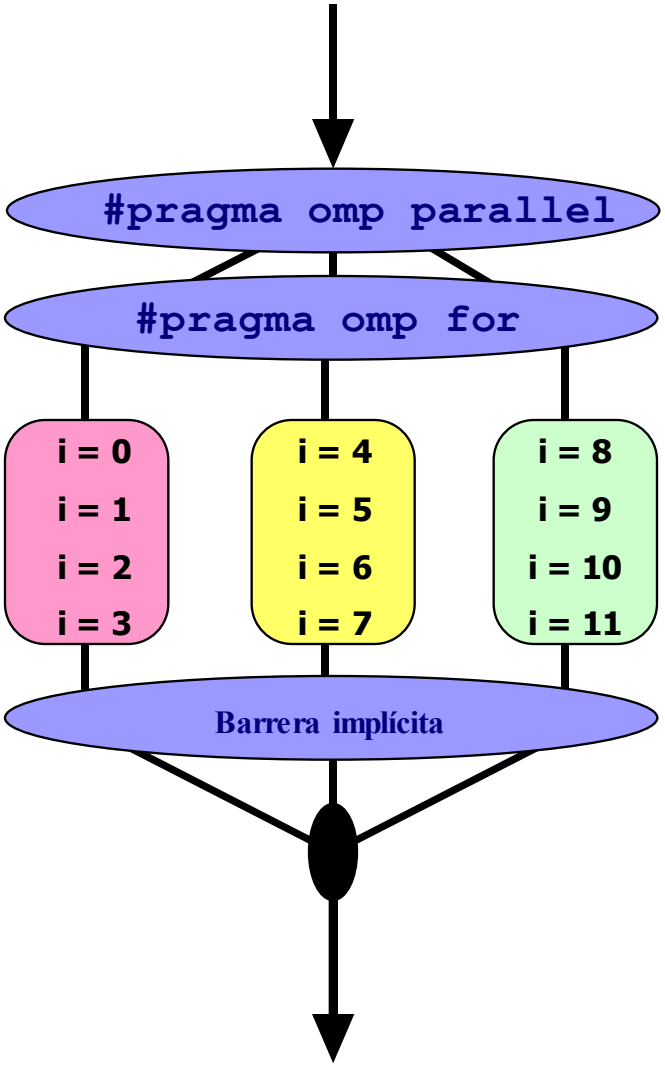
- Trabajo compartido es el término general usado en OpenMP para describir la distribución de trabajo entre hilos.
- Tres ejemplos de trabajo compartido en OpenMP son:
- Construcción omp for
- Construcción omp sections
- Construcción omp task

Automáticamente divide el trabajo entre hilos

Construcción omp for

```
// assume N=12
#pragma omp parallel
#pragma omp for
   for(i = 0, i < N, i++)
      c[i] = a[i] + b[i];
```

- Los hilos se asignan a un conjunto de iteraciones independientes
- Los hilos deben de esperar al final del bloque de construcción de trabajo en paralelo



Combinando pragmas

- Estos códigos son equivalentes

```
#pragma omp parallel
{
    #pragma omp for
    for (i=0;i< MAX; i++) {
       res[i] = huge();
    }
}
```

```
#pragma omp parallel for
    for (i=0;i< MAX; i++) {
         res[i] = huge();
    }
```

La cláusula Private

- Reproduce la variable por cada hilo
  - Las variables no son inicializadas; en C++ el objeto es construido por default
  - Cualquier valor externo a la región paralela es indefinido

```
void* work(float* c, int N) {
  float x, y; int i;
 #pragma omp parallel for private(x,y)
     for(i=0; i<N; i++) {
        x = a[i]; y = b[i];
        c[i] = x + y;
     }
}
```

La cláusula schedule

- La cláusula schedule afecta en como las iteraciones del ciclo se mapean a los hilos

```
schedule(static [,chunk])
```

  - Bloques de iteraciones de tamaño "chunk" a los hilos
  - Distribución Round Robin
  - Poca sobrecarga, puede causar desbalanceo de carga

```
schedule(dynamic[,chunk])
```

  - Los hilos toman un fragmento (chunk) de iteraciones
  - Cuando terminan las iteraciones, el hilo solicita el siguiente fragmento
  - Un poco más de sobrecarga, puede reducir el problema de balanceo de carga

```
schedule(guided[,chunk])
```

  - Planificación dinámica comenzando desde el bloque más grande
  - El tamaño de los bloques se compacta; pero nunca más pequeño que "chunk""

Ejemplo de la cláusula Schedule

```
#pragma omp parallel for schedule (static, 8)
    for( int i = start; i <= end; i += 2 )
    {
       if ( TestForPrime(i) )  gPrimesFound++;
    }
```

- Las iteraciones se dividen en bloques de 8
  - Si start = 3, el primer bloque es

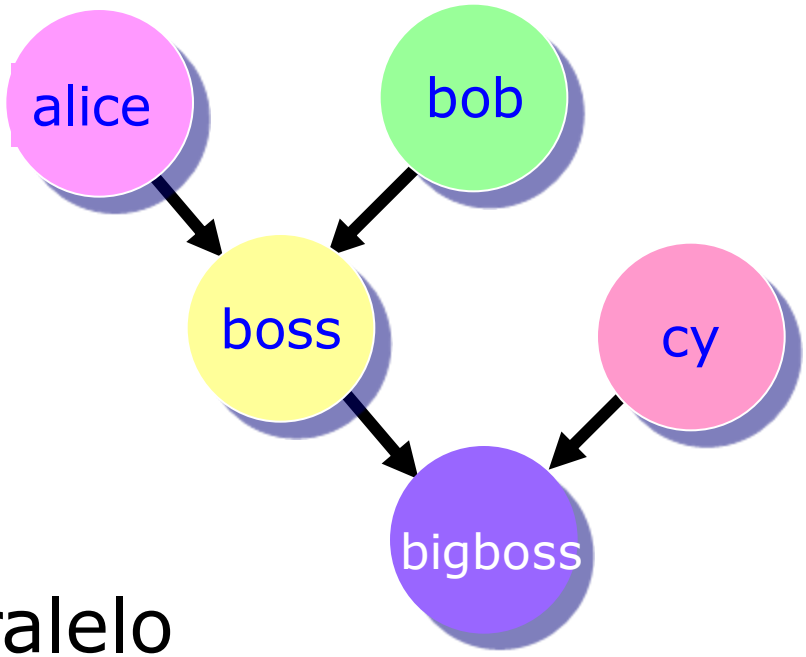
i={3,5,7,9,11,13,15,17}

Agenda

- ¿Qué es OpenMP?
  - Regiones Paralelas
  - Trabajo compartido
  - Ambiente de datos
  - Sincronización
- Tópicos avanzados opcionales

Descomposición de tareas

```
a = alice();
b = bob();
s = boss(a, b);
c = cy();
printf ("%6.2f\n",
  bigboss(s,c));
```



alice,bob, y cy
pueden realizarse en paralelo

Secciones omp

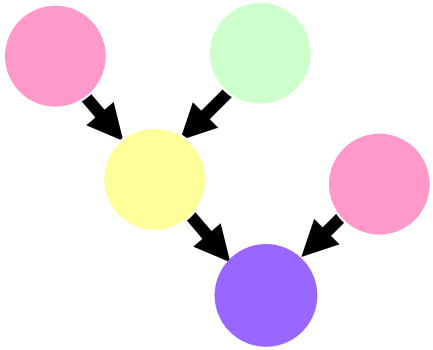
- **`#pragma omp sections`**
  - Debe estar dentro de una región paralela
  - Precede un bloque de código que contiene N bloques de código que pueden ser ejecutados concurrentemente por N hilos
  - Abarca cada sección de omp

Secciones omp

- **`#pragma omp section`**
  - Precede cada bloque de código dentro del bloque abarcado descrito anteriormente
  - Puede ser omitido por la primera sección paralela después del `pragma parallel sections`
  - Los segmentos de programa adjuntos se distribuyen para ejecución paralela entre hilos disponibles

Paralelismo a nivel funcional con secciones

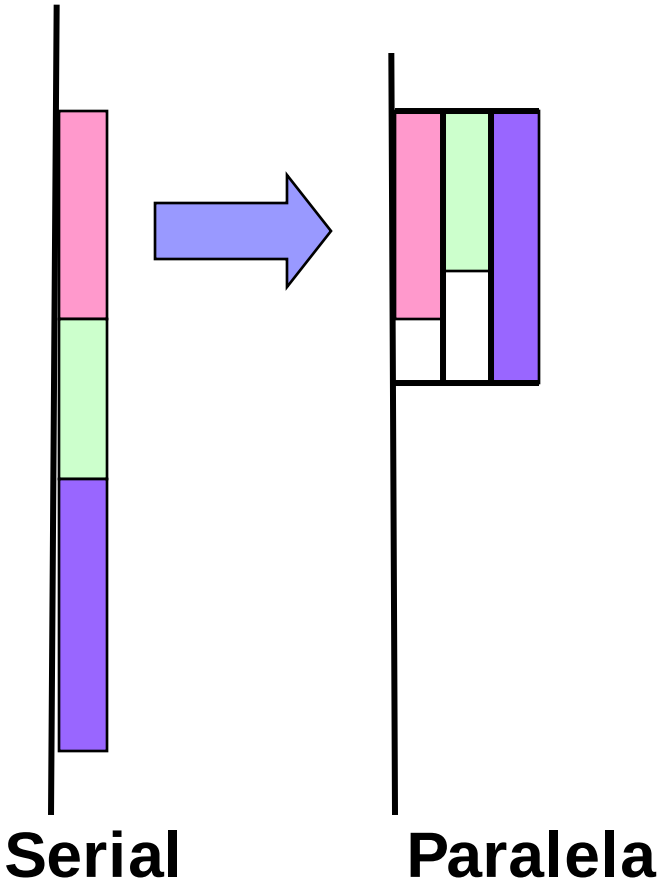
```
#pragma omp parallel sections
{
#pragma omp section  /* Optional */
    a = alice();
#pragma omp section
    b = bob();
#pragma omp section
    c = cy();
}

s = boss(a, b);
printf ("%6.2f\n",
          bigboss(s,c));
```



Ventajas de las secciones paralelas

- Secciones independientes de código se pueden ejecutar concurrentemente

```
#pragma omp parallel sections
{
    #pragma omp section
    phase1();
    #pragma omp section
    phase2();
    #pragma omp section
    phase3();
}
```



Agenda

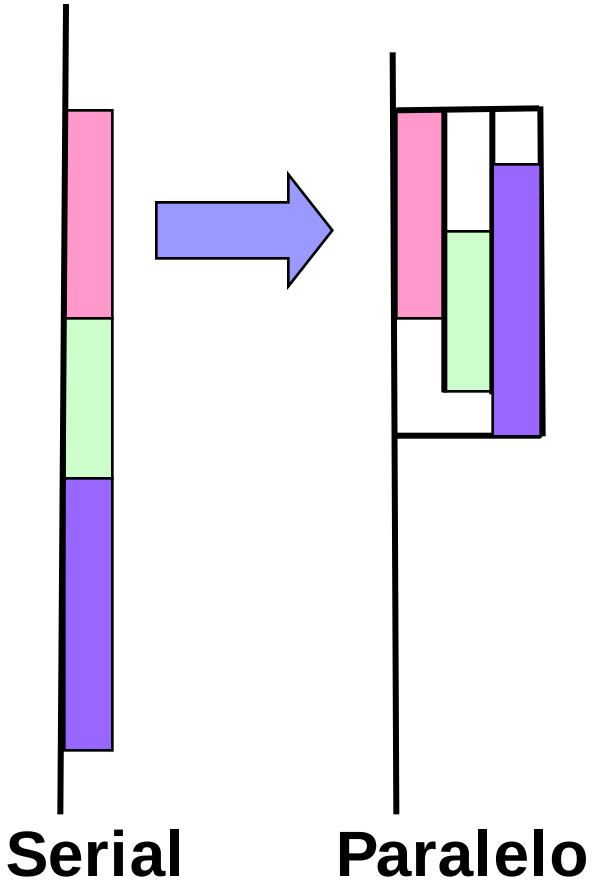
- ¿Qué es OpenMP?
  - Regiones Paralelas
  - Trabajo compartido
  - Ambiente de datos
  - Sincronización
- Tópicos avanzados opcionales

Nuevo Soporte de OpenMP

- Tareas – Lo principal en OpenMP 3.0
- Permite paralelización de problemas irregulares
  - Ciclos sin límite
  - Algoritmos recursivos
  - Productor/Consumidor

¿Qué son las tareas?

- Las tareas son unidades de trabajo independientes
- Los hilos se asignan para ejecutar trabajo en cada tarea
  - Las tareas pueden diferirse
- Las tareas pueden ejecutarse inmediatamente
- El sistema en tiempo de ejecución decide cual de las descritas anteriormente
  - Las tareas se componen de:
    - código para ejecutar
    - Ambiente de datos
    - Variables de control internas (ICV)



Ejemplo de task

```
#pragma omp parallel
// assume 8 threads
{
  #pragma omp single private(p)
  {
    …
    while (p) {
    #pragma omp task
      {
        processwork(p);
      }
      p = p->next;
    }
  }
}
```

Se crea un grupo de 8 hilos

Un hilo tiene acceso a ejecutar el ciclo while

El hilo que ejecuta el “ciclo while” crea una tarea por cada instancia de processwork()

Task – Visión explícita de una tarea

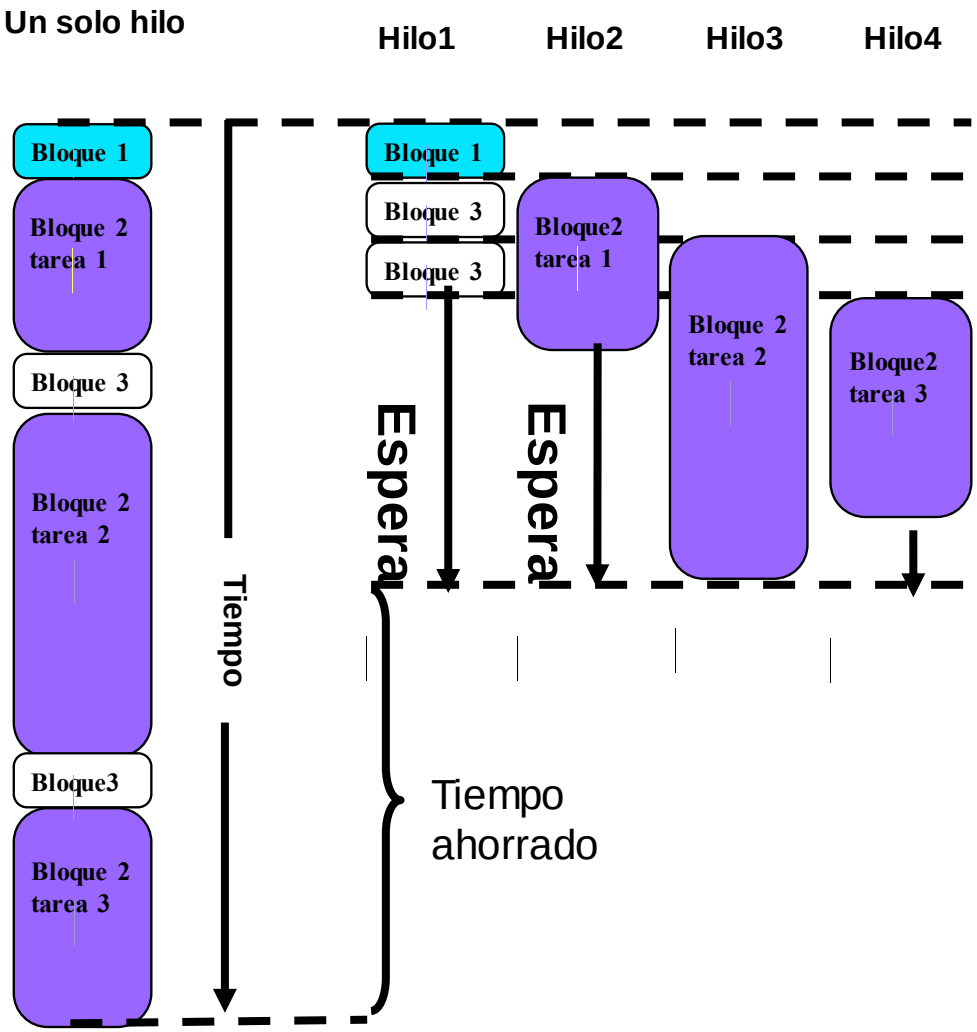
- Se crea un equipo de hilos en el omp parallel
- Un solo hilo se selecciona para ejecutar el ciclo while – a este hilo le llamaremos “L”
- El hilo L opera el ciclo while, crea tareas, y obtiene el siguiente apuntador
- Cada vez que L pasa el omp task genera una nueva tarea que tiene un hilo asignada
- Cada tarea se ejecuta en su propio hilo
- Todas las tareas se terminan en la barrera al final de la región paralela

```
#pragma omp parallel
{
  #pragma omp single
  {  // block 1
     node * p = head;
     while (p) {  //block 2
     #pragma omp task private(p)
        process(p);
     p = p->next;  //block 3
     }
  }
}
```

¿Qué tareas son útiles?

Tienen potencial para paralelizar patrones irregulares y llamadas a funciones recursivas

```
#pragma omp parallel
{
  #pragma omp single
  {  // block 1
     node * p = head;
     while (p) {  //block 2
     #pragma omp task
        process(p);
     p = p->next;  //block 3
     }
  }
}
```



Actividad 2 – Lista encadenada usando tareas

- Objetivo: Modifica la lista encadenada para implementar tareas para paralelizar la aplicación
- Sigue la lista encadenada de tareas llamada LinkedListTask en el documento de la práctica

```
while(p != NULL){
  do_work(p->data);
  p = p->next;
}
```

¿Cuándo las tareas se garantizan a ser completadas?

- Las tareas se garantizan a ser completadas:
- En las barreras de los hilos o tareas
  - En la directiva: **`#pragma omp barrier`**
  - En la directiva : **`#pragma omp taskwait`**

Ejemplo de terminación de tareas

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

Aquí se crean varias tareas foo – una para cada hilo

Se garantiza que todas las tareas foo terminan aquí

Aquí se crea una tarea bar

Se garantiza que la tarea bar termina aquí

Agenda

- ¿Qué es OpenMP?
  - Regiones Paralelas
  - Trabajo compartido
  - Ambiente de datos
  - Sincronización
- Tópicos avanzados opcionales

Alcance de los datos – ¿Qué se comparte?

- OpenMP usa un modelo de memoria compartida
- **Variable compartida -** una *variable* que pueden leer o escribir varios hilos
- La cláusula shared puede usarse para hacer elementos explícitamente compartidos
  - Las variables globales se comparten por default entre tareas
    - Variables globales, variables con alcance del namespace, variables estáticas, variables con calificador de constante que no tienen miembro mutable son compartidas, Variables estáticas que son declaradas en un alcance dentro del bloque de construcción son compartidas

Alcance de los datos – ¿Qué es privado?

- No todo se comparte...
  - Ejemplos de determinadas variables implícitamente privadas:
    - Las variables locales (stack) en funciones llamadas de regiones paralelas son PRIVADAS
    - Las variables automáticas dentro de un bloque de sentencias de omp son PRIVADAS
    - Las variables de iteración de ciclos son privadas
    - Las variables implícitamente declaradas privadas dentro de tareas serán tratadas como firstprivate

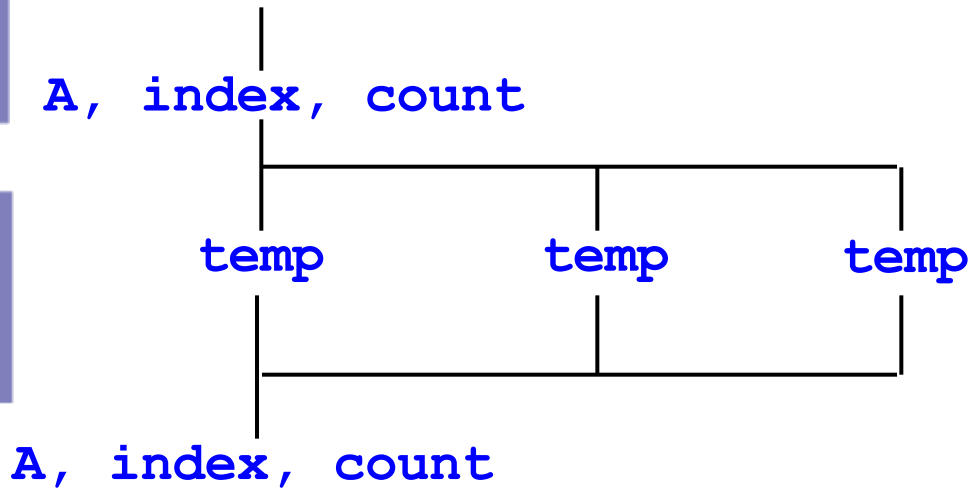
- La cláusula Firstprivate declara uno o más elementos a ser privados para una tarea, y inicializa cada uno de ellos con un valor

Un ejemplo de ambiente de datos

```
float A[10];
main ()
{
  integer index[10];
  #pragma omp parallel
  {
      Work (index);
  }
    printf ("%d\n", index[1]);
}
```

```
extern float A[10];
void Work (int *index)
{
  float temp[10];
  static integer count;
  <...>
}
```

***A*, *índex*, y *count* se comparten en todos los hilos, pero *temp* es local para cada hilo**



Problema con el Alcance de los datos – ejemplo de fib

```
int fib ( int n )
{

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

n es privada en ambas tareas

x es una variable privada
y es una variable privada

¿Qué es incorrecto?

No se pueden usar variables privadas fuera de las tareas

Ejemplo del alcance de datos – ejemplo de fib

```
int fib ( int n )
{

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

n es privada en ambas tareas

x & y son compartidas
Buena solución
necesitamos ambos valores para calcular sum

Problema con el alcance de datos – Recorrido de listas

```
List ml; //my_list
Element *e;
#pragma omp parallel
#pragma omp single
{
   for(e=ml->first;e;e=e->next)
#pragma omp task
      process(e);
}
```

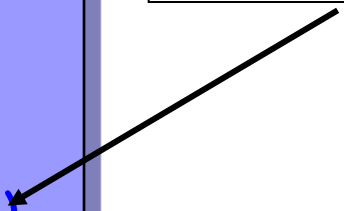
¿Qué está mal aquí?

Posible condición de concurso !
La variable compartida e
la actualizan múltiples tareas

Ejemplo de alcance de datos – Recorrido de listas

```
List ml; //my_list
Element *e;
#pragma omp parallel
#pragma omp single
{
   for(e=ml->first;e;e=e->next)
#pragma omp task firstprivate(e)
      process(e);
}
```

Buena solución – e es firstprivate



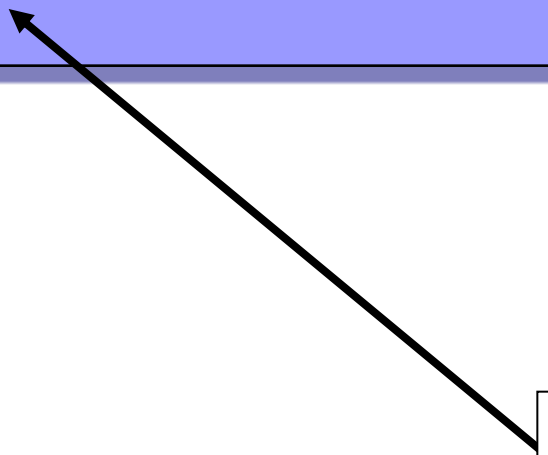
Ejemplo de alcance de datos – Recorrido de listas

```
List ml; //my_list
Element *e;
#pragma omp parallel
#pragma omp single private(e)
{
   for(e=ml->first;e;e=e->next)
#pragma omp task
      process(e);
}
```

Buena solución – e es privada

Ejemplo de alcance de datos – Recorrido de listas

```
List ml; //my_list
#pragma omp parallel
{
   Element *e;
   for(e=ml->first;e;e=e->next)
#pragma omp task
      process(e);
}
```



Buena solución – e es privada

Agenda

- ¿Qué es OpenMP?
  - Regiones Paralelas
  - Trabajo compartido
  - Ambiente de datos
  - Sincronización
- Tópicos avanzados opcionales

Ejemplo: Producto Punto

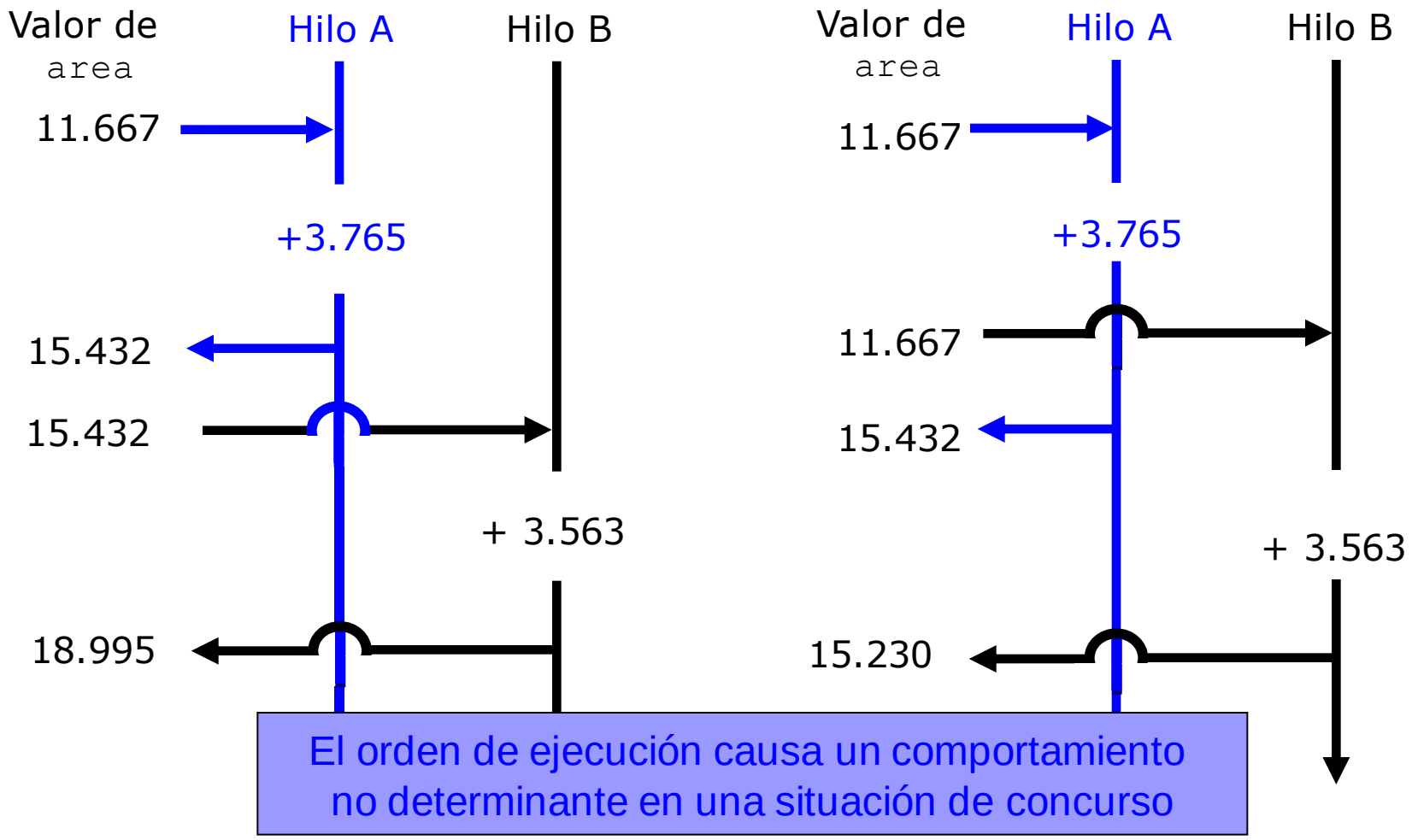
```
float dot_prod(float* a, float* b, int N)
{
  float sum = 0.0;
#pragma omp parallel for shared(sum)
   for(int i=0; i<N; i++) {
     sum += a[i] * b[i];
   }
  return sum;
}
```

¿Que está mal?

Condiciones de Concurso

- Una condición de concurso es un comportamiento no-determinístico causado cuando dos o más hilos acceden una variable compartida
- Por ejemplo, supón que el hilo A y el hilo B están ejecutando
- `area += 4.0 / (1.0 + x*x);`

Dos ejemplos



Proteger Datos Compartidos

- Debe proteger acceso a datos compartidos modificables

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

Cláusula OpenMP* Critical

```
#pragma omp critical [(lock_name)]
```

- Define una región crítica en un bloque estructurado

Los hilos esperan su turno –en un momento, solo uno llama `consum()` protegiendo RES de condiciones de concurso

Nombrar la sección crítica `RES_lock` es opcional

```
float RES;
#pragma omp parallel
{ float B;
#pragma omp for
  for(int i=0; i<niters; i++){
    B = big_job(i);
#pragma omp critical (RES_lock)
    consum (B, RES);
  }
}
```

Buena Práctica – Nombrar todas las secciones críticas

Cláusula de reducción OpenMP*

```
reduction (op : lista)
```

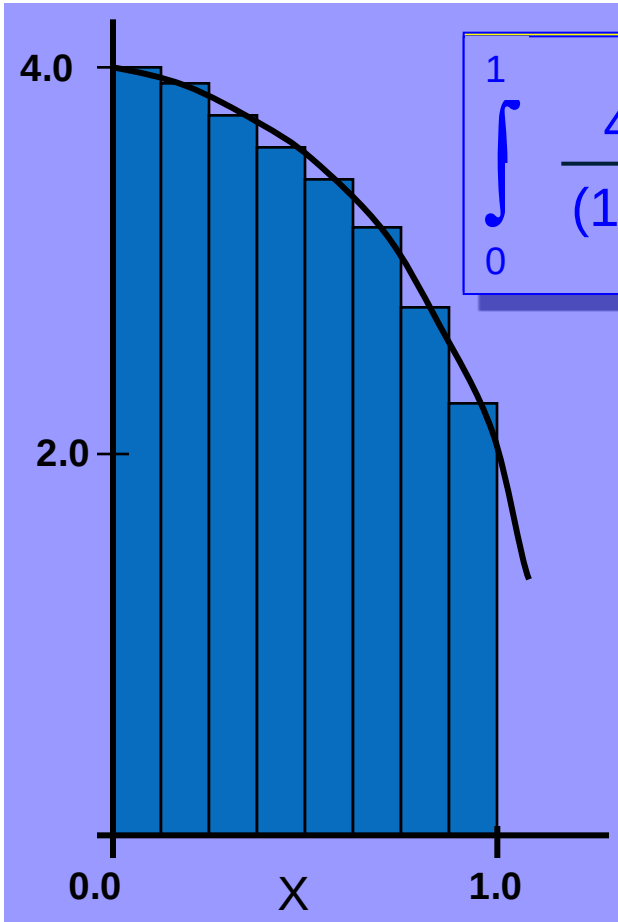
- Las variables en la “lista” deben ser compartidas en la región paralela
- Dentro un bloque parallel o de trabajo compartido:
  - Se crea una copia PRIVADA de cada variable de la lista y se inicializa de acuerdo a “op”
  - Estas copias se actualizan localmente por los hilos
  - Al final del bloque, las copias locales se combinan a través de la “op” en un solo valor con el valor que tenía la variable original COMPARTIDA

Ejemplo de reducción

```
#pragma omp parallel for reduction(+:sum)
   for(i=0; i<N; i++) {
     sum += a[i] * b[i];
   }
```

- Copia local de *sum* para cada hilo
- Todas las copias locales de *sum* se suman y se guardan en la variable “global”

Ejemplo de Integración Numérica



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

```
static long num_steps=100000; double
step, pi;

void main()
{   int i;
    double x, 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;
    printf("Pi = %f\n",pi);
}
```

C/C++ Operaciones de reducción

- Un rango de operadores asociativos y conmutativos pueden usarse con la reducción
- Los valores iniciales son aquellos que tienen sentido

Operador	Valor Inicial
+	0
*	1
-	0
^	0

Operador	Valor Inicial
&	~0
|	0
&&	1
||	0

Actividad 3 - Calcular Pi

```
static long num_steps=100000;
double step, pi;

void main()
{  int i;
   double x, 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;
   printf("Pi = %f\n",pi);
}
```

- Paraleliza el código de integración numérica usando OpenMP
- ¿Qué variables pueden compartirse?
- ¿Qué variables necesitan ser privadas?
- ¿Qué variables pueden usarse en reducciones?

Bloque de construcción Single

- Denota un bloque de código que será ejecutado por un solo hilo
  - El hilo seleccionado es dependiente de la implementación
- Barrera implícita al final

```
#pragma omp parallel
{
   DoManyThings();
#pragma omp single
   {
     ExchangeBoundaries();
   }  // threads wait here for single
   DoManyMoreThings();
}
```

Bloque de construcción Master

- Denota bloques de código que serán ejecutados solo por el hilo maestro
- No hay barrera implícita al final

```
#pragma omp parallel
{
   DoManyThings();
#pragma omp master
   {                  // if not master skip to next stmt
     ExchangeBoundaries();
   }
   DoManyMoreThings();
}
```

Barreras implícitas

- Varias sentencias de OpenMP* tienen barreras implícitas
  - Parallel – barrera necesaria – no puede removerse
  - for
  - single
- Las barreras innecesarias afectan el rendimiento y pueden removerse con la cláusula nowait
  - La cláusula nowait puede aplicarse a :
    - La cláusula For
    - La cláusula Single

Cláusula Nowait

```
#pragma omp for nowait
   for(...)
     {...};
```

```
#pragma single nowait
{ [...] }
```

- Cuando los hilos esperarían entren cómputos independientes

```
#pragma omp for schedule(dynamic,1) nowait
 for(int i=0; i<n; i++)
   a[i] = bigFunc1(i);

#pragma omp for schedule(dynamic,1)
 for(int j=0; j<m; j++)
   b[j] = bigFunc2(j);
```

Barreras

- Sincronización explícita de barreras
- Cada hilo espera hasta que todos lleguen

```
#pragma omp parallel shared (A, B, C)
{
    DoSomeWork(A,B); // Processed A into B
#pragma omp barrier

     DoSomeWork(B,C); // Processed B into C
}
```

Operaciones Atómicas

- Caso especial de una sección crítica
- Aplica solo para la actualización de una posición de memoria

```
#pragma omp parallel for shared(x, y, index, n)
   for (i = 0; i < n; i++) {
      #pragma omp atomic
        x[index[i]] += work1(i);
      y[i] += work2(i);
   }
```

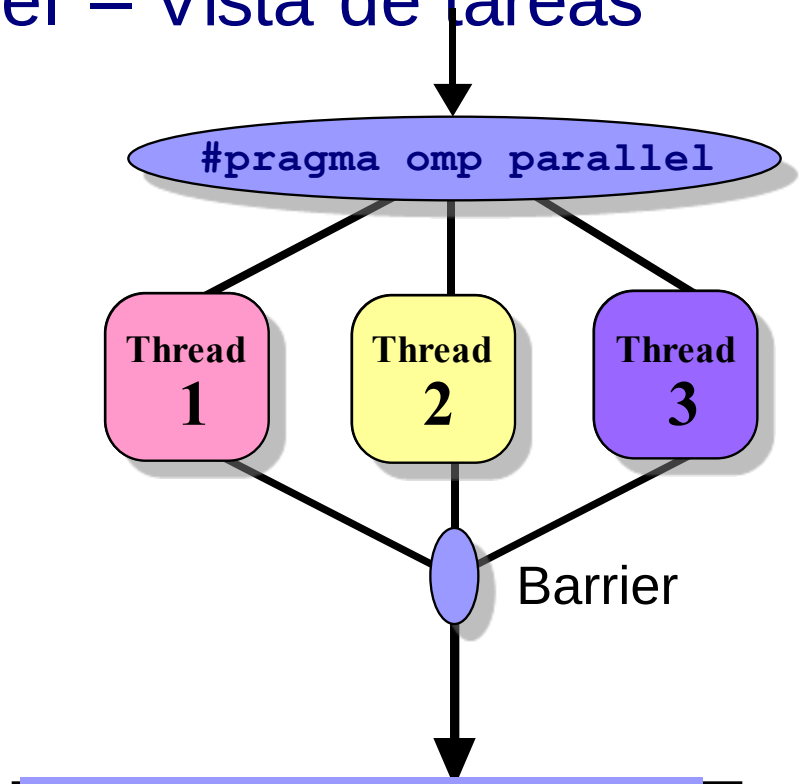
Agenda

- ¿Qué es OpenMP?
  - Regiones Paralelas
  - Worksharing
  - Ambiente de datos
  - Sincronización
- Tópicos avanzados opcionales

Conceptos Avanzados

Bloque de Construcción Parallel – Vista de tareas explícita

- Las tareas se crean en OpenMP incluso sin una directiva task explícita
- Veremos como las tareas se crean implícitamente para el fragmento de código que está abajo.
  - El hilo que encuentra la sentencia parallel empaca un conjunto de tareas implícitas
  - Se crea un conjunto de hilos.
  - Cada hilo en el equipo está asignado a una de las tareas (y vinculado a ella).
  - La barrera mantiene el hilo maestro original hasta que todas las tareas implícitas terminan




```
#pragma omp parallel
    {
        int mydata;

    }
```

Bloque de construcción Task

```
#pragma omp task [clause[[,]clause] ...]
                 bloque estructurado
```

Donde la *claúsula* **puede ser un:**

```
if (expresion)
untied
shared (lista)
private (lista)
firstprivate (lista)
default( shared | none  )
```

Tareas vinculadas y tareas no vinculadas

- Tareas vinculadas:
  - Una tarea vinculada se le asigna un hilo en su primera ejecución y este mismo hilo le da servicio a la tarea por su tiempo de vida.
  - Un hilo ejecutando una tarea vinculada, puede suspenderse, y enviarse a ejecutar otra tarea, pero eventualmente, el mismo hilo regresará a continuar la ejecución de su tarea vinculada originalmente.
  - Las tareas están vinculadas mientras no se declare desvincular explícitamente

Tareas vinculadas y tareas no vinculadas

- Tareas no vinculadas:
  - Una tarea no vinculada no tienen ninguna asociación a largo plazo con ningún hilo. Cualquier hilo que no esté ocupado puede ejecutar una tarea no vinculada. El hilo asignado para ejecutar una tarea no vinculada solo puede cambiar en un “punto de planificación de tareas”
  - Una tarea no vinculada se crea agregando “untied” a la cláusula tarea
  - Ejemplo: #pragma omp task untied

Cambio de tareas

- **Cambio de tareas** El acto de un hilo en cambiar de la ejecución de una tarea a otra tarea.

- El propósito de cambiar la tarea es distribuir hilos a lo largo de las tareas no asignadas en el equipo para evitar que se acumulen colas largas de tareas no asignadas

Cambio de tareas

- El cambio de tareas, para tareas vinculadas, solo puede ocurrir en puntos de planificación de tareas localizados dentro de los siguientes bloques de construcción
  - Se encuentran sentencias task
  - Se encuentran sentencias taskwait
  - Se encuentran directivas barrier
  - Regiones barrier implícitas
  - Al final de una región de tarea vinculada

- Las tareas no vinculadas tienen implementación dependiendo de los puntos de planificación

Ejemplo de cambio de tareas

- El hilo que ejecuta el “ciclo for”, que es el generador de tareas, genera muchas tareas en poco tiempo tal que...
- El hilo que es SINGLE está generando tareas y tendrá que suspender por un momento cuando el “conjunto de tareas” se llene
  - El intercambio de tareas se invica para iniciar el vaciado del “conjunto de tareas”
  - Cuando el “conjunto de tareas” se ha vaciado lo suficiente – la tarea en el bloque single puede seguir generando más tareas

```
#pragma omp single
{
    for (i=0; i<ONEZILLION; i++)
     #pragma omp task
         process(item[i]);
    }
}
```

Opciones adicionales – API de OpenMP*

- Obtener el número de hilo dentro de un grupo

```
int omp_get_thread_num(void);
```

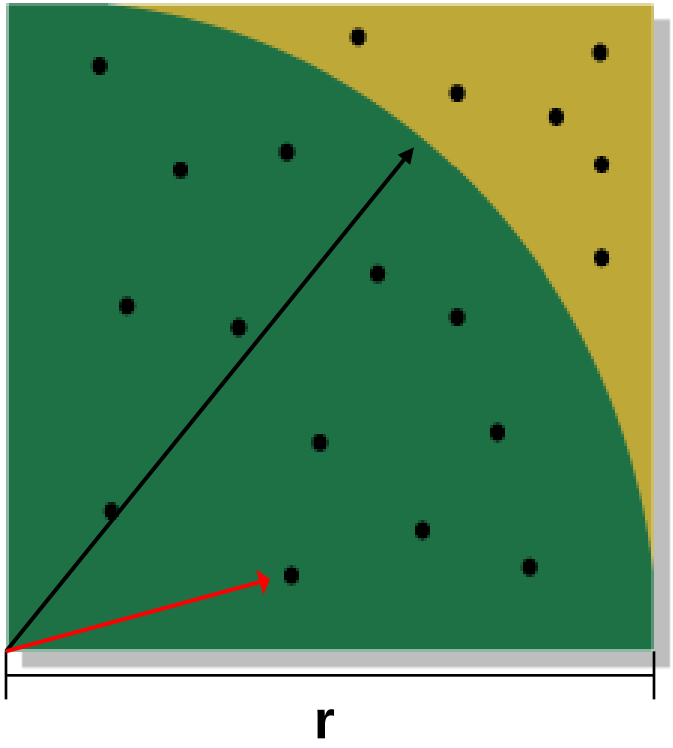
- Obtener el número de hilos dentro de un grupo

```
int omp_get_num_threads(void);
```

- Usualmente no necesario en los códigos de OpenMP
  - Pude hacer que el código no sea serialmmente consistente
  - Tiene usos específicos (debugging)
  - Se debe incluir el archivo de cabecera

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

Opciones adicionales - Monte Carlo Pi



$$\frac{\#\text{ of darts hitting circle}}{\#\text{ of darts in square}} = \frac{1/4\pi r^2}{r^2}$$

$$\pi = 4\frac{\#\text{ of darts hitting circle}}{\#\text{ of darts in square}}$$

```
loop 1 to MAX
   x.coor=(random#)
   y.coor=(random#)
   dist=sqrt(x^2 + y^2)
   if (dist <= 1)
      hits=hits+1

 pi = 4 * hits/MAX
```

Opcional – Haciendo Monte Carlo en Paralelo

```
hits = 0
call SEED48(1)
DO I = 1, max
    x = DRAND48()
    y = DRAND48()
    IF (SQRT(x*x + y*y) .LT. 1) THEN
       hits = hits+1
    ENDIF
END DO
pi = REAL(hits)/REAL(max) * 4.0
```

¿Cuál es el reto aquí?

Actividad Opcional 5: Calcular Pi

- Use la liberería Intel® Math Kernel Library (Intel® MKL) VSL:
  - Intel MKL's VSL (Vector Statistics Libraries)
  - VSL crea un arreglo, en vez de un solo número aleatorio
  - VSL puede tener varias semillas (una para cada hilo)

- Objetivo:
  - Usar lo básico de OpenMP para hacer el código de Pi paralelo
  - Escoge el mejor código para dividir las tareas
  - Categoriza propiamente todas las variables

Cláusula Firstprivate

- Variables inicializadas de una variable compartida
- Los objetos de C++ se construyen a partir de una copia

```
incr=0;
#pragma omp parallel for firstprivate(incr)
for (I=0;I<=MAX;I++) {
      if ((I%2)==0) incr++;
      A(I)=incr;
}
```

Cláusula Lastprivate

- Las variables actualizan la variable compartida usando el valor de la última iteración
- Los objetos de C++ se actualizan por asignación

```
void sq2(int n,
         double *lastterm)
{
  double x; int i;
  #pragma omp parallel
  #pragma omp for lastprivate(x)
  for (i = 0; i < n; i++){
     x = a[i]*a[i] + b[i]*b[i];
     b[i] = sqrt(x);
  }
  lastterm = x;
}
```

Cláusula Threadprivate

- Preserva el alcance global en el almacenamiento por hilo
- Usa copia para inicializar a partir del hilo maestro

```
struct Astruct A;
#pragma omp threadprivate(A)
…

#pragma omp parallel copyin(A)
  do_something_to(&A);
…

#pragma omp parallel
  do_something_else_to(&A);
```

Las copias privadas de "A" persisten entre regiones

20+ Funciones de librería

- Rutinas de ambiente en tiempo de ejecución:
  - Modifica/revisa el número de hilos

    `omp_[set|get]_num_threads()`

    `omp_get_thread_num()`

    `omp_get_max_threads()`
  - ¿Estamos en una región paralela?

    `omp_in_parallel()`
  - ¿Cuántos procesadores hay en el sistema?

    `omp_get_num_procs()`
  - Locks explícitos

    `omp_[set|unset]_lock()`
  - Y muchas más...

Funciones de librería

- Para arreglar el número de hilos usados en el programa
  - Establecer el número de hilos
  - Almacena el número obtenido

```
#include <omp.h>

void main ()
{
  int num_threads;
  omp_set_num_threads (omp_num_procs ());

  #pragma omp parallel
  {
    int id = omp_get_thread_num ();

    #pragma omp single
    num_threads = omp_get_num_threads ();

    do_lots_of_stuff (id);
  }
}
```

Solicita tantos hilos como haya procesadores disponibles.

Protégé esta operación porque los almacenamientos en memoria no son atómicos


intel®
Leap ahead™

BACKUP