SC-Camp 2011

Turrialba, Costa Rica, julio 11~15

Programación con OpenMP

Prof. Robinson Rivas-Suarez

Universidad Central de Venezuela

Material tomado de la Intel Software College

Objetivos

Al término de este módulo el estudiante será capaz de

- Crear hilos utilizando pragmas OpenMP
- Usar pragmas de sincronización OpenMP para coordinar la ejecución de los hilos y acceso a memoria

Agenda

¿Qué es OpenMP?

Regiones Paralelas

Bloques de construcción para trabajo en paralelo

Alcance de los datos para proteger datos

Sincronización Explícita

Cláusulas de Planificación

Otros bloques de construcción y cláusulas útiles

¿Qué es OpenMP*?

Directivas del compilador para programación multihilos

Es fácil crear hilos en Fortran y C/C++

Soporta el modelo de paralelismo de datos

Paralelismo incremental

Combina código serial y paralelo en un solo código fuente

¿Qué es OpenMP*?

```
C$OMP FLUSH
#pragma omp critical
C$OMP THREADPRIVATE(/ABC/)
CALL OMP_SET_NUM_THREADS(10)
C$OMP parallel do shared(a, b, c)
call omp_test_lock(jlok)
C$OMP MASTER
call OMP_INIT
C
C$OMP  SINGLE PRIV
ynamic"
C$OMP PARALLEL D
C$OMP ORDERED
C$OMP PARALLEL
IONS
#pragma omp parallel for private(A, B)
!$OMP  BARRIER
C$OMP PARALLEL COPYIN(/blk/)
C$OMP DO lastprivate(XX)
Nthrds = OMP_GET_NUM_PROCS()
omp_set_lock(lck)
```

http://www.openmp.org

La especificación actual es OpenMP 2.5

250 Páginas

(C/C++ y Fortran)

Arquitectura OpenMP*

Modelo fork-join

Bloques de construcción para trabajo en paralelo

Bloques de construcción para el ambiente de datos

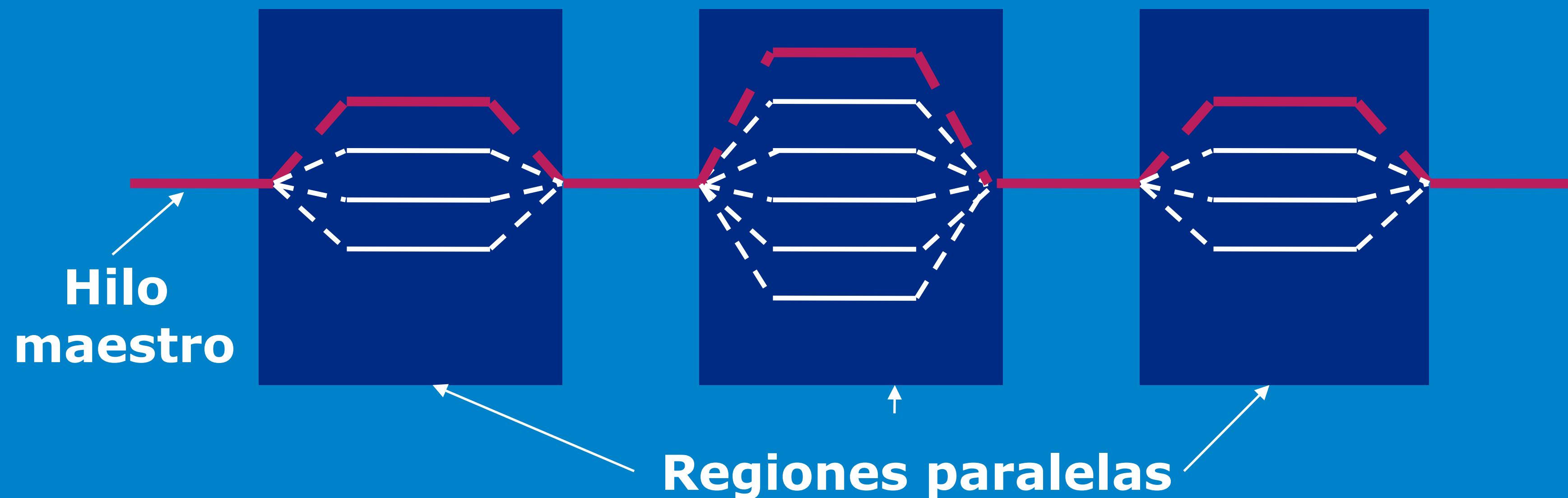
Bloques de construcción para sincronización

API (Application Program Interface) extensiva para afinar el control

Modelo de programación

Paralelismo fork-join:

- **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



Sintaxis del Pragma OpenMP*

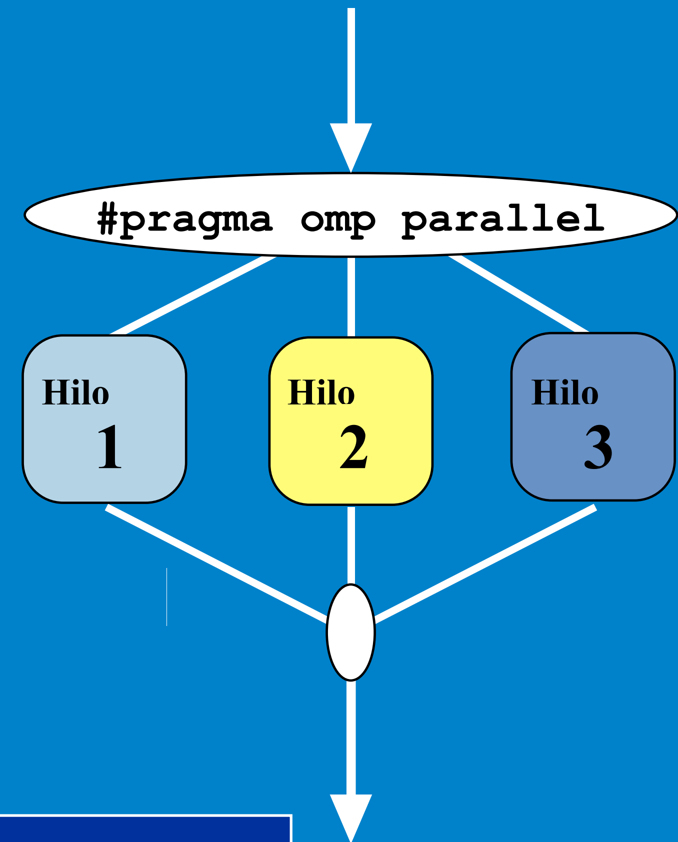
La mayoría de los bloques de construcción en OpenMP* son directivas de compilación o pragmas.

- En C y C++, los pragmas toman la siguiente forma:

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

Regiones Paralelas

- Define una región paralela sobre un bloque de código estructurado
- Los hilos se crean como `parallel`
- Los hilos se bloquean al final de la región
- Los datos se comparten entre hilos al menos que se especifique otra cosa



C/C++ :

```
#pragma omp parallel
    {
            bloque
    }
```

¿Cuántos hilos?

Establecer una variable de ambiente para el número de hilos

```
set OMP_NUM_THREADS=4
```

No hay un default estándar en esta variable

- En muchos sistemas:
  - # de hilos = # de procesadores
  - Los compiladores de Intel® usan este default

Actividad 1: Hello Worlds

Modificar el código serial de “Hello, Worlds” para ejecutarse paralelamente usando OpenMP*

```
int main()
{
    saludo();
}
int saludo()
{
    int i;

    for(i=0;i<10;i++)
    {
        printf("Hola mundo desde el hilo principal\n");
        sleep(1)
    }
}
```

Bloques de construcción de trabajo en paralelo

```
#pragma omp parallel
#pragma omp for
      for (i=0; i<N; i++){
            Do_Work(i);
      }
```

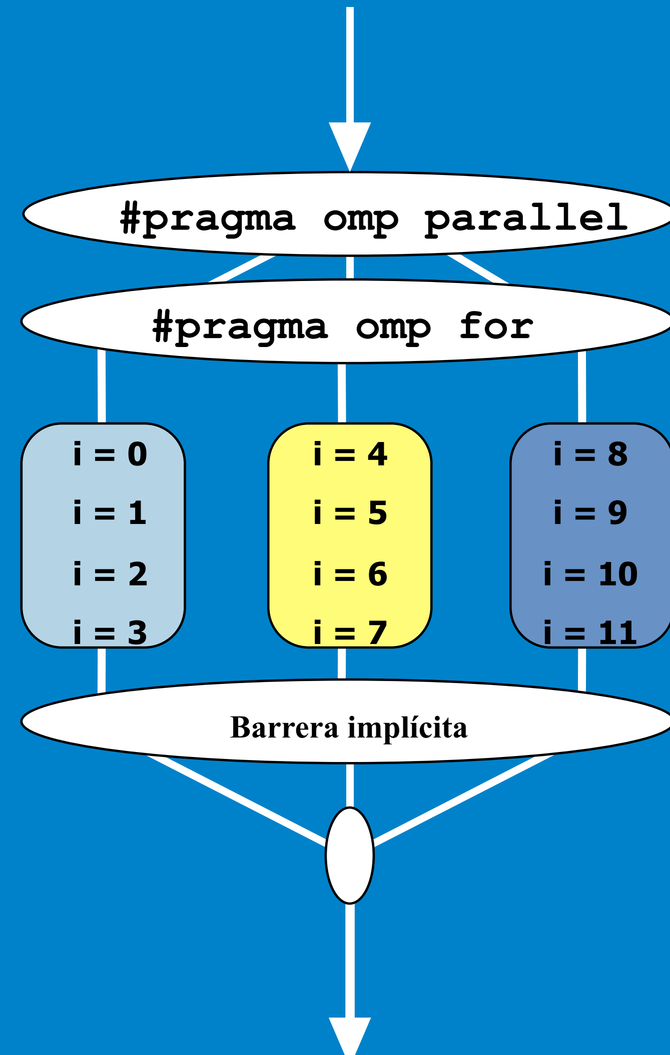
Divide las iteraciones del ciclo en hilos

Debe estar en la región paralela

Debe preceder el ciclo

Bloques de construcción de trabajo en paralelo

```
#pragma omp parallel
#pragma omp for
   for(i = 0; i < 12; 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

Ambos segmentos de código 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();
     }
```

Ambiente de datos

OpenMP usa un modelo de programación de memoria compartida

- La mayoría de las variables por default son compartidas.
- Las variables globales son compartidas entre hilo

Ambiente de datos

Pero, no todo es compartido...

- Las variables en el stack en funciones llamadas de regiones paralelas son PRIVADAS

- Las variables automáticas dentro de un bloque son PRIVADAS

- Las variables de índices en ciclos son privadas (salvo excepciones)
  - C/C+: La primera variable índice en el ciclo en ciclos anidados después de un **`#pragma omp for`**

Atributos del alcance de datos

El estatus por default puede modificarse

```
default (shared | none)
```

Clausulas del atributo de alcance

```
shared(varname,…)
```

```
private(varname,…)
```

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;
     }
}
```

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;
}
```

¿Qué es incorrecto?

Proteger datos compartidos

Debe proteger el acceso a los datos compartidos que son 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;
}
```

OpenMP* Bloques de construcción para regiones críticas

```
#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 R1 y R2 de de condiciones de concurso.

Nombrar las regiones críticas es opcional, pero puede mejorar el rendimiento.

```
float R1, R2;
#pragma omp parallel
{ float A, B;
#pragma omp for
  for(int i=0; i<niters; i++){
    B = big_job(i);
#pragma omp critical (R1_lock)
    consum (B, &R1);
    A = bigger_job(i);
#pragma omp critical (R2_lock)
    consum (A, &R2);
  }
}
```

OpenMP* Cláusula de reducción

```
reduction (op : list)
```

Las variables en "*list*" deben ser compartidas dentro de la región paralela

Adentro de parallel o el bloque de construcción de trabajo en paralelo:

- Se crea una copia PRIVADA de cada variable de la lista y se inicializa de acuerdo al "op"
- Estas copias son actualizadas localmente por los hilos
- Al final del bloque de construcción, las copias locales se combinan de acuerdo al "op" a un solo valor y se almacena en la variable COMPARTIDA original

Ejemplo de reducción

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

Una copia local de *sum* para cada hilo

Todas las copias locales de *sum* se suman y se almacenan en una variable “global”

C/C++ Operaciones de reducción

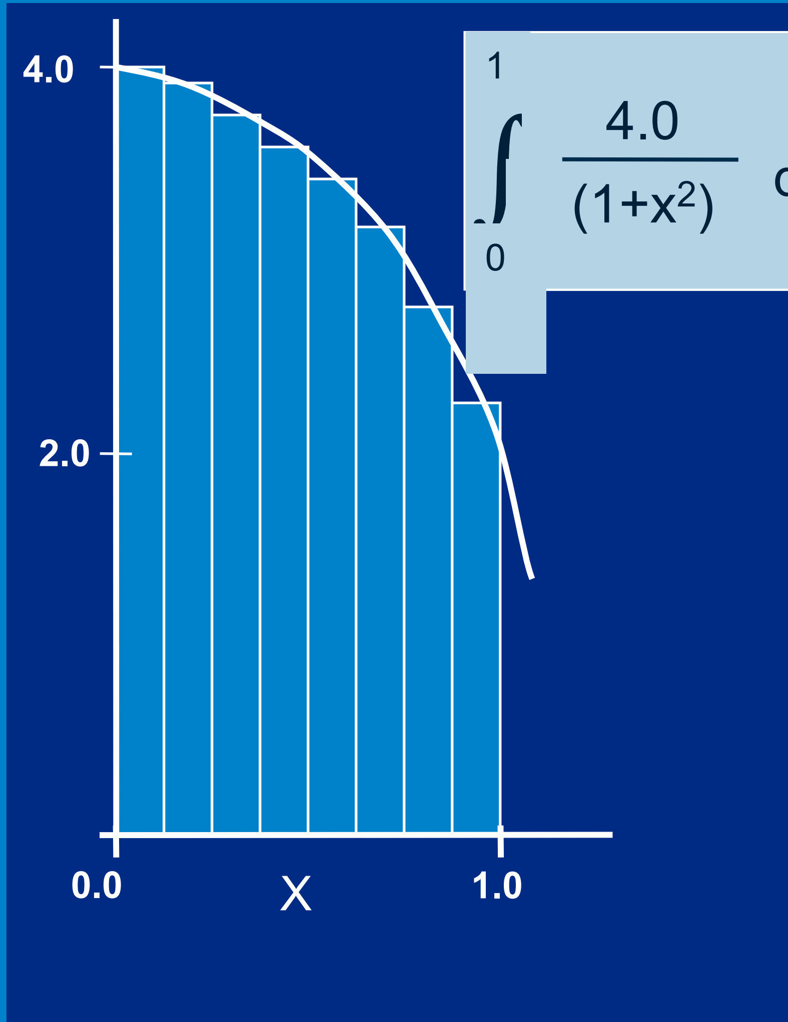
Un rango de operadores asociativos y conmutativos pueden usarse con la reducción

Los valores iniciales son aquellos que tienen sentido

Operador	Valor Inicial
+	0
*	1
-	0
^	0

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

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);
}
```

Actividad 2 - Calculando 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 se pueden compartir?

¿Qué variables deben ser privadas?

¿Qué variables deberían considerarse para reducción?

Asignando Iteraciones

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

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

- Los hilos timan un fragmento (chunk) de iteraciones
- Cuando terminan las iteraciones, el hilo solicita el siguiente fragmento

Asignando Iteraciones

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

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

Qué planificación utilizar

Cláusula Schedule	Cuando utilizar
STATIC	Predecible y trabajo similar por iteración
DYNAMIC	Impredecible, trabajo altamente variable por iteración
GUIDED	Caso especial de dinámico para reducir la sobrecarga de planificación

Ejemplo de la cláusula Schedule

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

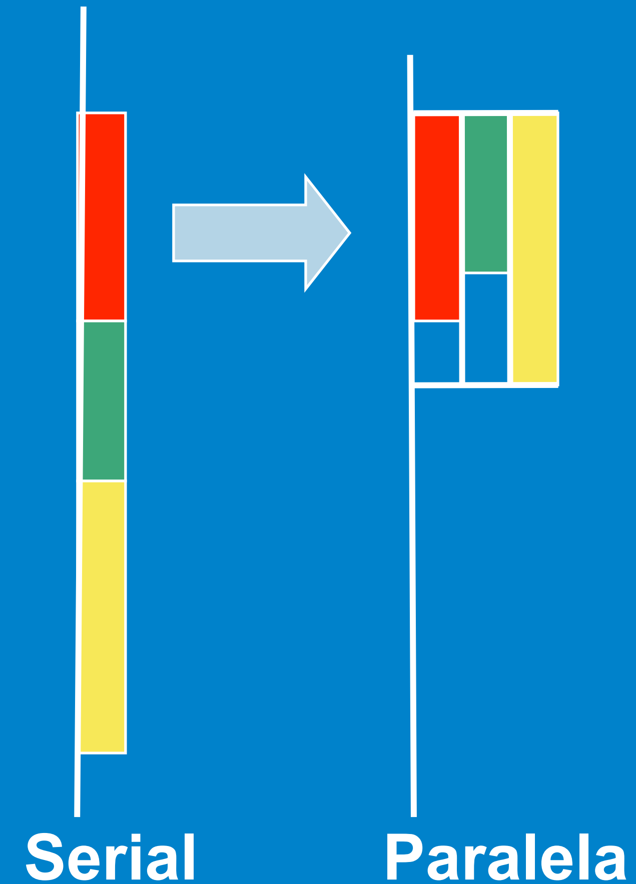
Las iteraciones se dividen en pedazos de 8

- Si start = 3, el primer pedazo es `i`={3,5,7,9,11,13,15,17}

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();
}
```



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

Varios bloques de construcción de OpenMP* tienen barreras implícitas

- `parallel`
- `for`
- `single`

Barreras innecesarias deterioran el rendimiento

- Esperar hilos implica que no se trabaja!

Suprime barreras implícitas cuando sea seguro con la cláusula **`nowait`**.

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);
	printf("Processed A into B\n");
#pragma omp barrier
	DoSomeWork(B,C);
	printf("Processed B into C\n");
}
```

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);
   }
```

API de OpenMP*

Obtener el número de hilo dentro de un equipo

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

Obtener el número de hilos en un equipo

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

Usualmente no se requiere para códigos de OpenMP

- Tiene usos específicos (debugging)
- Hay que incluir archivo de cabecera

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

Programación con OpenMP
¿Qué se cubrió?

OpenMP* es:

- Una aproximación simple a la programación paralela para computadoras con memoria compartida

Exploramos OpenMP para saber como:

- Hacer regiones de código en paralelo (**`omp parallel`**)
- Dividir el trabajo (**`omp for`**)
- Categorizar variables (**`omp private`**….)
- Sincronización (**`omp critical`**…)

Conceptos avanzados

Mas sobre OpenMP*

Bloques de construcción para el ambiente de datos

- **FIRSTPRIVATE**
- **LASTPRIVATE**
- **THREADPRIVATE**

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

Problemas de rendimiento

Los hilos ociosos no hacen trabajo útil

Divide el trabajo entre hilos lo más equitativamente posible

- Los hilos deben terminar trabajos paralelos al mismo tiempo

La sincronización puede ser necesaria

- Minimiza el tiempo de espera de recursos protegidos

Cargas de trabajo no balanceadas

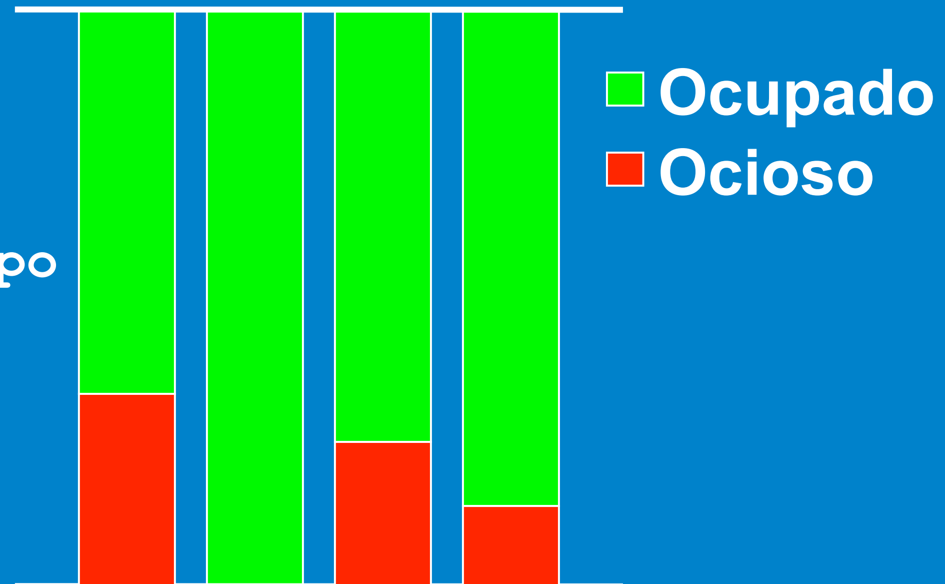
Cargas de trabajo desigual produce hilos ociosos y desperdicio de tiempo.

```
#pragma omp parallel
{

#pragma omp for
  for( ; ; ){

  }

}
```

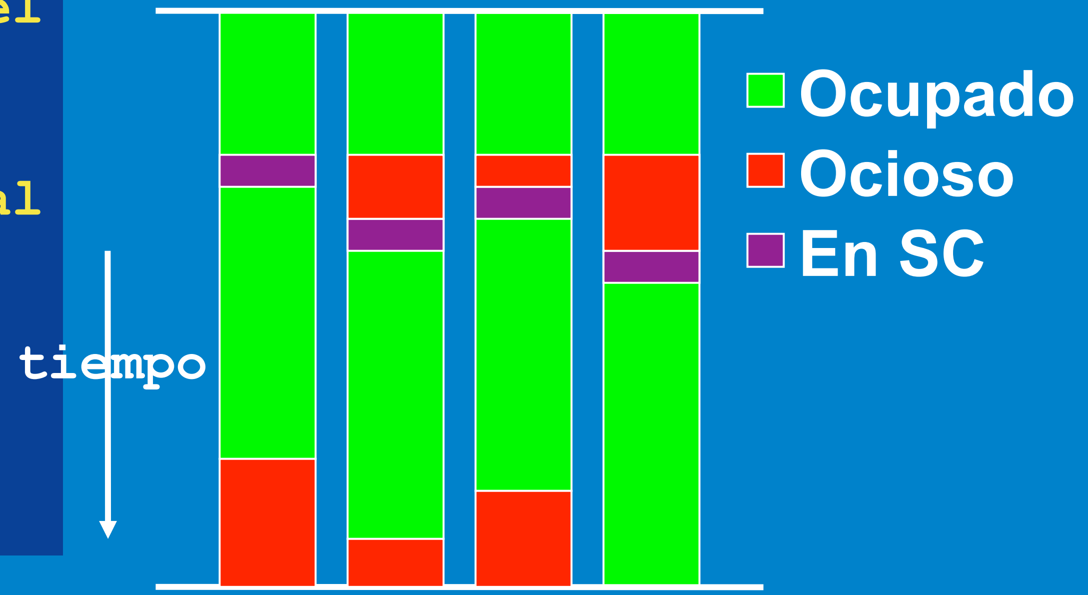


Sincronización

Tiempo perdido por locks

```
#pragma omp parallel
{

#pragma omp critical
  {
   ...
  }
  ...
}
```



Afinando el rendimiento

Los profilers usan muestreo para proveer datos sobre el rendimiento.

Los profilers tradicionales están limitados para usarse con códigos de OpenMP*:

- Miden tiempo del CPU, no tiempo real
- No reportan contención de objetos de sincronización
- No pueden reportar carga de trabajo desbalanceada
- Muchos de ellos no tienen todo el soporte de OpenMP

Los programadores necesitan profilers específicamente diseñadas para OpenMP.

Planificación estática: Hacerlo por uno mismo

Debe conocerte:

- Número de hilos (Nthrds)
- Cada identificador ID de cada hilo (id)

Calcular iteraciones (start y end):

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

¿Preguntas?

Material tomado de la Intel Software College