Introducción a la Programación de Memoria Compartida
.. con OpenMP®

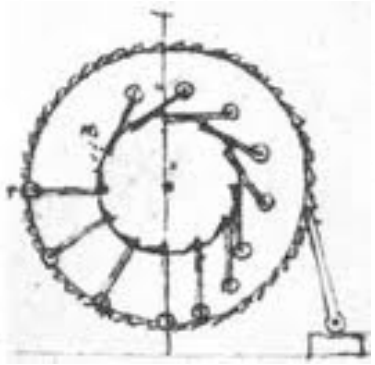
Carlos Jaime BARRIOS HERNANDEZ, PhD.

Escuela de Ingeniería de Sistemas e Informática

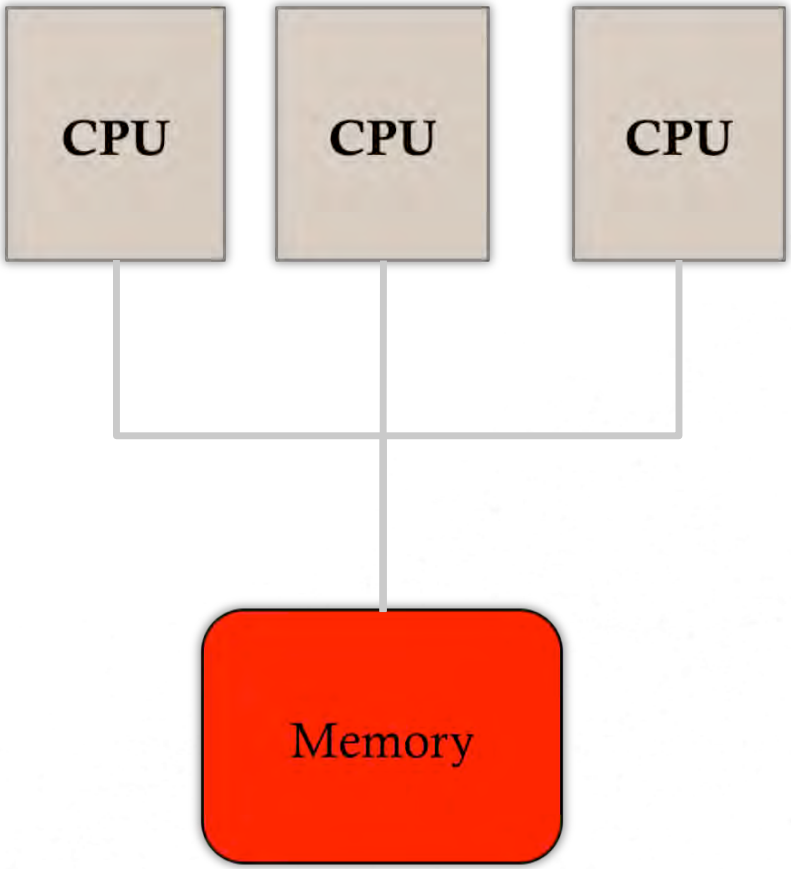
Universidad Industrial de Santander

Las herramientas antes que el diseño

- OpenMP® como herramienta de aprendizaje para entender el modelo de programación de memoria compartida.
- A partir de la experiencia técnica pensar en el modelo teórico.



El Modelo de Memoria Compartida



- Los Procesadores Interactúan con otro mediante variables compartida.
- OpenMP es un estándar para programación de memoria compartida

OpenMP®

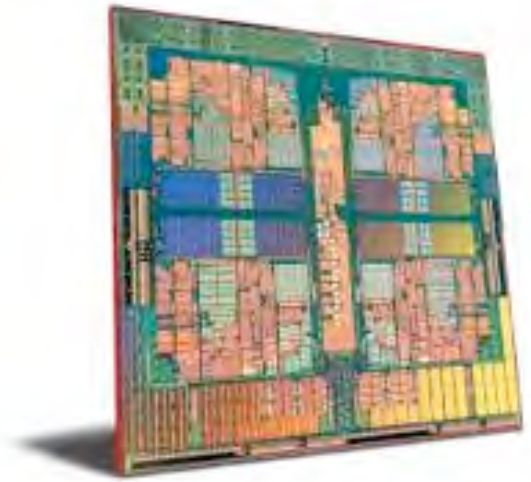
- OpenMP es una especificación para implementaciones portables de paralelismo en FORTRAN y C/C++
- Las especificación contiene provee directivos de compilador para programación de multihilos, variables de ambiente y rutinas de biblioteca que controlan paralelismo
  - Nivel de Cores
  - Nivel de Procesadores
- Soporta el modelo de paralelismo de datos
- Paralelismo Incremental
- Combina código serial y paralelo en un solo código fuente.

The OpenMP® API specification for parallel programming

www.openmp.org

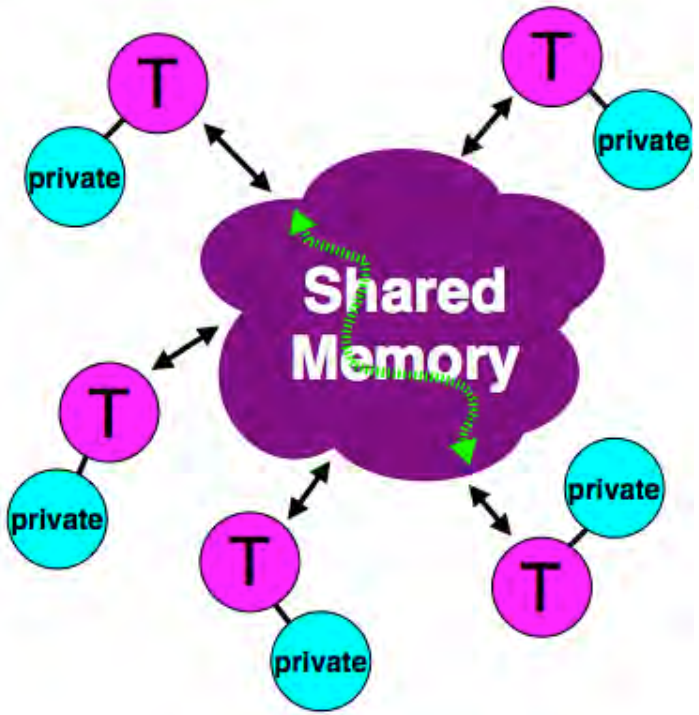
OpenMP es ideal para Arquitecturas Mu

- Modelo de Hilos y de Memoria mapeado naturalmente
- Ligero
- Robusto
- Disponible y usado para múltiples códigos sobre diferentes tecnologias disponibles (Intel, AMD)



Modelo de Memoria de OpenMP

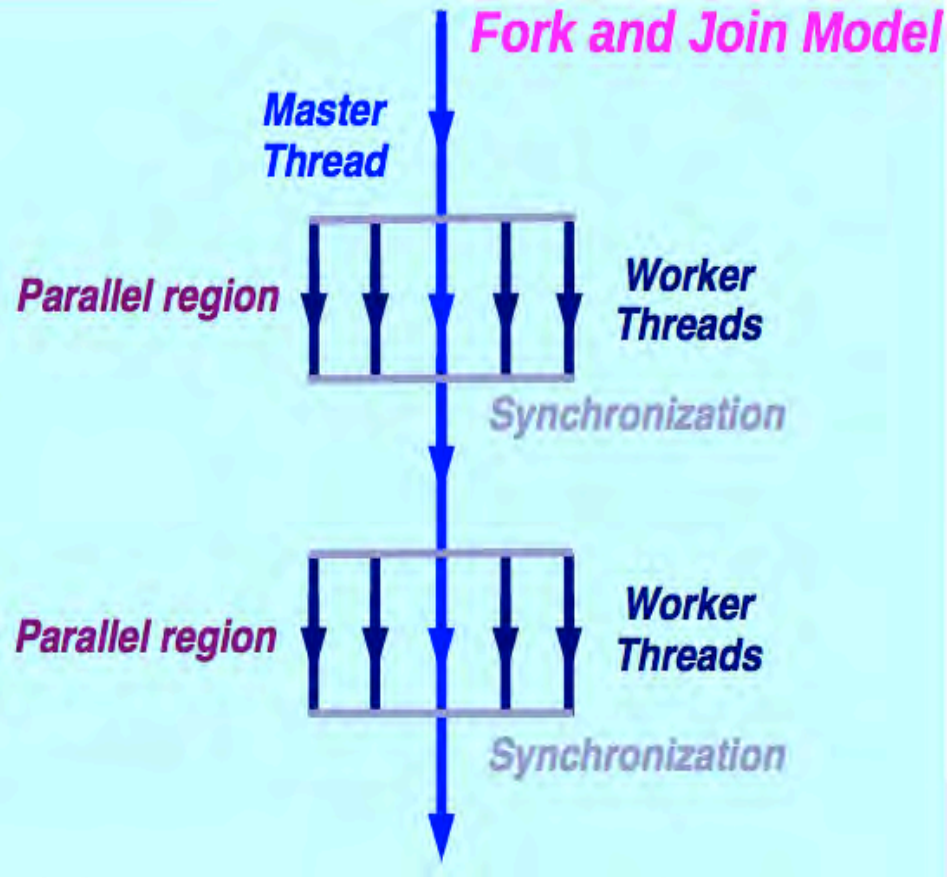
- Todos los hilos tienen acceso a la misma memoria global compartida
- Los datos pueden ser públicos o privados
- Datos privados pueden ser accedidos únicamente por su propio hilo
- Transferencia de Datos transparente al programador
- Sincronización es implícita



Tomado de An Overview of OpenMP – SUN Microsystems

Arquitectura de OpenMP

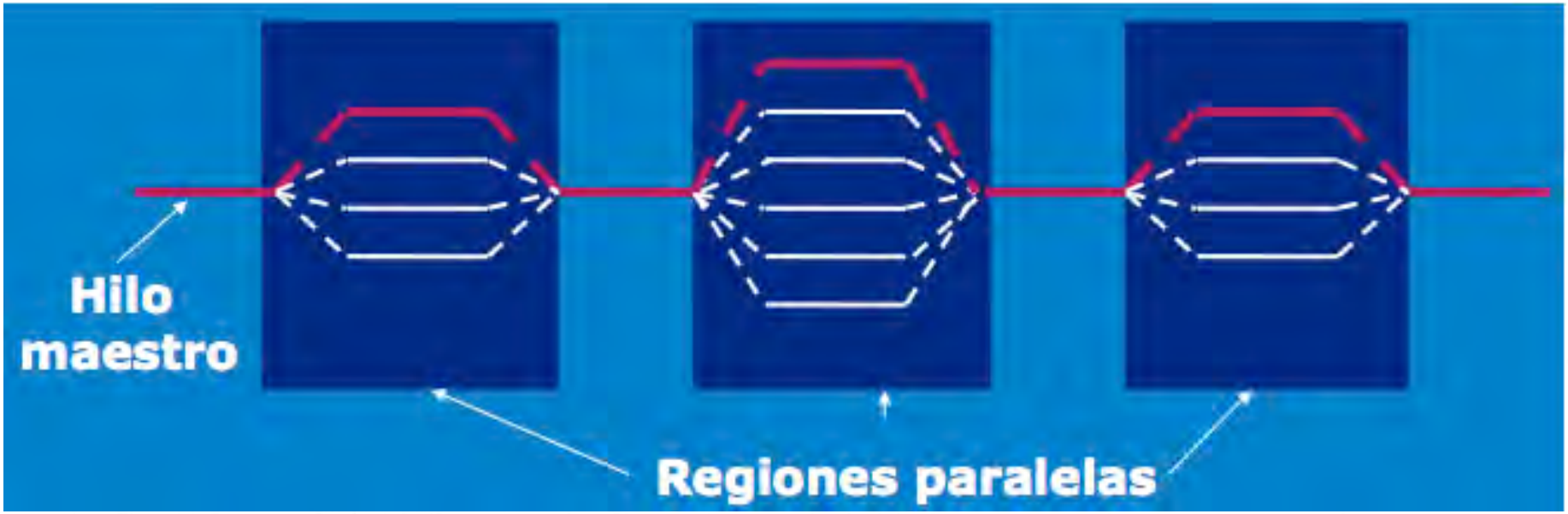
- Modelo Fork-Join
- Bloques de construcción para el trabajo en paralelo
- Bloques de construcción para el ambiente de datos
- Bloques de construcción para la sincronización
- API extensiva para afinar el control



Modelo de Ejecución - Tomado de An Overview of OpenMP – SUN Microsystems

Modelo de Programación

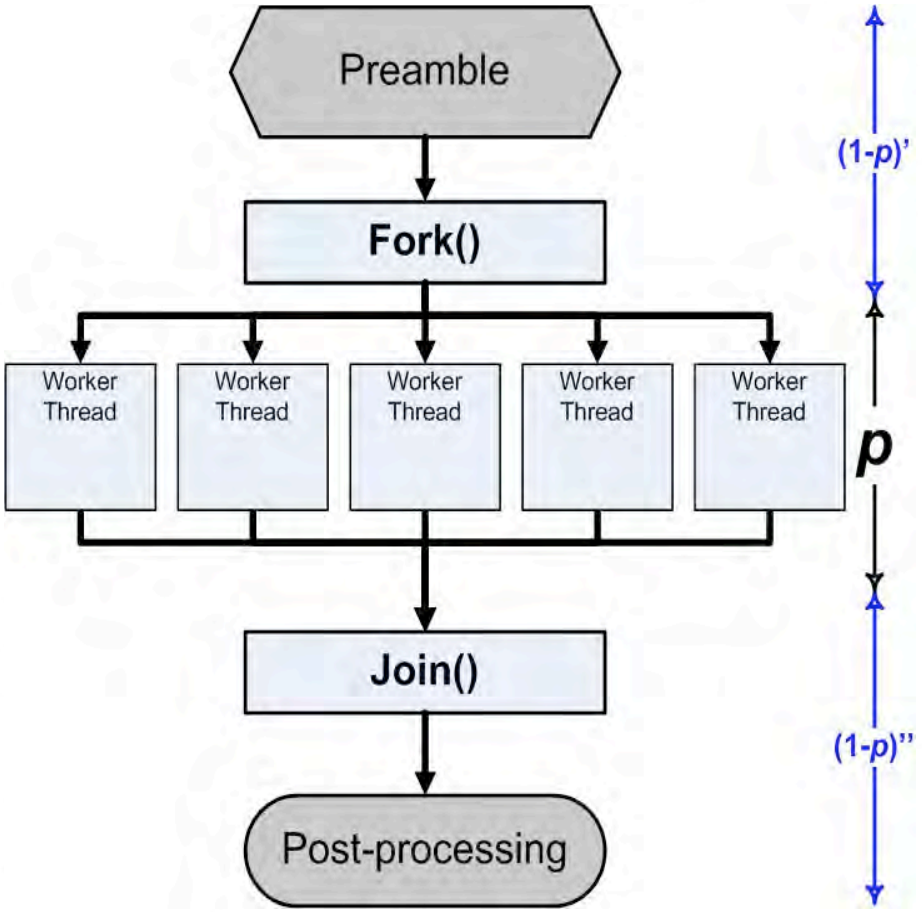
- Paralelismo Fork-Join:
  - El hilo maestro se divide en un conjunto de hilos como sea necesario
  - El paralelismo se añade incrementalmente: el programa secuencial se convierte en un programa paralelo



Tomado de Programación en OpenMP – Robinson Rivas SC-CAMP 2011

Modelo de Programación

- Inicialmente solamente el hilo maestro está activo
- El hilo maestro ejecuta el código secuencial
- Fork: El hilo maestro crea hilos adicionales para ejecutar el código paralelo
- Join: Al finalizar de código paralelo, los hilos creados mueren o se suspenden



Parallel Scalability Isn't Child's Play, Part 2: Amdahl's Law vs. Gunther's Law

Pragmas

- Una pragma es un directivo al compilador.
- La sintaxis es

#pragma omp <el resto de la pragma>

Ejemplo:

#pragma omp parallel for

es una directriz que dice al compilador que trate a paralelizar el bucle for

La Pragma for Paralelo

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

- El compilador tiene que verificar que cuando se ejecuta, habrá disponible la información necesaria para llevar a cabo las iteraciones

Tomado del Curso de Open MP de Doroty Bollman http://pegasus.uprm.edu/bollman/

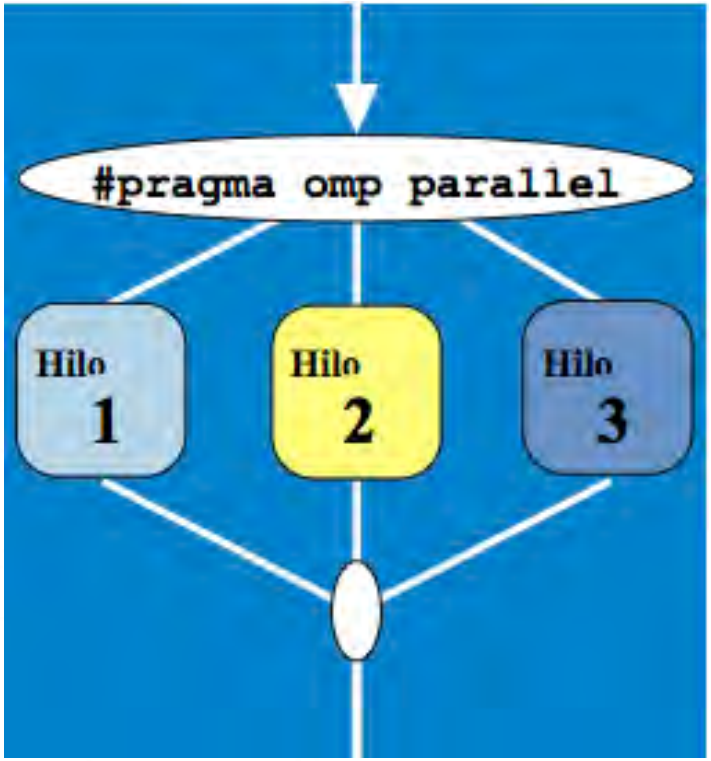
La sintaxis de la pragma for paralelo

$$\text{for}(\text{indice} = \textit{primero};\text{indice} \geq \left\{\begin{array}{c} < \\ <= \\ > \\ >= \end{array}\right\} \textit{ultimo};\left\{\begin{array}{c} \text{indice}++ \\ ++\text{indice} \\ \text{indice}-- \\ --\text{indice} \\ \text{indice} += \textit{inc} \\ \text{indice} -= \textit{inc} \\ \text{indice} = \text{indice} + \textit{inc} \\ \text{indice} = \textit{inc} + \text{indice} \\ \text{index} = \text{indice} - \textit{inc} \end{array}\right\}$$

Tomado del Curso de Open MP de Doroty Bollman http://pegasus.uprm.edu/bollman/

Regiones Paralelas

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



Tomado de Programación en OpenMP – Robinson Rivas SC-CAMP 2011

Variables Compartidas y Variables Privadas

- Una variable compartida tiene la misma dirección en el contexto de ejecución de cada hilo.
- Una variable privada tiene una dirección distinta en el contexto de ejecución de cada hilo.
- Un hilo no puede accesar las variables privadas de otro hilo.

Un par de Programas Simples

```
#include <studio.h>
#include “omp.h”
int main
{
#pragma omp parallel
   {
      printf(“E pur si mouve\n);
   }
}
```

Diferencia entre Datos Locales y Compartidos

Un ejemplo

```
#include <studio.h>
#include “omp.h”
int main
{
int = 5 // Variable compartida
#pragma omp parallel
    {
int c; // Variable local o privada para cada hilo
c = omp_get_thread_num();
printf(“c=%d, i= %d\n”,c,i)
    }
}
```

Salida:

```
c = 0, i =5
c = 2, i = 5
c = 1, i = 5
```

OMP_NUM_THREADS

- Establece el numero de hilos usando una variable de ambiente
- No hay un estándar por defecto en esta variable
  - # de hilos = # de procesadores = # Cores
  - Los compiladores INTEL usan esto por defecto

```
set OMP_NUM_THREADS=4
```

Función omp_set_num_threads

- Se usa para asignar el número de hilos a ser activos en secciones paralelas del código
- Se puede llamar en varios puntos del programa

```
void omp_set_num_threads (int t)
```

omp_get_thread_num()

- Todo hilo tiene una identificación que es el número del hilo
- omp_get_thread_num()
  devuelve el número del hilo

Función omp_get_num_procs

- Devuelve el número de procesadores fisicos que están dispondible para el uso del programa paralelo

```
int omp_get_num_procs (void)
```

Hello World

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

 int main (int argc, char *argv[]) {
   int p,th_id;
   p=omp_get_num_procs();
   omp_set_num_threads(p);
  #pragma omp parallel private(th_id);
    {
     th_id = omp_get_thread_num();
     printf("Hello World from thread %d\n", th_id);
     }
  return 0;
 }
```

Para compilar y ejecutar

- Compilar: cc –openmp –o hello.c hello
- Ejecutar: Hay que especificar el número de hilos

  Fuera del programa con

  setenv OMP_NUM_THREADS = *número de hilos*

  Dentro del programa con

  omp_set_num_threads(*número de hilos*)

Otro Ejemplo

- For-Loop con iteraciones independientes

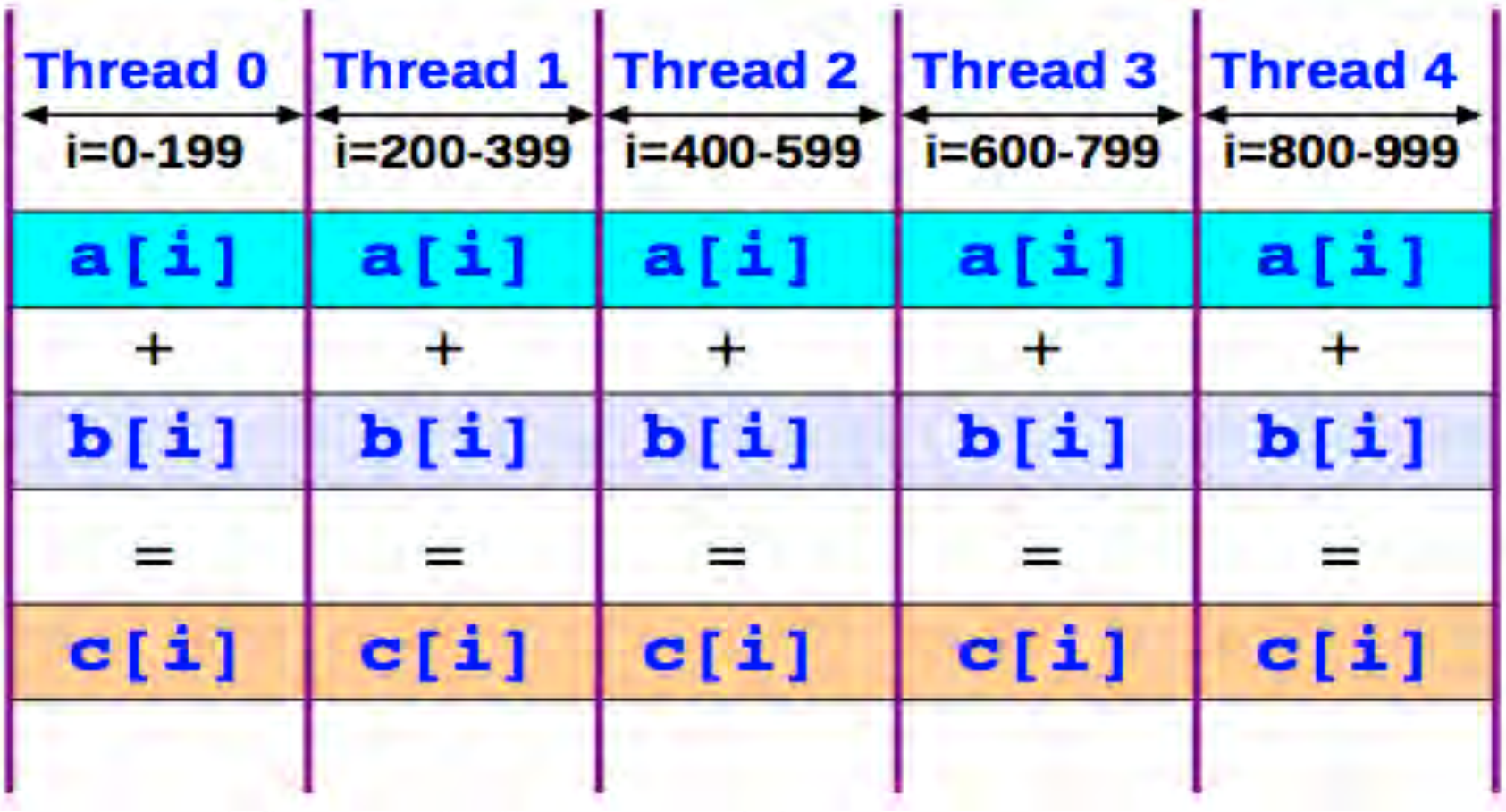
```
for (int i=0; i<n; i++)
  c[i] = a[i] + b[i];
```

- For-Loop paralelizado usando un pragma de OpenMP

```
#pragma omp parallel for
for (int i=0; i<n; i++)
    c[i] = a[i] + b[i];
```

cc –fopenmp source.c – o source
Setenv OMP_NUM_THREADS 5
source.out

Ejemplo de Ejecución Paralela



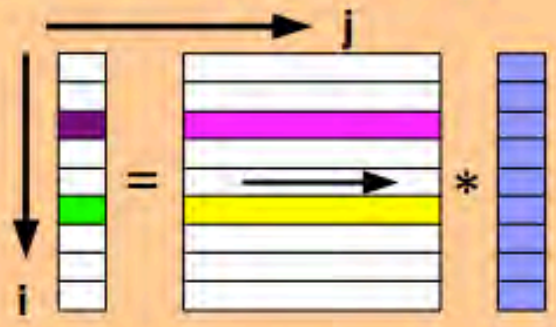
Ejemplo de Ejecución - Tomado de An Overview of OpenMP – SUN Microsystems

Componentes de OpenMP® 2.5

Directivas	Ambiente de Ejecución	Variables de Ambiente
Región Paralela	Numero de Hilos	Numero de Hilos
Trabajo Compartido (Worksharing)	ID del Hilo	Tipo de Calendarizador
Atributos de Datos Compartidos : private, firstprivate, lastprivate, shared, reduction	Ajuste Dinamico de Hilos	Ajuste Dinamico de Hilos
Orfandad (Oprhaning)	Paralelismo Anidado	Paralelismo Anidado
	Limitador del tiempo de reloj – Temporizador	
	Bloqueo	

Ejemplo: Matrix -Vector

```
#pragma omp parallel for default(none) \
                  private(i,j,sum) shared(m,n,a,b,c)
for (i=0; i<m; i++)
{
   sum = 0.0;
   for (j=0; j<n; j++)
     sum += b[i][j]*c[j];
   a[i] = sum;
}
```



TID = 0

```
for (i=0,1,2,3,4)
 i = 0
sum = Σ  b[i=0][j]*c[j]
   a[0] = sum
 i = 1
sum = Σ  b[i=1][j]*c[j]
   a[1] = sum
```

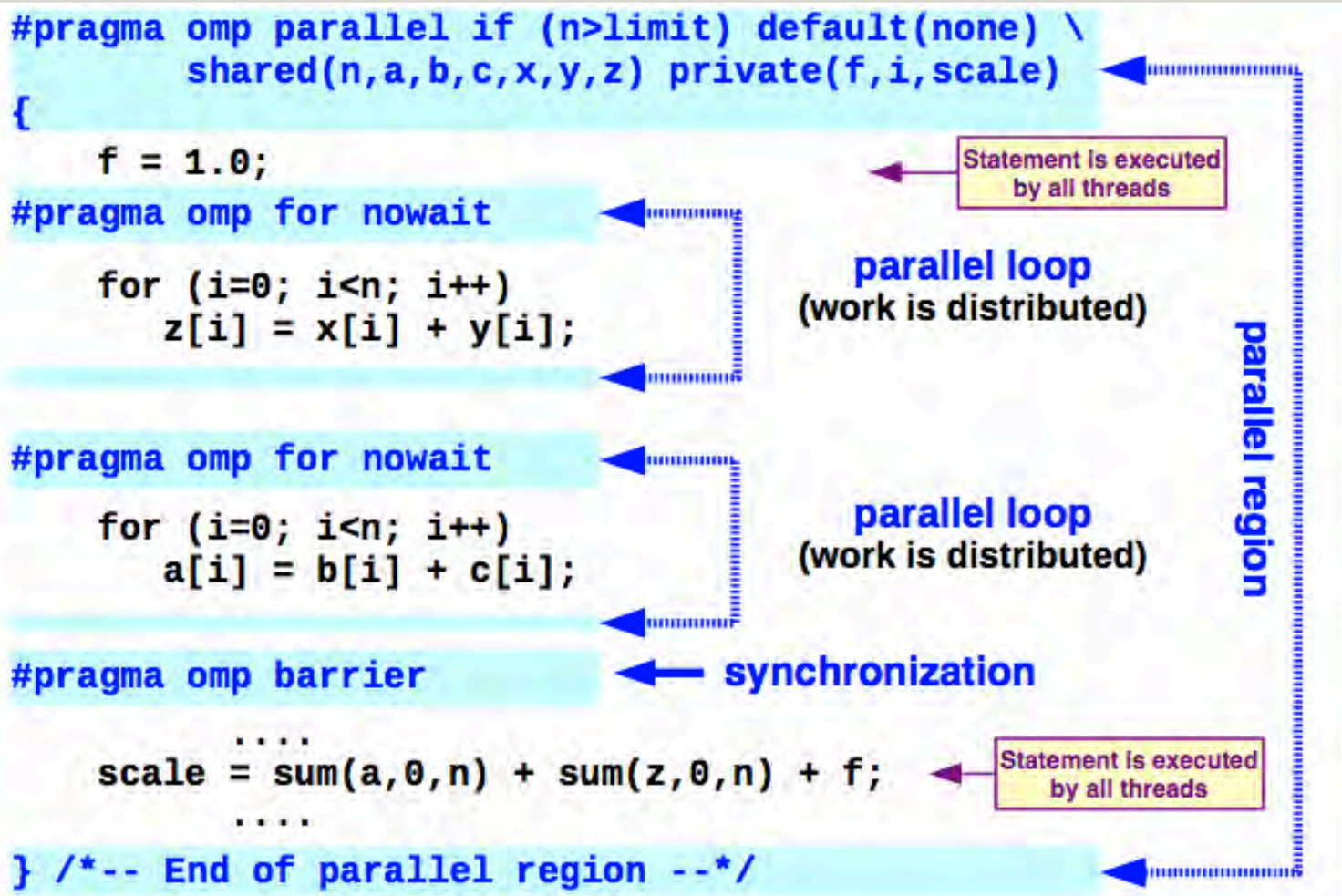
TID = 1

```
for (i=5,6,7,8,9)
 i = 5
sum = Σ  b[i=5][j]*c[j]
   a[5] = sum
 i = 6
sum = Σ  b[i=6][j]*c[j]
   a[6] = sum
```

... etc ...

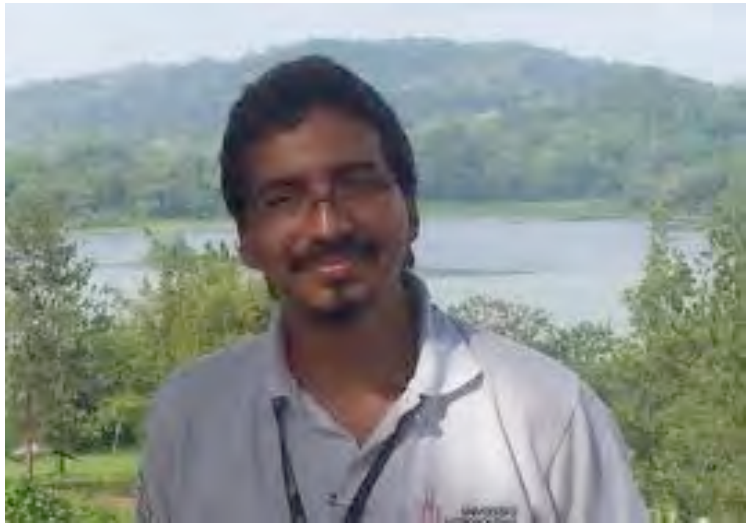
Tomado de An Overview of OpenMP – SUN Microsystems

Un Ejemplo mas Elaborado…



Tomado de An Overview of OpenMP – SUN Microsystems

No hay que pasar al sucio lo que esta ya en limpio
No hay que pasar al sucio lo que esta ya en limpio
No hay que pasar al sucio lo que esta ya en limpio
No hay que pasar al sucio lo que esta ya en limpio
No hay que pasar al sucio lo que esta ya en limpio
No hay que pasar al sucio lo que esta ya en limpio
No hay que pasar al sucio lo que esta ya en limpio
No hay que pasar al sucio lo que esta ya en limpio
No hay que pasar al sucio lo que esta ya en limpio
No hay que pasar al sucio lo que esta ya en limpio
No hay que pasar al sucio lo que esta ya en limpio



Robinson Rivas, MSc.
Profesor Universidad Central de Venezuela (UCV)
Instructor – Organizador SC-CAMP
Líder del Grupo de Paralelismo y Sistemas Distribuidos UCV
Microsoft Grantt Instructor (Sistemas Distribuidos)

Bloques de construcción de trabajo en paralelo

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

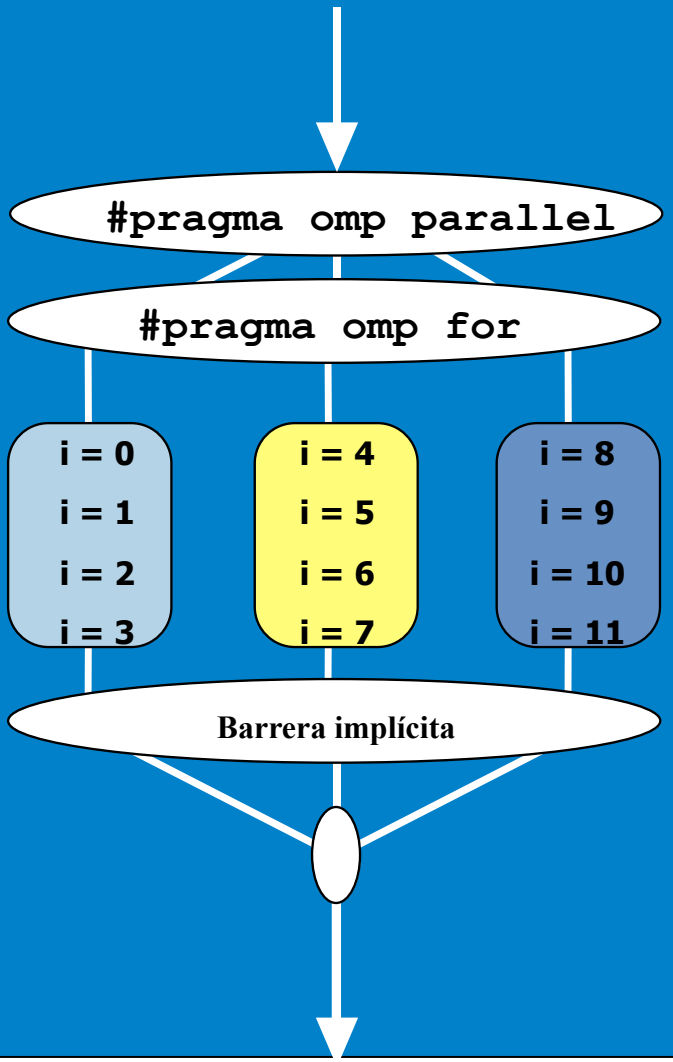
Divide las iteraciones del ciclo en hilos

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

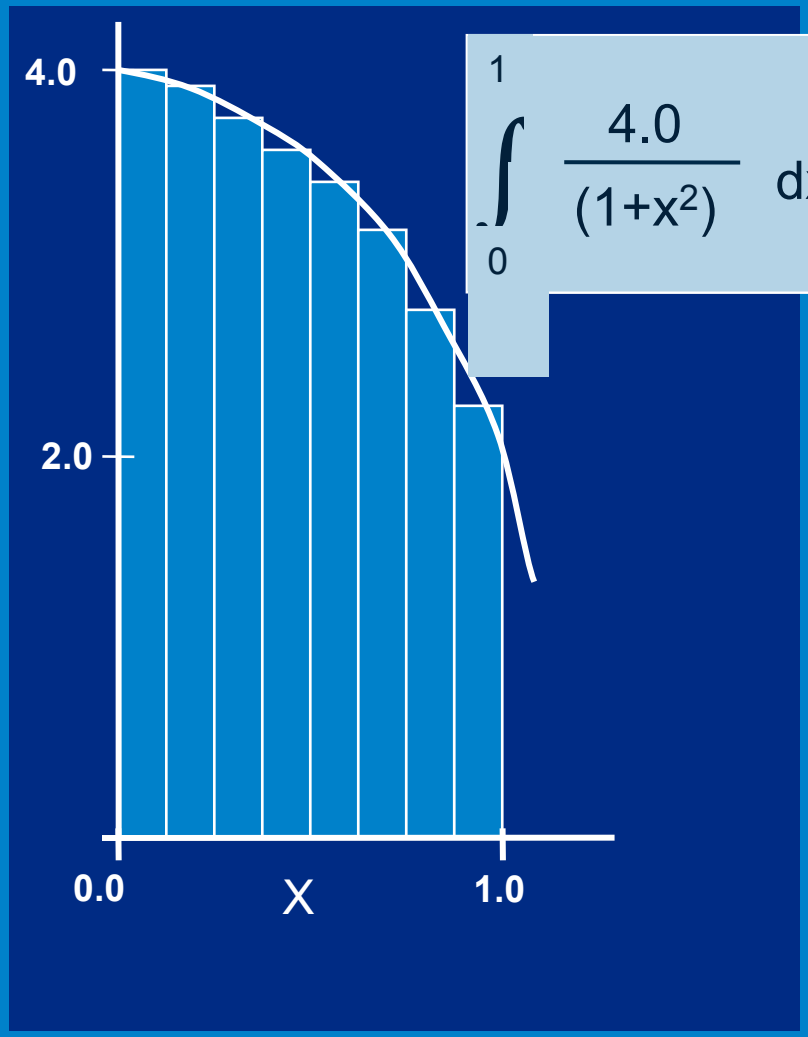
Un rango de operadores asociativos y conmutativos pueden usarse con la reducción

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++;
    }
```

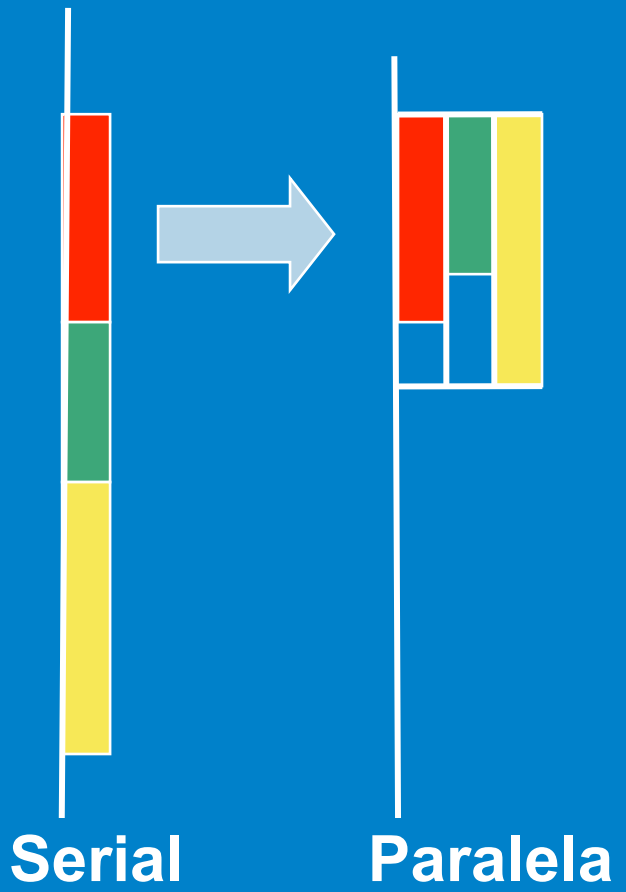
Las iteraciones se dividen en pedazos de 8

- 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

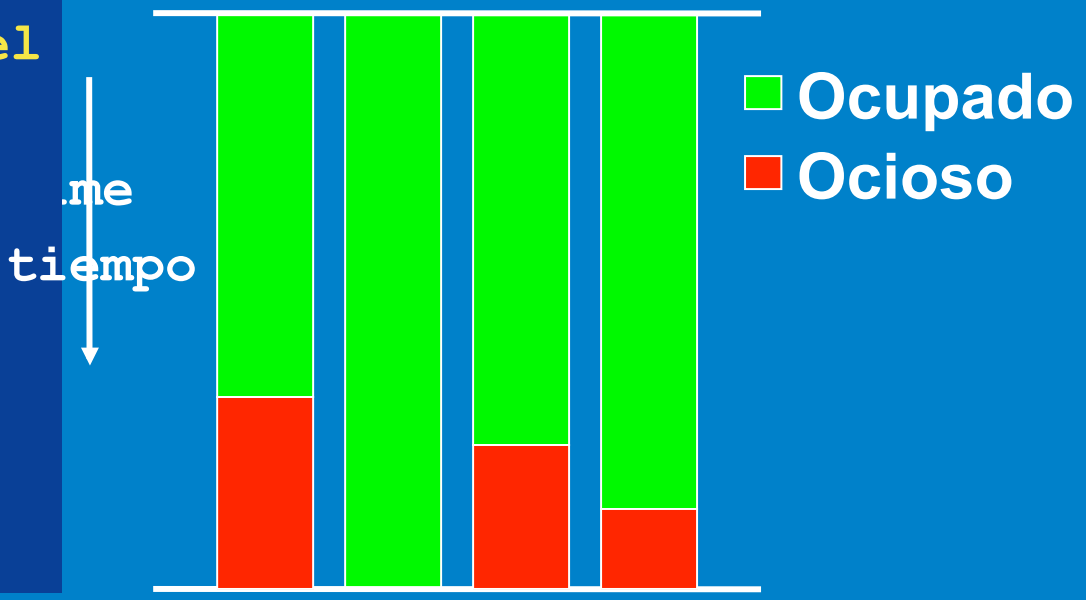
Cargas de trabajo desigual produce hilos ociosos y desperdicio de tiempo.

```
#pragma omp parallel
{

#pragma omp for
  for( ; ; ){

  }

}
```

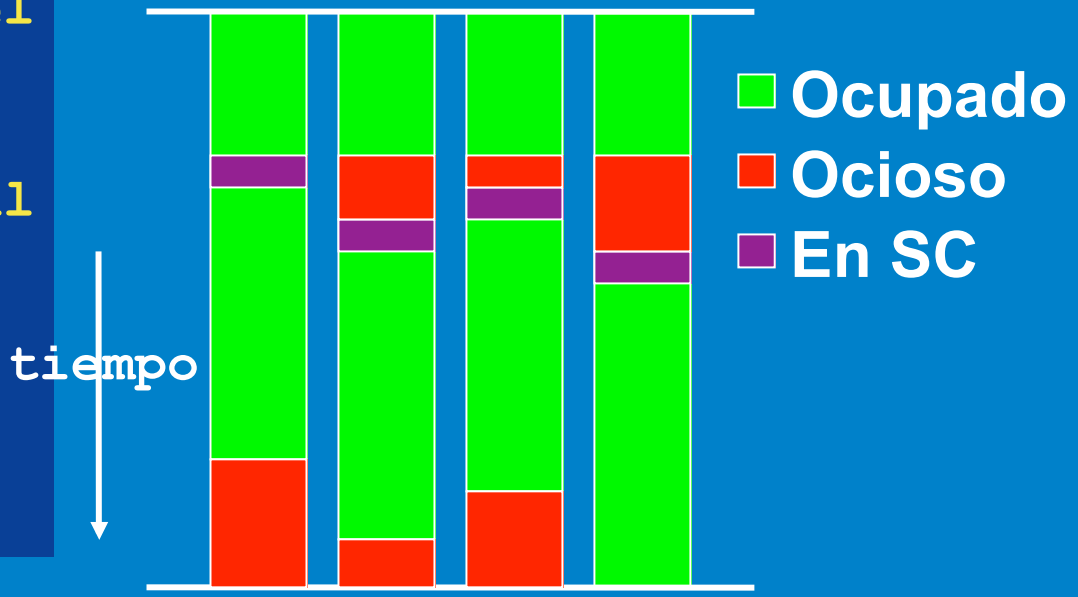


Sincronización

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];}
}
```

Evaluacion de Rendimiento en OpenMP

- Ejemplo: Medir Tiempos

```
double empezar,terminar;
empezar=omp_get_wtime( );
… algun código
terminar=omp_get_wtime();
printf("TIEMPO=%lf\n",empezar-terminar)
```

El resultado es en segundos.

Paralelismo Funcional

- OpenMP nos permite asignar hilos distintos a partes distintas del código (paralelismo funcional)

$$y_i^{(1)} = \frac{y_{i+1} - y_{i-1}}{2d} - \frac{2\,d^3}{2\,d\,3!} y_i^{(3)} + \ldots;$$

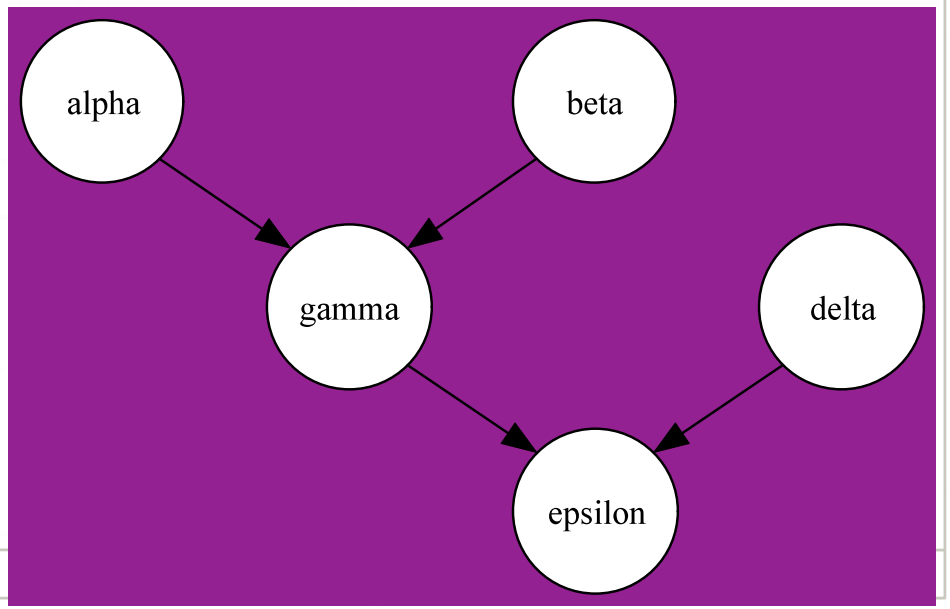
$$y_i^{(1)} = \frac{y_{i+1} - y_i}{d} - \frac{d^2}{2!\,d} y_i^{(2)} + \ldots;$$

$$y_i^{(1)} = \frac{y_i - y_{i-1}}{d} + \frac{d^2}{2!\,d} y_i^{(2)} + \ldots$$

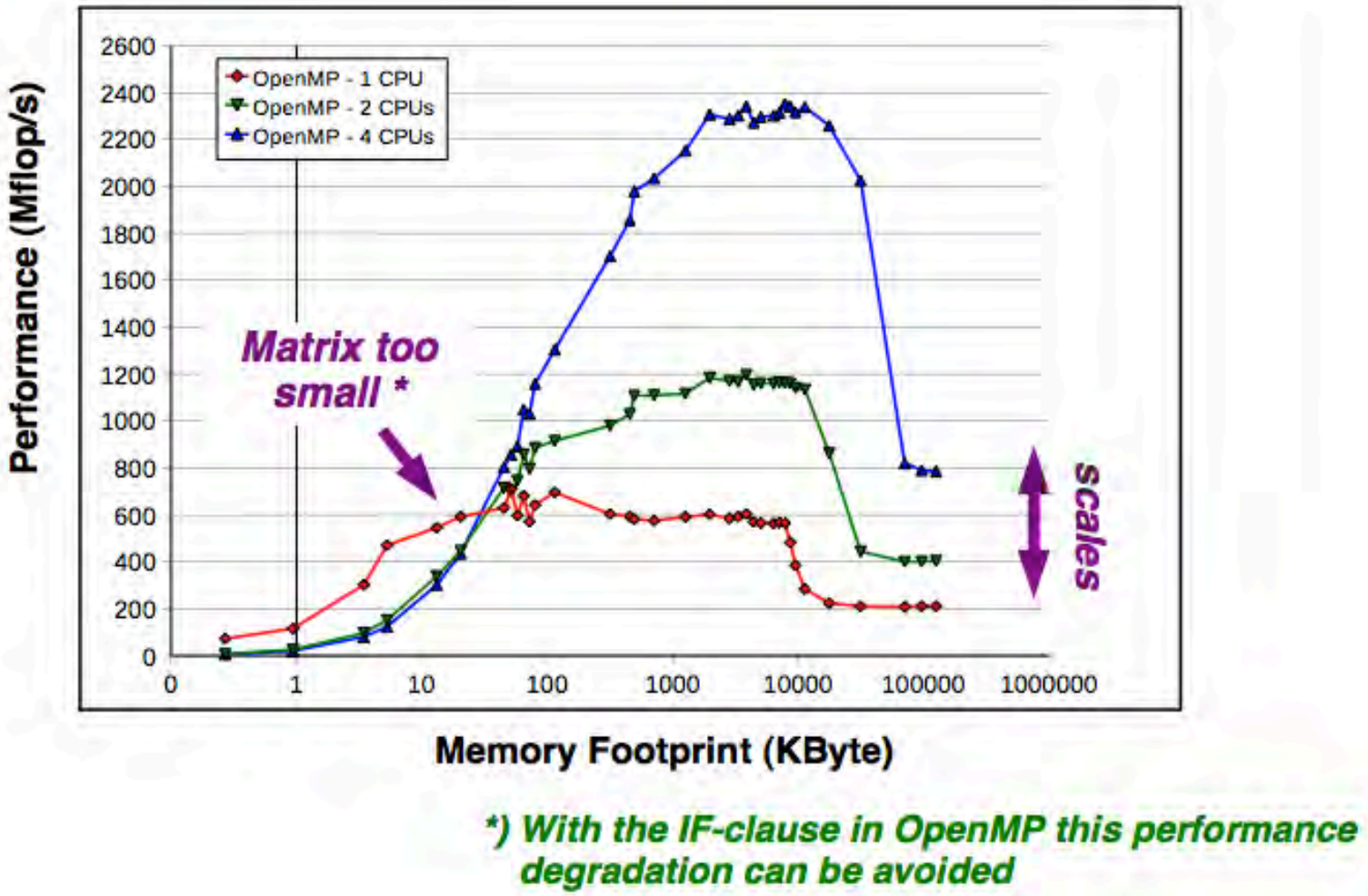
Ejemplo de Paralelismo Funcional

```
v = alpha();
w = beta();
x = gamma(v, w);
y = delta();
printf ("%6.2f\n", epsilon(x,y));
```

Se puede ejecutar alpha, beta, and delta en paralelo.



Rendimiento de OpenMP



Tomado de An Overview of OpenMP – SUN Microsystems

Conclusiones

- OpenMP permite explotar no solo paralelismo de datos, sino también de tareas.
- OpenMP debe considerarse cuando:
  - Se vayan a programar cores y es no es fácilmente la determinación la granularidad del problema, pero si el uso de memoria compartida.
- La decisión de cuantos hilos usar o no y de cómo sincronizarlos es tomada por el compilador y no necesariamente por el programador.

Lecturas Recomendadas

- Parallel Scalability Isn't Child's Play, Mark B. Friedman
  - Parte 1: http://blogs.msdn.com/b/ddperf/archive/2009/03/16/parallel-scalability-isn-t-child-s-play.aspx
  - Parte 2: http://blogs.msdn.com/b/ddperf/archive/2009/04/29/parallel-scalability-isn-t-child-s-play-part-2-amdahl-s-law-vs-gunther-s-law.aspx
- An Introduction to OpenMP, Robinson Rivas, SC-CAMP 2011 http://www.sc-camp.org/_pdf/OpenMP%20-%20sccamp%202011.pdf
- OpenMP® Official Site: http://www.openmp.org
- OpenMP Tutorial at Lawrence Livermore National Laboratory: https://computing.llnl.gov/tutorials/openMP/