OpenMP

Cómputo de Alto Desempeño

¿Qué es OpenMP?

- API para programación multiproceso en computadoras con arquitectura de memoria compartida, en múltiples plataformas.
- Formada por
  - directivas del compilador
  - bibliotecas de funciones
  - variables de ambiente,

No un lenguaje.

- Basado en el modelo de hilos.

Open MP

Definido por un grupo de proveedores de hardware y de software AMD

(OpenMP **ARB 1997)**

- BCS - Barcelona Supercomputing Center
- CAPS-Entreprise
- Convey Computer
- Cray
- Fujitsu
- HP
- IBM
- Intel
- Microsoft
- NEC
- NVIDIA
- Oracle Corporation
- Signalogic
- The Portland Group, Inc.
- Texas Instruments

Mas información en

www.openmp.org

Algunos Compiladores

Compañía	Compilador	Información
GNU	gcc	Usado en Linux, Solaris, AIX, MacOSX, Windows OpenMP 3.1 y soportado desde GCC 4.7 –fopenmp
IBM	XL C/C++ / Fortran	Usado en AIX y Linux.
Oracle	C/C++ / Fortran	Solaris y Linux
Intel	C/C++ / Fortran (10.1)	Windows, Linux y MacOSX. /Qopenmp y -openmp

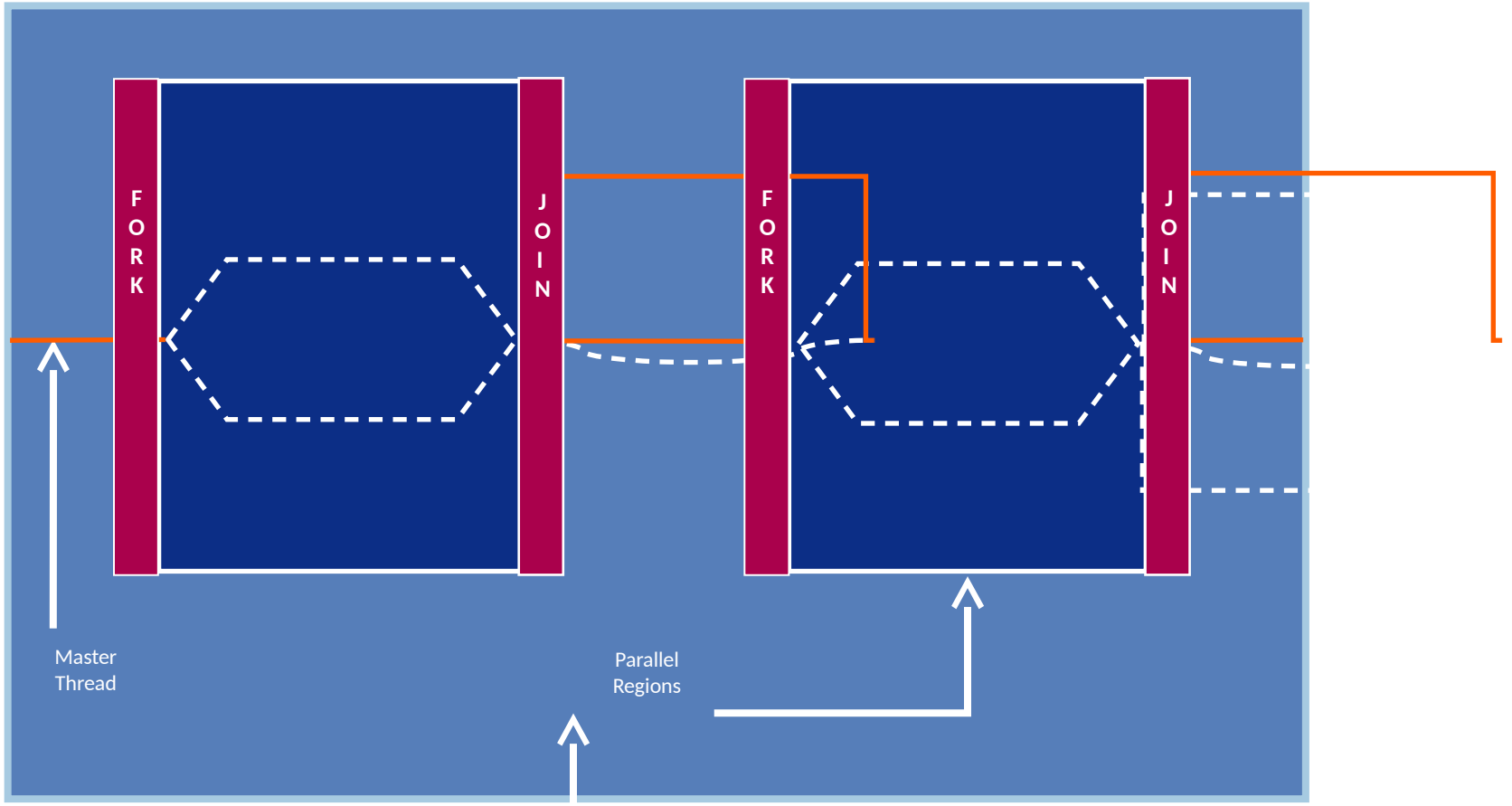
¿Qué es OpenMP?

- Modelo de programación paralelo.
- Paralelismo de memoria compartida.
- Extensiones para lenguajes de programación existentes (C,C++, Fortran)
- Combina código serial y paralelo en un solo archivo fuente.

Arquitectura de OpenMP

- Fork/join (Maestro esclavo)
- Trabajo Y Datos Compartido
- Sincronización

Fork / Join

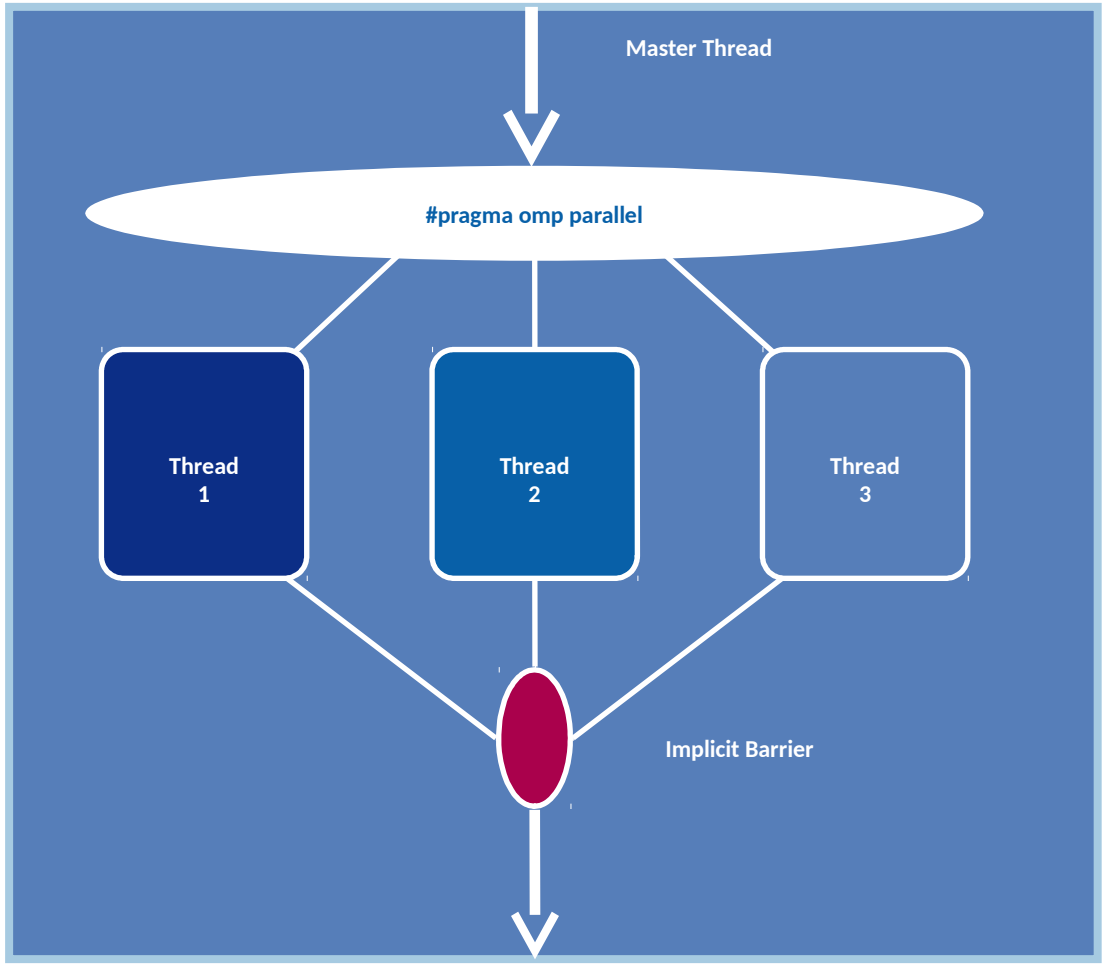


Sintaxis

- Directivas o pragmas
  - #pragma omp *construct [clause [clause]…]*
  - *Clausulas: Especifican atributos para compartir datos y calendarización*

Una pragma en C o C++ es un directivo al compilador.

Regiones paralelas



Regiones paralelas

- Los hilos son creados desde el pragma *parallel*.
- Los datos son compartidos entre los hilos.

C/C++ :

```
#pragma omp parallel
   {
      block
   }
```

¿Cuántos hilos?

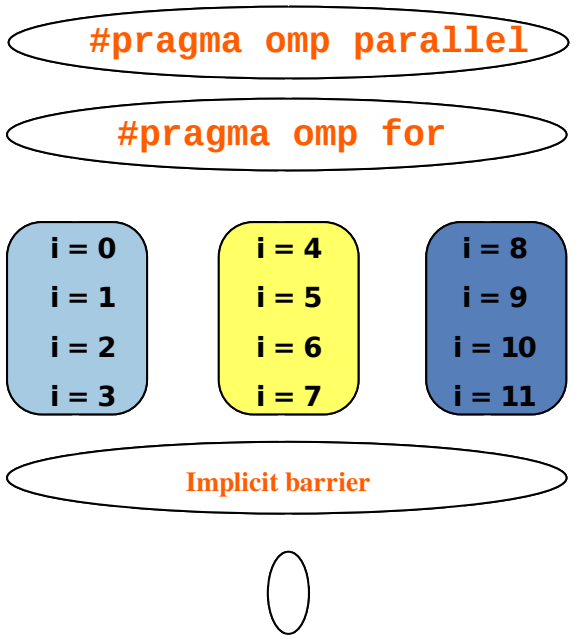
- Num. Hilos = Num. Procesadores o núcleos
- Intel lo usa de esta forma.
- Se definen más hilos con la variable de ambiente *OMP_NUM_THREADS*.

Work - Sharing

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

- Divide los ciclos de la iteración entre los hilos.
- Debe estar especificada en una región paralela
- Debe estar antes del ciclo.

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



Combinando Pragmas

- Estos dos 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();
    }
```

Algunas funciones de ambiente

omp.h

- void omp_set_num_threads(int nthreads)
- int omp_get_num_threads(void)
- int omp_get_max_threads(void)
- int omp_get_thread_num(void)
- int omp_get_num_procs(void)

¿Quién soy yo? ¿Cuántos somos?

- Cada hilo paralelo se identifica por un número. El 0 es el *hilo maestro*.

- Dos funciones de la bibliotreca omp.h:

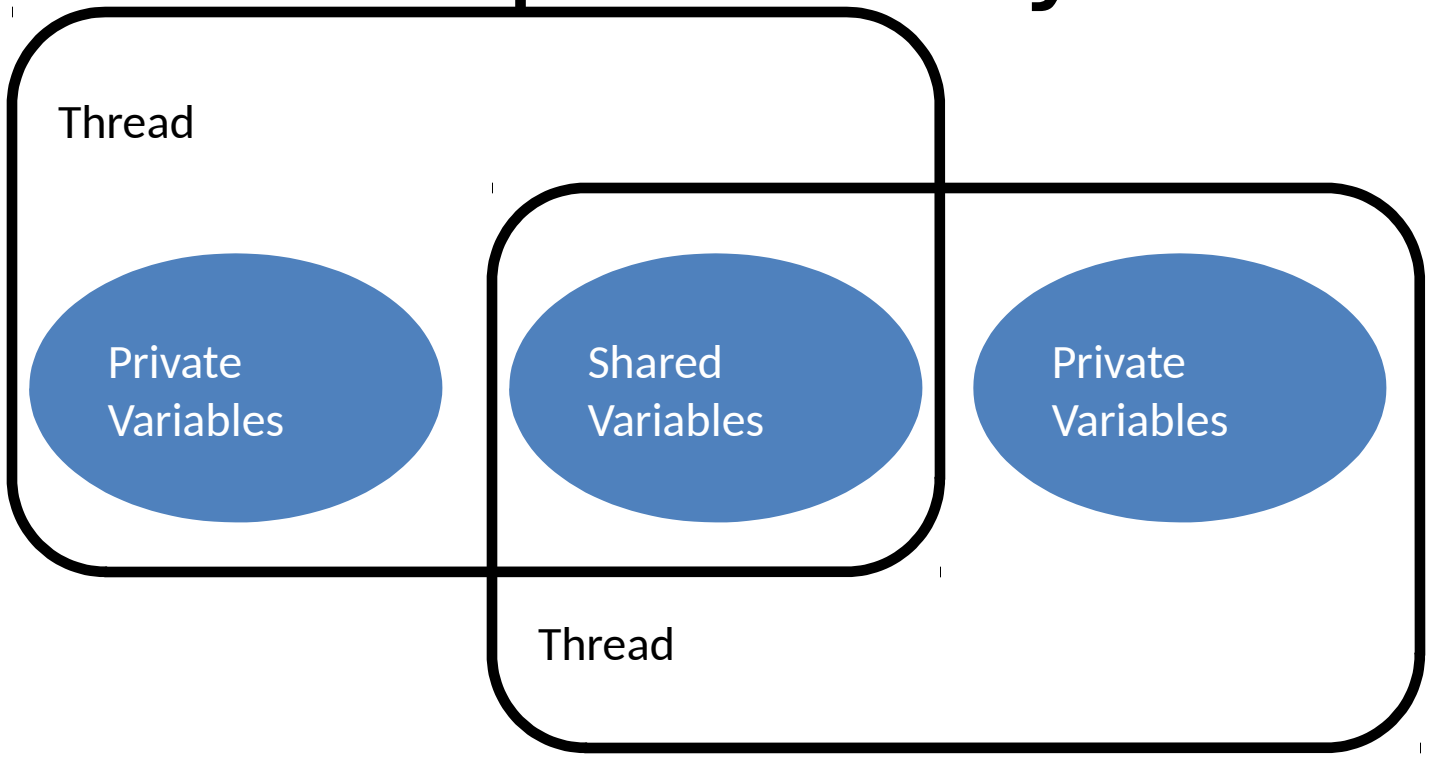
```
tid = omp_get_thread_num();
```

devuelve el identificador del *thread*.

```
nth = omp_get_num_threads();
```

devuelve el número de hilos generados.

Variables Compartidas y Privadas



Atributos de alcance

Clausulas shared y private

- **`shared(varname,…)`**
- **`private(varname,…)`**

Clausulas

- **`shared(X)`**

  Se declara la variable **`X`** como compartida por todos los *hilos*.

  Sólo existe una copia, y todos los *hilos* acceden y modifican dicha copia.

  - **`private(Y)`**

    Se declara la variable **`Y`** como privada en cada *hilo*. Se crean P copias, una por *hilo*(sin inicializar!).

    Se destruyen al finalizar la ejecución de los *hilos*.

Clausula private

- Las variables no se inicializan
- Cualquier valor externo a la región paralela se coloca como 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 problemas con private

- Se requiere realizar el producto entre dos vectores de dimensión n.

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

- ¿Cúal es el problema?

Región crítica

- Una región o sección crítica es una secuencia de instrucciones que no debe ser interrumpida por otros proceso
- Constructor critical

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

Constructor critical

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

// Sintaxis: #pragma omp critical [(lock_name)]
```

¿Cuál será el problema aquí?

Ejercicio- paralelizar

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

#define SIZE 10

int main()
{
   int i;
   int max;
   int a[SIZE];

   for (i = 0; i < SIZE; i++)
   {
      a[i] = rand();
      printf("%d\n", a[i]);
   }

   max = a[0];

      for (i = 1; i < SIZE; i++)
      {
         if (a[i] > max)
         {
max = a[i];
            }

      }

   printf("max = %d\n", max);
}
```

Asignando Iteraciones

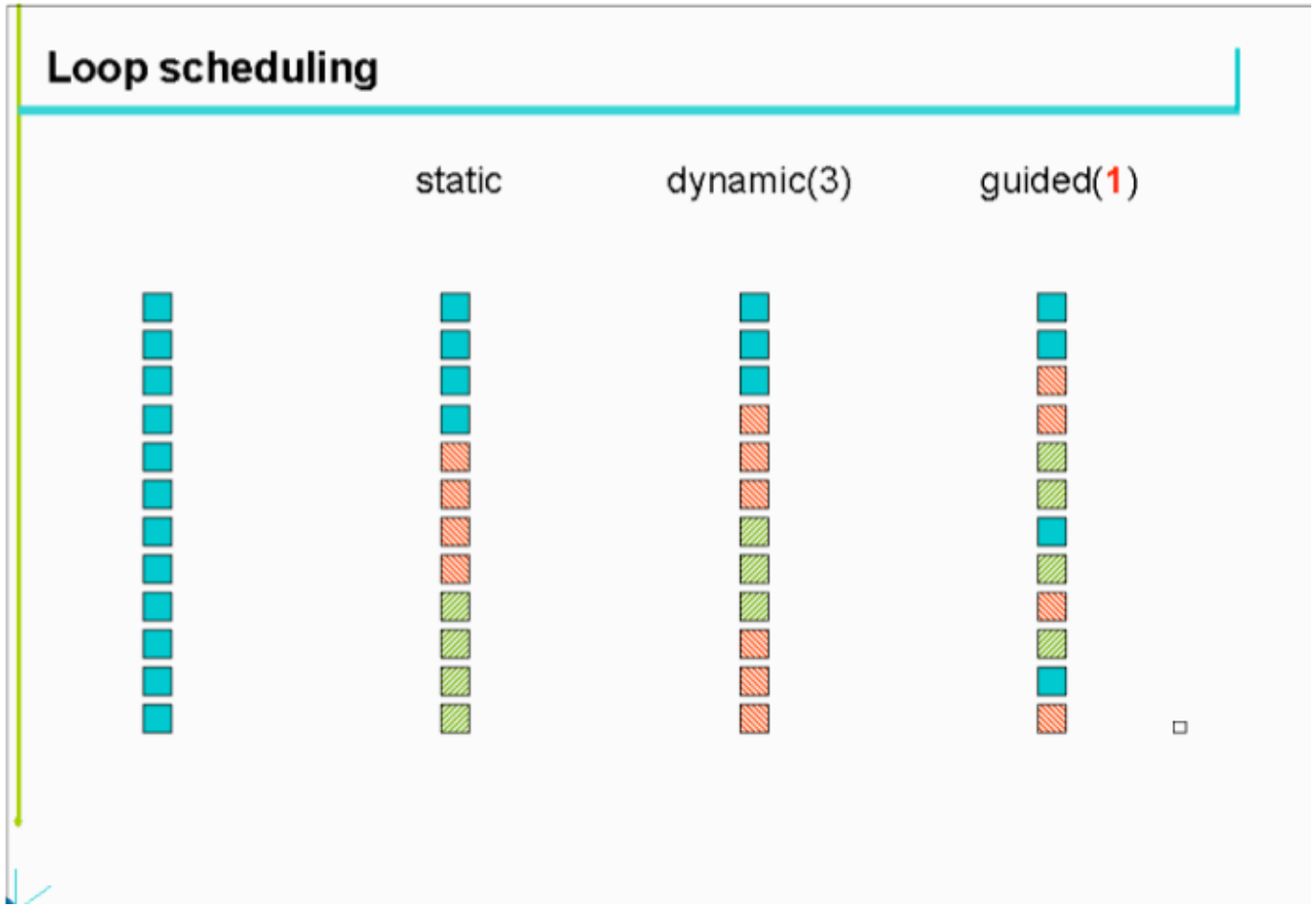
- La cláusula **schedule** permite dividir las iteraciones de los ciclos entre los hilos, indica como las iteraciones se asignan a los hilos.

Clausula schedule

- *schedule(static ,[chunk])*
  - Bloques de iteraciones de tamaño *“chunk”* a los hilos
  - Distribución Round robin
- *schedule (dynamic,[chunk])*
  - Los hilos toman un numero “*chunk” de iteraciones, cuando estan sin trabajo.*
- *schedule(guided,[chunck])*

Cada thread toma iteraciones dinámicamente y

  - progresivamente va tomando menos iteraciones.


Loop scheduling
static
dynamic(3)
guided(1)

Ejemplo

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

Ejercicio probar diferentes asignaciones

```
#include<omp.h>
#include<stdio.h>
#include<stdlib.h>
#define CHUNKSIZE 5
#define N 20

int main (int argc, char *argv[]) {

int nthreads, tid, i, chunk;
float a[N], b[N], c[N];



for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;
```

```
#pragma omp parallel shared(  ) private(  )
{
    tid=omp_get_thread_num();

     if (tid == 0)  {
   nthreads=omp_get_num_threads();
      printf("Numero de hilos = %d\n",      nthreads);
         }
    printf("Iniciando Hilo - %d ...\n",tid);

  #pragma omp for schedule( ,  )
    for (i=0; i<N; i++) {
   c[i] = a[i] + b[i];
   printf("Hilo %d: c[%d]= %f\n",tid,i,c[i]);
     }
  }
}
```

Clausula reduction

- Las operaciones de reducción son comunes en muchas aplicaciones paralelas.
- Utilizan variables a las que acceden todos los procesos/hilos y sobre las que se efectúa alguna operación de “acumulación” en modo atómico.

Cláusula reduction

#pragma omp reduction(operator:variable)

```
#pragma omp parallel private(X) reduction(+:sum)
{
    X = …
    …
    sum = sum + X;
    …
}
```

La operador de reducción a utilizar.

OJO: no se sabe en qué orden se va a ejecutar la operación
--> debe ser conmutativa (cuidado con el redondeo).

Operaciones utilizados en reducciones (C/C++)

Operator	Initial Value
+	0
*	1
-	0
^	0

Operator	Initial Value
&	0
|	0
&&	1
||	0

Actividad

- Realizar un programa que realice el producto punto de dos vectores de dimensión n.
- Paralelizar el programa.

Ejemplo- Ejercicio

- Cálculo del numero Pi
- $\pi = \int_0^1 4/(1+x^2)\, dx$

- La Regla de rectángulo consiste de estimar el área debajo de la curva y=4/(1+x2) entre x=0 y x=1 mediante áreas de rectángulos

Paralelizar utilizando OpenMP

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

long long num_steps = 1000000000;
double step;

int main(int argc, char* argv[])
{

  double x, pi, sum=0.0;
  int i;
  for (i=0; i<num_steps; i++)
  {
  x = (i + .5)*step;
  sum = sum + 4.0/(1.+ x*x);
  }

  pi = sum*step;

  printf("El valor de Pi es %15.12f\n",pi);
  return 0;
}
```

Para medir tiempo en OpenMP

```
double empezar,terminar;

empezar=omp_get_wtime( );

…código

terminar=omp_get_wtime();

printf(“TIEMPO=%lf\n”,empezar-terminar)
```

El resultado es en segundos.

Una Solución

```
#include <omp.h>
static long num_steps = 100000; double
  step;
#define NUM_THREADS 2
void main ()
{
    int i; double x, pi, sum[NUM_THREADS];
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS)
```

```
#pragma omp parallel
{
double x; int id;
id = omp_get_thread_num();
sum[id] = 0;
 #pragma omp for
 for (i=id;i< num_steps; i++){
  x = (i+0.5)*step;
  sum[id] += 4.0/(1.0+x*x);
   }
}
for(i=0, pi=0.0;i<NUM_THREADS;i++)
pi += sum[i] * step;
}
```

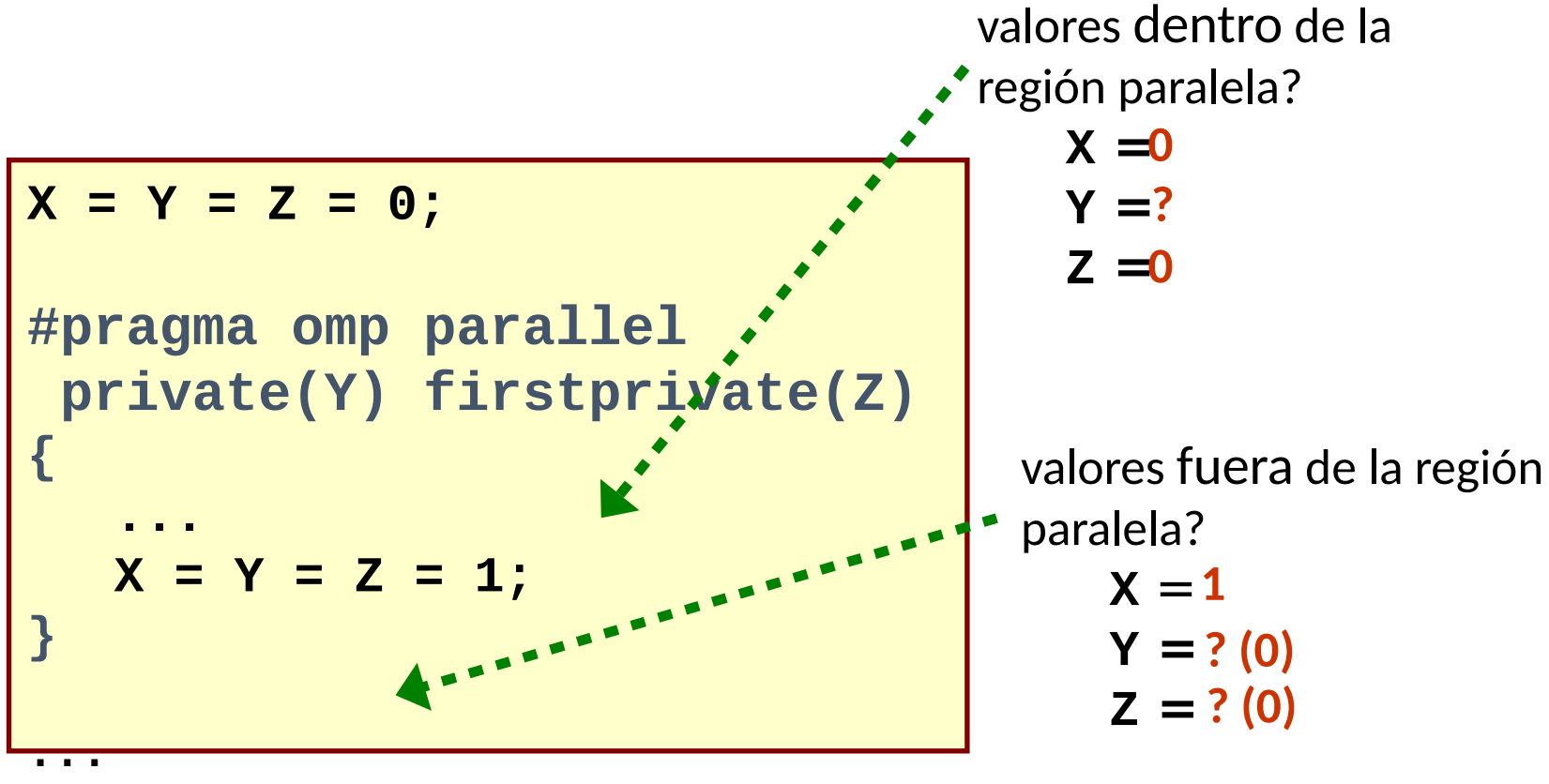
Otra Solución

```
#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 2
void main ()
{
  int i; double x, pi, sum = 0.0;
  step = 1.0/(double) num_steps;
  omp_set_num_threads(NUM_THREADS);
  #pragma omp parallel for reduction(+:sum) private(x)
   for (i=1;i<= num_steps; i++){
   x = (i-0.5)*step;
   sum = sum + 4.0/(1.0+x*x);
}
pi = step * sum;
}
```

Clausula firstprivate

- Las variables privadas no se inicializan y al terminar la región paralela tienen un valor indefinido.
- Para inicializar una variable privada se utiliza firstprivate.

Ejemplo


X = Y = Z = 0;
#pragma omp parallel
private(Y) firstprivate(Z)
{
...
X = Y = Z = 1;
}
...
valores dentro de la región paralela?
X = 0
Y = ?
Z = 0
valores fuera de la región paralela?
X = 1
Y = ? (0)
Z = ? (0)

¿Cómo se puede paralelizar el siguiente código?

```
x[0] = complex_function();
for (i=0;j<n;i++) {
  for (j=1;j<4;j++)
    x[j]=g(i,x[j-1]);
  sol[i] = x[1] – x[3];}
```

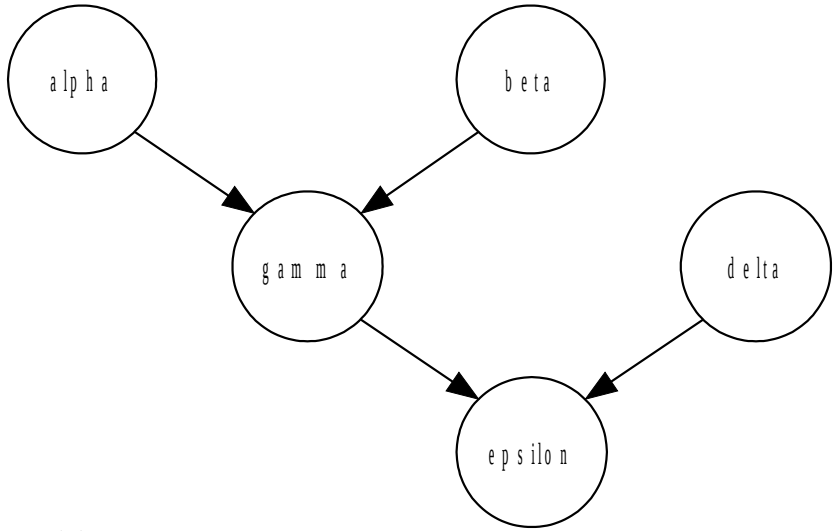
Se pueden dividir las itereciones del ciclo exterior si j y x son privadas. Sin embargo, x[0] se necesita en la primera iteración del bucle interior.

Una solución

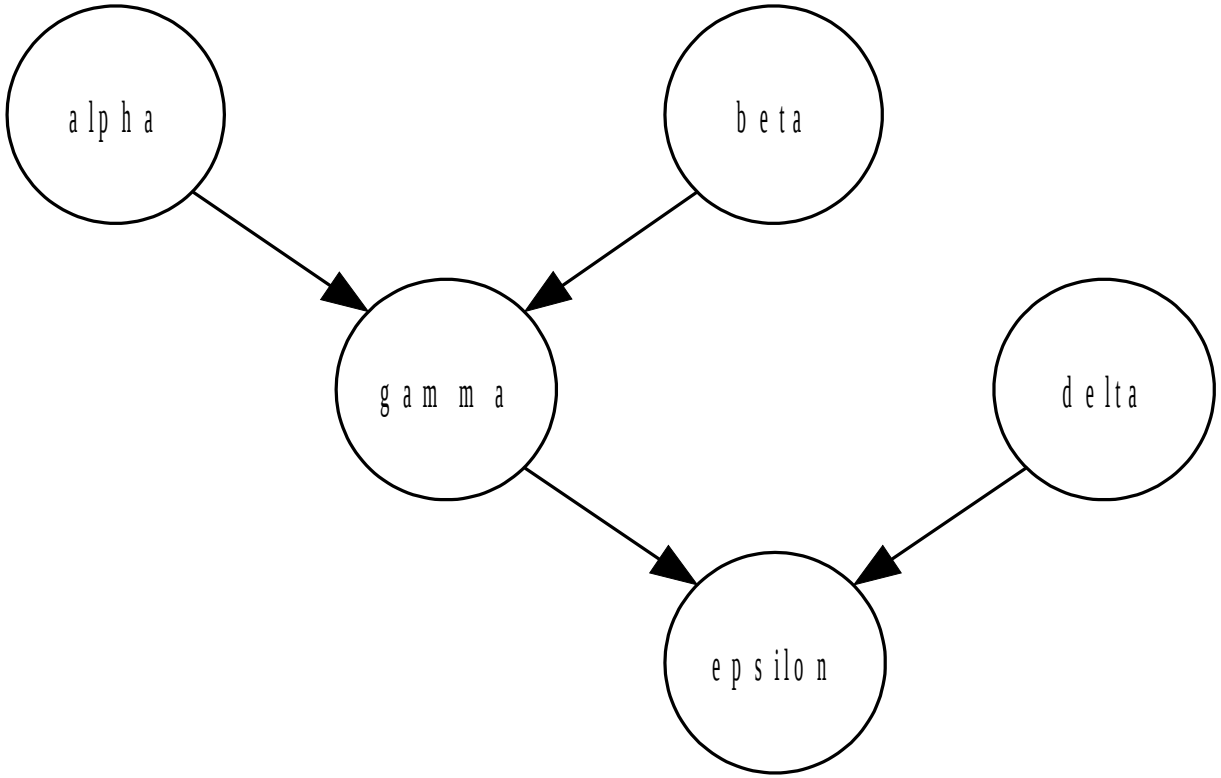
```
x[0] = complex_function();
#pragma omp parallel for private[j]
 firstprivate(x)
for (i-0;j<n;i++) {
  for (j=1;j<4;j++)
    x[j]=g(i,x[j-1]);
  sol[i] = x[1] – x[3];}
```

Descomposicion Funcional

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



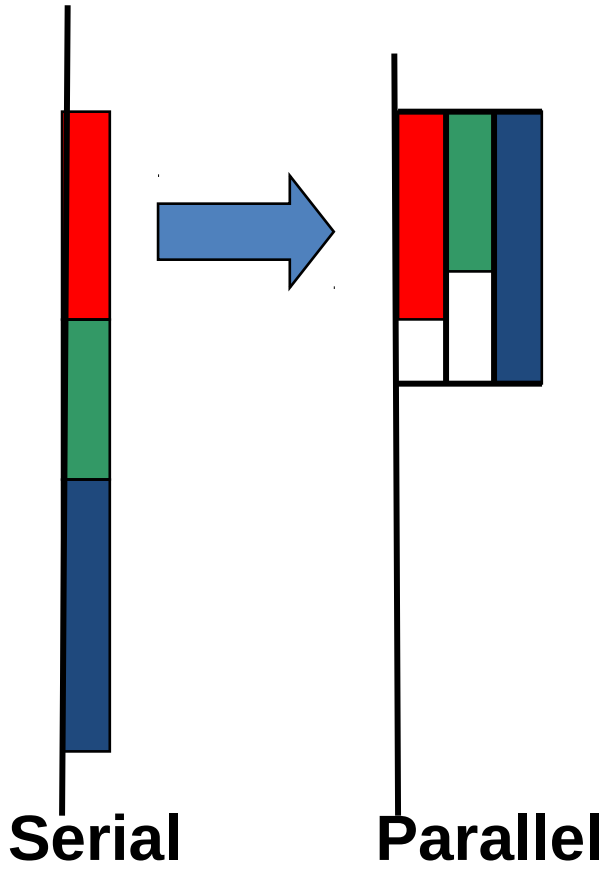
¿Cómo se paraleliza?



Paralelismo funcional Secciones Paralelas

- Secciones independientes de código, se pueden ejecutar de forma concurrente

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



Constructor section

Permite usar paralelismo de funcional (*descomposición funcional*).

Reparte secciones de código independiente a *hilos* diferentes.

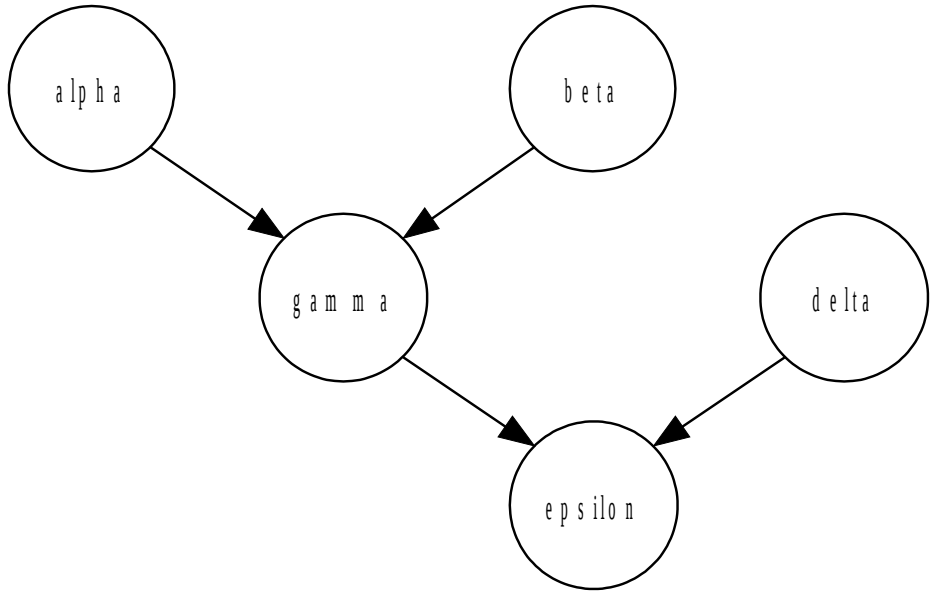
Cada sección paralela es ejecutada por un sólo *hilo*, y cada *hilo* ejecuta ninguna o alguna sección.

Una barrera implícita sincroniza el final de las secciones o

Posible solución

```
#pragma omp parallel sections
   {
#pragma omp section  /* Opcional */
      v = alpha();
#pragma omp section
      w = beta();
#pragma omp section
      y = delta();
   }
   x = gamma(v, w);
   printf ("%6.2f\n", epsilon(x,y));
   }
```

Otra Posibilidad



Ejecutar alpha y beta en paralelo. Ejecutar gama y delta en paralelo.

Otra posibilidad

```
#pragma omp parallel sections
{
#pragma omp section
     v=alpha();
#pragama omp section
     w=beta();
}
#pragma omp parallel sections
  {
    #pragma omp section
       y=delta();
    #pragma omp section
       x=gamma(v,w);
  }
printf(“%6.2f\n”,epsilon(x,y);
```

Con dos secciones

```
#pragma omp parallel
   {
   #pragma omp sections
      {
      #pragma omp section
         v = alpha();
      #pragma omp section
         w = beta();
      }
   #pragma omp sections
      {
      #pragma omp section
         x = gamma(v, w);
      #pragma omp section
         y = delta();
      }
   }
   printf ("%6.2f\n", epsilon(x,y));
```

Ejemplo de secciones

```
#pragma omp parallel sections
{
#pragma omp section
 for(i=0;i<n;i++)
      c[i]=a[i]+b[i];
#pragma omp section
     for(j=0;j<n;j++)
       d[j]=e[j]+f[j];
}
```

Una posibilidad mejor

```
#pragma omp parallel
{
  #pragma omp for
     for(i=0;i<n;i++)
       c[i]=a[i]+b[i];
 #pragma omp for
     for(j=0;j<n;j++)
       d[j]=e[j]+f[j];
}
```

o simplemente

```
#pragma parallel for
 for(i=0;i<n;i++)
      {
       c[i]=a[i]+b[i];
       d[i]=e[i]+f[i];
      }
```

Barreras Implícitas

- Algunos constructores tiene barreras implícitas
  - Parallel
  - For
  - Single
- Barreras no necesarias perjudican el desempeño

Ejercicio

- Paralelizar los dos códigos proporcionados, utilizando los constructores y clausulas vistos.

Constructor Barrier

- Barrera Explicita
- Cada hilo espera hasta que todos lleguen a la barrera

```
#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");
}
```

Constructor single

- Define un bloque básico de código, dentro de una región paralela, que debe ser ejecutado por un único hilo.
- Ejemplo, una operación de entrada/salida.

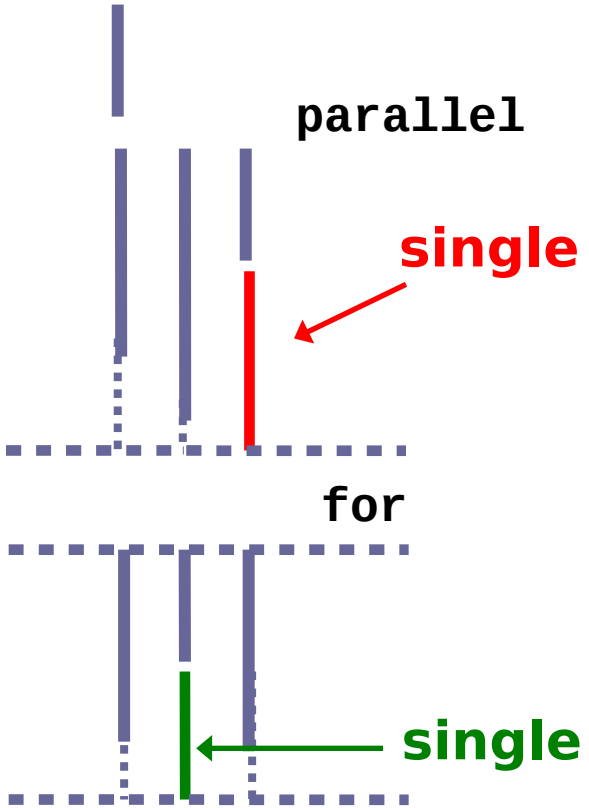
- No se especifica qué hilo ejecutará la tarea.

Constructor single

- #pragma omp single

```
#pragma omp parallel
{
    DoManyThings();
  #pragma omp single
    {
        ExchangeBoundaries();
    }   // Hilos esperan

    DoManyMoreThings();
}
```

```
#pragma omp parallel
{
  … ;
  #pragma omp single
  inicializar(A);

  #pragma omp for
  for(i=0; i<N; i++)
       A[i] = A[i] * A[i] + 1;

  … ;

  #pragma omp single
  copiar(B,A);
}
```



Constructor maestro

#pragma omp master { }

```
#pragma omp parallel
{
    DoManyThings();
    #pragma omp master
    {  // si no es el maestro,salta
        ExchangeBoundaries();
    }
    DoManyMoreThings();
}
```

```
#include <omp.h>
#include <stdio.h>
int main( )
{
      int a[5], i;
  #pragma omp parallel
{
       #pragma omp for
       for (i = 0; i < 5; i++)
       a[i] = i * i;
  #pragma omp master
       for (i = 0; i < 5; i++)
            printf_s("a[%d] = %d\n", i, a[i]);
       #pragma omp barrier
       #pragma omp for
  for (i = 0; i < 5; i++)
  a[i] += i;
  }
}
```

Barreras Implícitas

- Algunos constructores tiene barreras implícitas
  - Parallel
  - For
  - Single
- Barreras no necesarias perjudican el desempeño

Clausula nowait

- Permite ignorar barreras implicitas

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

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

```
#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);
```

Constructor atomic

- Asegura que una posición específica de memoria debe ser modificada de forma atómica, sin permitir que múltiples hilos intenten escribir en ella de forma simultánea. (sección critica).

- La sentencia debe tener una de las siguientes formas:

x <operacion-binaria> = <expr>

x++

++x

x---

---x

Ejemplo atomic

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

#define MAX 10

int main() {
  int count = 0;
  #pragma omp parallel num_threads(MAX)
  {
    #pragma omp atomic
    count++;
  }
  printf("Number of threads: %d\n", count);
}
```

Paralelismo anidado

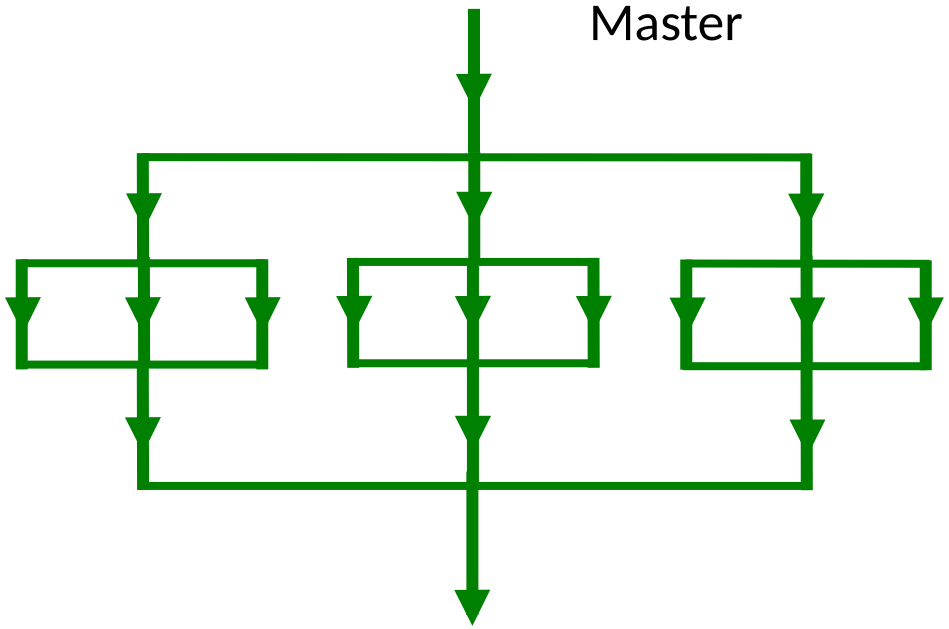
- Por defecto no es posible anidar regiones paralelas, hay que indicarlo explícitamente mediante:
  - una llamada a una función

  omp_set_nested(1);
  - una variable de entorno

  > export OMP_NESTED=TRUE

- Una función devuelve el estado de dicha opción:
- omp_get_nested(); (true o false)

Paralelismo anidado



Región paralela externa

Región paralela anidada

Región paralela externa

Paralelismo anidado

- OpenMP 3.0 mejora el soporte al paralelismo anidado:

- -La función omp_set_num_threads() puede ser invocada dentro de una región paralela para controlar el grado del siguiente nivel de paralelismo.

- Permite conocer el nivel de anidamiento mediante omp_get_level() y omp_get_active_level().

- Se puede tener acceso al identificador del padre de nivel n omp_get_ancestor_thread_num(n)y al número de threads en dicho nivel.

Paralelismo anidado

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

void report_num_threads(int level)
{
    #pragma omp single
  {
  printf("Level %d: number of threads in the team - %d\n",
  level, omp_get_num_threads());

    }
 }
```

```
int main() {
    omp_set_nested(1);
  #pragma omp parallel num_threads(2)
{
       report_num_threads(1);
          #pragma omp parallel num_threads(2)
       {
            report_num_threads(2);
            #pragma omp parallel num_threads(2)
             {
             report_num_threads(3);
             }
       }
 } return(0);
}
```

Variable de ambiente
SUNW_MP_MAX_POOL_THREADS

- Máximo numero de hilos esclavos en una región paralela
- Valor por defecto 1023

Referencias

- Parallel Programming in C with **MPI** and **OpenMP**. Michael J. Quinn. McGraw-Hill, 2003.
- Página Oficial de **OpenMP**: http://www.openmp.org
- Presentaciones Intel Corporation Shared Memory –model and Thread