COMPUTACIÓN PARALELA

3º GRADO EN INGENIERÍA INFORMÁTICA EN TECNOLOGÍAS DE LA INFORMACIÓN

Unidad Didáctica III. Arquitecturas de memoria compartida: OpenMP

Introducción

- Cualquier desarrollo paralelo necesita: COMUNICACIONES y SINCRONIZACIONES

- En memoria compartida las comunicaciones NO utilizan ningún *"sistema de comunicación"* entre computadores

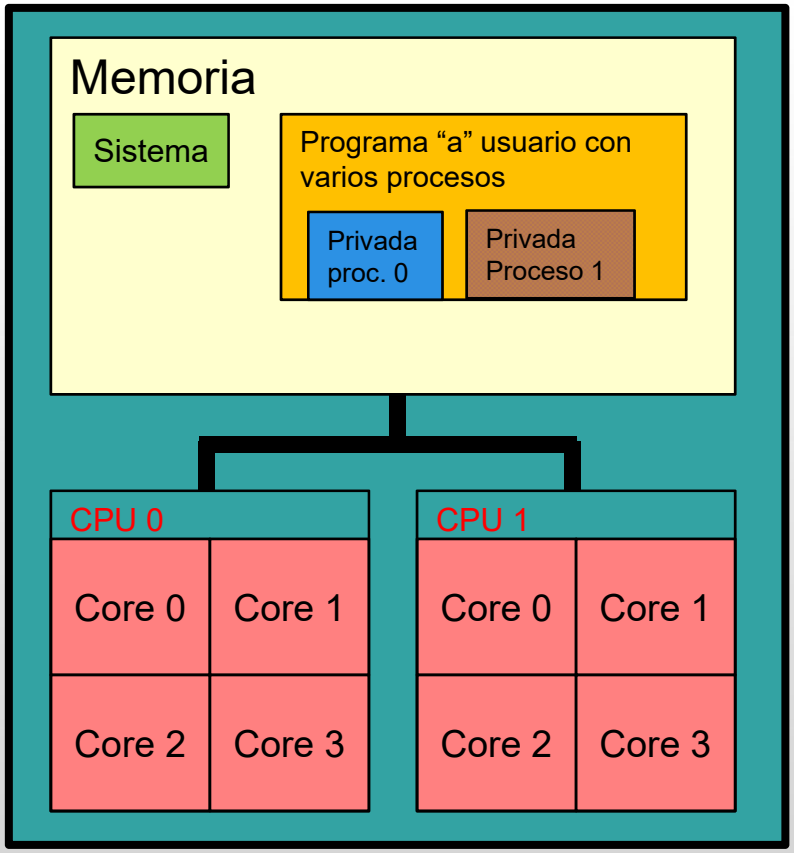
- Todas las comunicaciones se realizan a través de la **memoria**

- La memoria es común (compartida), pero se necesitará definir algunas variables como privadas …

- … la (correcta) gestión de la memoria será un aspecto clave en el correcto funcionamiento y en la eficiencia

Introducción

Nodo de cómputo multicore y multiprocesador



Programa “a” se ejecuta una vez pero pueden haber varios procesos

Cada proceso (ya veremos cómo generarlos) se ejecuta en un Core distinto

Cada proceso podrá acceder a toda la memoria común …

… pero podrán tener zonas de memoria privada.

Esas memorias no serán accesibles por el resto de procesos

El “mapeo” de los procesos en los cores puede ser realizado por SO o indicado por usuario

Introducción a OpenMP

- Modelo basado en la creación de hilos o threads* o procesos (que deben usar caminos “hardware” diferentes de ejecución)**
- Los diferentes hilos ejecutarán el mismo código (pero realizarán diferentes funcionalidades)
- Las diferentes funcionalidades podrán decidirse explícitamente (análogo a MPI) o “automatizadamente”

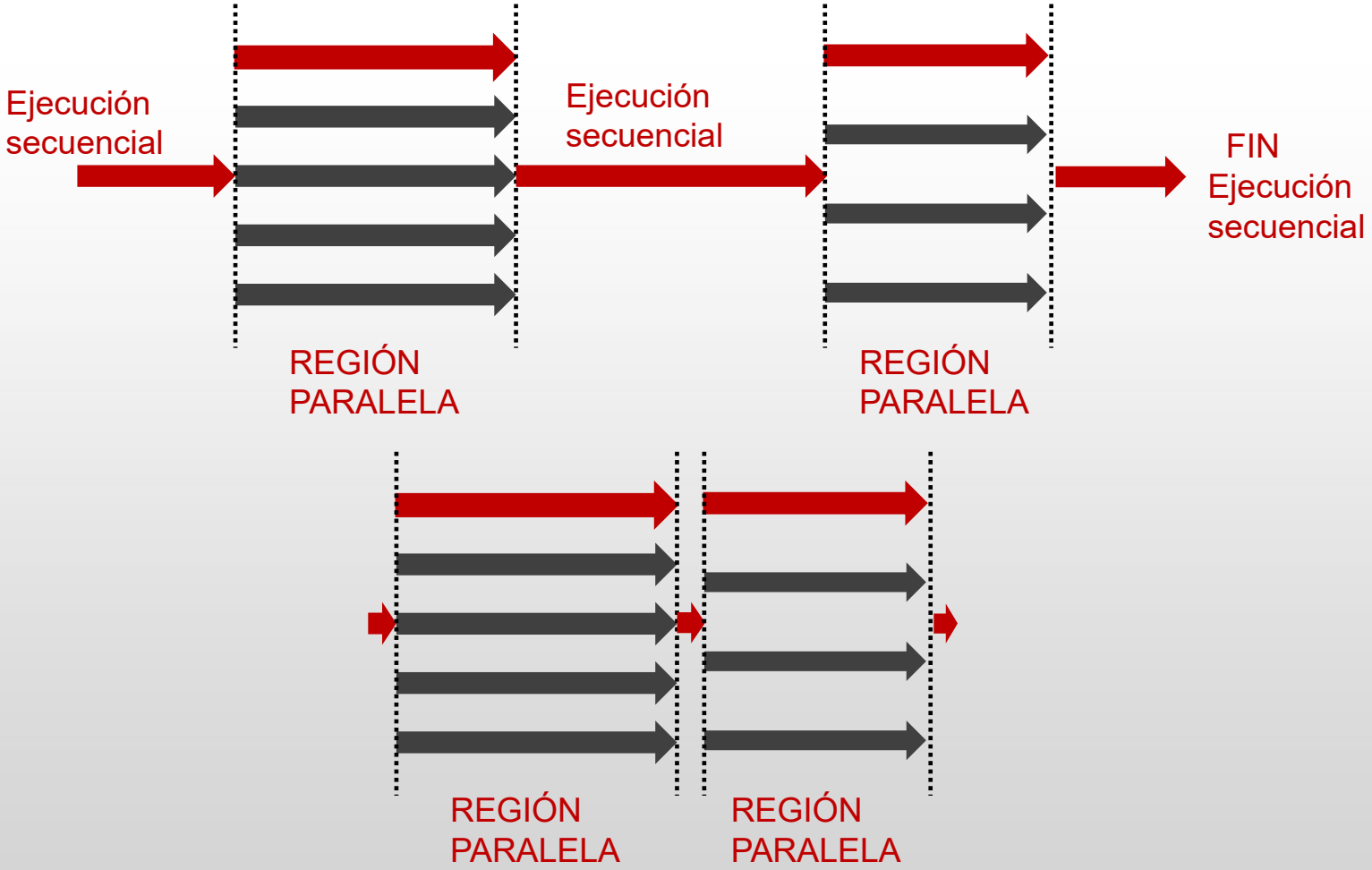
* OpenMP 3.0 permite el trabajo con tareas (“tasks”)

** No consideramos el uso hyperthreading o mecanismos similares

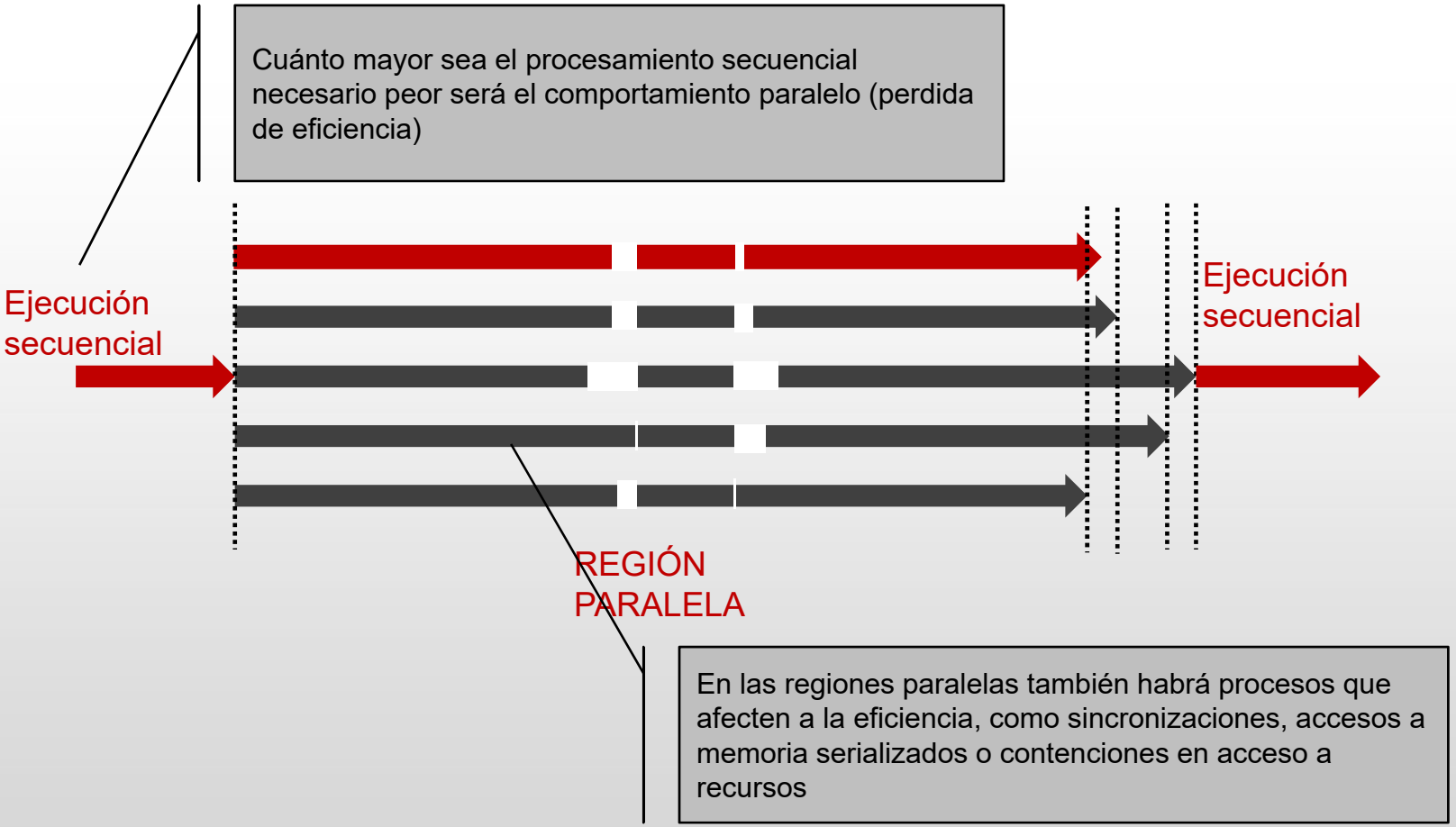
Vamos a ver (en “Introducción OpenMP”):

- Modelo de ejecución
- Compilación y ejecución
- Qué es una directiva
- Qué es una cláusula
- Qué es una variable de entorno
- Memoria compartida y memoria privada

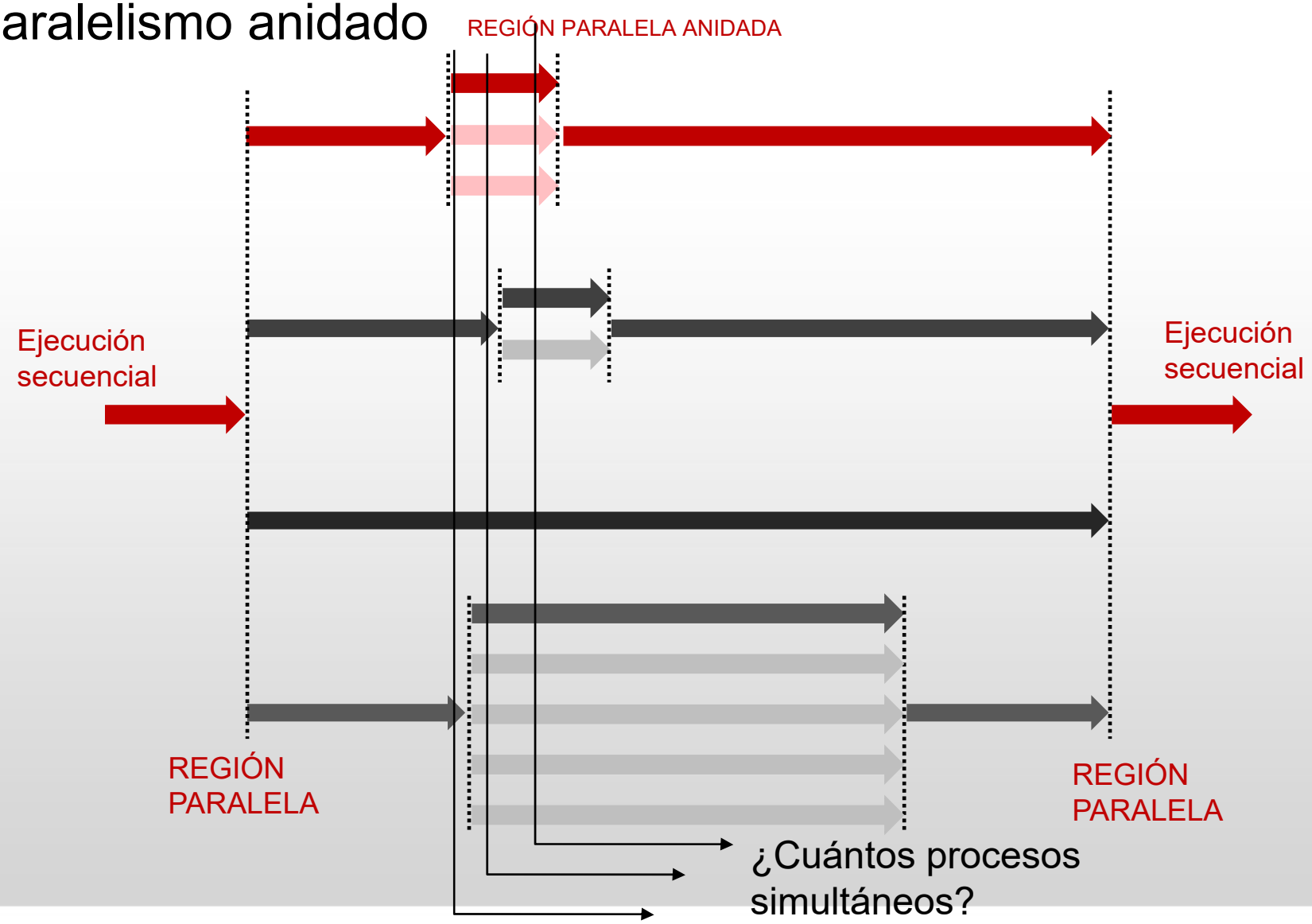
Modelo ejecución “fork-join”



Modelo ejecución “fork-join”



Paralelismo anidado



Introducción a OpenMP

Método basado en directivas para el procesamiento paralelo en arquitecturas de memoria compartida.

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

int main()
{
int i;
double *datos; //datos a procesar

…
#pragma omp for
   for(i=0;i<10000;i++)
    {
      función_procesamiento(&datos[i]);
    }
…
}
```

Directivas de preprocesador. Facilitan el desarrollo y/o compilación condicional

Directivas “pragma” para usar funcionalidades particulares del compilador, en este caso de OpenMP

#pragma omp → lo que sigue es una directiva OpenMP (en el ejemplo la directiva for)

Aunque OpenMP está basado en directivas consta de:

- Directivas
- Cláusulas (que varían el comportamiento de las directivas)
- Funciones o subrutinas (dependiendo del lenguaje de programación utilizado: C → funciones; Fortran → subrutinas)
- Variables de entorno (muchas de ellas toman el papel de "valor por defecto")

NO intrusivo

```
#include <omp.h>
#include <stdio.h>
int main()
{
int np;

…
#if defined (_OPENMP)
    np = omp_get_num_threads();
#endif
…
#pragma omp for
   for(i=0;i<10000;i++)
    {
      función_procesamiento(&datos[i]);
    }
…

}
```

La constante _OPENMP se define al compilar con OpenMP

No todo son directivas

Si el compilador no entiende (es decir no tiene habilitado) las directivas OpenMP es tratado como un comentario

RECORDATORIO MPI

Descomposición de dominio

Misma funcionalidad sobre diferentes datos

Clase 0:

Para todos los elementos de una matriz contar:

1. Elementos pares
2. Elementos impares
3. Números cuyas cifras sumen 15

Proceso 0 **Proceso 1** **Proceso 2**

Todos tareas 1, 2 y 3

Proceso 0

115	131	365	193	427	466	110	59	221	298	223	410
142	332	196	241	471	12	74	393	249	465	130	38
336	108	106	306	173	271	356	232	404	149	28	401
281	64	433	496	384	87	115	90	463	339	475	262
327	387	190	55	127	303	449	91	366	218	242	308
68	336	443	90	343	158	418	322	129	97	21	411
98	484	209	254	368	8	244	369	186	331	297	388
9	2	121	119	500	147	309	191	172	243	316	417
371	124	48	261	190	167	283	113	146	114	302	352
84	319	464	310	354	281	179	496	439	344	125	131
299	94	212	59	279	378	169	301	277	129	440	318
414	86	79	70	343	454	202	449	334	396	339	439

24	24	4

Proceso 1

24	24	1

Proceso 2

28	20	1

Comunicaciones a través de la memoria

Proceso 0 **Proceso 1** **Proceso 2**

Todos tareas 1, 2 y 3

> **Clase 0:**
> Para todos los elementos de una matriz contar:
> 1. Elementos pares
> 2. Elementos impares
> 3. Números cuyas cifras sumen 15

Memoria común

115	131	365	193	427	466	110	59	221	298	223	410
142	332	196	241	471	12	74	393	249	465	130	38
336	108	106	306	173	271	356	232	404	149	28	401
281	64	433	496	384	87	115	90	463	339	475	262
327	387	190	55	127	303	449	91	366	218	242	308
68	336	443	90	343	158	418	322	129	97	21	411
98	484	209	254	368	8	244	369	186	331	297	388
9	2	121	119	500	147	309	191	172	243	316	417
371	124	48	261	190	167	283	113	146	114	302	352
84	319	464	310	354	281	179	496	439	344	125	131
299	94	212	59	279	378	169	301	277	129	440	318
414	86	79	70	343	454	202	449	334	396	339	439

Proceso 1

24	24	1

24	24	4

Proceso 0

Proceso 2

28	20	1

A tener en cuenta:

- Los dos modos fundamentales de abordar un desarrollo paralelo no cambian (descomposición de dominio y funcional).
- Las comunicaciones van a seguir siendo necesarias, pero probablemente disminuirán en número y serán más rápidas…
- … ya que se realizarán a través de la memoria.
- El inicio y fin de la ejecución será siempre secuencial y se crearán una o varias regiones de ejecución paralela (modelo fork-join).
- La posibilidad de utilizar OpenMP depende de que el compilador (y por tanto el sistema) admita dicha característica especial.

A tener en cuenta:

- Para desarrollar un código paralelo eficiente hay que analizar las sincronizaciones y las comunicaciones.

- El rendimiento o el uso no óptimo de la memoria puede ocasionar pérdida de eficiencia.

Conclusión:

- Gran parte del estudio estará centrado en las opciones de gestión de memoria y la optimización de la misma…

- … ya que por regla general usar memoria privada será preferible a memoria compartida si la memoria se usa para lectura y escritura. Grandes cantidades de memoria no deben definirse como privada.

COMPILACIÓN:

El compilador NO es específico, para gcc ("GNU Compiler Collection") debemos compilar con la opción de compilación –fopenmp

El código fuente debe incluir el fichero omp.h (para gcc)

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

Ejemplo de compilación

```
gcc –fopenmp –o ejecutable fuente.c
```

RECORDAD: el parámetro de compilación y el fichero de encabezado varían en función del compilador utilizado

EJECUCIÓN:

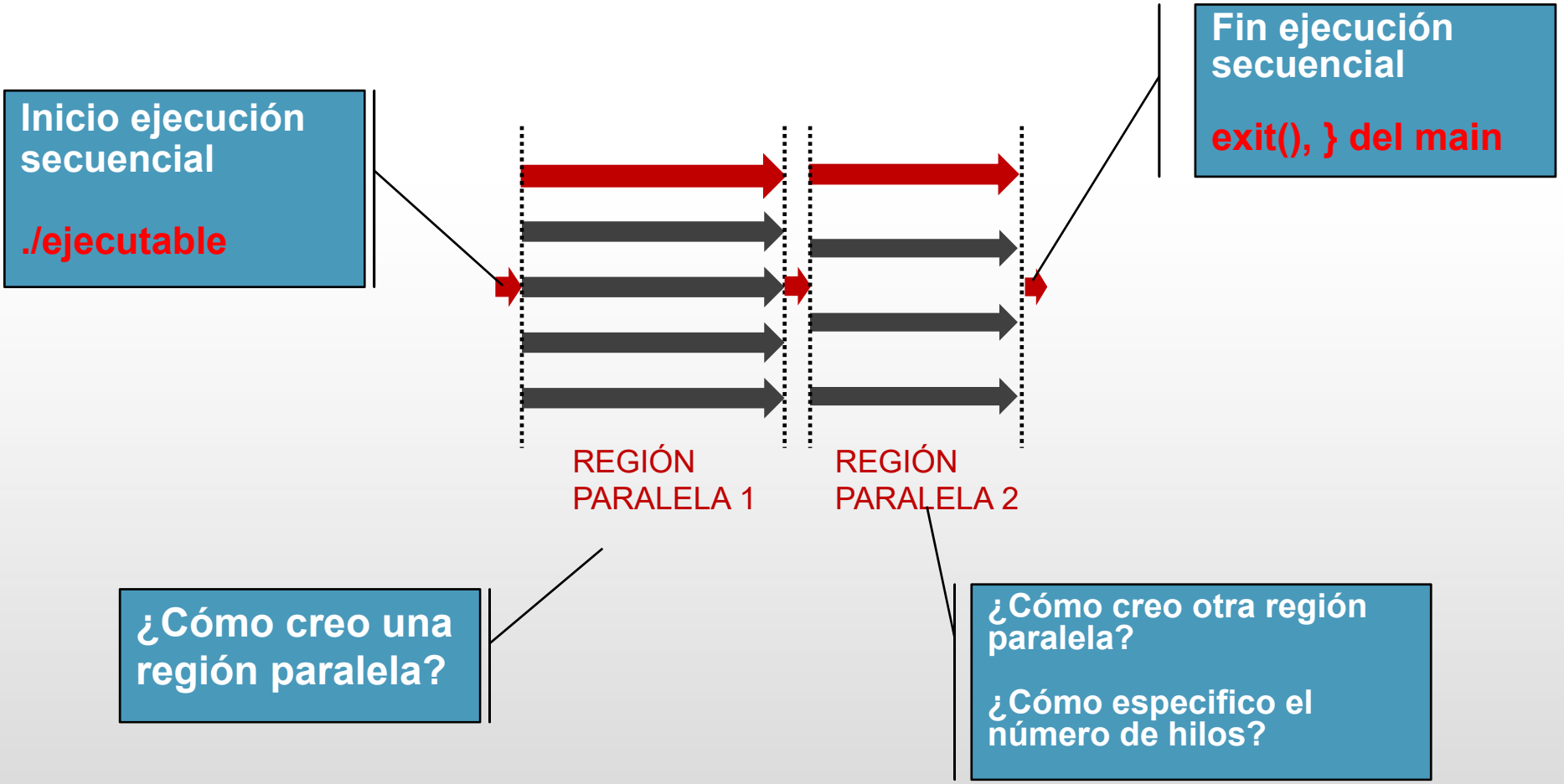
En un sistema Linux: ./ejecutable

¿Cómo especifico el número de hilos? (En MPI era “mpirun –np 4 ejemplo")

Modelo “fork-join” con posibilidad de “nested parallelism” →

→ El número de hilos se especifica para TODAS y CADA UNA de las regiones paralelas que se creen.

Introducción a OpenMP



Introducción a OpenMP

```
#include <omp.h>
int main()
{

    #pragma omp parallel
    {                                   //Inicio Región Paralela
        funcion1(…);
        …
        funcionN(…);
    }                                   //Fin Región Paralela
    …
    #pragma omp parallel
    {                                   //Inicio Región Paralela
        otrafuncion1(…);
        …
        otrafuncionN(…);
    }                                   //Fin Región Paralela
}
```

¿Cómo creo una región paralela?

Con la directiva parallel

¿Cómo creo otra región paralela?

Con la directiva parallel

Introducción a OpenMP

```
#include <omp.h>
int main()
{
    #pragma omp parallel
    {
        funcion1(…);
        …
        funcionN(…);
    }
    …
    #pragma omp parallel
    {
        otrafuncion1(…);
        …
        otrafuncionN(…);
    }
}
```

Si REPLICO la ejecución no reduzco tiempos. He de repartir el trabajo.

```
//Inicio Región Paralela
```

Es ejecutado por todos los hilos creados (5)

```
//Fin Región Paralela
```

```
//Inicio Región Paralela
```

Es ejecutado por todos los hilos creados (4)

```
//Fin Región Paralela
```

Introducción a OpenMP

```
#include <omp.h>
int main()
{
    int iam =0, np = 1;
    …
    #pragma omp parallel num_threads(5)
    {                                   //Inicio Región Paralela
        funcion1(…);
        …
        funcionN(…);
    }                                   //Fin Región Paralela
    …
    #pragma omp parallel num_threads(4)
    {                                   //Inicio Región Paralela
        otrafuncion1(…);
        …
        otrafuncionN(…);
    }                                   //Fin Región Paralela

}
```

¿Cómo especifico el número de hilos?

Con la cláusula num_threads

¿Cómo especifico el número de hilos?

Con la cláusula num_threads

Introducción a OpenMP

Recordemos:

- Tengo que repartir el trabajo asignado a cada región paralela entre los hilos creados para ejecutar dicha región

- Dos opciones para repartir el trabajo:
  - Explícita
  - Automatizada

- El reparto de trabajo explícito es análogo al realizado en MPI (ya sea usando una descomposición de dominio o funcional).

- El reparto automatizado es realizado por OpenMP según las directrices que especifiquemos (utilizando directivas y cláusulas)

Introducción a OpenMP

REPARTO DE TRABAJO EXPLÍCITO:

Clase 0:
Para todos los elementos de una matriz contar:
1. Elementos pares
2. Elementos impares
3. Números cuyas cifras sumen 15

Proceso 0 **Proceso 1** **Proceso 2**

Todos tareas 1, 2 y 3

Memoria común

115	131	365	193	427	466	110	59	221	298	223	410
142	332	196	241	471	12	74	393	249	465	130	38
336	108	106	306	173	271	356	232	404	149	28	401
281	64	433	496	384	87	115	90	463	339	475	262
327	387	190	55	127	303	449	91	366	218	242	308
68	336	443	90	343	158	418	322	129	97	21	411
98	484	209	254	368	8	244	369	186	331	297	388
9	2	121	119	500	147	309	191	172	243	316	417
371	124	48	261	190	167	283	113	146	114	302	352
84	319	464	310	354	281	179	496	439	344	125	131
299	94	212	59	279	378	169	301	277	129	440	318
414	86	79	70	343	454	202	449	334	396	339	439

Proceso 0 **Proceso 1** **Proceso 2**

24	24	4

24	24	1

28	20	1

¿Memoria de matriz?	Compartida	Todos acceden
Memoria de resultados	Privada (podría ser compartida)	Solo accede un hilo
¿Cómo reparto trabajo?	Reparto bloques de filas	Número e identificación de hilos
¿Cómo "reduzco el resultado?	Según memoria	Trabaja un hilo o trabajan todos

Ejemplo FUNCIONES: Obtener número e identificar hilos

```
#include <omp.h>
int main()
{
    int nt, iam;
    …
    #pragma omp parallel
    {

        nt = omp_get_num_threads();

        iam  = omp_get_thread_num();

        …

    } //Fin Región Paralela
    …
}
```

Devuelve el número de hilos de la región paralela en la que se llama a función.

¿Cómo especifico el número de hilos?

`//Inicio Región Paralela`

Devuelve el identificador de hilo

Entero 0 … ((nº hilos) -1)

Ejemplo VARIABLE DE ENTORNO: Número de hilos

```
#include <omp.h>
int main()
{
    int nt, iam;
    …
    #pragma omp parallel
    {                                   //Inicio Región Paralela

        nt = omp_get_num_threads();

        iam  = omp_get_thread_num();

        …

    } //Fin Región Paralela
    …
}
```

Clásula "num_threads" NO especificada

Variable de entorno:
OMP_NUM_THREADS

Memoria compartida vs Memoria privada:

```
#include <omp.h>
int main()
{
    int nt, iam, c_nt, num_pares;
    double* datos;
    …
    c_nt = 3; //podría ser cualquier valor
    #pragma omp parallel num_threads(c_nt)
    {
        nt = omp_get_num_threads();
        iam  = omp_get_thread_num();
        …
        num_pares = contarPares(datos,iniFila,finFila);
        …

    }
    …
}
```

Se debe calcular iniFila y finFila para llamar a la función contarPares.

Introducción a OpenMP

REPARTO DE TRABAJO EXPLÍCITO:

Clase 0:
Para todos los elementos de una matriz contar:
1. Elementos pares
2. Elementos impares
3. Números cuyas cifras sumen 15

Proceso 0 **Proceso 1** **Proceso 2**

Todos tareas 1, 2 y 3

Memoria común

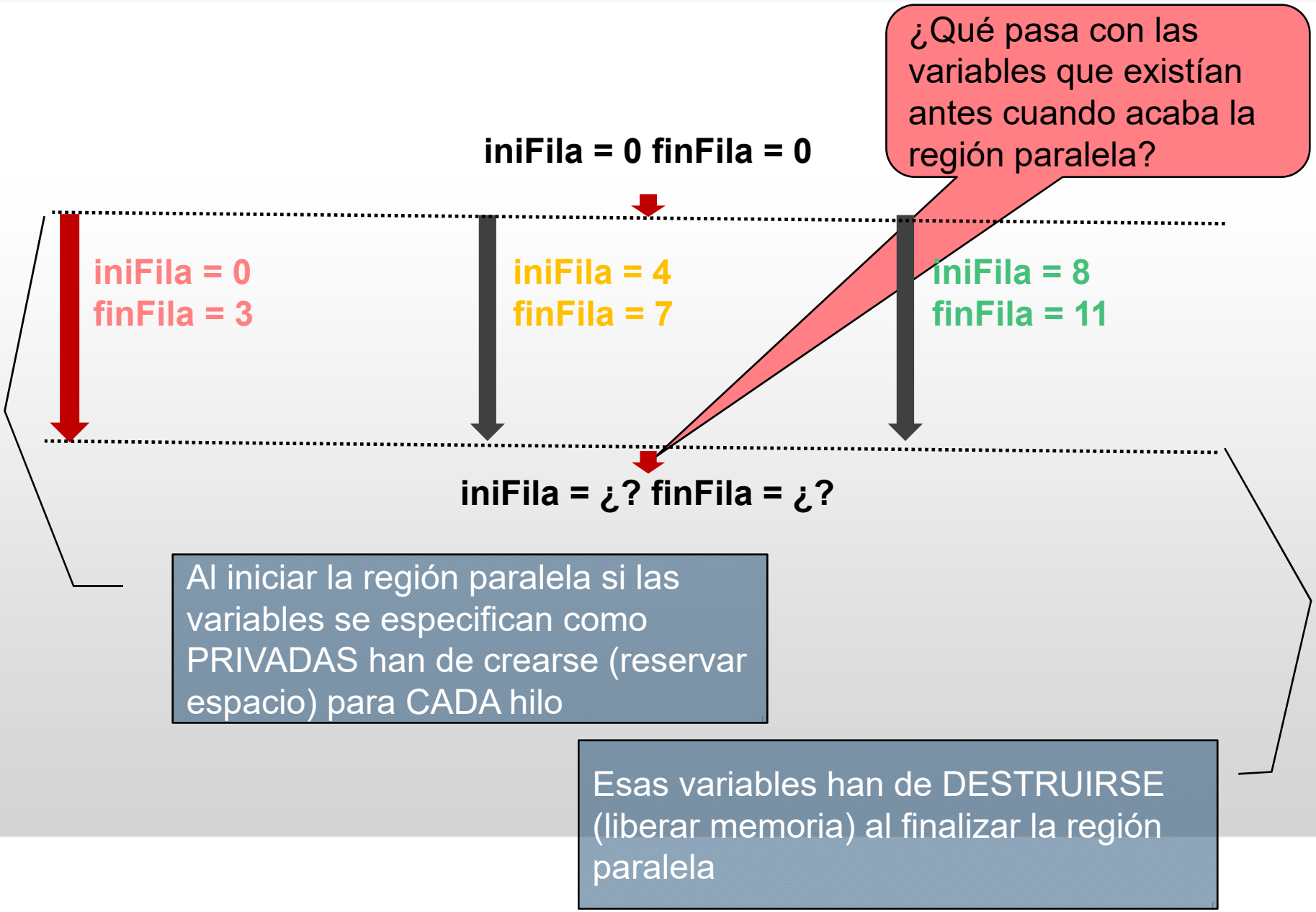
0	115	131	365	193	427	466	110	59	221	298	223	410
1	142	332	196	241	471	12	74	393	249	465	130	38
2	336	108	106	306	173	271	356	232	404	149	28	401
3	281	64	433	496	384	87	115	90	463	339	475	262
4	327	387	190	55	127	303	449	91	366	218	242	308
5	68	336	443	90	343	158	418	322	129	97	21	411
6	98	484	209	254	368	8	244	369	186	331	297	388
7	9	2	121	119	500	147	309	191	172	243	316	417
8	371	124	48	261	190	167	283	113	146	114	302	352
9	84	319	464	310	354	281	179	496	439	344	125	131
10	299	94	212	59	279	378	169	301	277	129	440	318
11	414	86	79	70	343	454	202	449	334	396	339	439

Proceso 0: **IniFila = 0 finFila = 3**

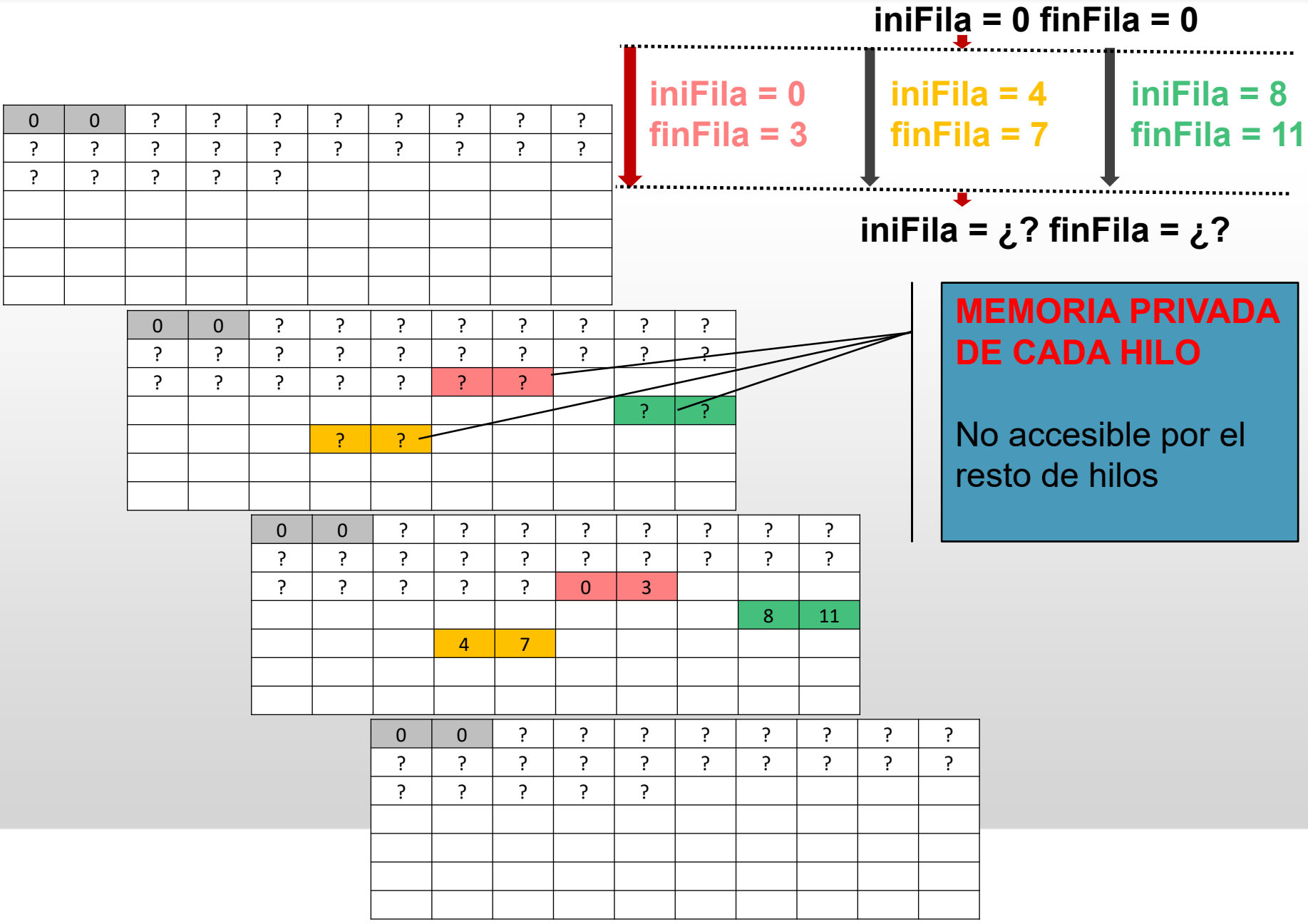
Proceso 1: **IniFila = 4 finFila = 7**

Proceso 2: **IniFila = 8 finFila = 11**

EJECUCIÓN SIMÚLTANEA → No puedo almacenar tres valores diferentes simultáneamente en una única variable

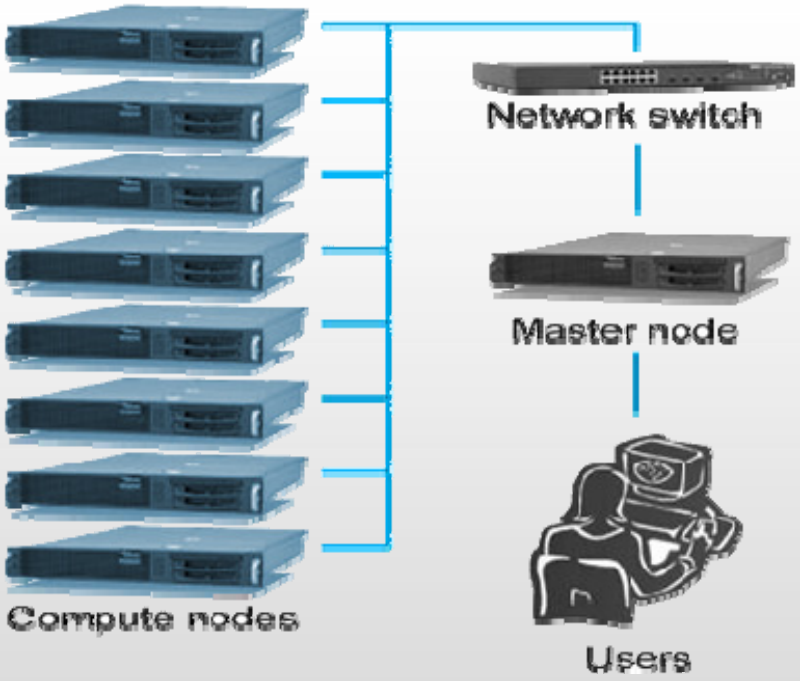

¿Qué pasa con las variables que existían antes cuando acaba la región paralela?
iniFila = 0 finFila = 0
iniFila = 0
finFila = 3
iniFila = 4
finFila = 7
iniFila = 8
finFila = 11
iniFila = ¿? finFila = ¿?
Al iniciar la región paralela si las variables se especifican como PRIVADAS han de crearse (reservar espacio) para CADA hilo
Esas variables han de DESTRUIRSE (liberar memoria) al finalizar la región paralela

Introducción a OpenMP



RECORDATORIO MPI

Simbólicamente nuestro modelo es:



- Conexión remota a nodo de acceso
- En el nodo de acceso compilamos y lanzamos
- Nuestro programa se ejecuta en los nodos de cómputo

Analicemos:

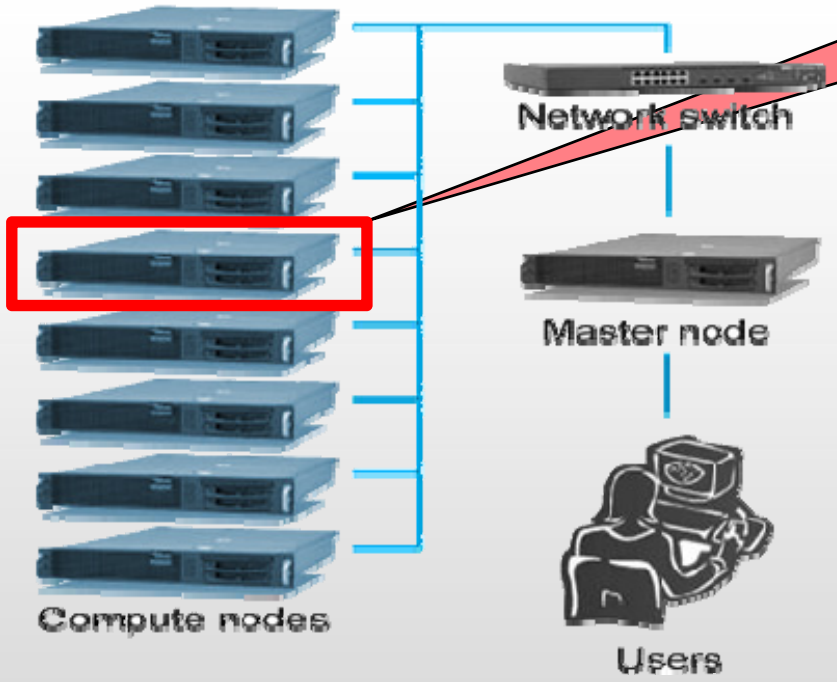
¿Puedo usar mi programa los dispositivos de interfaz humana comunes (monitor y teclado)?

¿En cuantos nodos se ejecuta mi ejecución?

¿Cuantos nodos ocupa mi ejecución?

Introducción a OpenMP

Simbólicamente nuestro modelo es:



SOLO 1 NODO DE CÓMPUTO
Nº máximo de procesos depende del número de cores

- Conexión remota a nodo de acceso
- En el nodo de acceso compilamos y lanzamos
- Nuestro programa se ejecuta en los nodos de cómputo

Analicemos:

¿Puedo usar mi programa los dispositivos de interfaz humana comunes (monitor y teclado)?

¿En cuantos nodos se ejecuta mi ejecución?

¿Cuantos nodos ocupa mi ejecución?

Introducción a OpenMP

Ejemplo consolidación #1

```
#include <omp.h>
int main()
{
    int iam = 0, np = 1;
    #pragma omp parallel num_threads(5)
    {                                   //Inicio Región Paralela
        printf("Hola Mundo\n");
    }                                   //Fin Región Paralela
}
```

¿Cuál será la salida de este programa?

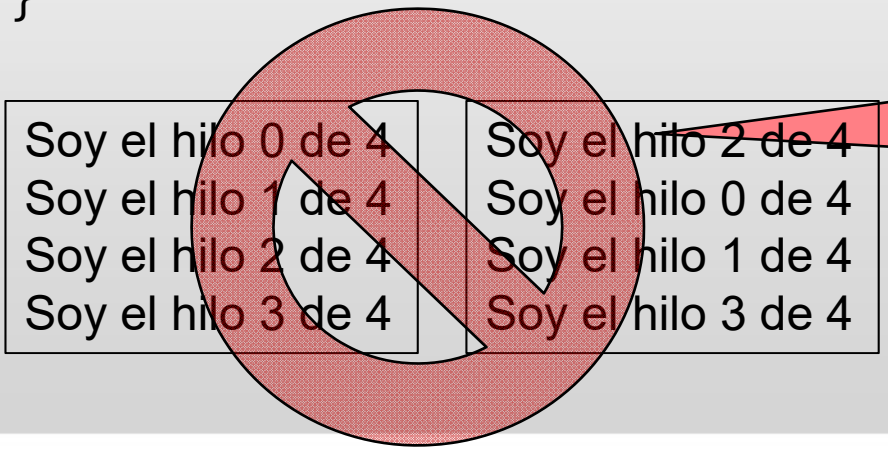
num_threads(5)	num_threads(4)	num_threads(c_nt)	num_threads(1)
Hola Mundo Hola Mundo Hola Mundo Hola Mundo Hola Mundo	Hola Mundo Hola Mundo Hola Mundo Hola Mundo	Hola Mundo Hola Mundo Hola Mundo Hola Mundo … Hola Mundo	Hola Mundo

Ejemplo consolidación #2

```
#include <omp.h>
int main()
{
    int iam = 99, np = 101;
    #pragma omp parallel num_threads(4)
    {
        np = omp_get_num_threads();
        iam = omp_get_thread_num();
        printf("Soy el hilo %d de %d hilos.\n",iam,np);
    }
}
```

¿Cuál será la salida de este programa?

¿Cuál es el valor de iam y np tras la región paralela?



NO puedo contestar si las variables (memoria) no está definidas como "compartidas" o "privadas"

Introducción a OpenMP

Ejemplo consolidación #2

```
#include <omp.h>
int main()
{
    int iam = 99, np = 101;
    #pragma omp parallel num_threads(4)
    {
        np = omp_get_num_threads();
        iam = omp_get_thread_num();
        printf("Soy el hilo %d de %d hilos.\n",iam,np);
    }
}
```

¿Cuál será la salida de este programa?

¿Cuál es el valor de iam y np tras la región paralela?

"iam" privada "np" privada	"iam" privada "np" compartida	"iam" compartida "np" privada	"iam" compartida "np" compartida
Soy el hilo 2 de 4 Soy el hilo 0 de 4 Soy el hilo 1 de 4 Soy el hilo 3 de 4	Soy el hilo 2 de 4 Soy el hilo 3 de 4 Soy el hilo 1 de 4 Soy el hilo 0 de 4	Soy el hilo 2 de 4 Soy el hilo 2 de 4 Soy el hilo 0 de 4 Soy el hilo 3 de 4	Soy el hilo 1 de 4 Soy el hilo 2 de 4 Soy el hilo 3 de 4 Soy el hilo 3 de 4

Introducción a OpenMP

Ejemplo consolidación #2

"iam" compartida

"np" privada

Soy el hilo 2 de 4
Soy el hilo 2 de 4
Soy el hilo 0 de 4
Soy el hilo 3 de 4

Proceso 0 Proceso 1 Proceso 2 Proceso 3

Antes de región paralela: iam=99 np=101

		Proceso 0	Proceso 1	Proceso 2	Proceso 3	Común
Instrucción	Proceso	np	np	np	np	iam
`np = omp_get_num_threads();` `np = omp_get_num_threads();` `np = omp_get_num_threads();` `np = omp_get_num_threads();`	0 1 2 3	4	4	4	4	99
`iam = omp_get_thread_num();` `iam = omp_get_thread_num();`	1 2	=	=	=	=	¿ 1 ó 2?
`printf("Soy el hilo %d de %d hilos.\n",iam,np);`	2	=	=	=	=	2
`printf("Soy el hilo %d de %d hilos.\n",iam,np);`	1	=	=	=	=	2
`iam = omp_get_thread_num();` `printf("Soy el hilo %d de %d hilos.\n",iam,np);`	0	=	=	=	=	0
`iam = omp_get_thread_num();` `printf("Soy el hilo %d de %d hilos.\n",iam,np);`	3	=	=	=	=	3

Soy el hilo 2 de 4

Soy el hilo 2 de 4

Soy el hilo 0 de 4

Soy el hilo 3 de 4

¿Cuál es el valor de iam y np tras la región paralela?

Introducción a OpenMP

Ejemplo consolidación #2

"iam" privada

"np" compartida

Soy el hilo 2 de 4
Soy el hilo 3 de 4
Soy el hilo 1 de 4
Soy el hilo 0 de 4

Proceso 0 Proceso 1 Proceso 2 Proceso 3

Antes de región paralela: iam=99 np=101

		Proceso 0	Proceso 1	Proceso 2	Proceso 3	Común
Instrucción	Proceso	iam	iam	iam	iam	Np
`np = omp_get_num_threads();` `np = omp_get_num_threads();`	2	?	?	?	?	¿4 ó 4?
`iam = omp_get_thread_num();` `np = omp_get_num_threads();`	1	?	1	?	?	¿4 ó 4?
`iam = omp_get_thread_num();`	3	?	=	?	3	4
`np = omp_get_num_threads();` `iam = omp_get_thread_num();`	0 2	?	=	2	=	¿4 ó 4?
`printf("Soy el hilo %d de %d hilos.\n",iam,np);`		?	=	=	=	4
`printf("Soy el hilo %d de %d hilos.\n",iam,np);`		?	=	=	=	4
`printf("Soy el hilo %d de %d hilos.\n",iam,np);`		?	=	=	=	4
`iam = omp_get_thread_num();` `printf("Soy el hilo %d de %d hilos.\n",iam,np);`		0	=	=	=	4

Soy el hilo 2 de 4
Soy el hilo 3 de 4
Soy el hilo 1 de 4
Soy el hilo 0 de 4

¿Cuál es el valor de iam y np tras la región paralela?

Introducción a OpenMP

En el desarrollo de aplicaciones paralelas en arquitecturas de memoria compartida es básico la gestión de las variables respecto a si se declaran como:

- Variables COMPARTIDAS (“shared”)
- Variables PRIVADAS: con posibilidad de diferentes comportamientos que varían el comportamiento visto hasta ahora

Recordemos que la memoria privada en escritura es más rápida ¿Por qué?

Modelo de memoria → CONSISTENCIA RELAJADA

(memoria compartida (variables globales))

- Hilo → visión de la memoria temporal (realmente nunca se trabaja directamente con la memoria principal)
- Hilo → almacenan las variables en caché (visión local de las variables “globales”)

En el ejemplo anterior cuando “np” se declaraba como “shared” supongamos que sólo un hilo escribe en “np” y a continuación vemos un ejemplo de funcionamiento real

Introducción a OpenMP

```
#include <omp.h>
int main()
{
        int iam = 99, np = 101;
        #pragma omp parallel num_threads(3) private(iam) shared(np)
        {
                #pragma omp single nowait //Sólo ejecuta el primer
                hilo
                {
                np = omp_get_num_threads();
                }

                iam = omp_get_thread_num();
                printf("Soy el hilo %d de %d hilos.\n",iam,np);
        }
}
```

El primer hilo que llega ejecuta la instrucción y guarda el valor correcto en SU visión local. El resto de hilos NO ejecuta dicha instrucción

Proceso 0		Proceso 1		Proceso 2		Común
instrucción	visión local (np)	inst.	visión local (np)	inst.	visión local (np)	np
	101	np=omp_get_num_threads();	3		101	101
	101	iam = omp_get_thread_num();	3			101
iam = omp_get_thread_num(); printf("Soy el hilo %d de %d hilos.\n",iam,np);	101					101
	101	printf("Soy el hilo %d de %d hilos.\n",iam,np);	3	iam = omp_get_thread_num();	101	3
	3		3		3	
	3		3	printf("Soy el hilo %d de %d hilos.\n",iam,np);	3	3

Otro hilo puede ejecutar antes de que se produzca el proceso de **"CONSISTENCIA DE MEMORIA"**

Se produce el proceso de **"CONSISTENCIA DE MEMORIA"**

Salida **"Soy el hilo 0 de 101"**

Salida **"Soy el hilo 2 de 3"**

Modelo de memoria → CONSISTENCIA RELAJADA

(memoria compartida (variables globales))

- Los procesos de CONSISTENCIA se realizan (por parte del sistema)…
- … pero no puedo asegurar que se producen exactamente en el momento que un hilo modifica una variable compartida
- Podré obligar a que se produzcan los procesos de consistencia

Introducción a OpenMP

```
#include <omp.h>
int main()
{
    int iam = 99, np = 101;
    #pragma omp parallel num_threads(3) private(iam) shared(np)
    {
        #pragma omp single nowait//Sólo ejecuta el primer hilo
        {
        np = omp_get_num_threads();
        }

        iam = omp_get_thread_num();

        printf("Soy el hilo %d de %d hilos.\n",iam,np);
    }
}
```

En este punto debería obligar a que se produjera el proceso de consistencia

Directivas OpenMP

Tras tener una visión global vamos a ver las directivas y cláusulas más importantes, que se pueden englobar en los siguientes grupos:

- Constructores paralelos
  - No automatizados
  - Automatizados
- Ejecución serializada y/o exclusiva
- Sincronización y/o consistencia de memoria
- Gestión de memoria

Directivas OpenMP

Como se ha visto las directivas se especifican mediante:

#pragma omp *directiva [clausulas]*

{

//Afecta al código englobado entre {}

}

- Las cláusulas son opcionales y puede haber más de una
- Si no hay llaves afecta únicamente a la sentencia inmediatamente inferior a la directiva

```
#include <omp.h>
int main()
{
	int iam = 99, np = 101;
	#pragma omp parallel num_threads(3) private(iam) shared(np)
	{
		#pragma omp single nowait //Sólo ejecuta el primer
		hilo
		{
		np = omp_get_num_threads();
		}

		iam = omp_get_thread_num();
		printf("Soy el hilo %d de %d hilos.\n",iam,np);
	}
}
```

```
#include <omp.h>
int main()
{
	int iam = 99, np = 101;
	#pragma omp parallel num_threads(3) private(iam) shared(np)
	{
		#pragma omp single nowait //Sólo ejecuta el primer hilo
		np = omp_get_num_threads();


		iam = omp_get_thread_num();
		printf("Soy el hilo %d de %d hilos.\n",iam,np);
	}
}
```

Directivas OpenMP

CONSTRUCTOR **PARALLEL**

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

- Se crea un grupo de threads, cuyo número está especificado por (según orden de preferencia):
  1. Cláusula "num_threads()" (el valor entre paréntesis puede ser una constante o una variable de tipo entero)
  2. o dinámicamente
  3. o por variable de entorno OPENMP_NUM_THREADS

- El que los pone en marcha actúa de maestro (identificador = 0 y más efectos que se verán en adelante)

- El código del bloque NO tiene restricciones y es ejecutado al completo por todos los procesos (aunque cada proceso puede seguir una secuencia de ejecución diferente)

- Hay barrera de sincronización implícita al final de la región.

CONSTRUCTOR **PARALLEL**

- Permite la cláusula if (“if (variable)”):

- Cuando dentro de una región hay otro constructor paralelo genero una región anidada. En la región anidada el master de esa región interna es el que la ha generado

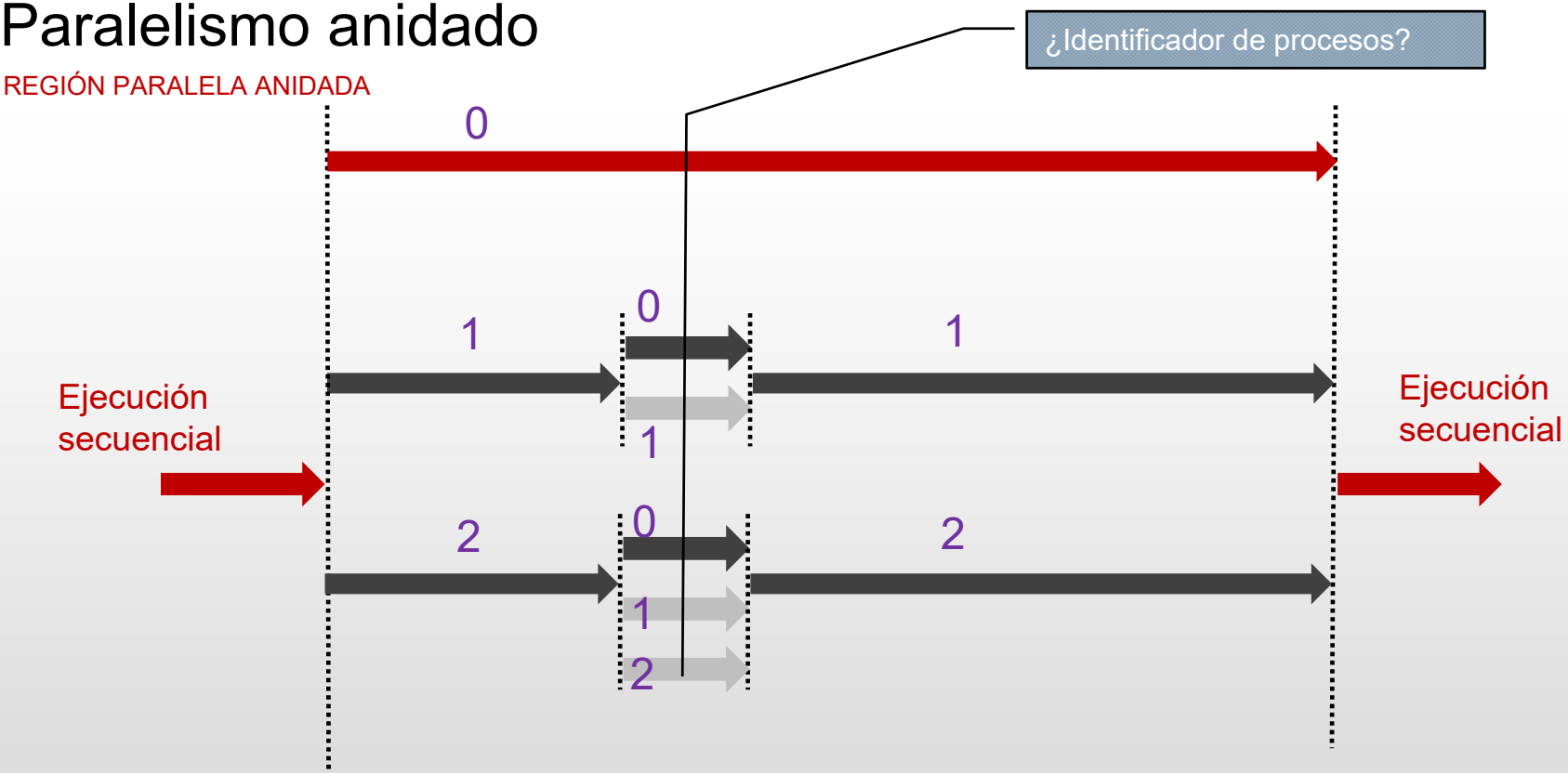
```
#include <omp.h>
int main()
{
     int iam = 99, np = 101;
     #pragma omp parallel num_threads(3) private(iam,np)
     {
          np = omp_get_num_threads();
          iam = omp_get_thread_num();
          if (iam >= 1)
               #pragma omp parallel   num_threads(iam+1)

               {

               }
     }
}
```

INTRODUCCION OPENMP

Paralelismo anidado



Directivas OpenMP

CONSTRUCTOR AUTOMATIZADOR **FOR**

```
#pragma omp for [cláusulas]
bucle for (imprescindible)
```

Paraleliza automáticamente un bucle for = Distribuye automáticamente el trabajo a realizar en un bucle for

```
#include <omp.h>
#include <stdio.h>
int main()
{
double* datos;
…
#pragma omp parallel num_threads(4) shared(datos)
   {
        …
        #pragma omp for
            for(i=0;i<10000;i++)
             {
               función_procesamiento(&datos[i]);
             }
        …
    }
}
```

	Ejemplo 1			
valores de i	Proceso 0	Proceso 1	Proceso 2	Proceso 3
inicial	0	2500	5000	7500
final	2500	5000	7500	10000

	Ejemplo 2			
valores de i	Proceso 0	Proceso 1	Proceso 2	Proceso 3
inicial	0	1250	2500	3750
final	1250	2500	3750	5000
inicial	5000	6250	7500	8750
final	6250	7500	8750	10000

CONSTRUCTOR AUTOMATIZADOR **FOR**

```
#include <omp.h>
#include <stdio.h>
int main()
{
double* datos;
…
#pragma omp parallel num_threads(3) shared(datos)
   {
        …
        // #pragma omp for
           for(i=0;i<10000;i++)
            {
              función_procesamiento(&datos[i]);
            }
        …
    }
}
```

	Ejemplo 2		
valores de i	Proceso 0	Proceso 1	Proceso 2
inicial	0	0	0
final	10000	10000	10000

Directivas OpenMP

```
#pragma omp for [cláusulas]
bucle for (imprescindible)
```

CONSTRUCTOR AUTOMATIZADOR **FOR**

- Debe estar DENTRO de una región paralela
- No admite {}, la siguiente instrucción debe ser un for
- El for debe cumplir que:
  - La parte de inicialización del for debe ser una asignación
  - La parte de incremento debe ser una suma o resta
  - La parte de evaluación es la comparación de una variable entera sin signo con un valor, utilizando un comparador mayor o menor (puede incluir igual).
  - Los valores que aparecen en las tres partes del for deben ser enteros.
  - No admite roturas de bucle (“break”)
- Hay barrera al final del for (hasta que no acaban todos ninguno continúa)
- Admite clausulas (entre otras la necesarias para decidir el reparto de trabajo)
- A nivel conceptual las iteraciones deben ser INDEPENDIENTES

Directivas OpenMP

Ejemplo consolidación #3

CONSTRUCTOR AUTOMATIZADOR **FOR**

```
#include <omp.h>
#include <stdio.h>
int main()
{
   int iam =0, var;

#pragma omp parallel num_threads(var) private(iam,i)
{
    iam = omp_get_thread_num();

    #pragma omp for
    for(i=0;i<10;i++)
     {
       printf("Hilo %d, realiza iteracion  %d \n",iam,i);
     }
}
}
```

¿Cuál es el la salida del ejemplo?

```
Hilo ? realiza iteración 0
Hilo ? realiza iteración 1
Hilo ? realiza iteración 2
Hilo ? realiza iteración 3
Hilo ? realiza iteración 4
Hilo ? realiza iteración 5
Hilo ? realiza iteración 6
Hilo ? realiza iteración 7
Hilo ? realiza iteración 8
Hilo ? realiza iteración 9
```

Directivas OpenMP

Ejemplo consolidación #3

CONSTRUCTOR AUTOMATIZADOR **FOR**

```
#include <omp.h>
#include <stdio.h>
int main()
{
   int iam =0, var;

#pragma omp parallel num_threads(2) private(iam,i)
{
    iam = omp_get_thread_num();

    #pragma omp for
    for(i=0;i<10;i++)
     {
       printf("Hilo %d, realiza iteración  %d \n",iam,i);
     }
}
}
```

¿Cuál es el la salida del ejemplo?

```
Hilo 0 realiza iteración 0
Hilo 1 realiza iteración 5
Hilo 1 realiza iteración 6
Hilo 0 realiza iteración 1
Hilo 1 realiza iteración 7
Hilo 0 realiza iteración 2
Hilo 1 realiza iteración 8
Hilo 0 realiza iteración 3
Hilo 0 realiza iteración 4
Hilo 1 realiza iteración 9
```

CONSTRUCTOR AUTOMATIZADOR **FOR**

Clausula de reparto de trabajo SCHEDULE (específica del constructor for)

- **schedule(static,**tamaño) las iteraciones se dividen según el tamaño, y la asignación se hace estáticamente a los threads. Si no se indica el tamaño se divide por igual entre los threads.

- **schedule(dynamic**,tamaño) las iteraciones se dividen según el tamaño y se asignan a los threads dinámicamente cuando van acabando su trabajo.

- **schedule(guided**,tamaño) las iteraciones se asignan dinámicamente a los threads pero con tamaños decrecientes, empezando en tamaño num_iter/np (para el primer proceso libre), después num_iter_restantes/np, el número de iteraciones a asignar no puede ser inferior a “tamaño”.

- **schedule(runtime)** deja la decisión para el tiempo de ejecución, y se obtiene de la variable de entorno OMP_SCHEDULE, que puede cambiarse en ejecución.

- **schedule(auto)** se deja la decisión al compilador

CONSTRUCTOR **FOR** – SCHEDULE – STATIC

```
#include <omp.h>
#include <stdio.h>
int main()
{
double* datos;
…
#pragma omp parallel num_threads(4) shared(datos)
   {
        …
        #pragma omp for
            for(i=0;i<10000;i++)
            {
              función_procesamiento(&datos[i]);
            }
        …
    }
}
```

schedule(static,2500)

	Ejemplo 1			
valores de i	Proceso 0	Proceso 1	Proceso 2	Proceso 3
inicial	0	2500	5000	7500
final	2500	5000	7500	10000

schedule(static,1250)

	Ejemplo 2			
valores de i	Proceso 0	Proceso 1	Proceso 2	Proceso 3
inicial	0	1250	2500	3750
final	1250	2500	3750	5000
inicial	5000	6250	7500	8750
final	6250	7500	8750	10000

schedule(static,9000)

	Ejemplo 4			
valores de i	Proceso 0	Proceso 1	Proceso 2	Proceso 3
inicial	0	9000		
final	9000	10000		

schedule(static,950)

	Ejemplo 3			
valores de i	Proceso 0	Proceso 1	Proceso 2	Proceso 3
inicial	0	950	1900	2850
final	950	1900	2850	3800
Inicial	3800	4750	5700	6650
Final	4750	5700	6650	7600
Inicial	7600	8550	9500	
Final	8550	9500	10000	

CONSTRUCTOR FOR – SCHEDULE – DYNAMIC

```
#include <omp.h>
#include <stdio.h>
int main()
{
double* datos;
…
#pragma omp parallel num_threads(4) shared(datos)
   {
        …
        #pragma omp for
            for(i=0;i<10000;i++)
             {
               función_procesamiento(&datos[i]);
             }
        …
    }
}
```

schedule(dynamic,2500)

	Ejemplo 1			
valores de i	Proceso 0	Proceso 1	Proceso 2	Proceso 3
inicial	0	2500	5000	7500
final	2500	5000	7500	10000

	Ejemplo 2			
valores de i	Proceso 0	Proceso 1	Proceso 2	Proceso 3
inicial	2500	7500	0	5000
final	5000	10000	2500	7500

	Ejemplo 3			
valores de i	Proceso 0	Proceso 1	Proceso 2	Proceso 3
inicial	2500		0	5000
final	5000		2500	7500
inicial	7500			
final	10000			

CONSTRUCTOR FOR – SCHEDULE – GUIDED

schedule(guided,500)

```
#include <omp.h>
#include <stdio.h>
int main()
{
double* datos;
…
#pragma omp parallel num_threads(4) shared(datos)
   {
        …
        #pragma omp for
            for(i=0;i<10000;i++)
             {
               función_procesamiento(&datos[i]);
             }
        …
    }
}
```

	Ejemplo			
	Iteraciones Restantes	Bloque asignar	Inicial	Final
Paso 1	10000	2500	0	2500
Paso 2	7500	1875	2500	4375
Paso 3	5625	1406	4375	5781
Paso 4	4219	1054	5781	6835
Paso 5	3165	791	6835	7626
Paso 6	2374	593	7626	8219
Paso 7 (NO)	1781	445	8219	8664
Paso 7		500	8219	8719
Paso 8		500	8719	9219
Paso 9		500	9219	9719
Paso 10 (NO)		500	9719	10219
Paso 10		281	9719	10000

	Ejemplo			
valores de i	Proceso 0	Proceso 1	Proceso 2	Proceso 3
inicial	0	4375	2500	5781
final	2500	5781	4375	6835
inicial	8219	9219	7626	6835
final	8719	9719	8219	7626
inicial	9719			8719
final	10000			9219
inicial				
final				

CONSTRUCTOR AUTOMATIZADOR **SECTION**

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

- Cada sección se ejecuta por un thread.
- Hay barrera al final de “sections”.
- Hay una serie de cláusulas (**private, firstprivate, lastprivate, …**) para indicar la forma en que se accede a las variables
- Es un AUTOMATIZADOR con especial relevancia ¿para qué tipo de descomposición?

CONSTRUCTOR AUTOMATIZADOR **SECTION**

```
#pragma omp parallel num_threads(2) private(iam)
{
    iam = omp_get_thread_num();
    #pragma omp sections
    {
        #pragma omp section
        {
        printf("El thread %d realiza la seccion 1. \n",iam);
        for (i=0;i<100000;i++) datos[0]++;
        }
        #pragma omp section
        printf("El thread %d realiza la seccion 2. \n",iam);
        #pragma omp section
        printf("El thread %d realiza la seccion 3. \n",iam);
        #pragma omp section
        printf("El thread %d realiza la seccion 4. \n",iam);
        #pragma omp section
        printf("El thread %d realiza la seccion 5. \n",iam);
    }//sections
}//parallel
```

¿Cuántos “printf”?

¿Depende del número de hilos?

¿Qué hilo realiza cada “section”?

Directivas OpenMP

CONSTRUCTOR **PARALLEL** COMBINADO CON **FOR** y **SECTIONS**

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

- Es forma abreviada de directiva **parallel**, cuya región está compuesta por únicamente un **for paralelo.** Admite las **mismas** cláusulas (menos **nowait**).

```
#pragma omp parallel sections [cláusulas]
```

- Es forma abreviada de directiva **parallel**, cuya región está compuesta por únicamente una región **sections.** Admite las **mismas** cláusulas (menos **nowait**).

CONSTRUCTOR EJECUCIÓN SECUENCIAL SINGLE

```
#pragma omp single [cláusulas]
bloque
```

- El bloque se ejecuta por un único thread.
- No tiene por qué ser el maestro.
- Hay barrera* al final a no ser que se utilice la cláusula nowait.
- Debe encontrarse dentro de una región paralela

* No hemos visto como implementar barreras pero el concepto ya debe estar claro

CONSTRUCTOR EJECUCIÓN SECUENCIAL SINGLE

```
int main()
{
   int iam =0, np = 1, suma=0;
   int datos[4],i = 0, j = 0;
   //datos → i * 10
#pragma omp parallel num_threads(4) shared(datos,suma) private(iam,np,i)
{
    iam = omp_get_thread_num();
    np = omp_get_num_threads();
    #pragma omp single
    {
    printf ("El trabajo SINGLE solo lo realiza el hilo: %d\n",iam);   **
    suma = 0;
    for (i=0;i<4;i++){
        suma += datos[i];
        }
    }
    printf ("Despues de  SINGLE el valor de suma = %d para el hilo: %d\n",suma,iam);
}//parallel
```

¿Cuál será la salida para 3 hilos?

¿Puedo determinar la salida de la línea **?

CONSTRUCTOR EJECUCIÓN SECUENCIAL SINGLE

```
int main()
{
   int iam =0, np = 1, suma=0;
   int datos[4],i = 0, j = 0;
   //datos → i * 10
#pragma omp parallel num_threads(4) shared(datos,suma) private(iam,np,i)
{
    iam = omp_get_thread_num();
    np = omp_get_num_threads();
    #pragma omp single nowait
    {
    printf ("El trabajo SINGLE solo lo realiza el hilo: %d\n",iam);   **
    suma = 0;
    for (i=0;i<4;i++){
        suma += datos[i];
        }
    }
    printf ("Despues de  SINGLE el valor de suma = %d para el hilo: %d\n",suma,iam);
}//parallel
```

La clausula NOWAIT no es exclusiva del constructor *single*. El funcionamiento será el mismo en aquellos constructores que lo admitan

¿Qué ha cambiado respecto al ejemplo anterior?

CONSTRUCTOR EJECUCIÓN SECUENCIAL MASTER

```
#pragma omp master [cláusulas]
bloque
```

- El bloque se ejecuta por un único thread.
- El thread que ejecuta el código es el master (rango = 0).
- NO hay ninguna barrera implícita en el constructor
- Debe encontrarse dentro de una región paralela

CONSTRUCTOR EJECUCIÓN SECUENCIAL MASTER

```
int main()
{
   int iam =0, np = 1, suma=0;
   int datos[4],i = 0, j = 0;
   //datos → i * 10
#pragma omp parallel num_threads(4) shared(datos,suma) private(iam,np,i)
{
    iam = omp_get_thread_num();
    np = omp_get_num_threads();
    #pragma omp master
    {
    printf ("El trabajo MASTER solo lo realiza el hilo: %d\n",iam);   **
    suma = 0;
    for (i=0;i<4;i++){
        suma += datos[i];
        }
    }
    printf ("Despues de  MASTER el valor de suma = %d para el hilo: %d\n",suma,iam);
}//parallel
```

¿Cuál será la salida para 3 hilos?

¿Puedo determinar la salida de la línea **?

CONSTRUCTOR EJECUCIÓN SECUENCIAL ORDERED

```
#pragma omp ordered [NO cláusulas]
bloque
```

- El bloque se ejecuta en orden secuencial.
- El trabajo por tanto se serializa (penaliza la eficiencia)
- El constructor debe estar dentro de un bucle **for**
- El constructor debe estar dentro de un bucle **for ordenado,** debe incluir la cláusula *ordered*

CONSTRUCTOR EJECUCIÓN SECUENCIAL ORDERED

```
int main()
{
   int iam =0, np = 1;
   int datos[100],i = 0, j = 0;

#pragma omp parallel private(iam, np,i)
{
    iam = omp_get_thread_num();
    #pragma omp for ordered
    for(i=0;i<5;i++)
    {
        printf("\tSoy el thread %d, antes del ordered en la iteracion %d\n",iam,i);
      #pragma omp ordered
      {
        printf("\t\tSoy el thread %d, actuando en la iteracion %d\n",iam,i);
        sleep(1);
      }
      printf("\tSoy el thread %d, despues del ordered en la iteracion %d\n",iam,i);
    }
}//parallel
}
```

¿Qué puedo asegurar de la ejecución paralela?

CONSTRUCTOR EJECUCIÓN SECUENCIAL ORDERED

```
int main()
{
   int iam =0, np = 1;
   int datos[100],i = 0, j = 0;

#pragma omp parallel private(iam, np,i)
{
    iam = omp_get_thread_num();
    #pragma omp for ordered
    for(i=0;i<5;i++)
    {
      #pragma omp ordered
       {
         printf("\t\tSoy el thread %d, actuando en la iteracion %d\n",iam,i);
         sleep(1);
       }
}
}//parallel
}
```

¿Voy a obtener algún beneficio temporal?

Viendo el ejemplo vemos que ordered se comporta como un constructor y como una cláusula

CONSTRUCTOR EJECUCIÓN SECUENCIAL CRITICAL

```
#pragma omp critical [name]
bloque
```

- El bloque es ejecutado por todos los procesos

- Implica exclusión mutua en la ejecución, es decir solo un proceso puede estar en ejecución en un momento dado de la sección crítica

- Puede usarse un nombre para identificar la sección crítica y que la exclusión mutua no se aplique a diferentes regiones críticas con diferentes identificadores (nombres)

CONSTRUCTOR EJECUCIÓN SECUENCIAL CRITICAL

```
int main()
{
   int iam =0, np = 1;
   int dato,i = 0, suma=0;

#pragma omp parallel num_threads(4) private(iam,np,i,dato) shared(suma)
{
    iam = omp_get_thread_num();
    dato = iam;
    #pragma omp critical
      {
        suma+=dato;
      }
     printf("\tSoy el thread %d, valor actual de suma: %d\n",iam,suma);
}//parallel
printf("\t\tSoy el thread %d, valor FINAL de suma: %d\n",iam,suma);
}
```

¿Cuál es la salida de dentro la región paralela?

¿Cuál es la salida tras la región paralela?

¿Depende del número de hilos?

CONSTRUCTOR EJECUCIÓN SECUENCIAL CRITICAL

```
int main()
{
   int iam =0, np = 1;
   int dato,i = 0, suma=0;

#pragma omp parallel num_threads(4) private(iam,np,i,dato) shared(suma)
{
    iam = omp_get_thread_num();
    dato = iam;
    //#pragma omp critical
       {
         suma+=dato;
       }
     printf("\tSoy el thread %d, valor actual de suma: %d\n",iam,suma);
}//parallel
printf("\t\tSoy el thread %d, valor FINAL de suma: %d\n",iam,suma);
}
```

¿En qué se diferencia del caso anterior?

CONSTRUCTOR DE EJECUCIÓN SECUENCIAL ATOMIC

```
#pragma omp atomic
x bin_op = expr
```

Afecta a una solo expresión de tipo escalar
(bin_op → + * - / & ^ | << >>
expr → ++ --)

Actualiza la memoria atómicamente, es decir antes de leer y tras efectuar la operación de escritura.

El efecto es análogo al *critical* pero con un overhead muy inferior

Sólo una variable puede estar en la expresión

CONSTRUCTOR EJECUCIÓN SECUENCIAL ATOMIC

```
int main()
{
   int iam =0, np = 1;
   int dato,i = 0, suma=0;

#pragma omp parallel num_threads(4) private(iam,np,i,dato) shared(suma)
{
    iam = omp_get_thread_num();
    dato = iam;
    #pragma omp atomic
      {
        suma+=dato;
      }
     printf("\tSoy el thread %d, valor actual de suma: %d\n",iam,suma);
}//parallel
printf("\t\tSoy el thread %d, valor FINAL de suma: %d\n",iam,suma);
}
```

Hay dos variables

CONSTRUCTOR EJECUCIÓN SECUENCIAL ATOMIC

```
int main()
{
   int iam =0, np = 1;
   int dato,i = 0, suma=0;

#pragma omp parallel num_threads(4) private(iam,np,i,dato) shared(suma)
{
    iam = omp_get_thread_num();
    dato = iam;
    #pragma omp atomic
      {
        suma++;
      }
     printf("\tSoy el thread %d, valor actual de suma: %d\n",iam,suma);
}//parallel
printf("\t\tSoy el thread %d, valor FINAL de suma: %d\n",iam,suma);
}
```

CUIDADO: el funcionamiento exacto puede depender del compilador

CONSTRUCTOR DE SINCRONIZACIÓN BARRIER

```
#pragma omp barrier
```

- Es una barrera cuyo objetivo es la sincronización
- Los hilos se paran al llegar a ella y …
- … continúan cuando ya han llegado todos
- Además asegura una visión consistente de la memoria (incluye implícitamente un proceso de consistencia)

CONSTRUCTOR DE SINCRONIZACIÓN BARRIER

```
int main()
{
   int iam =0, np = 1;
   int datos[100],i = 0, j = 0;
   int suma[4];
   int tam_bloque,ini,fin,suma_global=0;
   for(i=0;i<tam;i++) datos[i] = i;
#pragma omp parallel num_threads(4) private(iam,np,i,ini,fin) shared(j,suma,datos,suma_global)
{
    iam = omp_get_thread_num();
    np = omp_get_num_threads();
    tam_bloque = tam/np;  ini = tam_bloque * iam;  fin = tam_bloque * (iam + 1); suma[iam] = 0;

    for(i=ini;i<fin;i++)   {suma[iam] += datos[i]; }



}//parallel
}
```

Variables

Inicialización de datos

CONSTRUCTOR DE SINCRONIZACIÓN BARRIER

```
int main()
{
   int iam =0, np = 1;
   int datos[100],i = 0, j = 0;
   int suma[4];
   int tam_bloque,ini,fin,suma_global=0;
   for(i=0;i<tam;i++) datos[i] = i;
#pragma omp parallel num_threads(4) private(iam,np,i,ini,fin) shared(j,suma,datos,suma_global)
{
    iam = omp_get_thread_num();
    np = omp_get_num_threads();
    tam_bloque = tam/np;  ini = tam_bloque * iam;  fin = tam_bloque * (iam + 1); suma[iam] = 0;

    for(i=ini;i<fin;i++)   {suma[iam] += datos[i]; }

}
```

Distribución de trabajo como en MPI

Trabajo como en MPI, valores parciales

CONSTRUCTOR DE SINCRONIZACIÓN BARRIER

```
int main()
{
   int iam =0, np = 1;
   int datos[100],i = 0, j = 0;
   int suma[4];
   int tam_bloque,ini,fin,suma_global=0;
   for(i=0;i<tam;i++) datos[i] = i;
#pragma omp parallel num_threads(4) private(iam,np,i,ini,fin) shared(j,suma,datos,suma_global)
{
    iam = omp_get_thread_num();
    np = omp_get_num_threads();
    tam_bloque = tam/np;  ini = tam_bloque * iam;  fin = tam_bloque * (iam + 1); suma[iam] = 0;

    for(i=ini;i<fin;i++)   {suma[iam] += datos[i]; }
```

¿Cuántas veces se hace?

¿Cuántas veces hay que hacerlo?

```
      for (i=0;i<np;i++) suma_global+= suma[i];
```

Reducción de valores parciales

```
    printf("Valor final de suma global %d (hilo: %d).\n",suma_global,iam);
}//parallel
}
```

CONSTRUCTOR DE SINCRONIZACIÓN BARRIER

```
int main()
{
   int iam =0, np = 1;
   int datos[100],i = 0, j = 0;
   int suma[4];
   int tam_bloque,ini,fin,suma_global=0;
   for(i=0;i<tam;i++) datos[i] = i;
#pragma omp parallel num_threads(4) private(iam,np,i,ini,fin) shared(j,suma,datos,suma_global)
{
    iam = omp_get_thread_num();
    np = omp_get_num_threads();
    tam_bloque = tam/np;  ini = tam_bloque * iam;  fin = tam_bloque * (iam + 1); suma[iam] = 0;

    for(i=ini;i<fin;i++)   {suma[iam] += datos[i]; }


 #pragma omp master
{
       for (i=0;i<np;i++) suma_global+= suma[i];
 }
 printf("Valor final de suma global %d (hilo: %d).\n",suma_global,iam);
}//parallel
}
```

¿Qué valores está sumando?

¿Almacenan su valor final correcto?

CONSTRUCTOR DE SINCRONIZACIÓN BARRIER

```
int main()
{
   int iam =0, np = 1;
   int datos[100],i = 0, j = 0;
   int suma[4];
   int tam_bloque,ini,fin,suma_global=0;
   for(i=0;i<tam;i++) datos[i] = i;
#pragma omp parallel num_threads(4) private(iam,np,i,ini,fin) shared(j,suma,datos,suma_global)
{
    iam = omp_get_thread_num();
    np = omp_get_num_threads();
    tam_bloque = tam/np;  ini = tam_bloque * iam;  fin = tam_bloque * (iam + 1); suma[iam] = 0;

    for(i=ini;i<fin;i++)   {suma[iam] += datos[i]; }

 #pragma omp barrier
 #pragma omp master
{
       for (i=0;i<np;i++) suma_global+= suma[i];
 }
 printf("Valor final de suma global %d (hilo: %d).\n",suma_global,iam);
}//parallel
}
```

Directivas OpenMP

CONSTRUCTOR DE ACTUALIZACIÓN DE MEMORIA FLUSH

```
#pragma omp flush [lista]
```

Visión local se actualiza al valor correcto

- Asegura que las variables de la lista son actualizadas para todos los threads que ejecutan el flush.

- Si no se especifica la lista se actualizan todas las variables compartidas

- El flush existe implícito en otras sentencias, final de barrier, entrada y salida de región *critical* o región *ordered*, al salir de parallel, for, sections, single,…

- El flush puede ser ejecutado por un único hilo y en ese caso o trasmite sus cambios a las variables globales (no a la visión local de los otros) o actualiza su visión local de la variable global.

(En cada caso habrá que comprobar si existe, en caso de no existir y necesitarse se incluirá por código)

Directivas OpenMP

CONSTRUCTOR DE ACTUALIZACIÓN DE MEMORIA FLUSH

```
int main()
{
   int iam =0, np = 1;
   int datos[100],i = 0, j = 0;
   for (i=0;i<5;i++) datos[i]=0;
#pragma omp parallel num_threads(4) private(iam, np,i) shared(j,datos)
{
    iam = omp_get_thread_num();
    datos[iam] = 10*iam +1;
    #pragma omp flush(datos)
    printf("\tSoy el thread %d. Array: %d %d %d%d\n",iam,datos[0],datos[1],datos[2],datos[3]);
}//parallel
}
```

Si al ejecutar el printf cada hilo seguro que imprime la información de todos los hilos que ya han escrito.

Cada hilo realiza el flush en momentos distintos

Cláusulas OpenMP

CLÁUSULAS DE ALCANCE DE DATOS PRIVATE

private (lista): no se inicializan al entrar en la región paralela ni se guarda su valor al final

```
int main()
{
  int iam =22, np = 33;
#pragma omp parallel num_threads(4) private(np,iam)
{
    #pragma omp single
    printf("\t\tDENTRO valor de iam y np: %d -  %d\n",iam,np);

    iam = omp_get_thread_num();
    np = omp_get_num_threads();
    printf("\tSoy el thread %d de un grupo de %d\n",iam,np);
}//parallel
printf("\t\tFUERA valor de iam y np: %d -  %d\n",iam,np);
}
```

Cláusulas OpenMP

CLÁUSULAS DE ALCANCE DE DATOS SHARED

- shared (lista): las variables incluidas en lista son variables globales, no se crea ninguna variable, el espacio de memoria es el original.

Todos los hilos trabajan con el mismo espacio de memoria, hay que controlar los accesos a dichas variable en escritura (critical, ordered,…) y en lectura (flush). NUNCA se puede permitir escrituras que se puedan producir simultáneamente

- default (shared | none): indica como se tratarán las variables para las que no se especifique su comportamiento

El uso de none deja sin especificar variables, que podrán ser especificadas por otro medio o llegar a causar error en ejecución (por no "existir" en la región paralela)

Cláusulas OpenMP

CLÁUSULAS DE ALCANCE DE DATOS FIRSTPRIVATE

- firstprivate (lista): se inicializan al entrar en la región paralela con el valor de la varible. Pero NO se guarda su valor al final.

```
int main()
{
   int iam, np=0;
   int fp=14;
#pragma omp parallel num_threads(4) firstprivate(fp) private(np,iam) default(none)
{
    iam = omp_get_thread_num();
    np = omp_get_num_threads();

    fp += ++iam;

    printf("\t\tDENTRO valos de fp: %d - para  %d\n",fp,iam);

}//parallel
printf("\t\tFUERA valor de fp: %d - para  %d\n",fp,np);
}
```

Cláusulas OpenMP

CLÁUSULAS DE ALCANCE DE DATOS LASTPRIVATE

- **lastprivate (lista):** se aplica a bucles for paralelos o a sections.

La variable se actualiza al finalizar la estructura paralela en la que se englobe

La variable es tipo shared pero en una región se desea el comportamiento parecido a private.

Cláusulas OpenMP

CLÁUSULAS DE ALCANCE DE DATOS LASTPRIVATE

```
int main()
{
   int iam, np=0, i;
   int lp=14;
#pragma omp parallel num_threads(4)  private(np,iam,i) shared(lp) default(none)
{
    iam = omp_get_thread_num();
    np = omp_get_num_threads();

    #pragma omp for schedule(static,250) lastprivate(lp)
    for (lp=0;lp<1000;lp++){
        i++;
        if((iam==2) && (lp==748)){
           printf("A punto de acabar\n");
           sleep(5);
        }
    }

    printf("\t\tDENTRO valos de lp: %d - para  %d\n",lp,iam);

}//parallel
printf("\t\tFUERA valor de lp: %d - para  %d\n",lp,np);
}
```

Cláusulas OpenMP

CLÁUSULAS DE ALCANCE DE DATOS COPYPRIVATE

- copyprivate (lista): aplicado a variables privadas (para hacer algo que hemos dicho que no se podía)

Copia el contenido de una variable privada de un hilo al resto de hilos

Asociado a estructura single

La estructura single decide el hilo que copia el valor, es decir el que ejecuta la sección single (normalmente ese hilo modifica el valor dentro de la región single, ¿una región single incluye un barrier?)

Cláusulas OpenMP

CLÁUSULAS DE ALCANCE DE DATOS COPYPRIVATE

```
int main()
{
   int iam, np=0, i, tp;

#pragma omp parallel num_threads(4)  private(np,iam,i,tp)  default(none)
{
    iam = omp_get_thread_num();
    np = omp_get_num_threads();
    tp = (iam + 10) * 2;
    printf("\tDENTRO valor de tp: %d - en proceso %d\n",tp,iam);

    #pragma omp single copyprivate(tp)
     {
     tp = (iam + 1) * 100;
     printf("\tDENTRO SINGLE valor de tp: %d - en proceso %d\n",tp,iam);
     }

    printf("\tTRAS SINGLE valor de tp: %d - en proceso %d\n",tp,iam);


}//parallel
}
```

Cláusulas OpenMP

CLÁUSULAS DE ALCANCE DE DATOS THREADPRIVATE

threadprivate (lista):

Las variables son privadas en cada hilo

Se crean sólo para los hilos “esclavos”

Se copia el contenido de la variable del “root” al resto

El valor final de la variable del “root” permanece al acabar la región paralela (la variable no se ha creado y no se destruye)

La variable debe ser global

Cláusulas OpenMP

CLÁUSULAS DE ALCANCE DE DATOS THREADPRIVATE

```
int fp=14;

int main()
{
   int iam, np=0;
#pragma omp threadprivate(fp)
#pragma omp parallel num_threads(4)  private(np,iam) default(none)
{
    iam = omp_get_thread_num();
    np = omp_get_num_threads();

    fp += ++iam;

    printf("\t\tDENTRO valos de fp: %d - para  %d\n",fp,iam);

}//parallel
printf("\t\tFUERA valor de fp: %d - para  %d\n",fp,np);
}
```

Cláusulas OpenMP

CLÁUSULAS DE ALCANCE DE DATOS COPYIN

copyin (lista): se aplica a regiones paralelas (que pueden incluir for o section)

Copia el contenido de una variable (que tiene que ser threadprivate) a todos los hilos (nada nuevo de momento porque eso ya lo hacía threadprivate)

Ver en el ejemplo dos regiones paralelas anidadas, en la primera si se copia pero en la segunda depende de esta cláusula (es la única opción en la región anidada interna)

Cláusulas OpenMP

CLÁUSULAS DE ALCANCE DE DATOS COPYIN

```
int tp=14;

int main()
{
  int iam, np=0, i;
#pragma omp threadprivate(tp)

#pragma omp parallel num_threads(2)  private(np,iam,i)  default(none)
{
   iam = omp_get_thread_num();
   np = omp_get_num_threads();
   omp_set_nested(1);
   printf("\t\tDENTRO 1 valor de tp: %d - para  %d\n",tp,iam);
   #pragma omp master
     tp=23;
   #pragma omp parallel num_threads(2) copyin(tp)
   printf("\t\tDENTRO 2 valor de tp: %d - para  %d\n",tp,iam);

}//parallel
printf("\t\tFUERA valor de tp: %d - para  %d\n",tp,np);
}
```

USAR NUEVO EJEMPLO Y NUEVA EXPLICACION QUE ESTA EN SERVIDOR RANDALL

Cláusulas OpenMP

CLÁUSULAS DE ALCANCE DE DATOS COPYIN

```
#include <omp.h>
#include <stdio.h>
int fp=14;
int main(){

                int iam=1, np=5,i;
#pragma omp threadprivate(fp)
                printf("\t\tP0: %d - %d  - %d\n",fp,iam,i);

for (i=0;i<3;i++)
      {
#pragma omp parallel num_threads(i+3) private(np,iam) shared(i) default(none) copyin(fp)
                {
                        iam = omp_get_thread_num();
                        np = omp_get_num_threads();
                        printf("\t\tHilo %d Antes: %d - %d  \n",iam,fp,i);
                        fp += (iam+1)*10;
                        printf("\t\tHilo %d Despues: %d - %d \n",iam,fp,i);
                }//parallel

                fp+=1000;
                printf("\t\tP2: %d - %d\n\n\n\n",fp,np);
      }
}
```

Cláusulas OpenMP

CLÁUSULAS DE ALCANCE DE DATOS REDUCTION

reduction (operador:lista):

Se aplica a variables “shared”

Cada hilo calcula el valor parcial (como si fuera privada), se hace la operación con esos valores y se actualiza el valor de la variable global.

La actualización final se realiza al salir del constructor

Implica una inicialización de la variable (acorde a la operación a realizar)

Operadores: +, -, *, max , min , .and. , .or. , .eqv. , .neqv., .iand. , .ior. , .ieor

Cláusulas OpenMP

CLÁUSULAS DE ALCANCE DE DATOS REDUCTION

reduction (operador:lista):

Los operadores dependen del compilador y su versión (por ejemplo gcc inferior a 4.7 no admite “min, max”)

Ejemplo reducciones múltiples:

```
reduction (+:dato1,dato2)
reduction (+:dato1) reduction(*:dato3)
```

Cláusulas OpenMP

CLÁUSULAS DE ALCANCE DE DATOS REDUCTION

```
int main()
{
   int iam, np=0, i, prod_int=0;
   int data1[10];
   int data2[10];
   for (i=0;i<10;i++){
      data1[i] = i;
      data2[i] = i*2;
    }

#pragma omp parallel num_threads(4) shared(prod_int,data1,data2) private(np,iam,i)  default(none)
{
    iam = omp_get_thread_num();
    np = omp_get_num_threads();
    #pragma omp for schedule(static,2) reduction(+:prod_int)
    for (i=0;i<10;i++){
        prod_int += data1[i] * data2[i];
    }
}//parallel
printf("\t\tFUERA valor de prod_int: %d\n",prod_int);
}
```

Cláusulas OpenMP

Clause	Directive					
	PARALLEL	DO/for	SECTIONS	SINGLE	PARALLEL DO/for	PARALLEL SECTIONS
IF	●				●	●
PRIVATE	●	●	●	●	●	●
SHARED	●	●			●	●
DEFAULT	●				●	●
FIRSTPRIVATE	●	●	●	●	●	●
LASTPRIVATE		●	●		●	●
REDUCTION	●	●	●		●	●
COPYIN	●				●	●
COPYPRIVATE				●		
SCHEDULE		●			●	
ORDERED		●			●	
NOWAIT		●	●	●		

Funciones y variables de entorno OpenMP

void omp_set_num_threads(int num_threads);

int omp_get_num_threads(void);

int omp_get_max_threads(void); (caminos software)

int omp_get_thread_num(void);

int omp_get_num_procs(void); (caminos hardware)

int omp_in_parallel(void);

Funciones y variables de entorno OpenMP

void omp_set_dynamic(int); ¿Qué es dinámico?

int omp_get_dynamic(void);

void omp_set_nested(int);

int omp_get_nested(void);

double omp_get_wtime(void);

double omp_get_wtick(void);

y otras muchas…

Funciones y variables de entorno OpenMP

OMP_SCHEDULE (schedule(runtime))

OMP_NUM_THREADS

OMP_DYNAMIC

OMP_NESTED

COMPUTACIÓN PARALELA

3º GRADO EN INGENIERÍA INFORMÁTICA EN TECNOLOGÍAS DE LA INFORMACIÓN

Unidad Didáctica III. Arquitecturas de memoria compartida: OpenMP