Transparencias adicionales S2

Especificación OpenMP

- Estándar de facto para programación en memoria compartida

http://www.openmp.org

- Especificaciones:
  - Fortran: 1.0 (1997), 2.0 (2000)
  - C/C++: 1.0 (1998), 2.0 (2002)
  - Fortran/C/C++: 2.5 (2005), 3.0 (2008), **3.1 (2011)**, 4.0,(2013), 4.5 (2015), 5.0 (2018)
- Antecedentes:
  - Estándar ANSI X3H5 (1994)
  - HPF, CMFortran

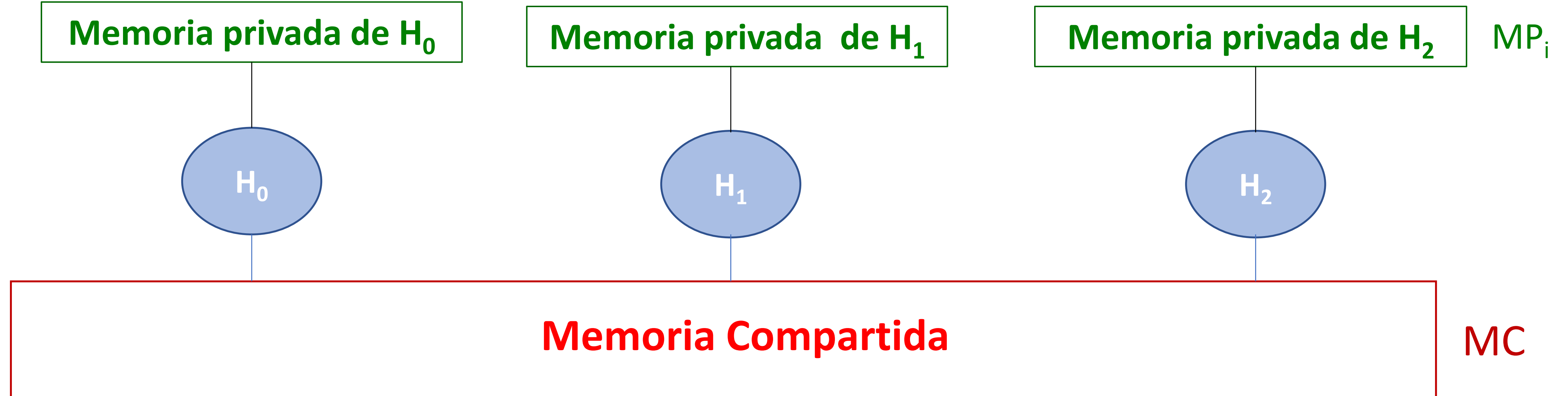
Arquitectura OpenMP

MP_i ≡ Memoria Privada del hilo H_i ={variables privadas del hilo H_i}

MC ≡ Memoria Compartida por todos los hilos={variables compartidas}

Hilo ≡ Proceso ligero que puede ejecutar una secuencia de tareas

- Cada hilo tiene su propio contexto de ejecución, su propia pila, compartiendo con otros hilos recursos del sistema
- Cada hilo tiene sus propias variables privadas
- Todos tienen acceso a las variables compartidas



Modelo de programación en OpenMP

- El código fuente debe contener al principio: #include <omp.h>
- Se pueden usar funciones específicas para el programador:
  - omp_get_num_threads(): devuelve el número de hilos
  - omp_get_thread_num(): devuelve el identificador del hilo que invoca a esta función
  - Etc.
- Usa directicas de compilación. Sintaxis: #pragma omp **directiva claúsulas**
  - #pragma omp parallel …… /* la llave de bloque **{** debe ir en la siguiente línea de código*/

```
{
 ….. /*Región Paralela RP*/
}
```

  - #pragma omp parallel for …..

```
for(i=vi; i<vf;inc)
{
  …..
 }
```

  - #pragma omp parallel sections /* la llave de bloque **{** debe ir en la siguiente línea de código*/

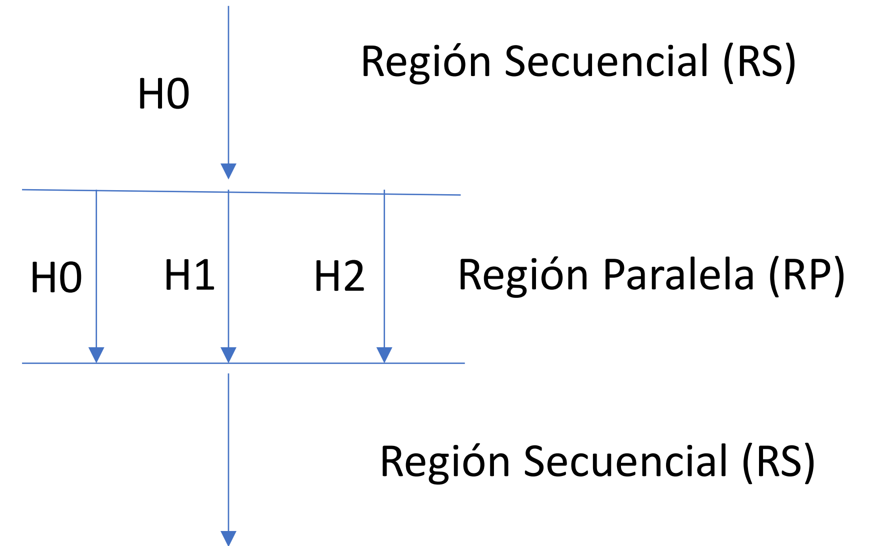
```
{
….
}
```

- Tiene variables de entorno
  - OMP_NUM_THREADS=4 OMP_SCHEDULE=guided ./pr: Ejecución con 4 hilos y una planificación de tipo guided del ejecutable pr

Modelo de ejecución en OpenMP

- Los hilos ejecutan concurrentemente el mismo código en una región paralela (ejemplo con 3 hilos)

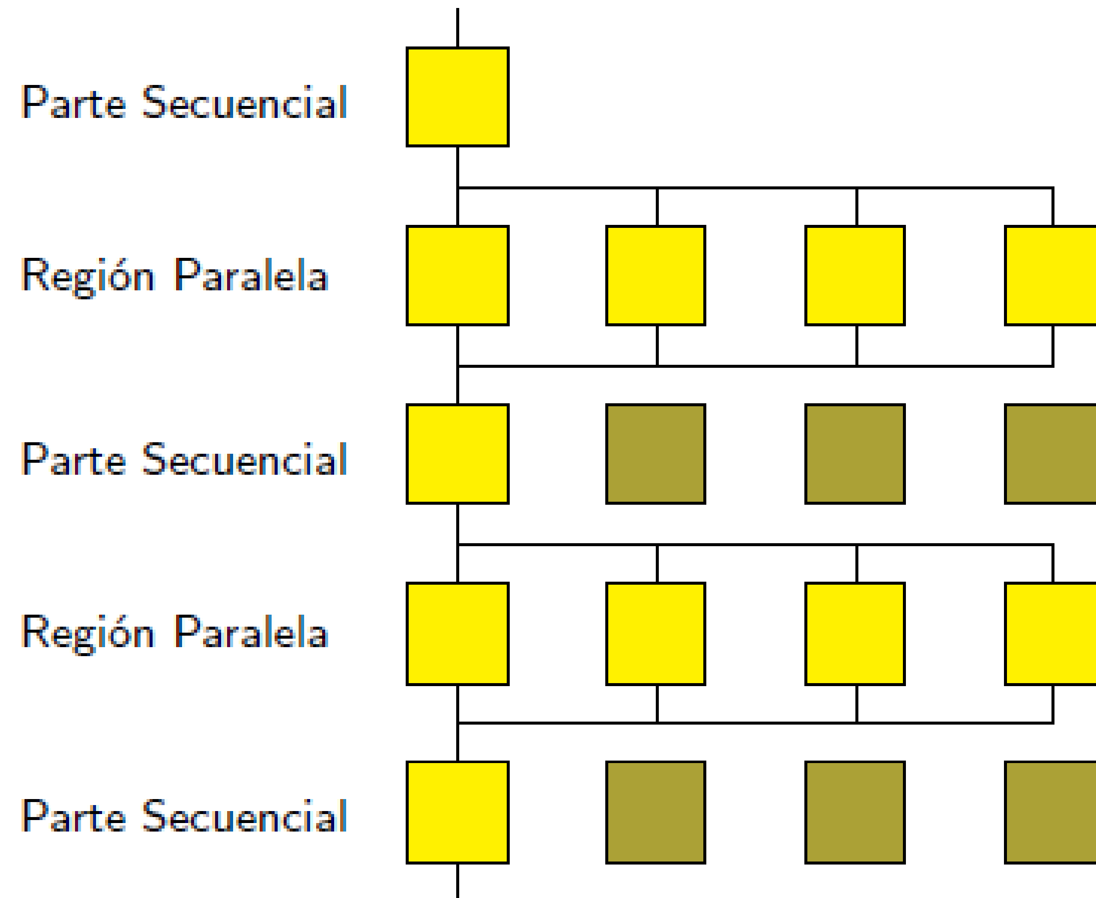
```
..../*Región Secuencial (RS)*/
#pragma omp parallel ......
  {
   ..... /*Región Paralela (RP)*/
  }
..../*Región Secuencial (RS)*/
```



- El hilo maestro H0 ejecuta la Región Secuencial (RS)
- El hilo maestro H0 cuando ejecuta la directiva, crea/activa los hilos
- Todos los hilos ejecutan concurrentemente el mismo código de la Región Paralela (RP) Pueden hacer tareas distintas, dependiendo de su identificador
- Hay una barrera implícita al final de la RP: todos los hilos esperan a que acaben todos
- En la siguiente RS el hilo maestro H0 ejecuta el código contenido en RS, quedando los otros hilos inactivos

Modelo de ejecución en OpenMP

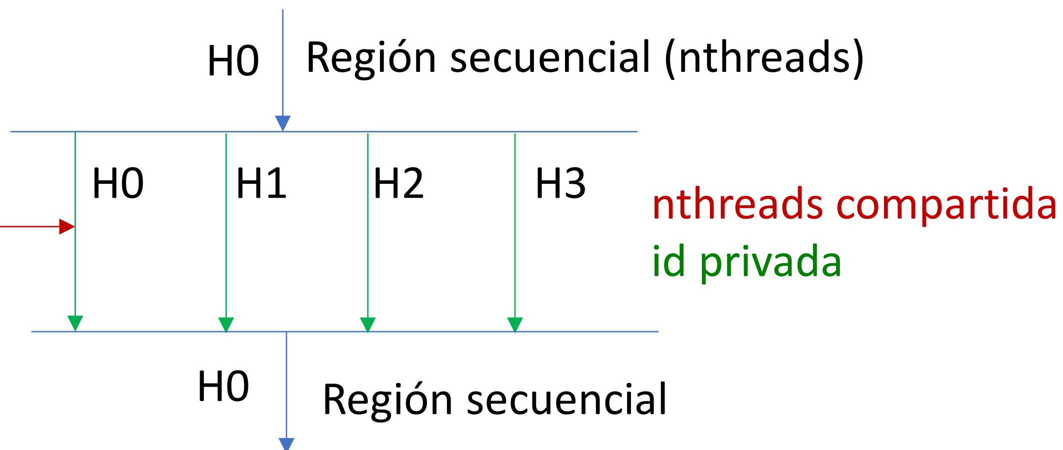
- En OpenMP los hilos desactivados no se destruyen, quedan a la espera de ser activados en la siguiente Región Paralela (RP)
- Los hilos creados en una RP forman parte de un equipo (team)
- La creación de una RP supone un overhead para el tiempo de ejecución, pues se tienen que crear/activar, sincronizar y desactivar los hilos del equipo



Ejemplo programa OpenMp

hola_openMP.c

```
#include <stdio.h>
#include <omp.h>
int main ()  {
  int nthreads;
  #pragma omp parallel /*Crea un Región Paralela, activando los hilos*/
  {
    int id = omp_get_thread_num(); // cada uno obtiene su identificador de hilo
    nthreads = omp_get_num_threads(); // todos obtienen el número de hilos
    printf("Hola desde el thread %d de %d threads\n", id, nthreads);
  }
}
```



Compilación y ejecución (ordenador personal)

```
jjibanez@MSI:~/cpa/CPA_Actual/OpenMP/codigos$ gcc -fopenmp -o hola hola_openMP.c
jjibanez@MSI:~/cpa/CPA_Actual/OpenMP/codigos$ ./hola
Hola desde el thread 1 de 8 threads
Hola desde el thread 7 de 8 threads
Hola desde el thread 3 de 8 threads
Hola desde el thread 6 de 8 threads
Hola desde el thread 0 de 8 threads
Hola desde el thread 5 de 8 threads
Hola desde el thread 4 de 8 threads
Hola desde el thread 2 de 8 threads
jjibanez@MSI:~/cpa/CPA_Actual/OpenMP/codigos$ OMP_NUM_THREADS=16 ./hola
Hola desde el thread 1 de 16 threads
Hola desde el thread 7 de 16 threads
Hola desde el thread 5 de 16 threads
Hola desde el thread 2 de 16 threads
Hola desde el thread 3 de 16 threads
Hola desde el thread 4 de 16 threads
Hola desde el thread 6 de 16 threads
Hola desde el thread 8 de 16 threads
Hola desde el thread 9 de 16 threads
Hola desde el thread 15 de 16 threads
Hola desde el thread 10 de 16 threads
Hola desde el thread 0 de 16 threads
Hola desde el thread 11 de 16 threads
Hola desde el thread 14 de 16 threads
Hola desde el thread 13 de 16 threads
Hola desde el thread 12 de 16 threads
```

Modelo de programación en OpenMP

- En un bloque paralelo for, las iteraciones se reparten entre los hilos; por defecto, el reparto es por bloques; por lo tanto, la variable iteradora debe ser privada
- En el siguiente ejemplo, suponiendo 3 hilos en el equipo, el reparto sería H0(i=0, i=1), H1(i=2, i=3) y H2(i=4, i=5):

```
#pragma omp parallel for
for (i=0; i<6;i++){
....
}
```

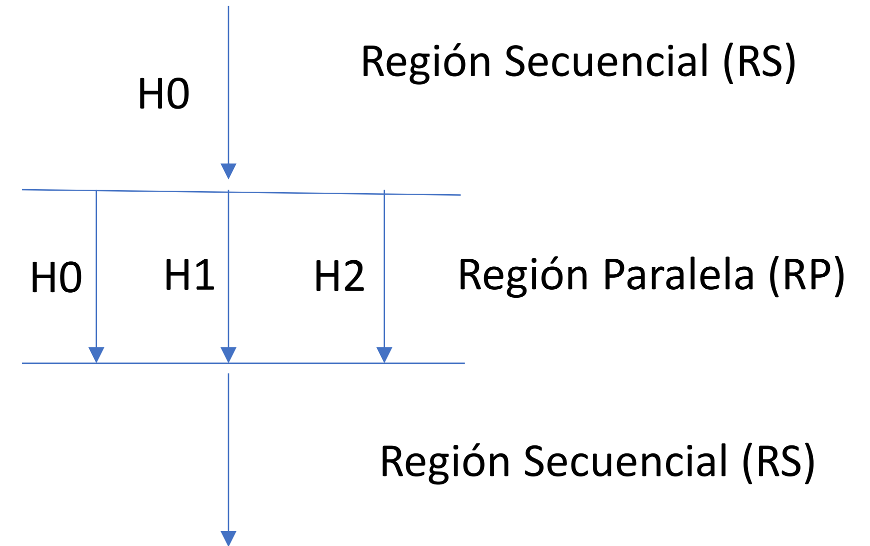
- El número de iteraciones no puede depender de que se cumpla o no una cierta condición, pues en ese caso no es conocido a priori el número de iteraciones a repartir
- Por ejemplo, en el siguiente código se produciría un error de compilación:

```
#pragma omp parallel for .....
  for(i=0; i<N&&x<3;++i)
  {
   /*x varia en cada iteración*/
   .....
  }
```

Modelo de ejecución

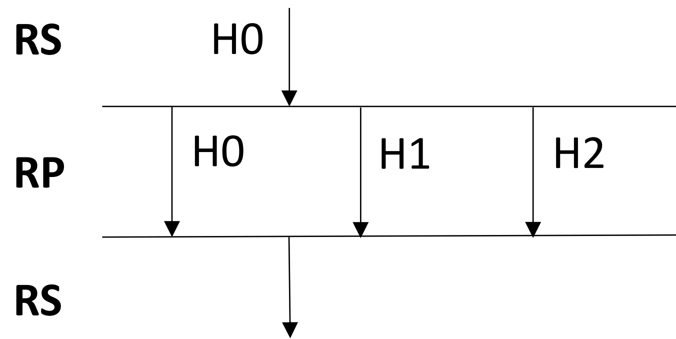
- for paralelo (ejemplo con 3 hilos)

```
▪#pragma omp parallel for ……
  for(i=0;i<N;i++)
 {
   ….. /*Región Paralela RP*/
  }
```



- El hilo maestro H0 ejecuta la Región Secuencial (RS)
- El hilo maestro cuando ejecuta la directiva, crea los hilos
- Todos los hilos ejecutan concurrentemente el mismo código de la Región Paralela (RP)
- Las iteraciones se reparten entre los hilos por defecto mediante bloques: el hilo H0 ejecutaría el primer bloque de iteraciones (0 hasta N/3-1), el segundo el segundo bloque (N/3 hasta 2N/3-1) y el tercero el tercer y último bloque (2N/3 hasta N-1)
- Hay una barrera implícita al final de la RP: todos los hilos esperan acabar
- En la siguiente RS el hilo maestro ejecuta el código contenido en RS, quedando los otros hilos inactivos

Ámbito de las variables en la Región Paralela (RP)



- Con ámbito de una variable en OpenMP nos refierimos a cómo se comporta esa variable dentro de una región paralela: si es compartida entre todos los hilos o si cada hilo tiene su propia copia (es privada para cada hilo).
- Las variables declaradas en una **Región Secuencial (RS)** son **compartidas** por defecto cuando se ejecuta el código en una **Región Paralela (RP)** posteriores.
  - **Excepción**: las variables de iteración en los bucles paralelos **for** se convierten en privadas en la **RP**
- Las variables declaradas dentro de una **RP** son **privadas** al ejecutar el código en esa región
- Cuando un hilo **ejecuta una función** en la **RP**, las variables locales declaradas en esa función son **privadas**.
  - **Excepción:** las variables de tipo **estático** (declaradas con **static**) son compartidas

Ejemplo de alcance(tipo) de las variables

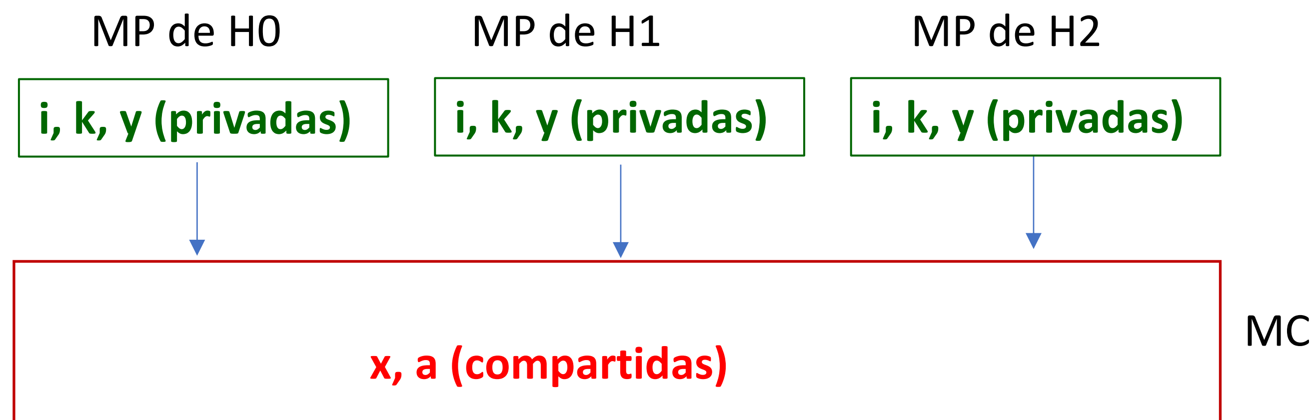
```
double f(int k){
   static double a=0;
   double y;
   a+= k;
   .....
   return y;
}

void main(){
   int i=3;
   double x[6];
   leer(x);
   #pragma omp parallel for
     for (i=0; i <6; i ++){
        int k= i * i;
        x[i] = f(k);
     }
   i=2*i;/*i valdrá 6*/
....
}
```

RS

RP

RS

Contenidos de las memorias privadas y compartida cuando se ejecuta el código de la Región Paralela RP:



MP ≡ Memoria Privada de un hilo
MC ≡ Memoria Compartida por todos los hilos

Nota: La variable i es privada cuando se ejecuta el código de RP

Variables compartidas que las convertimos en privadas en la RP

- Si queremos que una variable que está declarada en la región secuencial anterior a la región paralela RP sea privada en RP se usa la claúsula **private**
  - El valor de esa variable en la RP queda indefinido (no “hereda” el valor que tuviera en la anterior RS)
  - Para que “herede” el valor que tuviera en la RS anterior, se debe utilizar la claúsula **firstprivate**
  - El valor final que tuviera la variable declarada como privada en la RP no se hereda en la siguiente RS. En la siguiente RS el valor de la variable es el mismo que la tuviera en la RS anterior
  - Es posible que ese valor sea heredado, usando la claúsula **lastprivate**, pero únicamente para las directivas **parallel for** y **sections**

Variables compartidas que las hacemos privadas en la RP

```
int i=5;/*i vale 5 en la 1ª RS*/
#pragma omp parallel private(i)
{/*valor inicial de i indefinido en la RP*/
…
}
/*el valor de i es 5 en la 2ª RS*/
```

```
int i, k;
double alpha[10], z[10];
………
#pragma omp parallel for lastprivate(i)
for (i=0; i<10; i++) {
   z[i] = alpha*x[i];
}
k= i; /*k=i=10*/
```

```
int i=5; /*i vale 5 en la 1ª RS*/
#pragma omp parallel firsprivate(i)
{/*el valor inicial de i  es 5 en la RP*/
   i=i+1; /* i pasa a valer 6*/
……
}
/*El valor de i es 5 (el mismo valor que tenía en la anterior RS*/
```

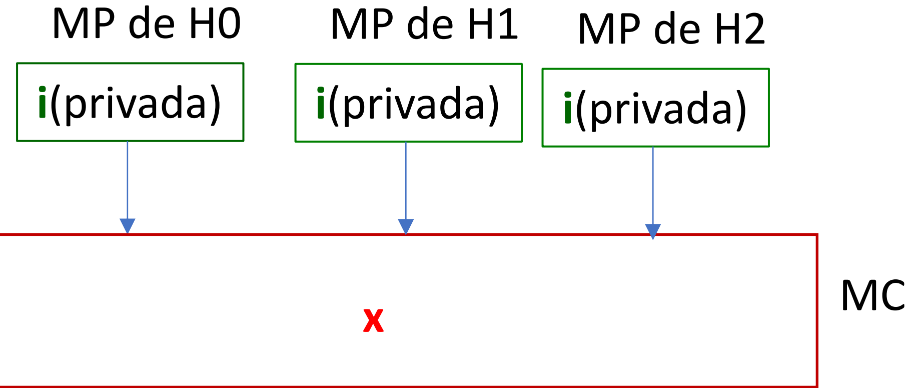
- Recordad que las variables declaradas en la RP son privadas
- Son equivalentes los siguientes códigos:

```
int i;
#pragma omp parallel private(i)
{
…
}
```

```
#pragma omp parallel
{
int i;
…
}
```

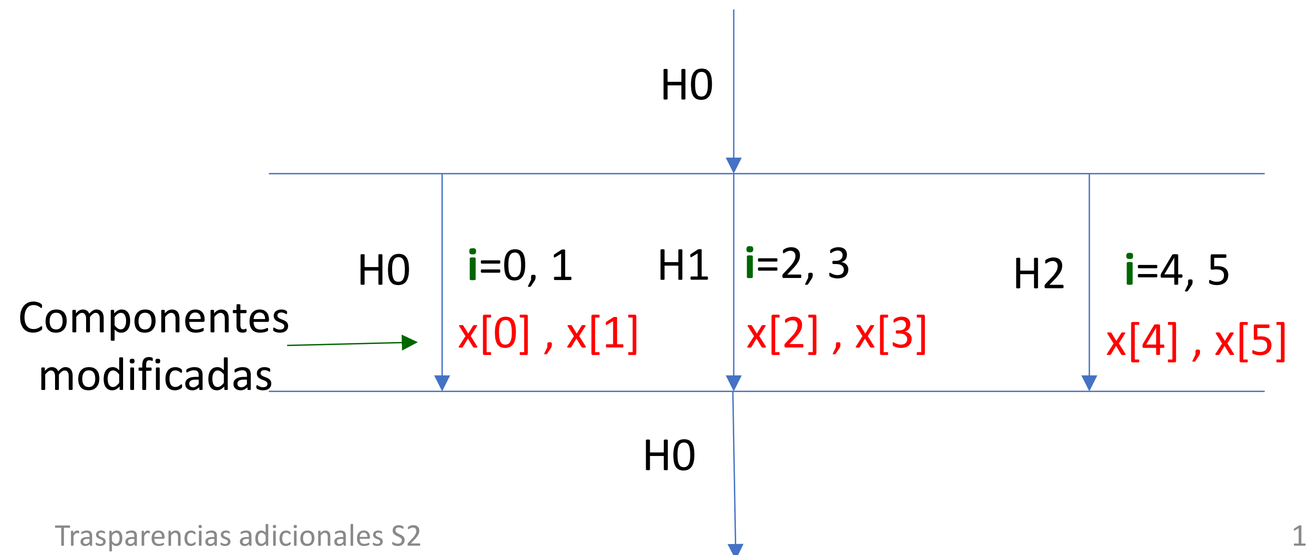
Ejemplo ejecución bucle paralelo for(con tres hilos)

```
int i;
double x[6];
leer(x);/*lee el vector x*/

.....

#pragma omp parallel for
for (i=0; i<6; i++)
  x[i] = 3*x[i];
```



MP ≡ Memoria Privada de un hilo

MC ≡ Memoria Compartida por todos los hilos



Accesos a Memoria Compartida en una Región Paralela

- El acceso concurrente a una variable compartida puede producir una condición de carrera:
  - El resultado final puede ser incorrecto
  - Es de naturaleza no determinista

- Ejemplo: dos hilos ejecutan ++i, siendo i una variable compartida. Se pueden obtener dos resultados distintos (resultado no determinista):

H0 carga i en un registro: 0	H0 carga i en un registro: 0
H0 incrementa registro: 1	H1 carga i en un registro: 0
H0 almacena el valor en i: 1	H0 incrementa registro: 1
H1 carga i en un registro: 1	H1 incrementa registro: 1
H1 incrementa registro: 2	H0 almacena el valor en i: 1
H1 almacena el valor en i: 2	H1 almacena el valor en i: 1

Accesos a Memoria Compartida en una Región Paralela

- En una Región Paralela (RP) hay que comprobar las variables que cambian su valor. Si una de esas variables debe ser compartida, habrá que crear una región crítica que impida el acceso simultaneo a esa variable por parte de los hilos. Si esa variable debe ser privada, la declararemos privada mediante la cláusula **private**.
- Se suele detectar cuando en la región paralela RP hay instrucciones del tipo

  v= expresión;

  donde v es una variable compartida

  - Cuando se trata de modificar componentes de un vector compartido hay que comprobar que las iteraciones están repartidas entre los hilos

- Ejemplo (condición de carrera en la variable x)

```
double x=3;
#pragma omp parallel /*x es compartida*/
for (i=0; i<n; i++) {
      x+=0.1
}
```

- Ejemplo (no hay condición de carrera en el vector x)

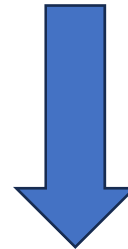
```
double x[6];
Leer(x);
#pragma omp parallel for
for (i=0; i<6; i++) {
      x[i]=2*x[i];
}
/* No hay condición de carrera, pues las iteraciones están repartidas, los hilos modifican diferentes componentes de x*/
```

Operaciones de reducción

- Las variables declaradas con la clausula **reduction** se usan en los bucles for, anidados o no, cuando estos bucles calculan sumatorios(+,−), productorios(*) y cálculo de máximos o mínimos (max o min), de manera que cada hilo calcula una parte del cálculo (usando para ello diferentes iteraciones)

```
double x[N], s=s0; /*s0 conocido*/
int i;
leer(x);
for (i=0; i<N; i++)
     s = s + x[i];
```

$$s = s_0 + \sum_{i=0}^{N-1} x_i$$



Paralelizar el cálculo del sumatorio

```
double x[N], s=s0; /*s0 conocido*/
int i;
leer(x);
#pragma omp paralell for reduction(+:s)
for (i=0; i<N; i++)
     s = s + x[i];
```

$$\boldsymbol{s} = s_0 + \sum_{i(\mathrm{H}_0)} x_i + \sum_{i(\mathrm{H}_1)} x_i + \sum_{i(\mathrm{H}_2)} x_i$$

En la RS la variable s tiene valor inicial s_0, en la RP cada hilo calcula una suma parcial, y cuando se ejecuta la siguiente RS, el valor de s es la suma de s_0 y de la suma de las sumas parciales

Operaciones de reducción

```
double x[N], p=p0; /*p0 conocido*/
leer(x);
int i;
for (i=0; i<N; i++)
      p = p * x[i];
```

$$p = p_0 \cdot \prod_{i=0}^{N-1} x_i$$

Paralelizar el cálculo del producto de componentes

```
double x[N], p=p0; /*p0 conocido*/
leer(x);
int i;
#pragma omp paralell for reduction(*:p)
for (i=0; i<N; i++)
      p = p * x[i];
```

$$p = p_0 \cdot \prod_{i(H_0)} x_i \cdot \prod_{i(H_1)} x_i \cdot \prod_{i(H_2)} x_i$$

Operaciones de reducción

```
double x[N], m;
int i;
Leer(x);
m=x[0];
for (i=1; i<N; i++)
     if (x[i]>m)
          m=x[i];
```

$$\boldsymbol{m} = \max_{i=0,\cdots,N-1} \{x_i\}$$

Paralelizar el cálculo del máximo del vector x

```
double x[N], m;
int i;
leer(x)
m=x[0];
#pragma omp paralell for reduction(max:m)
for (i=1; i<N; i++)
     if (x[i]>m)
          m=x[i];
```

$$\boldsymbol{m} = \max(\mathrm{m}, \max_{i(\mathrm{H}_0)}\{x_i\}, \max_{i(\mathrm{H}_1)}\{x_i\}, \max_{i(\mathrm{H}_2)}\{x_i\})$$

Operaciones de reducción

```
double x[N], m; /*x conocido*/
int i;
m=x[0];
for (i=1; i<N; i++)
    if (x[i]<m)
        m=x[i];
```

$$\boldsymbol{m} = \min_{i=0,\cdots,N-1} \{x_i\}$$

Paralelizar el cálculo del mínimo del vector x

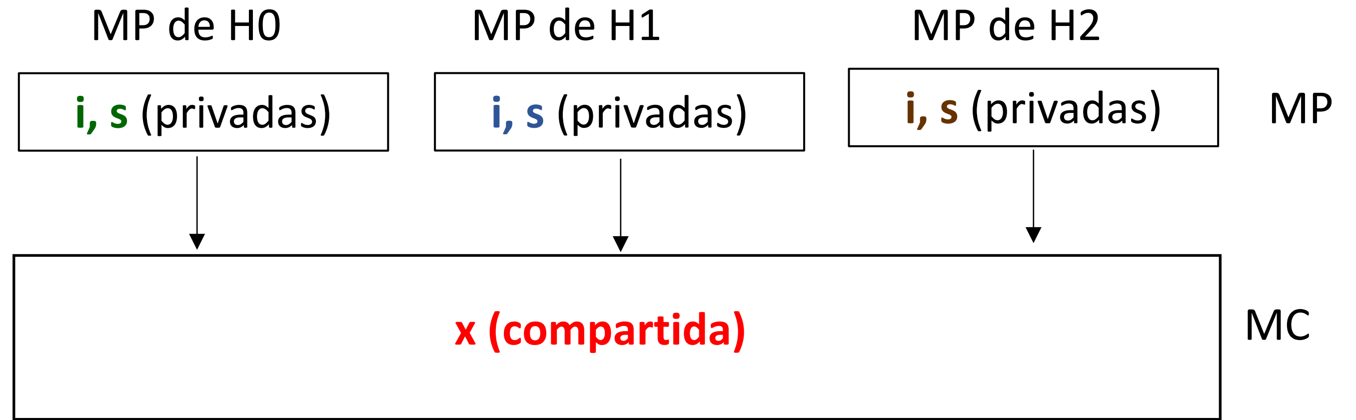
```
double x[N], m; /*x conocido*/
int i;
m=x[0];
#pragma omp paralell for reduction(min:m)
for (i=1; i<N; i++)
    if (x[i]<m)
        m=x[i];
```

$$\boldsymbol{m} = \min(\mathrm{m}, \min_{i(\mathrm{H}_0)}\{x_i\}, \min_{i(\mathrm{H}_1)}\{x_i\}, \min_{i(\mathrm{H}_2)}\{x_i\})$$

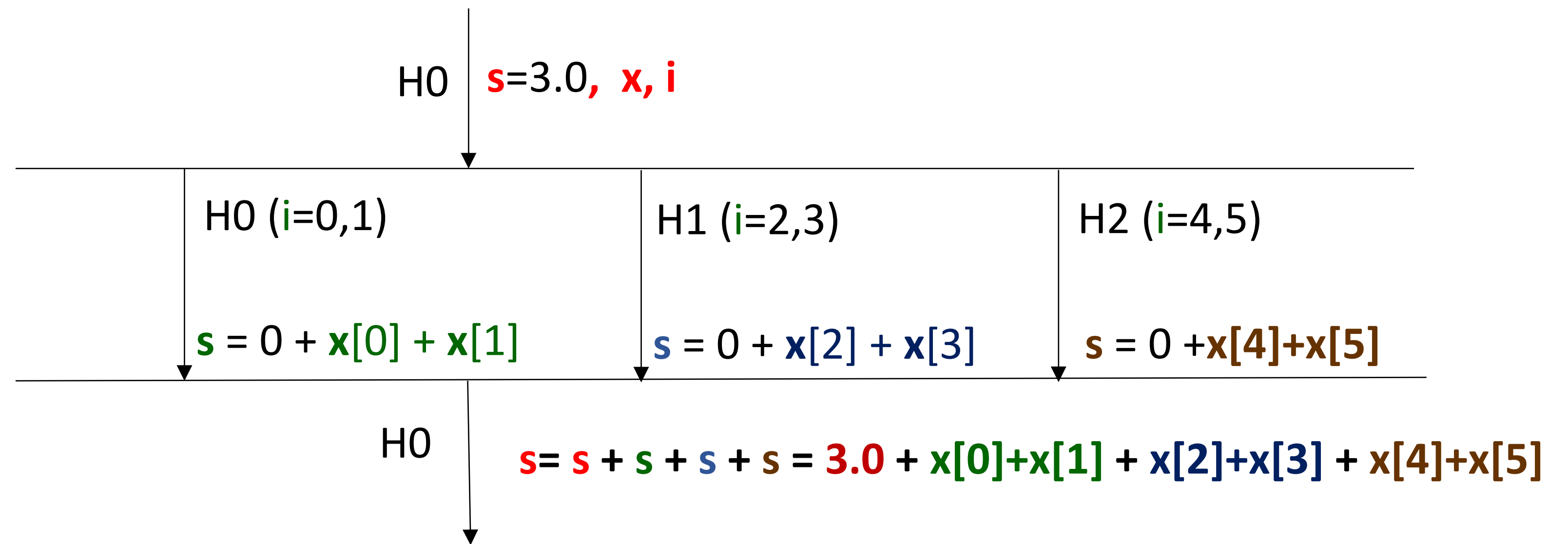
Ejemplo de la traza de una reducción: suma de las componentes de un vector

```
double s = 3.0;
double x[6];
int i;
leer(x);
for (i=0; i<6; i++)
   s = s + x[i];
printf("suma=%f\n", s);
```

$$s = 3.0 + \sum_{i=0}^{5} x_i$$




```
double s = 3.0;
double x[6];
int i;
leer(x);
#pragma omp parallel for reduction(+:s)
   for (i=0; i<6; i++)
      s = s + x[i];
printf("suma=%f\n", s);
```



Operaciones de reducción

Operación	reduction(+: v)	reduction(–: v)	reduction(*: v)	reduction(max:v)	reduction(min: v)
Valor inicial en cada hilo en la **RP**:	v=0	v=0	v=1	v=valor menor posible	v=valor mayor posible
v= valor en la **RS** anterior **v**= valor en la **RS** posterior	$\boldsymbol{v} = v + \sum_{i=0}^{N-1} x_i$	$\boldsymbol{v} = v - \sum_{i=0}^{N-1} x_i$	$\boldsymbol{v} = v \prod_{i=0}^{N-1} x_i$	$\boldsymbol{v} = \max(v, \max_{i=0,\cdots,N-1}\{x_i\})$	$\boldsymbol{v} = \min(v, \min_{i=0,\cdots,N-1}\{x_i\})$

- Las iteraciones del bucle paralelo **for** con variable iteradora i se reparten entre los hilos (por defecto, un reparto por bloques): cada hilo hace parte de la suma, resta, producto, determinación de su máximo o determinación de su mínimo
- Se pueden declarar varias variables de reducción para una misma operación:

  Ejemplo: reduction(+: v1, v2,…,vn)
- Pueden existir varias reducciones en la directiva de OpenMP: reduction(+: ….) reduction(*: ….)

Tipos de variables al paralelizar un código secuencial

Cuando se quiere paralelizar un código secuencial, hay que determinar el tipo de variable (**ámbito**) correcto de las variables en la Región Paralela (RP). En general:

- **Variables cuyo ámbito debe ser privado**
  - Declaradas antes de la RP:
    - Las variables iteradores de los bucles **for** que aparezcan dentro de la RP son implícitamente privadas
    - Las variables que aparezcan a la izquierda de una asignación en la RP puede que sea necesario privatizarlas. En este caso, debemos convertirlas en privadas mediante la clausula **private**
  - Declaradas en la RP: directamente son privadas
- **Variables compartidas**
  - Normalmente está declaradas antes de la región paralela, pero si deben ser compartidas en la RP, se puede producir una condición de carrera. En este caso, se solucionará usando las directivas **critical** o una **atomic**
  - Si se trata de un array compartido y los hilos van a modificar diferentes componentes del array, no hay problema de condición de carrera
- **Variables de reducción en bucles paralelos for**
  - Una variable será una **reducción** para un bucle **paralelo for** cuando dicha variable se inicializa antes de un bucle **for** y dicha variable es la suma, producto, máximo o mínimo de términos que dependen de la variable iteradora del bucle **for** paralelo

- Ejemplo: paralelizar los bucles exterior e interior del siguiente código

```
int i, j;
double s=0.0, A[N][N];
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        s=s+A[i][j];
```

$$s = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} A_{ij}$$

- Se puede paralelizar cualquiera de los bucles, pero no a la vez

Paralelización bucle exterior

```
int i, j;
double s=0.0, A[N][N];
#pragma omp parallel for   reduction (+: s) private(j)
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        s=s+A[i][j];
/* La variable i se convierte en privada automáticamente*/
/*La variable j debe ser privada, pues si no habría una condición de carrera en j*/
```

Paralelización bucle interior

```
int i, j;
double s=0.0, A[N][N];
for (i=0; i<N; i++)
  #pragma omp parallel for  reduction (+: s)
    for (j=0; j<N; j++)
        s=s+A[i][j];
/*La variable j se convierte en privada
automáticamente */
```

Ejemplo: paralelización en bucles anidados

```
int i, j;
double s, A[N][N], x[N];
for (i=0; i<N; i++){
      s=0.0;
     for (j=0; j<N; j++)
         s=s+A[i][j];
     x[i]=s;
}
```

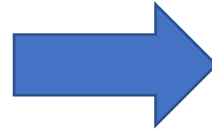
Para cada i, se calcula x[i]= $s = \sum_{j=0}^{N-1} A_{ij}$

```
/*Paralelización bucle externo*/
int i, j;
double s, A[N][N], x[N];
#pragma omp parallel for private (s, j)
for (i=0; i<N; i++){
      s=0.0;
      for (j=0; j<N; j++)
           s=s+A[i][j];
      x[i]=s;
}
```

```
/*Paralelización bucle interno*/
int i, j;
double s, A[N][N], x[N];
for (i=0; i<N; i++){
      s=0.0;
      #pragma omp parallel for reduction (+: s)
      for (j=0; j<N; j++)
           s=s+A[i][j];
      x[i]=s;
}
```

Ejercicio 1: paralelización del bucle externo

```
int i, j;
double  v[N], A[N][N], B[N][N];
Leer(A,B);
for(i=0;i<N;i++){
   v[i]=0.0;
   for (j=0;j<N;j++)
      v[i]+=A[i][j]*B[i][j];
}
```



```
int i, j;
double v[N] , A[N][N], B[N][N];
Leer(A,B);
#pragma omp paralell for   private(j)
for(i=0; i<N; i++){
   v[i]=0.0;
   for (j=0;j<N;j++)
      v[i]+=A[i][j]*B[i][j];
}
```

Para que no haya condición de carrara la variable j debe ser privada

$$v_i = \sum_{j=0}^{N-1} A_{ij} B_{ij}$$

i=0, 1, ...,*N*-1

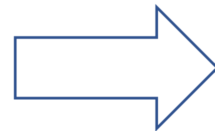
$Repartir\ las\ iteraciones\ bucle\ i\ (N = 6, 3\ \text{hilos})$

H_0: v_0, v_1
H_1: v_2, v_3
H_2: v_4, v_5

Paralelizar bucle i

Ejercicio 2: paralelizar el bucle externo

```
int i, j;
double s, v[N], A[N][N], B[N][N];
Leer(A,B);
for(i=0; i<N; i++){
    s=0.0;
    for (j=0; j<N; j++)
        s+=A[i][j]*B[i][j];
    v[i]=s;
}
```



```
int i, j;
double s, v[N], A[N][N], B[N][N];
Leer(A,B);
#pragma omp paralell for private(j, s)
for(i=0; i<N; i++){
    s=0.0;
    for (j=0; j<N; j++)
        s+=A[i][j]*B[i][j];
    v[i]=s;
}
```

las variables j y s deben ser privadas

$$v_i = s = \sum_{j=0}^{N-1} A_{ij} B_{ij}$$

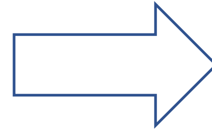
Repartir el cálculo de las componentes del vector v entre los hilos

Ejercicio 3: paralelizar el bucle interno

```
int i, j;
double  v[N], A[N][N], B[N][N];
Leer(A,B);
for(i=0; i<N; i++){
   s=0.0;
   for (j=0; j<N; j++)
      s+=A[i][j]*B[i][j];
   v[i]=s;
}
```

$$v_i = s = \sum_{j=0}^{N-1} A_{ij} B_{ij}$$

i=0, 1, ...,N-1

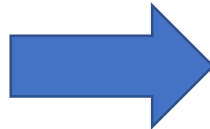


```
int i, j;
double v[N] , A[N][N], B[N][N];
Leer(A,B);
for(i=0; i<N; i++){
   s=0.0;
   #pragma omp paralell for  reduction(+: s)
   for (j=0;j<N;j++)
      s+=A[i][j]*B[i][j];
   v[i]=s;
}
```

Ejercicio 4 (paralelización de bucles)

```
double p, A[N];
int i;
Leer(A);
p=1;
for(i=0; i<N; i++)
   p=p*A[i];
```

Hacer que el producto total sea realizado entre todos los hilos:



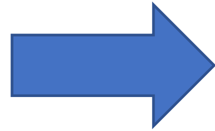
```
double p, A[N];
int i;
Leer(A);
p=1.0;
#pragma omp parallel for reduction(*:p)
for(i=0; i<N; i++)
   p=p*A[i];
```

$$p = \prod_{i=0}^{N-1} A_i$$

Ejercicio 5 (paralelización de bucles)

```
double m, v[N];
int i;
Leer(v);
m=v[0];
for(i=1;i<N; i++)
   if (v[i]>m)
        m=v[i];
```

$$m = \max_{i=0,\cdots,N-1} \{v_i\}$$



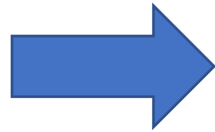
```
double m, v[N];
int i;
Leer(v);
m=v[0];
#pragma omp parallel for reduction(max: m)
for(i=1;i<N; i++)
   if (v[i]>m)
        m=v[i];
```

Máximo de las componentes de un vector: Reducción sobre m

Ejercicio 6 (paralelización de bucles)

```
double m, v[N];
int i;
Leer(v);
m=v[0];
for(i=1;i<N; i++)
   if (v[i]<m)
         m=v[i];
```

Mínimo de las componentes de un vector: Reducción sobre m



```
double m, v[N];
int i;
Leer(v);
m=v[0];
#pragma omp parallel for  reduction(min: m)
for(i=1;i<N; i++)
   if (v[i]<m)
         m=v[i];
```

$$m = \min_{i=0,\cdots,N-1} \{v_i\}$$

Ejercicio 7 (paralelización de bucles anidados)

Indica cómo se paralelizaría mediante OpenMP cada uno de los tres bucles del siguiente código. ¿Cuál de las tres formas de paralelizar será la más eficiente y por qué?

```
double fun_mat(double a[n][n], double b[n][n], double c[n][n]){
    int i, j, k;
    double aux, p=1.0;
    for (i=0; i<n; i++) {
        for (j=0; j<n; j++) {
            aux=0.0;
            p= p*a[i][j];
            for (k=0; k<n; k++)
                aux += a[i][k] * b[k][j];
            c[i][j] = aux;
        }
    }
    return s;
}
```

Bucle i: `#pragma omp parallel for reduction(*:p) private(j, aux, k)`

Bucle j: `#pragma omp parallel for reduction(*:p) private(aux, k)`

Bucle k: `#pragma omp parallel for reduction(+:aux)`

$$p = \prod_{i=0}^{n-1} \prod_{j=0}^{n-1} a_{ij}$$

→ Reducciones * de p:
Bucle paralelo de i
Bucle paralelo de j

$$aux = \sum_{k=0}^{n-1} a_{ik} b_{kj}$$

→ Reducción + de aux:
Bucle paralelo de k

La paralelización más eficiente es la del for externo, pues solo se crea 1 RP. En las otras dos paralelizaciones se crean N RPs (bucle j) y N*N RPs (bucle k)

Ejercicio 8 Dependencia entre iteraciones

Paralelizar de la forma más eficiente la siguiente función:

```
double funcion(double A[M][N])
{
    int i, j;
        for (i=0; i<M-1; i++)
            for (j=0; j<N; j++)
                A[i][j] = 2.0 * A[i+1][j];
}
```

- Si se intenta paralelizar el bucle externo de variable iteradora **i**, ya que hay una dependencia de datos entre las iteraciones
- Una posibilidad consistiría en paralelizar el bucle interno de **variable iteradora j**

Solución más eficiente

Intercambiar los bucles entre sí y paralelizar el más externo que ahora es el de la **variable iteradora j**

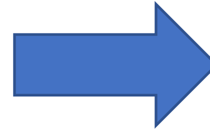
```
double funcion(double A[M][N])
{
    int i, j;
    #pragma omp parallel for  private(i)
    for (j=0; j<N; j++)
        for (i=0; i<M-1; i++)
            A[i][j] = 2.0 * A[i+1][j];
}
```

Ejercicio 9 (paralelización de bucles)

Dado el siguiente código:

```
int i;
#pragma paralell for
for (i=0; i++; i<N){
    ........
}
```

¿Cómo podríamos conocer el identificador del hilo que ha realizado la última iteración (iteración N-1)?



```
int i, id;
#pragma paralell for lastprivate(id)
for (i=0; i++; i<N){
    id=omp_get_thread_num();
    ........
}
```

Garantizar Suficiente Trabajo

- La paralelización de bucles supone un overhead: creación/activación, sincronización y desactivación de hilos
- En bucles muy sencillos, el overhead puede ser mayor que el tiempo de cálculo
- Cláusula **if**: permite crear o no una región paralela dependiendo de una condición
- Ejemplo:

```
#pragma omp parallel for if(n>5000)
for (i=0; i<n; i++)
    z[i] = a*x[i] + y[i];
```

Solo se ejecuta de forma paralela si el valor de n es mayor que 5000

Planificación (reparto iteraciones en bucles paralelos)

- Idealmente, todas las iteraciones cuestan lo mismo y a cada hilo se le asigna aproximadamente el mismo número de iteraciones, realizando por defecto un reparto por bloques
- En ocasiones, se puede producir desequilibrio de la carga, con la consiguiente pérdida de prestaciones, cuando las iteraciones tienen diferente coste computacional
- En OpenMP es posible especificar la planificación
- Puede ser de dos tipos:
  - Estática: las iteraciones se asignan a hilos antes de la ejecución del bucle paralelo
  - Dinámica: la asignación se adapta a la ejecución actual
- La planificación se realiza a nivel de rangos contiguos de iteraciones (chunks)

Cláusula schedule

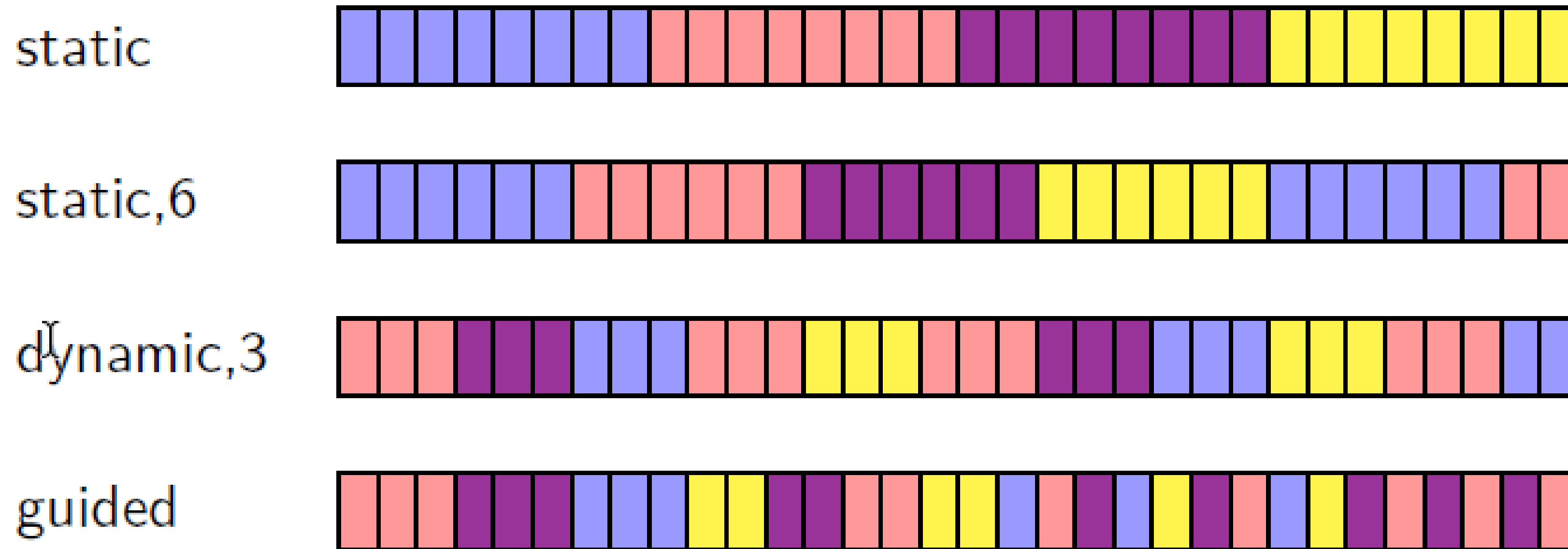
- Sintaxis de la cláusula de planificación schedule:

schedule(tipo[,**chunk**])

- Tipos:
  - **static** sin **chunk:** reparto por bloques (planificación por defecto)
  - **static** (con **chunk**): reparto cíclico de las iteraciones (round-robin) de grupos de iteraciones de tamaño chunk
  - **dynamic** (**chunk** opcional, por defecto 1): se asigna un grupo de iteraciones de tamaño **chunk** a un hilo cuando este hilo queda ocioso (first come,first-served)
  - **guided** (**chunk** mínimo opcional): como **dynamic** pero el tamaño del rango va decreciendo exponencialmente ($\propto$ nrest/nhilos)
  - **runtime**: se especifica en tiempo de ejecución con la variable de entorno OMP_SCHEDULE. Esta planificación es fundamental cuando queremos modificar la planificación en tiempo de ejecución

Cláusula schedule

- Ejemplo: ejecución bucle paralelo de 32 iteraciones sobre 4 hilos



- Ejemplo: Si queremos en tiempo real lanzar la ejecución de tipo dinámico con un valor de chunk igual a 2 sobre 4 hilos del ejecutable **pr,** añadiríamos la claúsula schedule(runtime) en los bucles paralelos for y desde la línea de comandos ejecutaríamos:

OMP_NUM_THREADS=4 OMP_SCHEDULE=dynamic,2 ./pr

Ejercicio 10.a

```
#include <stdio.h>
#define N 100
double prod(double v[N]) {
  int i;
  double p=1.0;
  for (i=0;i<N;i++)
    p*=v[i]; /* equivalente a p=p*v[i]*/
  printf("El producto de las componentes  del vector es \n",p);
}
```

Paralelizar mediante OpenMP la función anterior de manera que cumpla los siguientes requisitos:

1. Utilice 4 hilos en la región paralela.
2. Use una distribución dinámica con bloques de tamaño igual a 2.
3. Imprima en pantalla el identificador del hilo que ha realizado el producto de la última iteración y el producto total calculado.

Ejercicio 10.a (paralelización de bucles)

1. Utilice 4 hilos en la región paralela.
2. Imprima en pantalla el identificador del hilo que ha realizado el producto de la última iteración y el producto total calculado.
3. Use una distribución dinámica con bloques de tamaño igual a 2.

```
#include <stdio.h>
#include <omp.h>
#define N 100
double prod(double v[N]) {
    int i;
    double p=1.0;
    omp_set_num_threads(4);
    int tid;
    #pragma omp parallel for lastprivate(tid)   reduction(*:p)   schedule(dynamic,2)
    for (i=0;i<N;i++) {
        tid = omp_get_thread_num();
        p=p*v[i]; /* equivalente a p=p*v[i]*/
    }
    printf("El producto de las componentes  del vector es \n",p);
    printf("El hilo que ha calculado el producto de la última iteración es %d\n",tid);
}
```

$$p = \prod_{i=0}^{N-1} v_i$$

Ejercicio 10.b

a. ¿Qué modificarías en el código del ejercicio para que la planificación se modificase en tiempo de ejecución?
b. Cómo harías que en tiempo de ejecución la planificación fuese estática con chunk igual a 3

Solución:

a. Cambiaría la clausula schedule(dynamic,2) por la clausula schedule(runtime)
b. Si el código del ejecutable fuese ej10, desde la línea de comandos del terminal pondría:

```
OMP_SCHEDULE=static,3  ./ej10
```

Cuestión 1-2

Dada la siguiente función:

```
void prodmv(double a[N], double c[N], double B[N][N]){
    int i, j;
    double sum;
    for (i=0; i<N; i++) {
        sum = 0.0;
        for (j=0; j<N; j++)
            sum += B[i][j] * c[j];
        a[i] = sum;
    }
}
```

$$a_i = sum = \sum_{j=0}^{N-1} B_{ij} c_j$$

a) Realiza la implementación paralela más eficiente mediante OpenMP del código dado.

b) Calcula los costes computacionales en flops de las implementaciones secuencial y paralela, suponiendo que el número de hilos p es un divisor de N.

c) Calcula el speedup y la eficiencia del código paralelo.

a)

```
void prodmv(double a[N], double c[N], double B[N][N]){
    int i, j;
    double sum;
    #pragma omp parallel for private(sum, j)
    for (i=0; i<N; i++) {
        sum = 0.0;
        for (j=0; j<N; j++)
            sum += B[i][j] * c[j];
        a[i] = sum;
    }
}
```

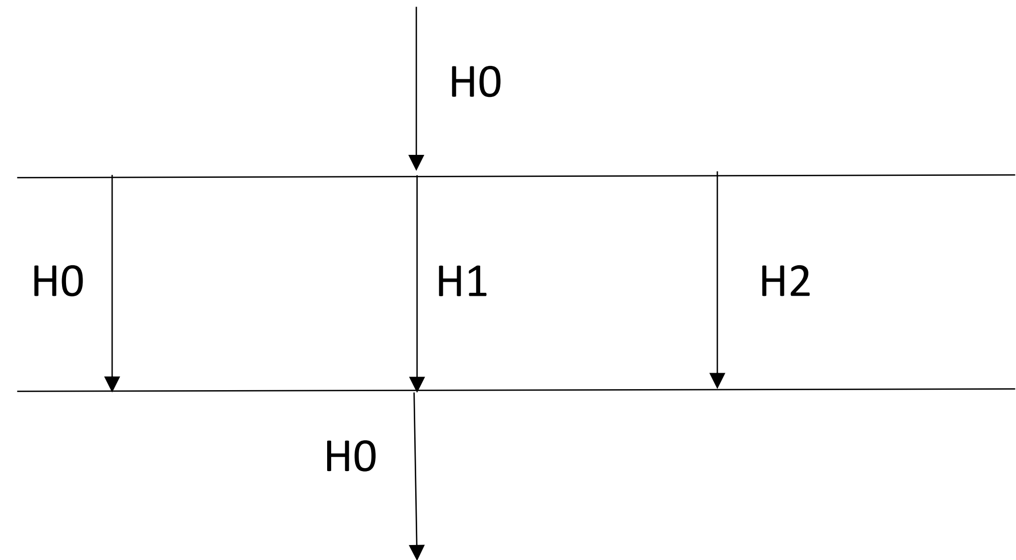
b) $t(N) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} 2 = \sum_{i=0}^{N-1} 2N = 2N^2$ flops

$$t(N,p) = \sum_{i=0}^{N/p-1} \sum_{j=0}^{N-1} 2 = \sum_{i=0}^{N/p-1} 2N = \frac{2N^2}{p} \text{ flops}$$

c) $S(N,p) = \frac{t(N)}{t(N,p)} = \frac{2N^2}{\frac{2N^2}{p}} = p \qquad E(N,p) = \frac{S(N,p)}{p} = \frac{p}{p} = 1$

Regiones paralelas (Trs 26-35 de S2)

```
#pragma omp parallel …….
{
….
}
```



Cláusulas:

- private, shared, reduction, if

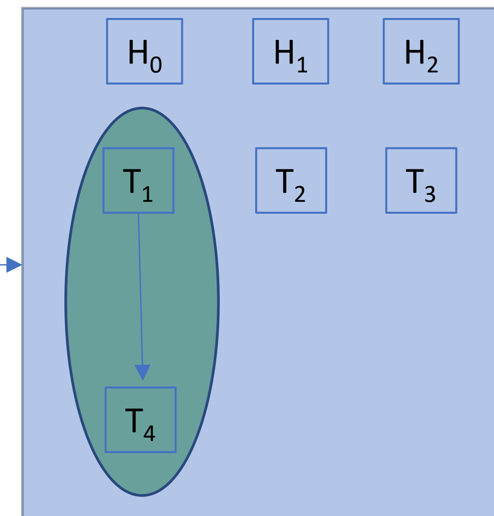
Notas:

- La primera llave { debe ir en la línea siguiente de #pragma omp parallel …….
- Todos los hilos ejecutan concurrentemente el código del bloque **parallel**, pero pueden realizar diferentes acciones dependiendo de su identificador
- Todos los hilos se sincronizan al finalizar las instrucciones del bloque (hay una barrera implícita al final de la construcción)

Regiones paralelas

```
#pragma omp parallel .....
{
        int id,nthreads; /*reparto mediante identificador*/
        id = omp_get_thread_num();
        nthreads = omp_get_num_threads();
        dowork(id, nthreads); /*Se ejecuta en paralelo la función dowork*/

        #pragma omp for .....
        /*Reparto de iteraciones (hay sincronización)*/
        for(....)
        {
        ...
        }

        #pragma omp single ....
        /*Solo el primer hilo que llega ejecuta el bloque (hay sincronización)*/
        {
        ...
        }

        #pragma omp master....
        /*El hilo H0 ejecuta el bloque de instrucciones (no hay sincronización)*/
        {
        ...
        }

        #pragma omp sections ......
         /*Reparto de tareas a partir del grafo de dependencias (hay sincronización)*/
        {/*En paralelo se ejecutan los códigos {T1;T4}, T2 y T4
              #pragma omp section
                     {T1;T4}
              #pragma omp section
                     T2;
              #pragma omp section
                     T3;
        }
}
```

- En una región paralela se pueden repartir las tareas mediante:
  - Los identificadores de los hilos
  - Construcciones del tipo **omp for**, **omp sections**, **omp single**, **omp master** ....
  - Todas las directivas excepto la **directiva master** tienen una **sincronización** al final de su bloque de instrucciones



Ejercicio 11 (Reparto de tareas mediante identificador)

Paraleliza la siguiente función:

```
int encontrar(int x[N], int valor){
    int i, encontrado=0;
    for (i=0; i<N && !encontrado; i++)
        if (x[i]==valor)
            encontrado=1;
    return encontrado;
}
```

No se puede paralelizar directamente el bucle **for**, pues no se puede determinar a priori el número de iteraciones a repartir al existir la condición !encontrado

Solución: realizar reparto explícito de iteraciones (cíclico) entre los hilos a partir de sus identificadores (ver sesión 3 de prácticas)

Ejemplo: 3 hilos y N=10

i=	0	1	2	3	4	5	6	7	8	9
H_i:	H_0	H_1	H_2	H_0	H_1	H_2	H_0	H_1	H_2	H_0

i(H0): 0, 3, 6, 9
i(H1): 1, 4, 7
i(H2): 2, 5, 8

Ejemplo: 4 hilos y N=10

i=	0	1	2	3	4	5	6	7	8	9
H_i:	H_0	H_1	H_2	H_3	H_0	H_1	H_2	H_3	H_0	H_1

i(H0): 0, 4, 8
i(H1): 1, 5, 9
i(H2): 2, 6
i(H3): 3, 7

Queremos que cada hilo ejecuta las iteraciones de su color

Ejercicio 11 (Reparto de tareas mediante identificador)

3 hilos

i=	0	1	2	3	4	5	6	7	8	9
H_i:	H_0	H_1	H_2	H_0	H_1	H_2	H_0	H_1	H_2	H_0

i(H0): 0, 3, 6, 9
i(H1): 1, 4, 7
i(H2): 2, 5, 8

4 hilos

i=	0	1	2	3	4	5	6	7	8	9
H_i:	H_0	H_1	H_2	H_3	H_0	H_1	H_2	H_3	H_0	H_1

i(H0): 0, 4, 8
i(H1): 1, 5, 9
i(H2): 2, 6
i(H3): 3, 7

Queremos modificar el bucle for para que cada hilo realice sus iteraciones (determinar vi e inc):

for(i=vi; i<N&&!encontrado; i+= inc)

hilo	vi=¿?
0	0
1	1
2	2
3	3

vi=hilo

nhilos	inc=¿?
3	3
4	4
5	5
6	6

inc=nhilos

```
int encontrar(int x[N], int valor){
    int i, encontrado=0;
    #pragma omp parallel  private(i)
    {
        int hilo= omp_get_num_thread();
        int  nhilos=omp_num_threads();
         for(i=hilo;  i<N&&!encontrado;  i+=nhilos)
            if (x[i]==valor)
                    encontrado=1;
        return encontrado;
    }
}
```

Regiones paralelas (reparto mediante **omp for**)

```
#pragma omp parallel .....
{
  ........
        #pragma omp for ....
         for(.......) {
              ...
         }
        .....
}
```

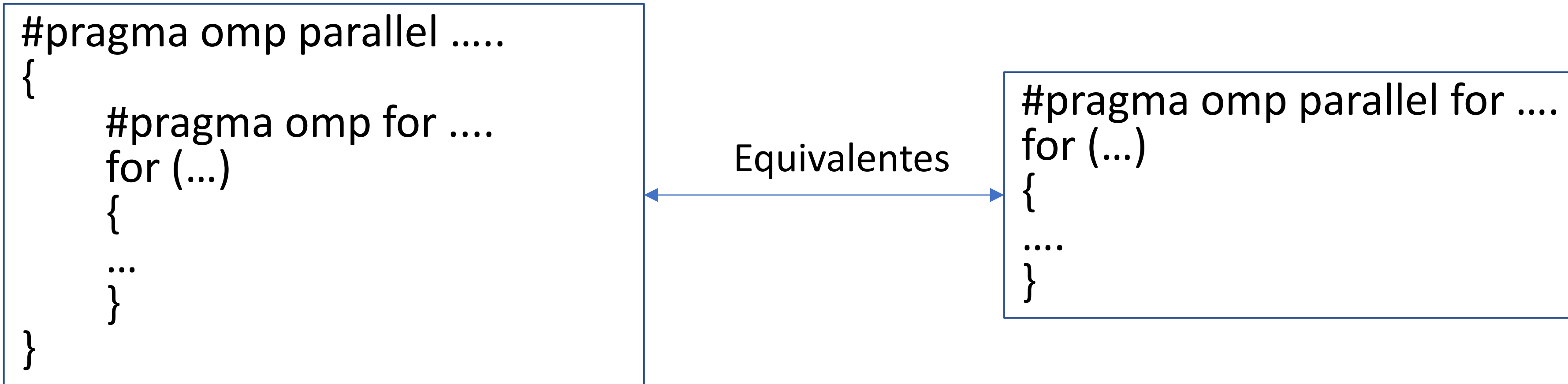
barrera

```
#pragma omp parallel .....
{
  ........
        #pragma omp parallel for ....
         for(.......) {
              ...
           }
  .......
}
```

- En una región paralela definida mediante **omp parallel** se puede repartir las iteraciones de un bucle **for** mediante la directiva **omp for**
- Hay una barrera implícita al final de la construcción (sincronización)
- Al igual que ocurría en la directiva **omp parallel for**, en la directiva **omp for** la variable iteradora de dicho bucle se convierte en privada

- Esta directiva puede usar las mismas cláusulas que tenía la directiva **omp parallel for**
- No usaremos la directiva **omp parallel for** dentro de una región paralela definida mediante **omp parallel** (no usaremos anidamiento paralelo)

Regiones paralelas (reparto mediante **omp for**)



- Cuando un bucle (T2) no depende del bucle anterior (T1), podemos eliminar la sincronización que tiene al final el bucle T1 poniendo la claúsula **nowait**

```
#pragma omp parallel …..
{
  ……..
      #pragma omp for …. nowait
       for(…….) /*T1*/
         {
           …
         }
      #pragma omp for …..
       for(…….) /*T2*/
         {
           …
         }
  …..
}
```

Regiones paralelas (directivas omp single y omp master)

- Cuando se usa la directiva **omp single** solo un hilo ejecuta el bloque de instrucciones (el primero que llega). Hay una barrera implícita
- Se pueden usar las siguientes cláusulas: **private, firstprivate, nowait**

```
#pragma omp parallel …..
{
        …
        #pragma omp single ….
        {
        …
        }
}
```

Hay una barrera implícita y por tanto una sincronización

- Cuando se usa la directiva **omp master** solo el hilo principal (H0) ejecuta el bloque de instrucciones, los demás no esperan (no hay barrera)

```
#pragma omp parallel …..
{
        …
        #pragma omp master
        {
        …
        }
}
```

Ejercicio 12 (Regiones paralelas)

Dada la función :

```
double fun( int n, double a[], double b[] ){
int i,ac,bc;
double asuma,bsuma,cota;
asuma = 0; bsuma = 0;
for (i=0; i<n; i++)
      asuma += a[i];
for (i=0; i<n; i++)
      bsuma += b[i];
cota = (asuma + bsuma) / 2.0 / n;
ac = 0; bc = 0;
for (i=0; i<n; i++) {
      if (a[i]>cota) ac++;
      if (b[i]>cota) bc++;
}
return cota/(ac+bc);
}
```

Paralelízala eficientemente mediante directivas OpenMP, usando para ello una sola región paralela

$$\text{asuma} = \sum_{i=0}^{n-1} a_i$$

$$\text{bsuma} = \sum_{i=0}^{n-1} b_i$$

No hay condición de carrera en las variables cota, ac y bc, pues todos los hilos escriben el mismo valor

```
double fun( int n, double a[], double b[] ){
    int i, ac, bc;
    double asuma,bsuma,cota;
    asuma = 0; bsuma = 0;
    #pragma omp parallel
    {
      #pragma omp for  reduction(+:asuma)  nowait
       for (i=0; i<n; i++)
            asuma += a[i];
      #pragma omp for  reduction(+:bsuma)
        for (i=0; i<n; i++)
           bsuma += b[i];
      cota = (asuma + bsuma) / 2.0 / n;
       ac = 0; bc = 0;
      #pragma omp for  reduction(+: ac, bc)
       for (i=0; i<n; i++) {
            if (a[i]>cota) ac++;
            if (b[i]>cota) bc++; }
    }
    return cota/(ac+bc);
}
```

Ejercicio 13

Dado el siguiente código:

```
double ej(double x[N], double y[N], double A[N][N]){
int i, j;
double aux,s=0.0;
for (i=0; i<N; i++)
     x[i] = x[i]*x[i];
for (i=0; i<N; i++)
     y[i] = 1.0+y[i];
for (i=0; i<N; i++)
     for (j=0; j<N; j++) {
          aux = x[i]-y[j];
          A[i][j] = aux;
          s += aux;
     }
return s;
}
```

$$s = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} A_{ij}$$

a) Paralelízala eficientemente mediante OpenMP, usando para ello una sola región paralela.

b) Calcula los costes computacionales en flops de las implementaciones secuencial y paralela, suponiendo que el número de hilos p es un divisor de N.

c) Calcula el speedup y la eficiencia del código paralelo.

a)

```
double ej(double x[N], double y[N], double A[N][N]){
int i, j;
double aux, s=0.0;
#pragma omp parallel
{
     #pragma omp for  nowait
     for (i=0; i<N; i++)
        x[i] = x[i]*x[i];
     #pragma omp for
     for (i=0; i<N; i++)
        y[i] = 1.0+y[i];
     #pragma omp for  private(j, aux) reduction(+:s)
     for (i=0; i<N; i++)
        for (j=0; j<N; j++) {
             aux = x[i]-y[j];
             A[i][j] = aux;
             s += aux;
        }
}
return s;
}
```

Ejercicio 13

```
double ej(double x[N], double y[N], double A[N][N]){
int i, j;
double aux, s=0.0;
#pragma omp parallel
{
	#pragma omp for nowait
	for (i=0; i<N; i++)
	    x[i] = x[i]*x[i];
	#pragma omp for
	for (i=0; i<N; i++)
	    y[i] = 1.0+y[i];
	#pragma omp for private(j, aux) reduction(+:s)
	for (i=0; i<N; i++)
	    for (j=0; j<N; j++) {
	        aux = x[i]-y[j];
	        A[i][j] = aux;
	        s += aux;
	    }
}
return s;
}
```

b) $$t(N) = \sum_{i=0}^{N-1} 1 + \sum_{i=0}^{N-1} 1 + \sum_{i=0}^{N-1}\sum_{j=0}^{N-1} 2 = N + N + 2N^2 \underset{N>>1}{\cong} 2N^2 \text{ flops}$$

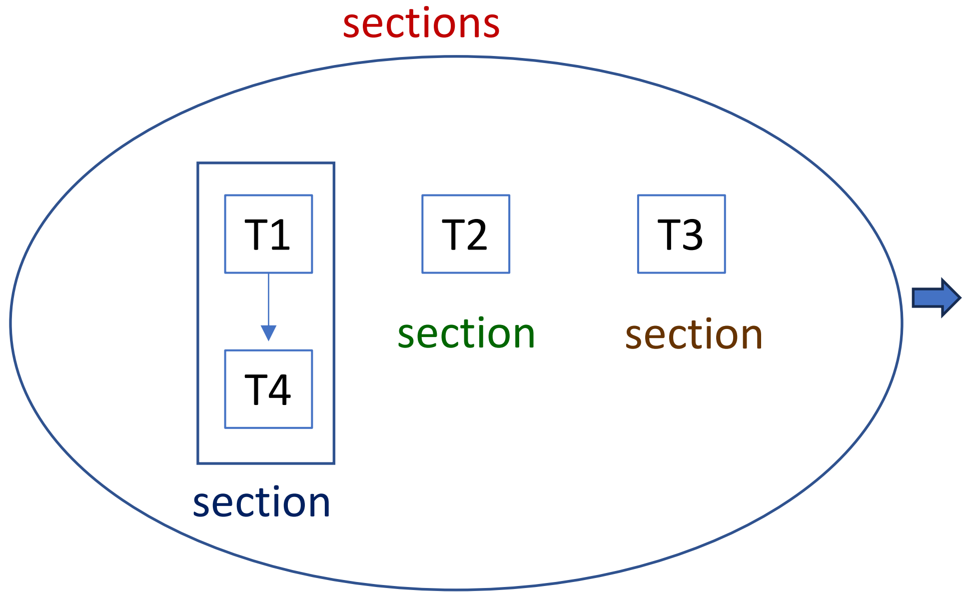
$$t(N,p) = \sum_{i=0}^{N/p-1} 1 + \sum_{i=0}^{N/p-1} 1 + \sum_{i=0}^{N/p-1}\sum_{j=0}^{N-1} 2 = \frac{N}{p} + \frac{N}{p} + \frac{2N^2}{p} \underset{N>>1}{\cong} \frac{2N^2}{p} \text{ flops}$$

c)

$$S(N,P) = \frac{t(N)}{t(N,p)} \underset{N>>1}{\cong} \frac{2N^2}{\frac{2N^2}{p}} = p$$

$$E(N,p) = \frac{S(N,p)}{p} \underset{N>>1}{\cong} \frac{p}{p} = 1$$

Regiones paralelas (reparto de tareas mediante directiva **sections**)

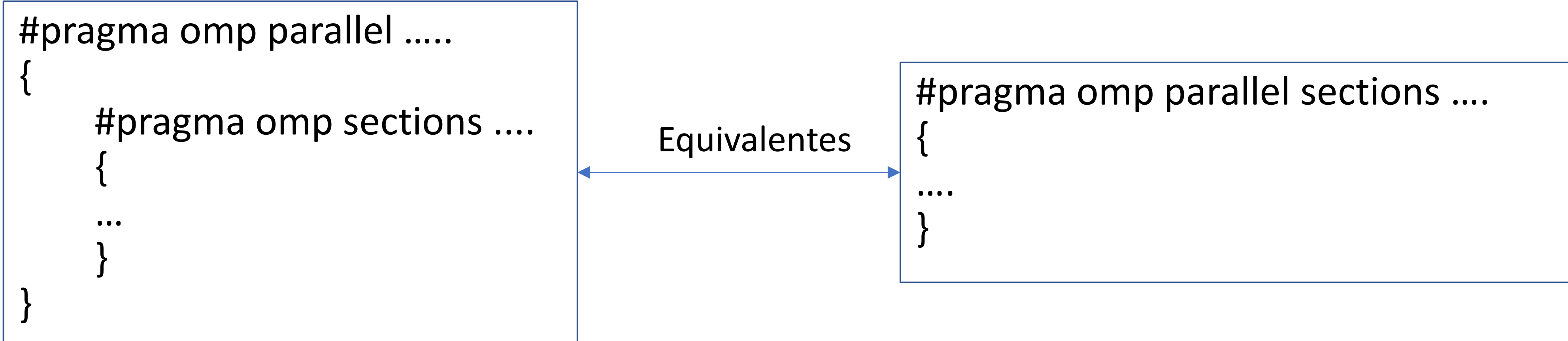



```
#pragma omp parallel .....
{
        .......
        #pragma omp sections ...... /*Construcción sections*/
        {
           #pragma omp section
                {T1;T4}
           #pragma omp section
                T2;
           #pragma omp section
                T3;
        }
        ......
}
```

- Cuando se usa una descomposición funcional de manera que se obtiene un grafo de dependencias, se usa la directiva **sections**, dentro de la cual hay varias directivas **section**
- En cada directiva **section** se especifican las tareas que se tienen que ejecutar secuencialmente (T1 y T4)
- Dos tareas que se encuentran en distintas directivas **section** se pueden ejecutar concurrentemente
- Hay una barrera implícita al final de la construcción
- Se pueden usar las siguientes cláusulas: **private, first/lastprivate, reduction, nowait**
- En el anterior grafo solo hay una dependencia entre las tareas T1 y T4: T1 y T4 las tiene que ejecutar un hilo, T2 la puede ejecutar otro hilo y lo mismo ocurre con T3

Regiones paralelas (reparto mediante **omp parallel sections**)

- Cuando hay una sola región paralela del tipo **sections**, se puede usar la directiva **omp parallel sections**



Ejercicio 14 (Regiones paralelas-secciones)

Sea el siguiente código secuencial en C:

```
x=f1(a);
y=f2(b);
z=f3(a,x,y);
t=f4(b,x,y);
z=f5(z,t);
```

donde tanto las variables como las funciones que aparecen se han definido previamente. Si los costes en flops de las funciones fi son los valores que aparecen en la siguiente tabla:

	f1	f2	f3	f4	f5
Flops	3n	2n	4n	3n	2n

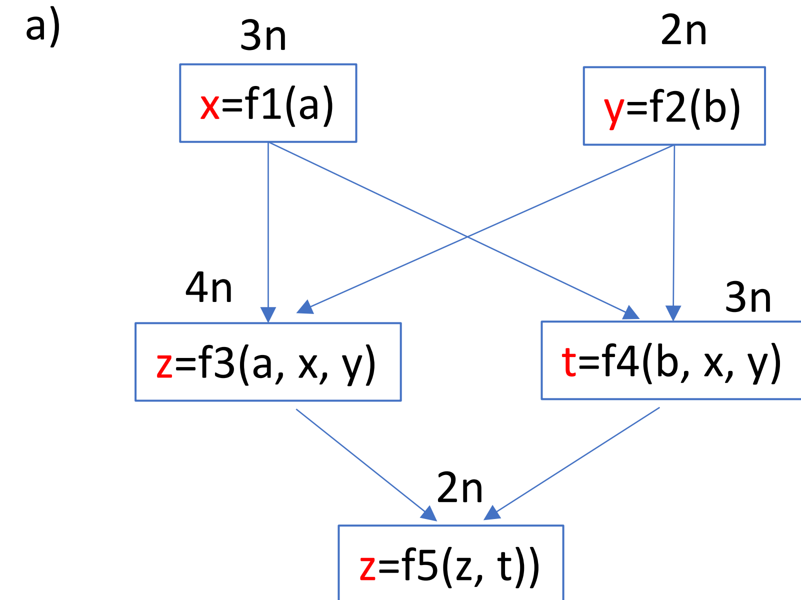
a) Dibuja el grafo de dependencias de las diferentes tareas.

b) Determina la longitud del camino crítico, el máximo grado de concurrencia y el grado medio de concurrencia.

c) Paraleliza de forma eficiente mediante OpenMP la función.

d) Determina el speedup y la eficiencia del código paralelo del apartado anterior al usar 2 hilos (n es el tamaño del problema).

a)



b)

Camino crítico: f1 $\rightarrow$ f3 $\rightarrow$ f5

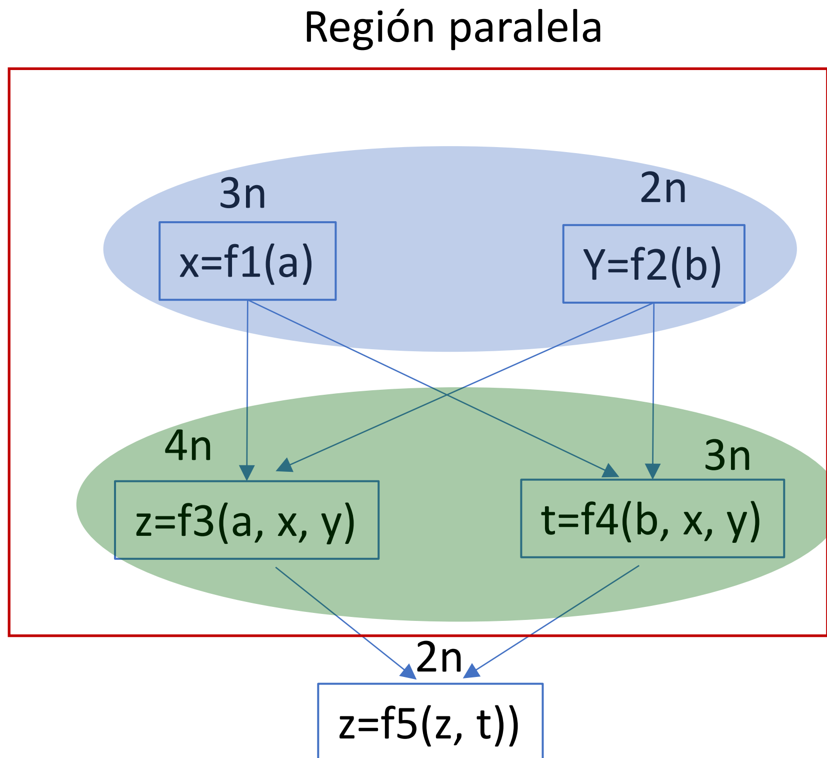
Longitud del camino crítico: 3n + 4n + 2n = 9n flops

Máximo grado de concurrencia: 2

Grado medio de concurrencia:

$$M = \frac{3n + 2n + 4n + 3n + 2n}{9n} = \frac{14}{9} = 1.\overset{\frown}{5}$$

Ejercicio 14 (Regiones paralelas-secciones)

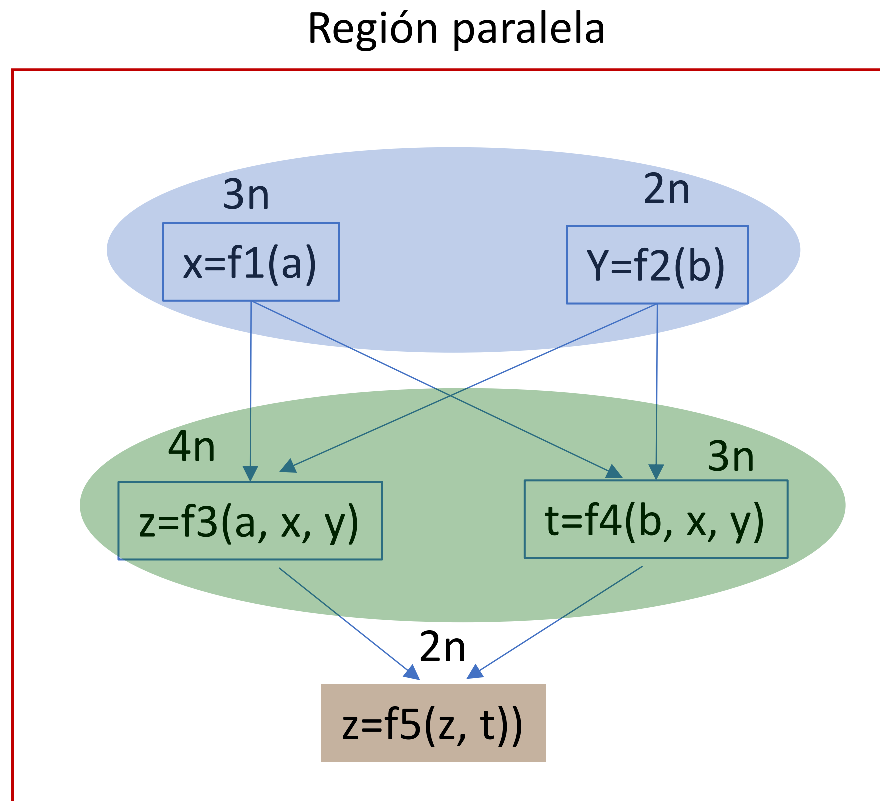


Apartado c)

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
            x=f1(a);
        #pragma omp section
            y=f2(b);
    }
    #pragma omp sections
    {
        #pragma omp section
            z=f3(a,x,y);
        #pragma omp section
            t=f4(b,x,y);
    }
}
z=f5(z,t);
```

Ejercicio 14 (Regiones paralelas-secciones)

Otra opción: crear una sola región paralela



Apartado c)

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
            x=f1(a);
        #pragma omp section
            y=f2(b);
    }
    #pragma omp sections
    {
        #pragma omp section
            z=f3(a,x,y);
        #pragma omp section
            t=f4(b,x,y);
    }
    #pragma omp single
      z=f5(z,t);
}
```

Ejercicio 15 (Regiones paralelas-secciones)

Sea el siguiente código:

```
double funcion(int n, double u[], double v[], double w[], double z[]){
   int i;
   double sv,sw,res;
   calcula_v(n,v); /* tarea T1 */
   calcula_w(n,w); /* tarea T2 */
   calcula_z(n,z); /* tarea T3 */
   calcula_u(n,u,v,w,z); /* tarea T4 */
   sv = 0;
   for (i=0; i<n; i++) sv = sv + v[i]; /* tarea T5 */
   sw = 0;
   for (i=0; i<n; i++) sw = sw + w[i]; /* tarea T6 */
   res = sv+sw;
   for (i=0; i<n; i++) u[i] = res*u[i]; /* tarea T7 */
   return res;
}
```

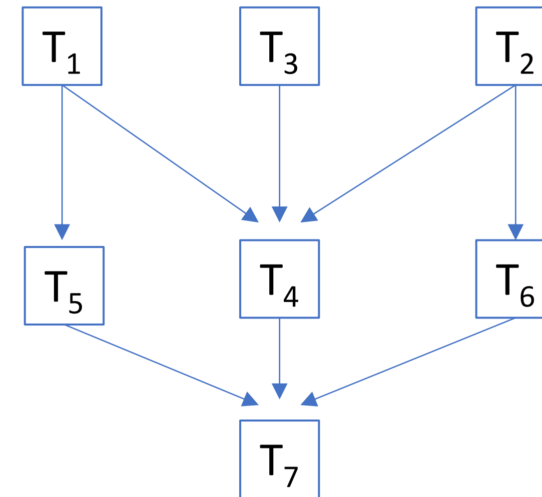
Las funciones **calcula_X** tienen como entrada los vectores que reciben como argumentos y con ellos modifican el vector X indicado. Cada función únicamente modifica el vector que aparece en su nombre.

Por ejemplo, la función calcula_u utiliza los vectores v, w y z para realizar unos cálculos que guarda en el vector u, pero no modifica ni v, ni w, ni z.

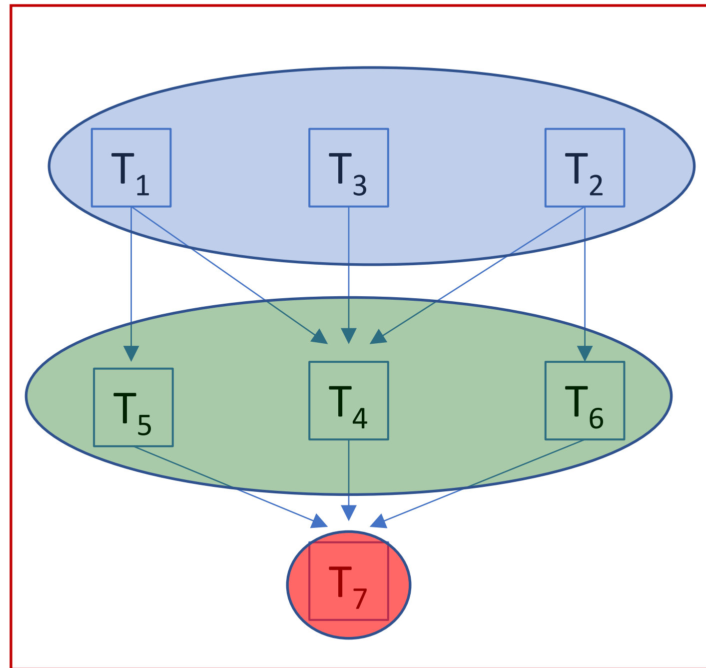
a) Dibuja el grafo de dependencias de las diferentes tareas.

b) Paraleliza la función de forma eficiente.

a)



Ejercicio 15 (Regiones paralelas-secciones)




```
double funcion(int n, double u[], double v[], double w[], double z[]){
int i;
double sv,sw,res;
#pragma omp parallel private(i)
{
 #pragma omp sections
 {
     #pragma omp section
         calcula_v(n,v); /* tarea T1 */
     #pragma omp section
         calcula_w(n,w); /* tarea T2 */
     #pragma omp section
         calcula_z(n,z); /* tarea T3 */
 }
 #pragma omp sections
 {
     #pragma omp section
         calcula_u(n,u,v,w,z); /* tarea T4*/
     #pragma omp section
     {
         sv = 0;
         for (i=0; i<n; i++) sv = sv + v[i]; /* tarea T5 */
     }
     #pragma omp section
     {
         sw = 0;
         for (i=0; i<n; i++) sw = sw + w[i]; /* tarea T6 */
     }
 }
 #pragma omp single
 {
     res = sv+sw;
     for (i=0; i<n; i++) u[i] = res*u[i]; /* tarea T7 */
 }
}
return res;
}
```

Otra posibilidad hubiese consistido en ejecutar res=sv+sw y el bucle for en la región secuencial (fuera de la región paralela); por lo tanto, no haría falta la directiva single

Ejercicio 16 (regiones críticas)

Paraleliza mediante OpenMP de dos formas distintas el siguiente código:

```
double A[N][N], maxi=0;
int i, j;
leer(A);/*función que lee los elementos de A*/
for (i=0; i<N; i++)
  for(j=1;j<N
    if (A[i][j]>maxi)
       maxi=A[i][j];
printf("El valor máximo de la matriz A es: %.2f\n",maxi);
```

```
double A[N][N], maxi=0;
leer(A);/*función que lee los elementos de A*/
int i, j;
#pragma omp parallel for private(j) reduction(max:maxi)
for (i=0; i<N; i++)
  for(j=1;j<N
    if (A[i][j]>maxi)
       maxi=aAi][j];
printf("El valor máximo de la matriz A es: %.2f\n",maxi);
```

```
double A[N][N], maxi=0;
int i, j;
leer(A);/*función que lee los elementos de A*/
#pragma omp parallel for private(j)
for (i=0; i<N; i++)
  for(j=1;j<N
    if (A[i][j]>maxi)
        #pragma omp critical
         if (A[i][j]>maxi)
            maxi=A[i][j];
printf("El valor máximo de la matriz A es: %.2f\n",maxi);
```

Ejercicio 17 (regiones críticas)

Paraleliza mediante OpenMP de dos formas distintas, si es posible, el siguiente código:

```
double A[N][N], maxi=0;
int i, j, imax=0, jmax;
leer(A);/*función que lee los elementos de A*/
for (i=0; i<N; i++) {
  for(j=1;j<N
    if (A[i][j]>maxi){
        maxi=A[i][j]; imax=i; jmax=j;
    }
printf("El valor máximo A[%d][%d]=%.2f\n", imax, jmax, maxi);
```

```
double A[N][N], maxi=0;
leer(A);/*función que lee los elementos de A*/
int i, j;
#pragma omp parallel for private(j) reduction(max:maxi)
for (i=0; i<N; i++) {
  for(j=1;j<N
     if (A[i][j]>maxi){
        maxi=A[i][j]; imax=i; jmax=j;
     }
printf("El valor máximo A[%d][%d]=%.2f\n", imax, jmax, maxi);
```

No se puede usar una reducción, pues, aunque calcula bien el valor de maxi, los valores de imax y jmax no se calculan bien, pues se produce una condición de carrera en las dos variables

```
double A[N][N], maxi=0;
int i, j;
leer(A);/*función que lee los elementos de A*/
#pragma omp parallel for private(j)
for (i=0; i<N; i++) {
  for(j=1;j<N
    if (A[i][j]>maxi)
        #pragma omp critical
         if (A[i][j]>maxi){
            maxi=A[i][j]; imax=i; jmax=j;
         }
printf("El valor máximo de la matriz A es: %.2f\n",maxi);
```

Ejercicio 18 (regiones críticas)

Paraleliza mediante OpenMP el siguiente código, utilizando una sola región paralela:

```
void normalize(double *a, int n){
        double maxi = a[0];
        int i;
        for (i=1; i<n; i++)
            if (maxi<a[i])
                  maxi=a[i];
        printf("Valor máximo=%f\n", maxi);
        for (i=0;i<n;i++) {
            a[i]=a[i]/maxi;
            printf("a[%d]=%f", i, a[i];
        }
}
```

de manera que

- se debe usar una planificación dinámica con chunk igual a 2 en el primer bucle for
- el valor máximo solo se debe imprimir una vez
- los valores de las componentes del vector **a** deben imprimirse ordenados.

a)

```
void normalize(double *a, int n){
        double maxi=a[0];
        int i;
        #pragma omp parallel
        {
           #pragma omp for  schedule(dynamic,2)
           for (i=1; i<n; i++)
                if (maxi<a[i])
                     #pragma omp critical
                          if (maxi<a[i])
                               maxi=a[i];
           #pragma omp single
             printf("Valor máximo=%f\n", maxi);
           #pragma omp for ordered
           for (i=0;i<n;i++) {
                a[i]=a[i]/maxi;
                     #pragma omp ordered
                     printf("a[%d]=%f", i, a[i]);
           }
         }
}
```

Ejercicio 19

Paraleliza mediante OpenMp el siguiente código:

```
double suma_v(double v[N]){
    int i;
    double suma=0.0;
    for (i=0;i<N;i++)
        suma+=v[i];
    return suma;
}
```

(a) Usando reducción

(b) Sin usar reducción

a)

```
double suma_v(double v[N]){
    int i;
    double suma=0.0;
    #pragma omp parallel for reduction(+: suma)
    for (i=0;i<N;i++)
        suma+=v[i];
    }
    return suma;
}
```

b)

```
double suma_v(double v[N]){
    int i;
    double suma=0.0;
    #pragma omp parallel
    {
      double  s=0.0;
       #pragma omp for
        for (i=0; i<N; i++)
            s+=v[i];
        #pragma atomic
            suma+=s;
    }
    return suma;
}
```

Ejercicio 20

La infinito-norma de una matriz se define como el máximo de las sumas de los valores absolutos de los elementos de cada fila:

$$\|A\|_\infty = \max_{i=0,1,\cdots,n-1} \left\{ \sum_{j=0}^{n-1} |a_{ij}| \right\}$$

La siguiente función calcula la infinito-norma de una matriz cuadrada:

```
double infNorm(double A[N][N]) {
    int i, j;
    double s, norm=0.0;
    for (i=0; i<N; i++) {
        s = 0.0;
        for (j=0; j<N; j++)
            s += fabs(A[i][j]);
        if (s>norm)
            norm = s;
    }
    return norm;
}
```

a) Realiza una implementación paralela mediante OpenMP de dicho algoritmo (sin usar reducciones).

Solución:

```
double infNorm(double A[N][N]){
    int i,j;
    double s, norm=0.0;
    #pragma omp parallel for  private(s, j)
    for (i=0; i<N; i++) {
        s = 0.0;
        for (j=0; j<N; j++)
            s += fabs(A[i][j]);
            if (s>norm)
                #pragma omp critical
                if (s>norm)
                    norm = s;
    }
    return norm;
}
```

Ejercicio 20

b) Realiza otra implementación paralela mediante OpenMP usando una reducción.

```
double infNorm(double A[N][N]) {
    int i, j;
    double s, norm=0.0;
    for (i=0; i<N; i++) {
        s = 0.0;
        for (j=0; j<N; j++)
            s += fabs(A[i][j]);
        if (s>norm)
            norm = s;
    }
    return norm;
}
```

```
double infNorm(double A[N][N]){
    int i, j;
    double s, norm=0.0;
    #pragma omp parallel for private(s, j) reduction(max:norm)
    for (i=0; i<N; i++) {
        s = 0.0;
        for (j=0; j<N; j++)
            s += fabs(A[i][j]);
        if (s>norm)
            norm = s;
    }
    return norm;
}
```

Ejercicio 20

c) Calcula el coste computacional en flops de la versión original secuencial y de la versión paralela desarrollada, suponiendo que la función **fabs** tiene un coste de 1 flop.

```
double infNorm(double A[N][N]) {
    int i, j;
    double s, norm=0.0;
    for (i=0; i<N; i++) {
        s = 0.0;
        for (j=0; j<N; j++)
            s += fabs(A[i][j]);
        if (s>norm)
            norm = s;
    }
    return norm;
}
```

```
double infNorm(double A[N][N]){
    int i, j;
    double s, norm=0.0;
    #pragma omp parallel for private(s, j) reduction(max:norm)
    for (i=0; i<N; i++) {
        s = 0.0;
        for (j=0; j<N; j++)
            s += fabs(A[i][j]);
        if (s>norm)
            norm = s;
    }
    return norm;
}
```

$$t(N)=\sum_{i=0}^{N-1}\sum_{j=0}^{N-1} 2 = \sum_{i=0}^{N-1} 2n = 2N^2 \text{ flops.}$$

$$t(N,p) = \sum_{i=0}^{N/p-1}\sum_{j=0}^{N-1} 2 = \sum_{i=0}^{N/p-1} 2n = \frac{2N^2}{p} \text{ flops}$$

$$S(n,p) = \frac{t(n)}{t(n,p)} = p. \quad E(n,p) = \frac{S(n,p)}{p} = 1$$

Ejercicio 21

El siguiente código permite ordenar un vector **v** de **n** números reales ascendentemente:

```
int ordenado = 0;
double a;
while( !ordenado ) {
    ordenado = 1;
    for( i=0; i<n-1; i+=2 )
        if( v[i]>v[i+1] ) {
            a = v[i]; v[i] = v[i+1]; v[i+1] = a;
            ordenado = 0;
        }
    for( i=1; i<n-1; i+=2 )
        if( v[i]>v[i+1] ) {
            a = v[i]; v[i] = v[i+1]; v[i+1] = a;
            ordenado = 0;
        }
}
```

a) Paraleliza mediante OpenMP el código anterior.

Solución:

```
int ordenado = 0;
double a;
while( !ordenado ) {
    ordenado = 1;
   #pragma omp parallel for  private(a)
    for( i=0; i<n-1; i+=2 )
        if( v[i]>v[i+1] ) {
            a = v[i]; v[i] = v[i+1]; v[i+1] = a;
            ordenado = 0;
        }
   #pragma omp parallel for  private(a)
    for( i=1; i<n-1; i+=2 ) {
        if( v[i]>v[i+1] ) {
            a = v[i]; v[i] = v[i+1];
            v[i+1] = a;
            ordenado = 0;
        }
```

Ejercicio 21

b) Modificar el código para contabilizar el número de intercambios que se producen, es decir, el número de veces que se entra en cualquiera de las dos estructuras if

```
int ordenado = 0;
double a;
while( !ordenado ) {
    ordenado = 1;
    for( i=0; i<n-1; i+=2 )
        if( v[i]>v[i+1] ) {
            a = v[i]; v[i] = v[i+1]; v[i+1] = a;
            ordenado = 0;
        }
    for( i=1; i<n-1; i+=2 )
        if( v[i]>v[i+1] ) {
            a = v[i]; v[i] = v[i+1]; v[i+1] = a;
            ordenado = 0;
        }
}
```

Solución:

```
int ordenado = 0,  cont=0 ;
double a;
while( !ordenado ) {
    ordenado = 1;
    #pragma omp parallel for private(a)  reduction(+: cont)
    for( i=0; i<n-1; i+=2 )
        if( v[i]>v[i+1] ) {
            a = v[i]; v[i] = v[i+1]; v[i+1] = a;
            ordenado = 0;
            cont++;
        }
    #pragma omp parallel for private(a)  reduction(+: cont)
    for( i=1; i<n-1; i+=2 ) {
        if( v[i]>v[i+1] ) {
            a = v[i]; v[i] = v[i+1];
            v[i+1] = a;
            ordenado = 0;
            cont++;
        }
```

Cuestión 1-8

```
#define N 6000
#define PASOS 6
double funcion1(double A[N][N], double b[N], double x[N])
{
    int i, j, k, n=N, pasos=PASOS;
    double max=-1.0e308, q, s, x2[N];
    for (k=0; k<pasos; k++) {
        q=1;
        for (i=0; i<n; i++) {
            s = b[i];
            for (j=0; j<n; j++)
                s -= A[i][j]*x[j];
            x2[i] = s;
            q *= s;
        }
        for (i=0;i<n;i++)
            x[i] = x2[i];
        if (max<q)
            max = q;
    }
return max;
}
```

a) Paraleliza el código usando OpenMP ¿Por qué lo haces así? Se valorarán más aquellas soluciones que sean más eficientes

Solución

a) El bucle más externo no se puede paralelizar, pues en cada iteración x se actualiza a partir de x de la iteración anterior (hay dependencia entre iteraciones)

En cada iteración del bucle externo con variable iteradora k, se calcula:

$$x2_i = s = b_i - \sum_{j=0}^{n-1} A_{ij} x_j \,, i = 0,..,n-1$$

$$q = \prod_{i=0}^{n-1} x2_i$$

$$x = x2$$ → x cambia en cada iteración del bucle k

$$\max = \max_k\{q\}$$

Cuestión 1-8

```
#define N 6000
#define PASOS 6
double funcion1(double A[N][N], double b[N], double x[N]){
    int i, j, k, n=N, pasos=PASOS;
    double max=-1.0e308, q, s, x2[N];
    for (k=0;k<pasos;k++) {
        q=1;
        for (i=0;i<n;i++) {
            s = b[i];
            for (j=0;j<n;j++)
                s -= A[i][j]*x[j];
            x2[i] = s;
            q *= s;
        }
        for (i=0;i<n;i++)
            x[i] = x2[i];
        if (max<q)
            max = q;
    }
return max;
}
```

$$x2_i = s = b_i - \sum_{j=0}^{n-1} A_{ij}x_j \, , i = 0, \ldots, n-1$$

$$q = \prod_{i=0}^{n-1} x2_i$$

$$x = x2$$

$$\max = \max_k \{q\}$$

a) Paralelización de los bucles for

```
double funcion1(double A[N][N], double b[N], double x[N]){
    int i, j, k, n=N, pasos=PASOS;
    double max=-1.0e308, q, s, x2[N];
    for (k=0;k<pasos;k++) {
        q=1;
        #pragma omp parallel for  private(s, j) reduction(*: q)
        for (i=0;i<n;i++) {
            s = b[i];
            for (j=0;j<n;j++)
                s -= A[i][j]*x[j];
            x2[i] = s;
            q *= s;
        }
        #pragma omp parallel for
        for (i=0;i<n;i++)
            x[i] = x2[i];
        if (max<q)
            max = q;
    }
return max;
}
```

Cuestión 1-8

```
#define N 6000
#define PASOS 6
double funcion1(double A[N][N], double b[N], double x[N]){
    int i, j, k, n=N, pasos=PASOS;
    double max=-1.0e308, q, s, x2[N];
    for (k=0; k<pasos; k++) {
        q=1;
        for (i=0; i<n; i++) {
            s = b[i];
            for (j=0; j<n; j++)
                s -= A[i][j]*x[j];
            x2[i] = s;
            q *= s;
        }
        for (i=0;i<n;i++)
            x[i] = x2[i];
        if (max<q)
            max = q;
    }
return max;
}
```

b) Indica el coste teórico (en flops) que tendría una iteración del bucle k del código secuencial

$$t(n) = \sum_{i=0}^{n-1}\left(\sum_{j=0}^{n-1} 2 + 1\right) = \sum_{i=0}^{n-1}(2n+1) = 2n^2 + n \approx 2n^2 \text{ flops}$$

c) Considerando una única iteración del bucle k (PASOS=1), indica el speedup y la eficiencia que podrá obtenerse con *p* hilos, suponiendo que hay tantos núcleos/procesadores como hilos y que N es un múltiplo exacto de *p*.

$$t(n,p) = \sum_{i=0}^{n/p-1}\left(\sum_{j=0}^{n-1} 2 + 1\right) = \frac{2n^2}{p} + \frac{n}{p} \approx \frac{2n^2}{p} \; flops$$

$$S(n,p) = \frac{2n^2}{\frac{2n^2}{p}} = p$$

E=1

Cuestión 2-7

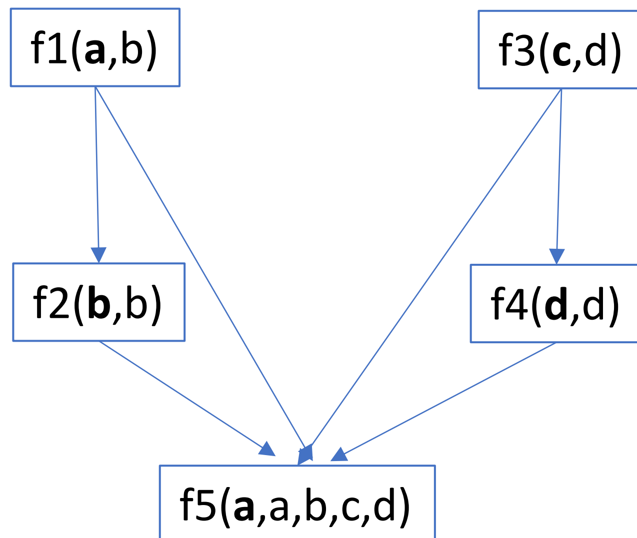
Dada la siguiente función:

```
void func(double a[],double b[],double c[],double d[]){
    f1(a, b);
    f2(b, b);
    f3(c, d);
    f4(d, d);
    f5(a, a, b, c, d);
}
```

El primer argumento de todas las funciones usadas es de salida y el resto de argumentos son argumentos de entrada. Por ejemplo, f1(a,b) es una función que a partir del vector b modifica el vector a.

a) Dibuja el grafo de dependencias de tareas e indica al menos 3 tipos diferentes de dependencias que aparezcan en este problema.

Solución: a)



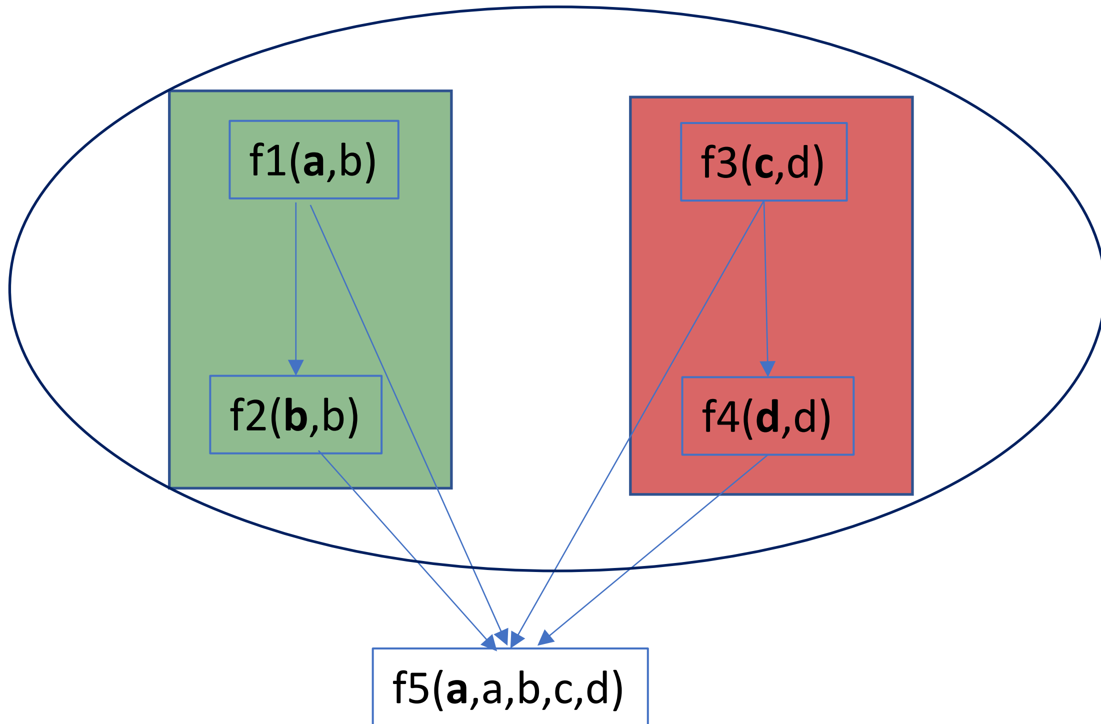
Dependencias de flujo: f5 depende de todas las demás

Antidependencias: f1→ f2 y f3→ f4

Dependencia de salida: f1→ f5

Cuestión 2-7

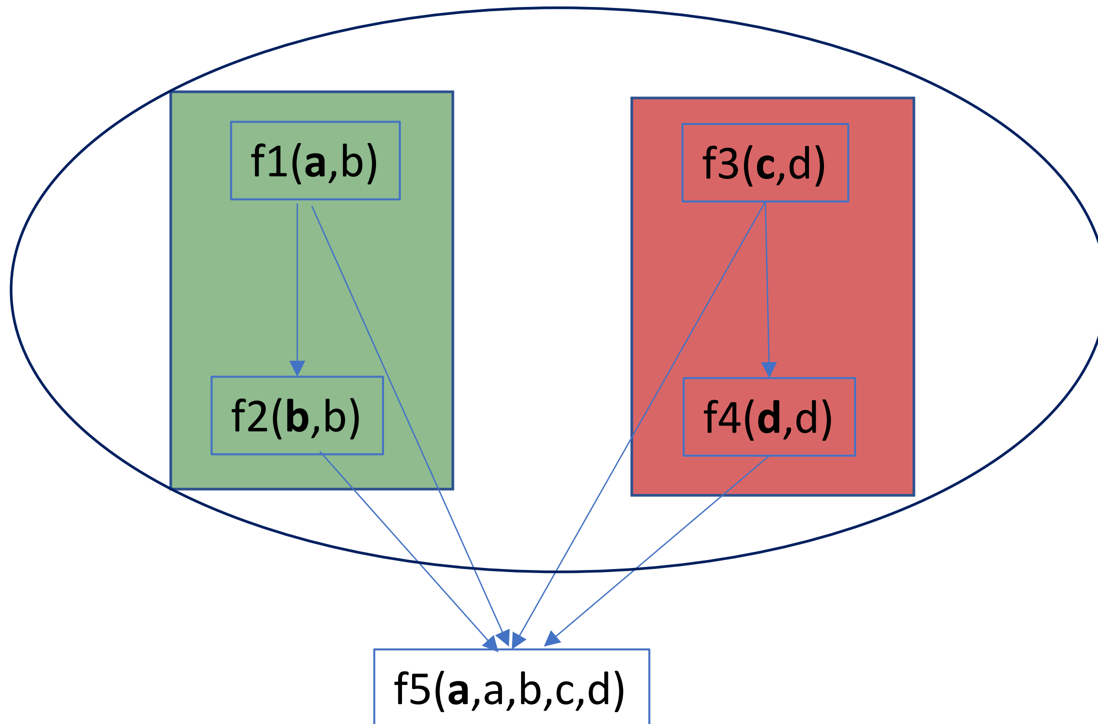
b) Paraleliza la función mediante directivas OpenMP




```
void func(double a[],double b[],double c[],double d[])
{
    #pragma omp parallel sections
    {
        #pragma omp section
        {
            f1(a,b);
            f2(b,b);
        }
        #pragma omp section
        {
            f3(c,d);
            f4(d,d);
        }
    }
    f5(a,a,b,c,d);
}
```

Cuestión 2-7

c) Suponiendo que todas las funciones tienen el mismo coste y que se dispone de un número de procesadores arbitrario, ¿cuál será el speedup máximo?



$$t_1 = 1 + 1 + 1 + 1 + 1 = 5$$

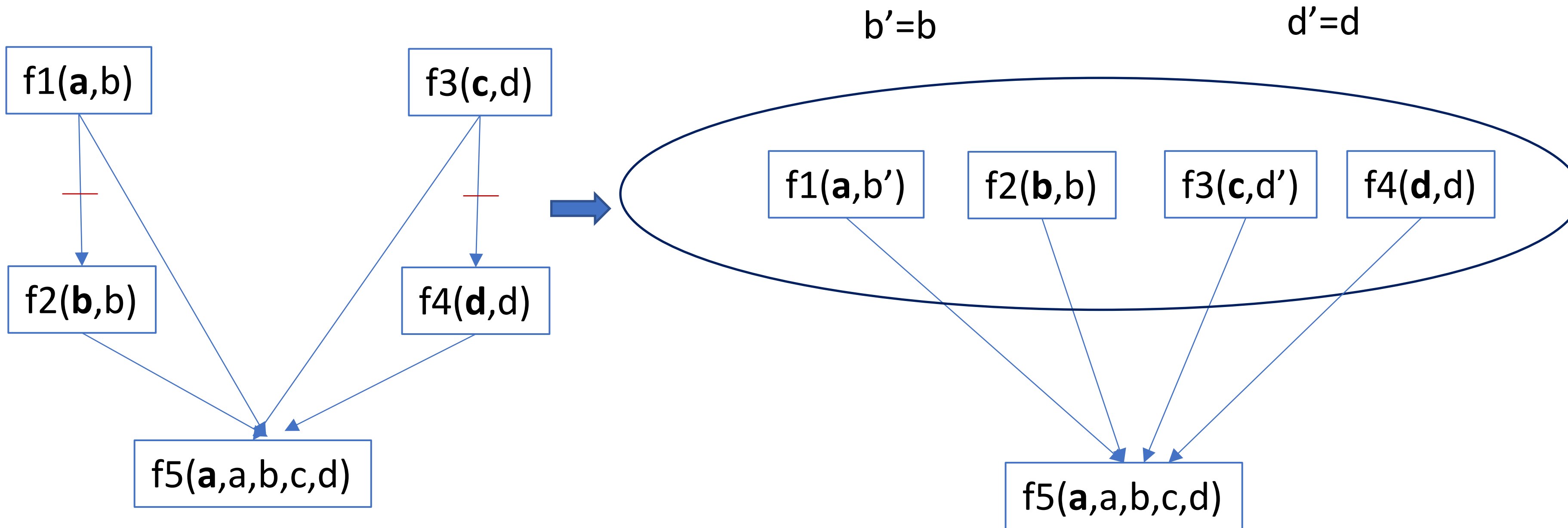
$$t_p = 2 + 1 = 3 \ (p \geq 2)$$

$$S_p = \frac{5}{3} = 1.\widehat{6} \ (p \geq 2)$$

Cuestión 2-7

d) ¿Se podría mejorar este speedup utilizando replicación de datos?

Si replicamos los datos b y d, entonces las anti-dependencias quedan eliminadas: Copiar b y d antes de ejecutar el código, de manera que en f1 y f3 usarán las copias y f2 y f4 los datos reales.



$$t_1 = 1 + 1 + 1 + 1 + 1 = 5$$

$$t_p = 1 + 1 = 2 (p \geq 4)$$

$$S_p = \frac{5}{2} = 2.5 \ (p \geq 4)$$

Cuestión 3-2

Dada la función:

```
void f(int n, double v[], double x[], int ind[]){
   int i;
   double aux;
   for (i=0; i<n; i++) {
      aux=f2(v[i]);
      if (aux>x[ind[i]])
            x[ind[i]] = aux;
   }
}
```

Paralelizar la función, teniendo en cuenta que f2 es una función muy costosa. Se valorará que la solución aportada sea eficiente.

Nota:

- Se asume que f2 no tiene efectos laterales y su resultado sólo depende de su argumento de entrada.
- El tipo de retorno de la función f2 es double.
- La función max devuelve el máximo de dos números.

```
void f(int n, double v[], double x[], int ind[]){
   int i;
   double aux;
   #pragma omp parallel for  private(aux)
   for (i=0; i<n; i++) {
      aux=f2(v[i]);
      if (aux>x[ind[i]]) {
         #pragma omp critical
         if (aux>x[ind[i]])
               x[ind[i]] = aux;
   }
}
```

Cuestión 1-11

Dada la siguiente función:

```
double f(double A[N][N], double B[N][N], double vs[N], double bmin) {
    int i, j;
    double x, y, aux, stot=0;
    for (i=0; i<N; i++) {
        aux = 0;
        for (j=0; j<N; j++) {
            x = A[i][j]*A[i][j]/2.0;
            A[i][j] = x;
            aux += x;
        }
        for (j=i; j<N; j++) {
            if (B[i][j]<bmin) y = bmin;
            else y = B[i][j];
            B[i][j] = 1.0/y;
        }
        vs[i] = aux;
        stot += vs[i];
    }
    return stot;
}
(
```

a) Paraleliza (eficientemente) el bucle i mediante OpenMP

Solución

$$x = \frac{A_{ij}^2}{2}$$

$$A_{ij} = x$$

$$aux = \sum_{j=0}^{N-1} A_{ij}$$

$$vs_i = aux$$

$$stot = \sum_{i=0}^{N-1} vs_i$$

Antes del bucle for (i=0; i<N; i++) pondríamos:

```
#pragma omp parallel for  private(aux, j, x, y) reduction(+:stot)
```

Cuestión 1-11 (Paralelismo de bucles)

b) Paraleliza (eficientemente) los dos bucles j mediante OpenMP.

```
double f(double A[N][N], double B[N][N], double vs[N], double bmin) {
    int i, j;
    double x, y, aux, stot=0;
    for (i=0; i<N; i++) {
        aux = 0;
        for (j=0; j<N; j++) {
            x = A[i][j]*A[i][j]/2.0;
            A[i][j] = x;
            aux += x;
        }
        for (j=i; j<N; j++) {
            if (B[i][j]<bmin) y = bmin;
            else y = B[i][j];
            B[i][j] = 1.0/y;
        }
        vs[i] = aux;
        stot += vs[i];
    }
    return stot;
}
```

$$A_{ij} = \frac{A_{ij}^2}{2}$$

$$aux = \sum_{j=0}^{N-1} A_{ij}$$

$$vs_i = aux = \sum_{j=0}^{N-1} A_{ij}$$

$$y = \max(Bij, bmin)$$

$$B_{ij} = 1/y$$

$$stot = \sum_{i=0}^{N-1} vs_i$$

```
double f(....) {
    int i, j;
    double x, y, aux, stot=0;
    for (i=0; i<N; i++) {
        aux = 0;
        #pragma omp parallel
        {
            #pragma omp for  private(x) reduction(+:aux) nowait
            for (j=0; j<N; j++) {
                x = A[i][j]*A[i][j]/2.0;
                A[i][j] = x;
                aux += x;
            }
            #pragma omp for  private(y)
            for (j=i; j<N; j++) {
                if (B[i][j]<bmin) y = bmin;
                else y = B[i][j];
                B[i][j] = 1.0/y;
            }
        }
        vs[i] = aux;
        stot += vs[i];
    }
    return stot;
}
```

Cuestión 1-11

c) Calcula el coste secuencial del código original.

```
double f(double A[N][N], double B[N][N], double vs[N], double bmin) {
    int i, j;
    double x, y, aux, stot=0;
    for (i=0; i<N; i++) {
        aux = 0;
        for (j=0; j<N; j++) {
            x = A[i][j]*A[i][j]/2.0;
            A[i][j] = x;
            aux += x;
        }
        for (j=i; j<N; j++) {
            if (B[i][j]<bmin) y = bmin;
            else y = B[i][j];
            B[i][j] = 1.0/y;
        }
        vs[i] = aux;
        stot += vs[i];
    }
    return stot;
}
```

$$\sum_{i=0}^{N-1}\left(\sum_{j=0}^{N-1} 3 + \sum_{j=i}^{N-1} 1 + 1\right) \approx \sum_{i=0}^{N-1}(3N + N - i)$$

$$= \sum_{i=0}^{N-1} 4N - \sum_{i=0}^{N-1} i \approx 4N^2 - \frac{N^2}{2} = \frac{7N^2}{2} \text{ flops}$$

Cuestión 1-11

d) Suponiendo que paralelizamos solo el primer bucle j, calcula el coste paralelo de dicha versión. Obtén el speedup y la eficiencia en el caso de que se disponga de N procesadores.

```
double f(.....) {
    int i, j;
    double x, y, aux, stot=0;
    for (i=0; i<N; i++) {
        aux = 0;
        for (j=0; j<N; j++) {
            x = A[i][j]*A[i][j]/2.0;
            A[i][j] = x;
            aux += x;
        }
        for (j=i; j<N; j++) {
            if (B[i][j]<bmin) y = bmin;
            else y = B[i][j];
            B[i][j] = 1.0/y;
        }
        vs[i] = aux;
        stot += vs[i];
    }
    return stot;
}
```

$$t(N,p) = \sum_{i=0}^{N-1} \left(\sum_{j=0}^{N/p-1} 3 + \sum_{j=i}^{N-1} 1 + 1 \right) \approx$$

$$\sum_{i=0}^{N-1} \left(\frac{3N}{p} + N - i \right) \approx \frac{3N^2}{p} + N^2 - \frac{N^2}{2} = \frac{3N^2}{p} + \frac{N^2}{2} \text{ flops}$$

$$p{=}N \Rightarrow t(N,N) \approx \frac{3N^2}{N} + \frac{N^2}{2} \approx \frac{N^2}{2} \text{ flops}$$

$$S(n,n) = \frac{\frac{7N^2}{2}}{\frac{N^2}{2}} = 7$$

$$E(n,n) = \frac{7}{N}$$

Cuestión 2-10

Se quiere paralelizar el siguiente programa mediante OpenMP, donde **genera** es una función previamente definida en otro lugar

```
double fun1(double a[], int n, int v0){
    int i;
    a[0] = v0;
    for (i=1;i<n;i++)
        a[i] = genera(a[i-1],i);
}
```

```
double compara(double x[], double y[], int n){
    int i;
    double s=0;
    for (i=0;i<n;i++)
        s += fabs(x[i]-y[i]);
    return s;
}
```

```
/* fragmento del programa principal (main) */
int i, n=10;
double a[10], b[10], c[10], x=5, y=7, z=11, w;
fun1(a,n,x); /* T1 */
fun1(b,n,y); /* T2 */
fun1(c,n,z); /* T3 */
x = compara(a,b,n); /* T4 */
y = compara(a,c,n); /* T5 */
z = compara(c,b,n); /* T6 */
w = x+y+z; /* T7 */
printf("w:%f\n", w);
```

a) Paraleliza el código de forma eficiente a nivel de bucles

a) Solución

- El bucle i de la función **fun1** no se puede paralelizar, pues hay dependencia entre iteraciones
- Se podría paralelizar el bucle i de la función **compara**:

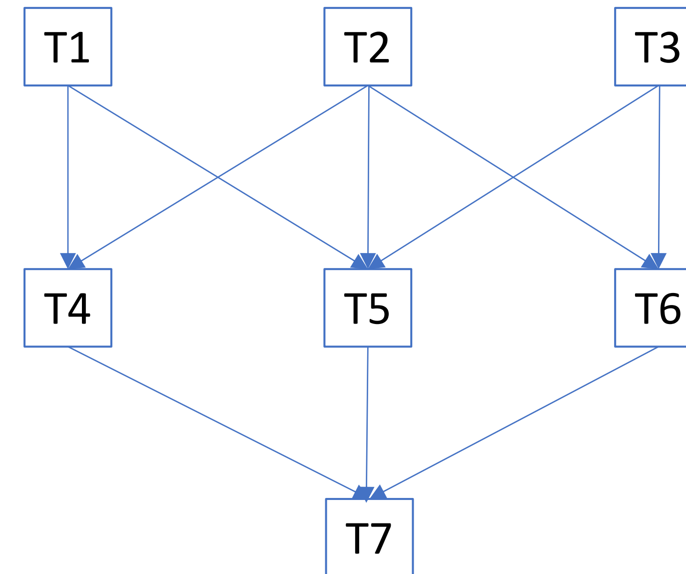
```
double compara(double x[], double y[], int n){
    int i;
    double s=0;
    #pragma omp parallel for reduction(+:s)
    for (i=0;i<n;i++)
        s += fabs(x[i]-y[i]);
    return s;
}
```

Cuestión 2-10

b) Dibuja el grafo de dependencias de tareas, según la numeración de tareas indicada en el código.

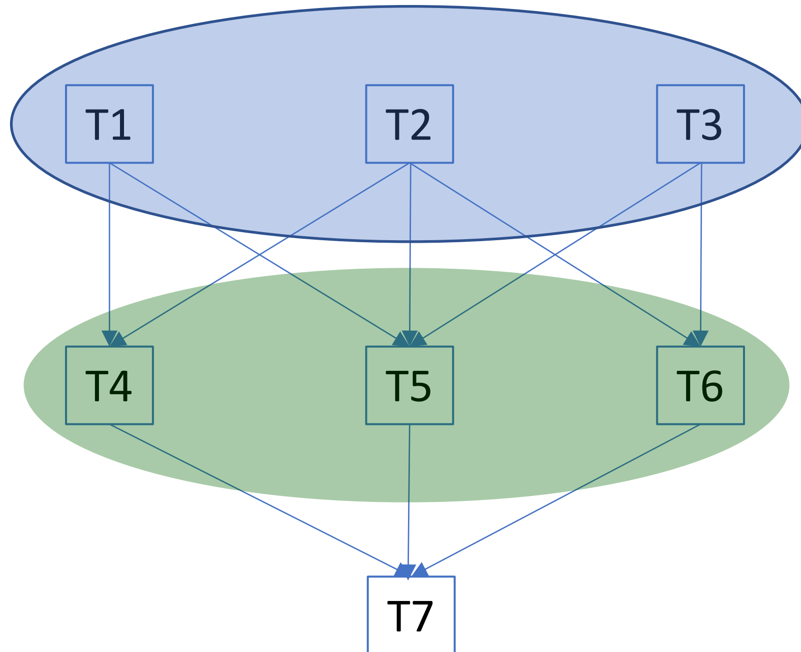
b) **Solución**

```
/* fragmento del programa principal (main) */
int i, n=10;
double a[10], b[10], c[10], x=5, y=7, z=11, w;
fun1(a,n,x); /* T1 */
fun1(b,n,y); /* T2 */
fun1(c,n,z); /* T3 */
x = compara(a,b,n); /* T4 */
y = compara(a,c,n); /* T5 */
z = compara(c,b,n); /* T6 */
w = x+y+z; /* T7 */
printf("w:%f\n", w);
```



Cuestión 2-10

c) Paraleliza el código de forma eficiente a nivel de tareas, a partir del grafo de dependencias anterior.




```
int i, n=10;
double a[10], b[10], c[10], x=5, y=7, z=11, w;
#pragma omp parallel sections
{
    #pragma omp section
        fun1(a,n,x); /*T1*/
    #pragma omp section
        fun1(b,n,y); /*T2*/
    #pragma omp section
        fun1(c,n,z); /*T3*/
}
#pragma omp parallel sections
{
    #pragma omp section
        x = compara(a,b,n); /*T4*/
    #pragma omp section
        y = compara(a,c,n); /*T5*/
    #pragma omp section
        z = compara(c,b,n); /*T6*/
}
w= x+y+z;  /*T7*/
printf("w:%f\n", w);
```

Nota: La implementación también se podría realizar creando una región paralela (**omp parallel**) y repartir las la ejecución de las tareas mediante 2 sections (**omp sections**), dejando T7 fuera de la región paralela.

Cuestión 2-10

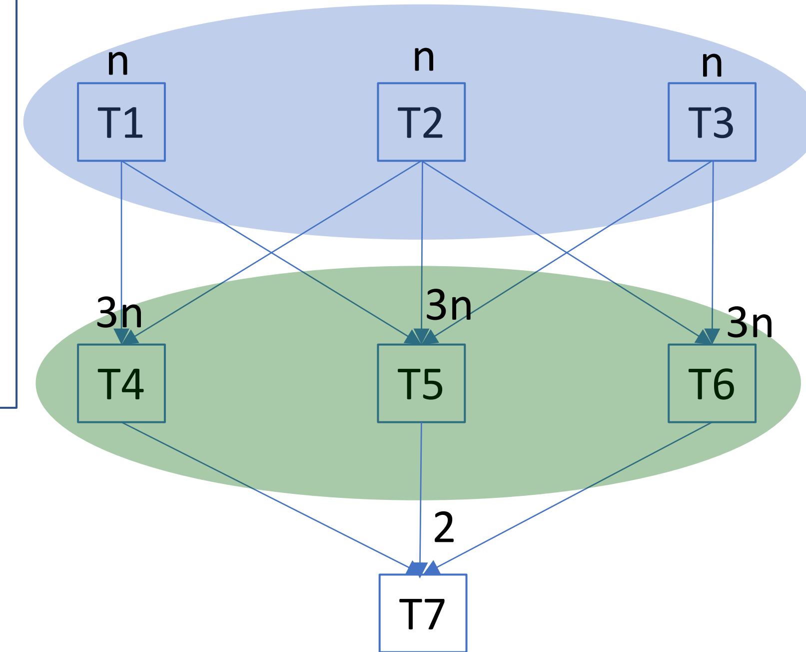
d) Obtén el tiempo secuencial (asume que una llamada a las funciones **genera** y **fabs** cuesta 1 flop) y el tiempo paralelo asumiendo que hay 3 procesadores. Calcular el speed-up y la eficiencia, suponiendo que tenemos 3 hilos.

```
double fun1(double a[], int n, int v0){
    int i;
    a[0] = v0;
    for (i=1;i<n;i++)
        a[i] = genera(a[i-1],i);
}
```

$$t_f = \sum_{i=1}^{n-1} 1 \approx n \text{ flops}$$

```
double compara(double x[], double y[], int n){
    int i;
    double s=0;
    for (i=0;i<n;i++)
        s += fabs(x[i]-y[i]);
    return s;
}
```

$$t_c = \sum_{i=0}^{n-1} 3 = 3n \text{ flops}$$




```
/* fragmento del programa principal (main) */
int i, n=10;
double a[10], b[10], c[10], x=5, y=7, z=11, w;
fun1(a,n,x); /* T1 */
fun1(b,n,y); /* T2 */
fun1(c,n,z); /* T3 */
x = compara(a,b,n); /* T4 */
y = compara(a,c,n); /* T5 */
z = compara(c,b,n); /* T6 */
w = x+y+z; /* T7 */
printf("w:%f\n", w);
```

$$t(n) = n + n + n + 3n + 3n + 3n + 2 \approx 12n \text{ flops}$$

$$t(n,3) = n + 3n + 2 \approx 4n \text{ flops}$$

$$S(n,3) \approx \frac{12n}{4n} = 3$$

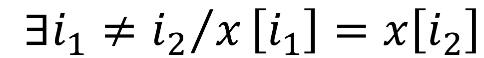
$$E(n,3) \approx \frac{3}{3} = 1$$

Cuestión 3-3

Dada la siguiente función, la cual busca un valor en un vector:

```
int buscar(int x[], int n, int valor){
   int i, pos=-1;
   for (i=0; i<n; i++)
      if (x[i]==valor)
         pos=i;
   return pos;
}
```

$$pos = \max_{0 \le i \le n-1} \{ i \; / \; x[i]=valor \}$$



Se pide paralelizarla mediante OpenMP. En caso de varias ocurrencias del valor en el vector, la implementación paralela debe devolver lo mismo que la implementación secuencial.

Solución (1):

```
int buscar(int x[], int n, int valor){
   int i, pos=-1;
   #pragma omp parallel for
   for (i=0; i<n; i++)
      if (x[i]==valor)
         if (i>pos)
            #pragma omp critical
              if (i>pos)
              pos=i;
   return pos;
}
```

Solución (2):

```
int buscar(int x[], int n, int valor){
   int i, pos=-1;
   #pragma omp parallel for reduction(max:pos)
   for (i=0; i<n; i++)
      if (x[i]==valor)
         pos=i;
   return pos;
}
```

Cuestión 3-7

Dado el siguiente fragmento de código, donde el vector de índices ind contiene valores enteros entre 0 y $m-1$ (siendo m la dimensión de x), posiblemente con repeticiones:

```
for (i=0; i<n; i++) {
   s = 0;
   for (j=0; j<i; j++) {
      s += A[i][j]*b[j];
   }
   c[i] = s;
   x[ind[i]] += s;
}
```

a) Realiza una implementación paralela mediante OpenMP, en la que se reparten las iteraciones del bucle externo.

```
#pragma omp parallel for private(s, j)
for (i=0; i<n; i++) {
   s = 0;
   for (j=0; j<i; j++) {
      s += A[i][j]*b[j];
   }
   c[i] = s;
   #pragma omp atomic
   x[ind[i]] += s;
}
```

Cuestión 3-7

b) Realiza una implementación paralela mediante OpenMP, en la que se reparten las iteraciones del bucle interno.

```
for (i=0; i<n; i++) {
    s = 0;
    #pragma omp parallel for reduction(+:s)
    for (j=0; j<i; j++) {
        s += A[i][j]*b[j];
    }
    c[i] = s;
    x[ind[i]] += s;
}
```

c) Para la implementación del apartado a), indica si cabe esperar que haya diferencias de prestaciones dependiendo de la planificación empleada. Si es así, ¿qué planificaciones serían mejores y por qué?

Solución: El bucle j hace i iteraciones, con lo que las iteraciones del bucle i serán más costosas cuanto mayor sea el valor de i. Por tanto, la planificación por defecto (estatic,0) no es la mejor. Sería mejor una planificación cíclica, dinámica o *guided*, con tamaño de chunk=1, por ejemplo.

Cuestión 1-4

Dada la siguiente función:

```
double funcion(double A[M][N]) {
int i,j;
double suma;
for (i=0; i<M-1; i++) {
      for (j=0; j<N; j++)
            A[i][j] = 2.0 * A[i+1][j];
}
suma = 0.0;
for (i=0; i<M; i++) {
      for (j=0; j<N; j++)
            suma = suma + A[i][j];
}
return suma;
}
```

Depedencias entre iteraciones en el bucle i

a)Indica su coste teórico en flops

b) Paralelízalo usando OpenMP. ¿Por qué lo haces así? Se valorarán más aquellas soluciones que sean más eficientes

```
double funcion(double A[M][N]){
int i,j;
double suma;
#pragma omp parallel for private(i)
for (j=0; j<N; j++)
      for (i=0; i<M-1; i++)
            A[i][j] = 2.0 * A[i+1][j];
suma = 0.0;
#pragma omp parallel for reduction(+:suma) private(j)
for (i=0; i<M; i++)
      for (j=0; j<N; j++)
      suma = suma + A[i][j];
return suma;
}
```

c) Indica el speedup que podrá obtenerse con *p* procesadores suponiendo *M* y *N* múltiplos exactos de *p*

Los costes de los dos bucles es el mismo, por lo tanto:

$$t=2\sum_{i=0}^{M-2} \sum_{j=0}^{N-1} 1 = 2\sum_{i=0}^{M-2} N = 2(M-1)N \cong 2MN \text{ flops}$$

$$t_p = \frac{2MN}{p} \longrightarrow S_p = \frac{t_1}{t_p} = \frac{2MN}{2M\,N/p} = p$$

Cuestión 1-4

d) Indica una cota superior del speedup (cuando *p* tiende a infinito) si no se paralelizara la parte que calcula la suma; es decir, sólo se paralelizara la primera parte y no la segunda.

$$t_p = MN + \frac{MN}{p} \text{ flops} \longrightarrow S_p = \frac{2MN}{MN + M\,N/p} = \frac{2}{1 + 1/p} = \frac{2p}{p+1}$$

Haciendo tender p a $+\infty$, resulta que S_p tiende a 2.

Cuestión 2-12

Teniendo en cuenta la definición de las siguientes funciones:

```
/* producto matricial C = A*B */
void matmult(double A[N][N], double B[N][N],double C[N][N]){
   int i,j,k;
   double suma;
   for (i=0; i<N; i++) {
      for (j=0; j<N; j++) {
         suma = 0.0;
         for (k=0; k<N; k++) {
            suma = suma + A[i][k]*B[k][j];
         }
         C[i][j] = suma;
      }
   }
}
```

```
/* simetriza una matriz como A+A' */
   void simetriza(double A[N][N]){
   int i,j;
   double suma;
   for (i=0; i<N; i++) {
      for (j=0; j<=i; j++) {
         suma = A[i][j]+A[j][i];
         A[i][j] = suma;
         A[j][i] = suma;
      }
   }
}
```

se pretende paralelizar el siguiente código:

```
matmult(X, Y, C1); /* T1 */
matmult(Y, Z, C2); /* T2 */
matmult(Z, X, C3); /* T3 */
simetriza(C1); /* T4 */
simetriza(C2); /* T5 */
matmult(C1, C2, D1); /* T6 */
matmult(D1, C3, D); /* T7 */
```

Teniendo en cuenta la declaración de las dos funciones, resulta que **matmult** modifica el tercer argumento y **simetriza** el único argumento que tiene

Cuestión 2-12

a) Realiza una paralelización basada en los bucles.

```
/* producto matricial C = A*B */
void matmult(double A[N][N], double B[N][N],double C[N][N]){
   int i,j,k;
   double suma;
   #pragma omp parallel for private(j, k, suma)
   for (i=0; i<N; i++) {
      for (j=0; j<N; j++) {
         suma = 0.0;
         for (k=0; k<N; k++) {
            suma = suma + A[i][k]*B[k][j];
         }
         C[i][j] = suma;
      }
   }
}
```

```
/* simetriza una matriz como A+A' */
   void simetriza(double A[N][N]){
   int i,j;
   double suma;
   #pragma omp parallel for private(j, suma)
   for (i=0; i<N; i++) {
      for (j=0; j<=i; j++) {
         suma = A[i][j]+A[j][i];
         A[i][j] = suma;
         A[j][i] = suma;
      }
   }
}
```

Cuestión 2-12

b) Calcula el coste en flops de cada una de las tareas y dibuja el grafo de dependencias de tareas, considerando en este caso que las tareas son cada una de las llamadas a matmult y simetriza y

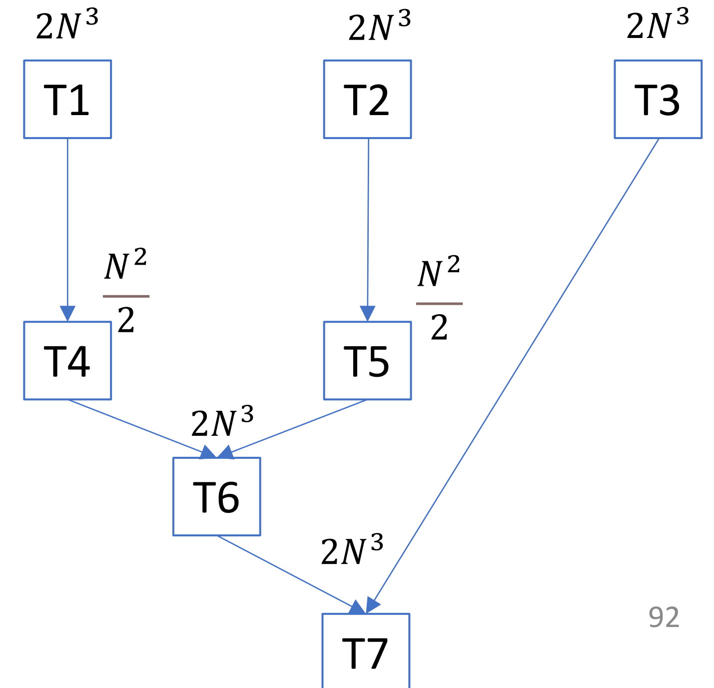
```
matmult(X, Y, C1); /* T1 */
matmult(Y, Z, C2); /* T2 */
matmult(Z, X, C3); /* T3 */
simetriza(C1); /* T4 */
simetriza(C2); /* T5 */
matmult(C1, C2, D1); /* T6 */
matmult(D1, C3, D); /* T7 */
```

```
/* producto matricial C = A*B */
void matmult(double A[N][N], double B[N][N],double C[N][N]){
    int i,j,k;
    double suma;
    for (i=0; i<N; i++) {
        for (j=0; j<N; j++) {
            suma = 0.0;
            for (k=0; k<N; k++) {
                suma = suma + A[i][k]*B[k][j];
            }
            C[i][j] = suma;
        }
    }
}
```

$$t_s(n) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} \sum_{k=0}^{N-1} 2 = 2N^3 \text{ flops}$$

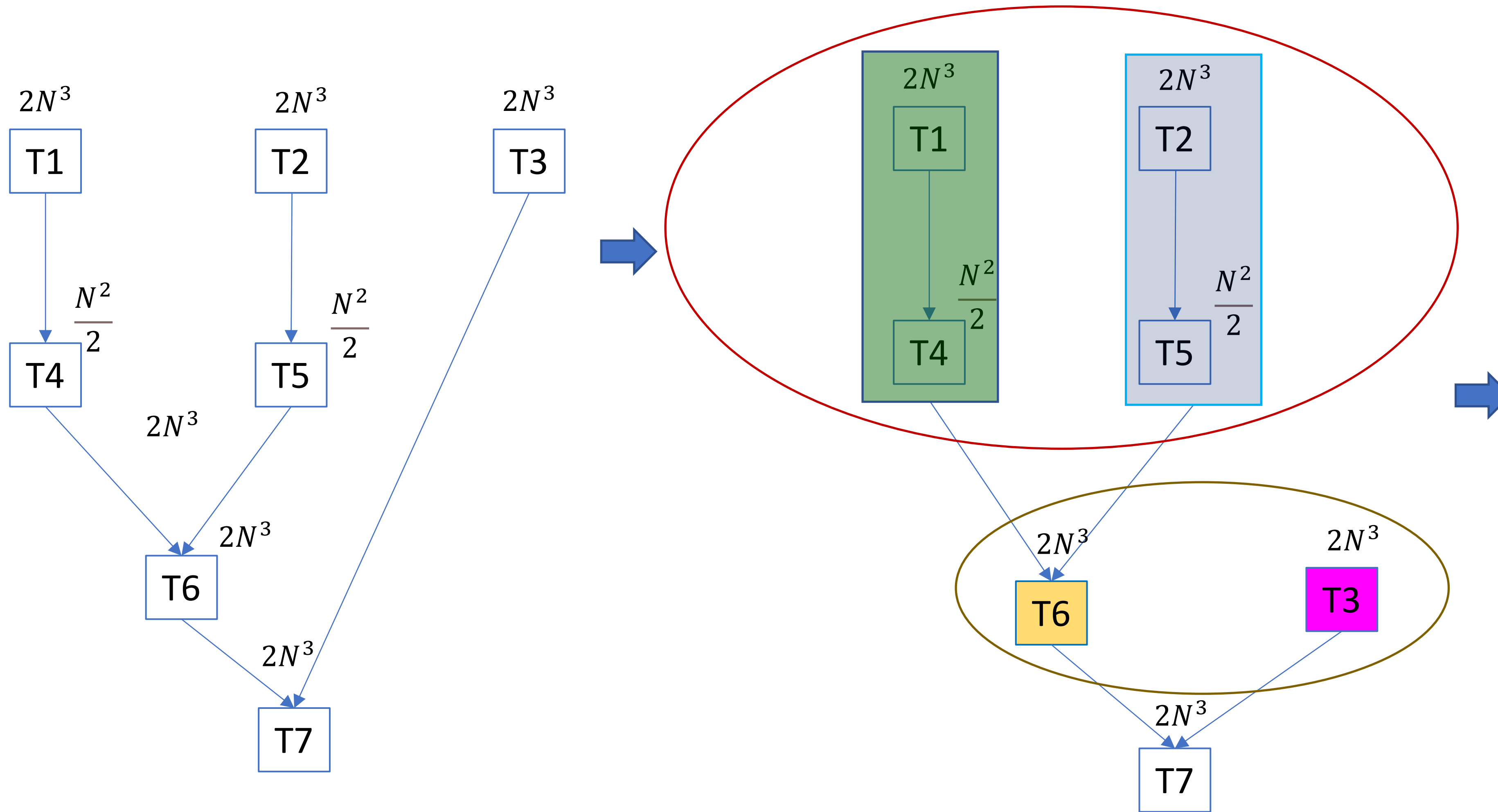
```
/* simetriza una matriz como A+A' */
void simetriza(double A[N][N]){
    int i,j;
    double suma;
    for (i=0; i<N; i++) {
        for (j=0; j<=i; j++) {
            suma = A[i][j]+A[j][i];
            A[i][j] = suma;
            A[j][i] = suma;
        }
    }
}
```

$$t_s(n) = \sum_{i=0}^{N-1} \sum_{j=0}^{i} 1 = \sum_{i=0}^{N-1} (i+1) = \sum_{i=0}^{N-1} i + \sum_{i=0}^{N-1} 1 \approx \frac{N^2}{2} + N \approx \frac{N^2}{2} \text{ flops}$$



Cuestión 2-12

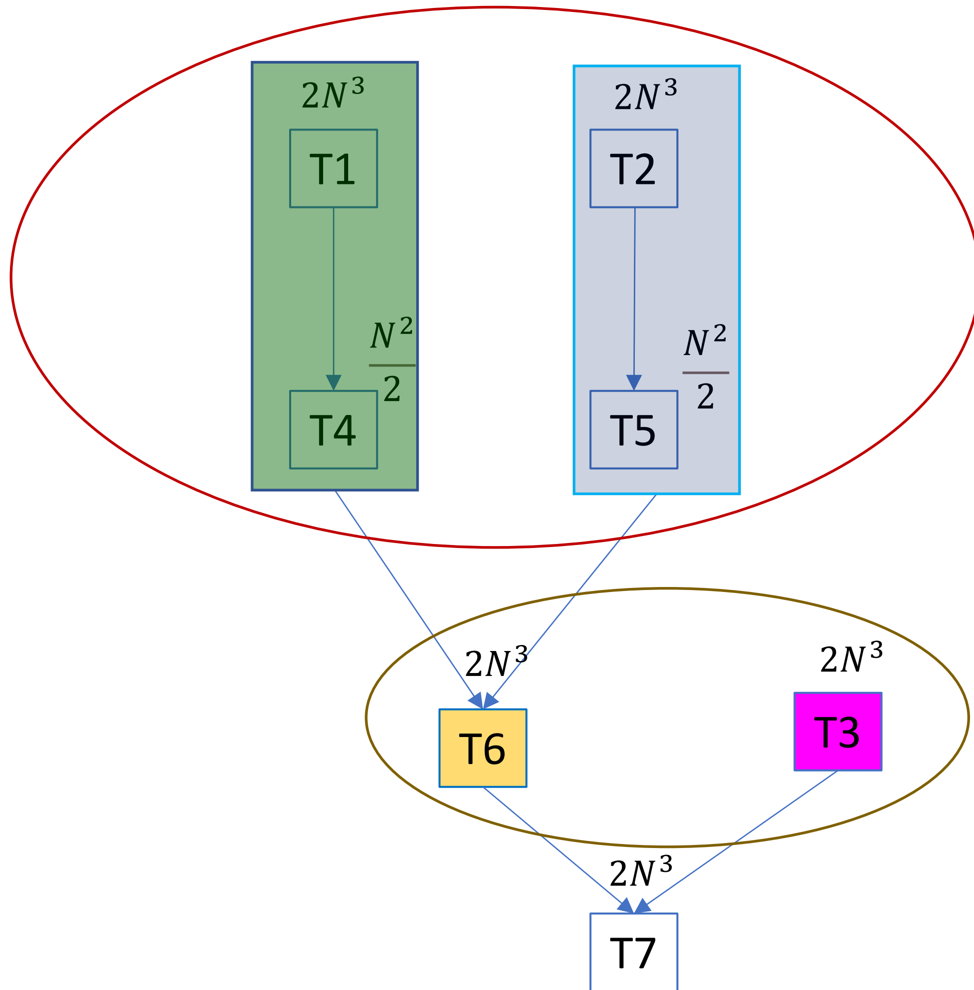
c) Realiza la paralelización basada en secciones, a partir del grafo de dependencias anterior




```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
            T1;
            T4;
        }
        #pragma omp section
        {
            T2;
            T5;
        }
    }
    #pragma omp sections
    {
        #pragma omp section
            T6;
        #pragma omp section
            T3;
    }
}
matmult(D1,C3,D); /*T7*/
```

Cuestión 2-12

d) Suponiendo que se dispone de 2 hilos, calcula el tiempo secuencial, el tiempo paralelo, el speed-up y la eficiencia de la implementación paralela desarrollada



$$t(\mathrm{N}) = 5 \cdot 2N^3 + 2\frac{N^2}{2} = 10N^3 + N^2 \text{ flops}$$

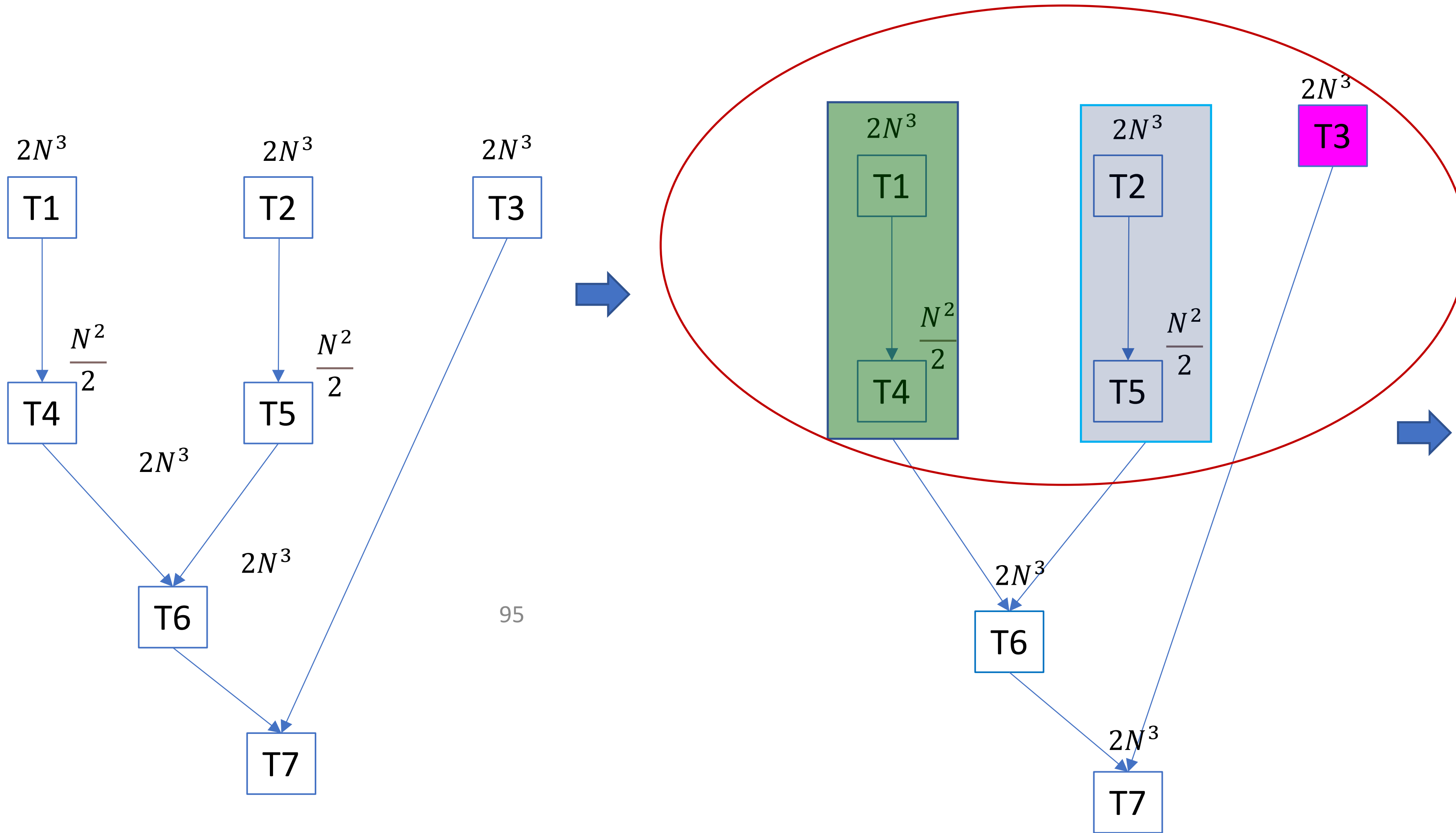
$$t(N,2) = 2N^3 + \frac{N^2}{2} + 2N^3 + 2N^3 = 6N^3 + \frac{N^2}{2} \text{ flops}$$

$$S(N,2) = \frac{10N^3 + N^2}{6N^3 + \frac{N^2}{2}} \approx \frac{10}{6} = 1.\overset{\frown}{6}$$

$$E(N,2) \approx \frac{1.\overset{\frown}{6}}{2} \approx 0.8\overset{\frown}{3}$$

Cuestión 2-12

d) Otra posibilidad. Los resultados de speed-up y eficiencia serían los mismos




```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
            T1;
            T4;
        }
        #pragma omp section
        {
            T2;
            T5;
        }
        #pragma omp section
            T3;
    }
}
T6;
T7;
```

Cuestión 3-8

La siguiente función normaliza los valores de un vector de números reales positivos de forma que los valores finales queden entre 0 y 1, utilizando el máximo y el mínimo

```
void normalize(double *a, int n){
   double mx, mn, factor;
   int i;
   mx = a[0], mn = a[0];
   for (i=1; i<n; i++) {
      if (mx<a[i]) mx=a[i];
   }
   for (i=1; i<n;i++)
      if (mn>a[i]) mn=a[i];
   factor = mx-mn;
   for (i=0;i<n;i++)
      a[i]=(a[i]-mn)/factor;
}
```

a) Paraleliza el programa con OpenMP de la manera más eficiente posible, mediante una única región paralela. Supóngase que el valor de n es muy grande y que se quiere que la paralelización funcione eficientemente para un número arbitrario de hilos
b) Incluye el código necesario para que se imprima una sola vez el número de hilos utilizados

a)

```
void normalize(double *a, int n){
   double mx, mn, factor;
   int i;
   mx = a[0], mn = a[0];
   #pragma omp parallel
   { (*)
      #pragma omp for reduction(max: mx) nowait
      for (i=1; i<n; i++)
         if (mx<a[i]) mx=a[i];
      #pragma omp for reduction(min:mn)
      for (i=1; i<n; i++)
         if (mn>a[i]) mn=a[i];
      factor = mx-mn;
      #pragma omp for
      for (i=0;i<n;i++) {
         a[i]=(a[i]-mn)/factor;
   }
}
```

b) Añadiríamos en (*) las líneas de código:

```
#pragma omp single
   printf("Num procs: %d\n", omp_get_num_threads());
```

Cuestión 1-5

Dada la siguiente función:

```
double fun_mat(double a[n][n], double b[n][n]){
int i,j,k;
double aux,s=0.0;
for (i=0; i<n; i++) {
      for (j=0; j<n; j++) {
            aux=0.0;
            s += a[i][j];
            for (k=0; k<n; k++)
                  aux += a[i][k] * a[k][j];
            b[i][j] = aux;
      }
}
return s;
}
```

(a) Indica cómo se paralelizaría mediante OpenMP cada uno de los tres bucles. ¿Cuál de las tres formas de paralelizar será la más eficiente y por qué?

Bucle i: `#pragma omp parallel for reduction(+:s)  private(j, k, aux)`

Bucle j: `#pragma omp parallel for reduction(+:s)  private(k, aux)`

Bucle k: `#pragma omp parallel for reduction(+:aux)`

La forma más eficiente consiste en paralelizar el bucle más externo, pues únicamente se crea una región paralela. En el caso de los bucles **j** y **k** se crean respectivamente n y n^2 regiones paralelas. Recordad que la creación de una región paralela supone la activación, sincronización y desactivación de hilos

Cuestión 1-5

b) Suponiendo que se paraleliza el bucle más externo, indica los costes a priori secuencial y paralelo, en flops, y el speedup suponiendo que el número de hilos coincide con n.

$$t(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \left(1 + \sum_{k=0}^{n-1} 2\right) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} (1 + 2n) \approx \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 2n = 2n^3 \text{ flops}$$

$$t(n,p) == \sum_{i=0}^{n/\mathrm{p}-1} \sum_{j=0}^{n-1} \left(1 + \sum_{k=0}^{n-1} 2\right) = \sum_{i=0}^{n/\mathrm{p}-1} \sum_{j=0}^{n-1} (1 + 2n) \approx \sum_{i=0}^{n/p-1} \sum_{j=0}^{n-1} 2n = \frac{2n^3}{p} \text{ flops}$$

Si $p = n$ → $t(n, n) = \frac{2n^3}{n} = 2n^2$ flops → $S(n, n) = \frac{n^2}{n} = n$

Cuestión 1-5

c) Añade las líneas de código necesarias para que se muestre en pantalla el número de iteraciones que ha realizado el hilo 0, suponiendo que se paraleliza el bucle más externo.

```
double fun_mat(double a[n][n], double b[n][n]){
int i,j,k;
double aux,s=0.0;
#pragma omp parallel for reduction(+:s)  private(j, k, aux)
for (i=0; i<n; i++) {
      for (j=0; j<n; j++) {
            aux=0.0;
            s += a[i][j];
            for (k=0; k<n; k++)
                  aux += a[i][k] * a[k][j];
            b[i][j] = aux;
      }
}
return s;
}
```

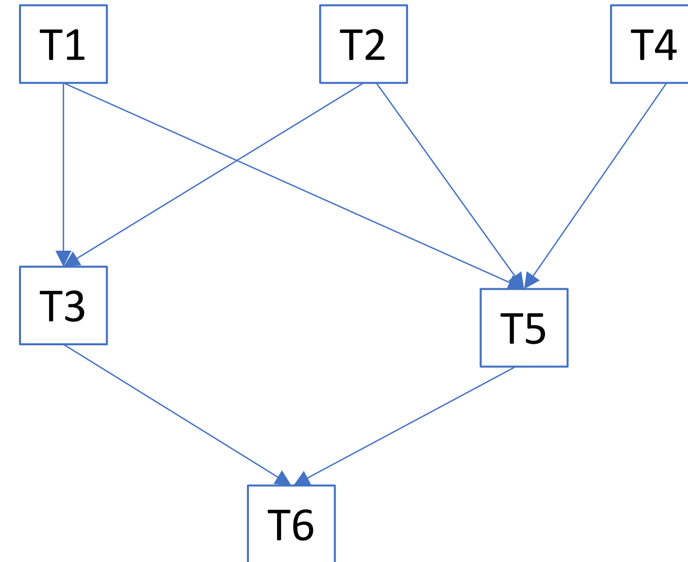
```
double fun_mat(double a[n][n], double b[n][n]){
int i,j,k;
double aux, s=0.0, iter=0, tid;
#pragma omp parallel for reduction(+:s)  private(j, k, aux, tid)
for (i=0; i<n; i++) {
       tid = omp_get_thread_num();
       if (tid==0) iter++;
      for (j=0; j<n; j++) {
            aux=0.0;
            s += a[i][j];
            for (k=0; k<n; k++)
                  aux += a[i][k] * a[k][j];
            b[i][j] = aux;
      }
}
printf("Iteraciones realizadas por el hilo 0 = %d\n", iter);
return s;
}
```

Cuestión 2-9

Dado el siguiente fragmento de código:

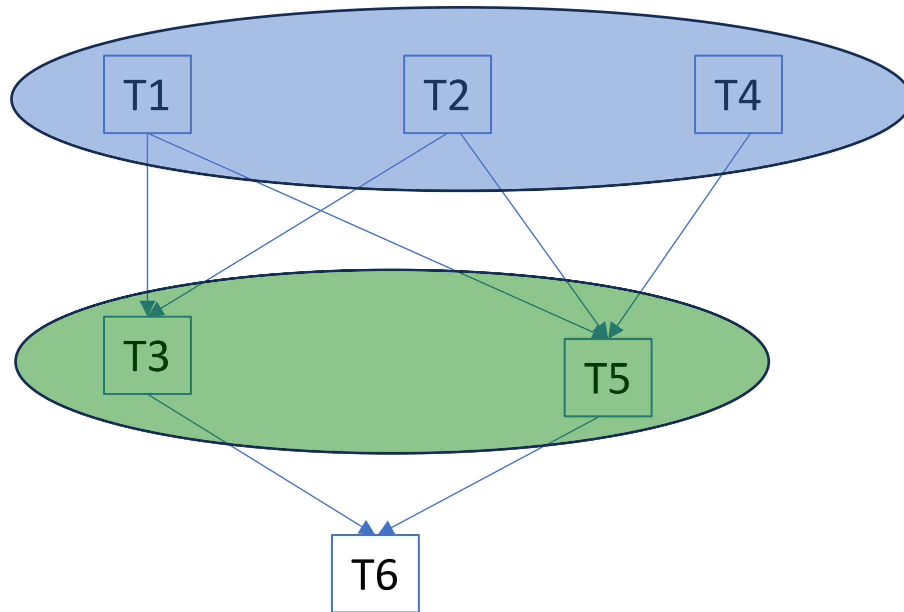
```
minx = minimo(x,n); /* T1 */
maxx = maximo(x,n); /* T2 */
calcula_z(z,minx,maxx,n); /* T3 */
calcula_y(y,x,n); /* T4 */
calcula_x(x,y,n); /* T5 */
calcula_v(v,z,x); /* T6 */
```

(a) Dibuja el grafo de dependencias de las tareas, teniendo en cuenta que las funciones minimo y máximo no modifican sus argumentos, mientras que las demás funciones modifican sólo su primer argumento.



Cuestión 2-9

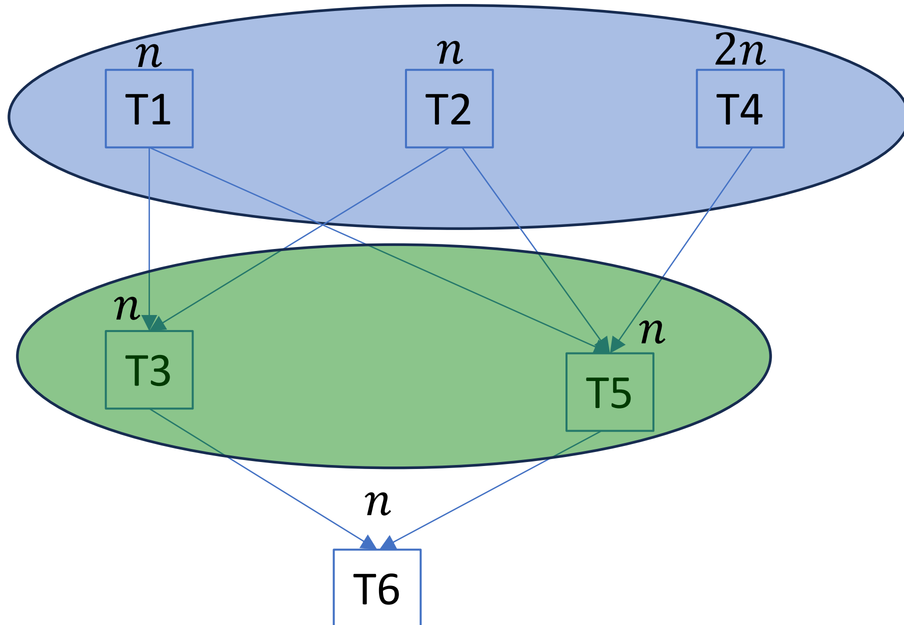
b) Paraleliza el código mediante OpenMP




```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
            minx = minimo(x,n); /* T1 */
        #pragma omp section
            maxx = maximo(x,n); /* T2 */
        #pragma omp section
            calcula_y(y,x,n); /* T4 */
    }
    #pragma omp sections
    {
        #pragma omp section
            calcula_z(z,minx,maxx,n); /* T3 */
        #pragma omp section
            calcula_x(x,y,n); /* T5 */
    }
}
calcula_v(v,z,x); /* T6 */
```

Cuestión 2-9

c) Si el coste de las tareas es de *n* flops, excepto el de la tarea 4 que es de 2*n* flops, indica la longitud del camino crítico, el grado máximo de concurrencia, y el grado medio de concurrencia. Obtén el speedup y la eficiencia de la implementación del apartado anterior, si se ejecutara con 5 procesadores.



Camino crítico: T4 $\rightarrow$T5 $\rightarrow$ T6 $\Rightarrow$ $L = 2n + n + n = 4n$ flops

Grado máximo de concurrencia: 3

Grado medio de concurrencia: $\frac{7n}{4n} = 1.75$

Speedup: $S_p = \frac{7n}{4n} = 1.75$

Eficiencia: $E_p = \frac{1.75}{5} = 0.35$

Cuestión 3-9

Dada la siguiente función:

```
int funcion(int n, double v[]){
    int i, pos_max=-1;
    double suma, norma, aux, max=-1;
    suma = 0;
    for (i=0;i<n;i++)
        suma = suma + v[i]*v[i];
    norma = sqrt(suma);
    for (i=0;i<n;i++)
        v[i] = v[i] / norma;
    for (i=0;i<n;i++) {
        aux = v[i];
        if (aux < 0) aux = -aux;
        if (aux > max) {
            pos_max = i; max = aux;
        }
    }
    return pos_max;
}
```

a) Paralelízala con OpenMP, usando una única región paralela
b) ¿Qué añadirías para garantizar que en todos los bucles las iteraciones se reparten de 2 en 2 entre los hilos?

a)

```
int funcion(int n, double v[]){
  int i, pos_max=-1;
  double suma, norma, aux, max=-1;
  suma=0;
  #pragma omp parallel
  {
    #pragma omp for reduction(+:suma)
      for (i=0;i<n;i++)
        suma = suma + v[i]*v[i];
    norma = sqrt(suma);
    #pragma omp for
      for (i=0; i<n; i++)
        v[i] = v[i] / norma;
    #pragma omp for private(aux)
      for (i=0; i<n; i++) {
        aux = v[i];
      if (aux < 0) aux = -aux;
      if (aux > max){
        #pragma omp critical
          if (aux > max)
            {pos_max = i; max = aux;}
      }
  }
  return pos_max;
}
```

b) Añadir schedule(static,2) o schedule(dynamic,2) en las directivas **omp for**

Cuestión 1-7

a)Paraleliza de la forma más eficiente el siguiente código:

```
#define EPS 1e-9
#define N 128
int fun(double a[N][N], double b[], double x[], int n, int nMax){
int i, j, k;
double err=100, aux[N];
for (i=0;i<n;i++)
     aux[i]=0.0;
for (k=0;k<nMax && err>EPS;k++) {
     err=0.0;
     for (i=0;i<n;i++) {
          x[i]=b[i];
          for (j=0;j<i;j++)
               x[i]-=a[i][j]*aux[j];
          for (j=i+1;j<n;j++)
               x[i]-=a[i][j]*aux[j];
          x[i]/=a[i][i];
          err+=fabs(x[i]-aux[i]);
     }
     for (i=0;i<n;i++)
          aux[i]=x[i];
}
return k<nMax;
}
```

Solución a): El bucle k no se puede paralelizar, pues al contener la condición err>EPS, a priori no se pueden establecer el nº de iteraciones a repartir.

El bucle i anidado en el bucle k si se puede paralelizar:

```
#pragma omp parallel for private(j) reduction(+:err)
```

b) Calcula el coste computacional secuencial, el coste paralelo y el speed-up de una iteración del bucle k

$$t(n) = \sum_{i=0}^{n-1}\left(\sum_{j=0}^{i-1} 2 + \sum_{j=i+1}^{n-1} 2 + 3\right) = \sum_{i=0}^{n-1}(2n+1) \approx 2n^2 \text{ flops}$$

$$t(n,p) = \sum_{i=0}^{n/p-1}\left(\sum_{j=0}^{i-1} 2 + \sum_{j=i+1}^{n-1} 2 + 3\right) = \sum_{i=0}^{n/p-1}(2n+1) \approx \frac{2n^2}{p} \text{ flops}$$

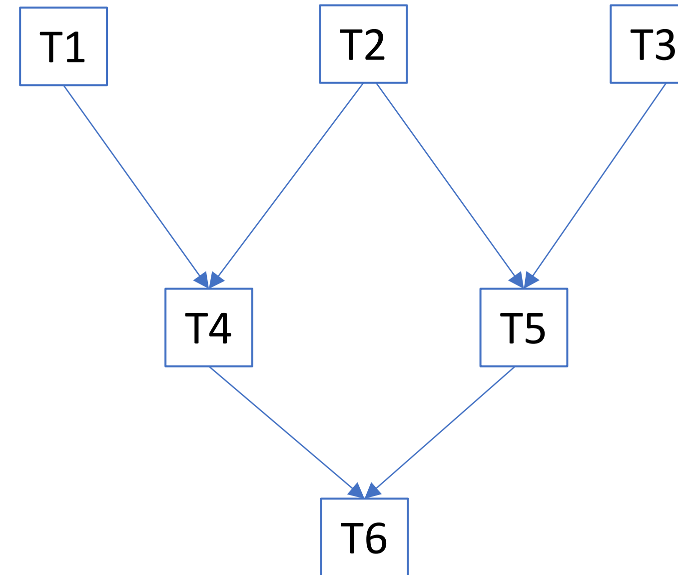
$$S(n,p) = \frac{2n^2}{\frac{2n^2}{p}} = p$$

Cuestión 2-8

En la siguiente función, T1, T2, T3 modifican x, y, z, respectivamente.

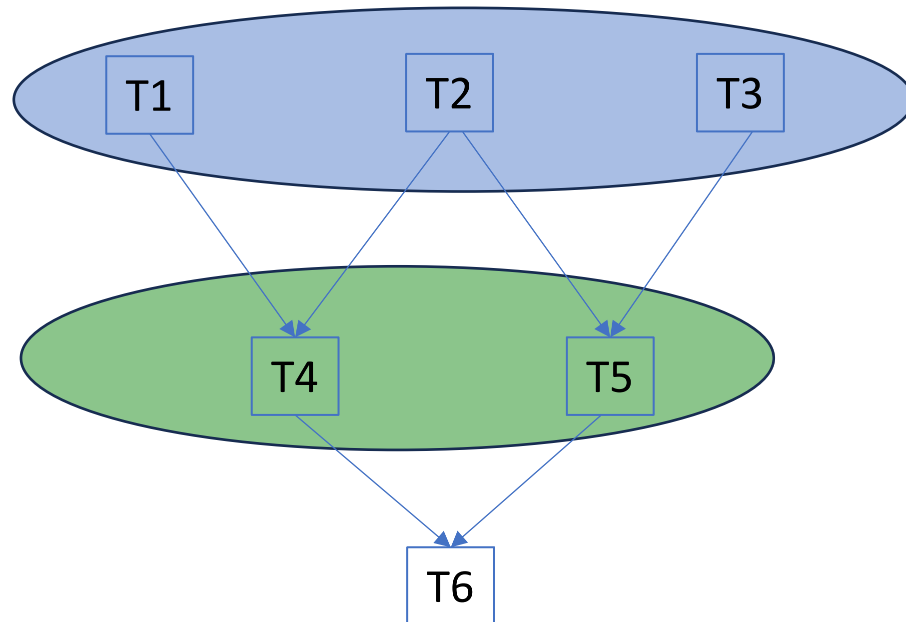
```
double f(double x[], double y[], double z[], int n) {
int i, j;
double s1, s2, a, res;
T1(x,n); /* Tarea T1 */
T2(y,n); /* Tarea T2 */
T3(z,n); /* Tarea T3 */
for (i=0; i<n; i++) {/* Tarea T4 */
      s1=0;
      for (j=0; j<n; j++) s1+=x[i]*y[i];
      for (j=0; j<n; j++) x[i]*=s1;
}
for (i=0; i<n; i++) {/* Tarea T5 */
      s2=0;
      for (j=0; j<n; j++) s2+=y[i]*z[i];
      for (j=0; j<n; j++) z[i]*=s2;
}
/* Tarea T6 */
a=s1/s2;
res=0;
for (i=0; i<n; i++) res+=a*z[i];
return res;
}
```

a) Dibuja el grafo de dependencia de las tareas.



Cuestión 2-8

Realiza una paralelización mediante OpenMP a nivel de tareas (no de bucles), basándote en el grafo de dependencias.




```
void f(double x[], double y[], double z[], int n){
int i, j;
double s1, s2, a, res;
#pragma omp parallel private(i, j)
{/*las variables i y j se deben de privatizar*/
    #pragma omp sections
    {
        #pragma omp section
            T1(x,n); /* Tarea T1 */
        #pragma omp section
            T2(y,n); /* Tarea T2 */
        #pragma omp section
            T3(z,n); /* Tarea T3 */
    }
    #pragma omp sections
    {
        #pragma omp section
            /* Tarea T4 */
        #pragma omp section
            /* Tarea T5 */
    }
}
/* Tarea T6 */
return res;
}
```

Cuestión 2-8

c) Indica el coste a priori del algoritmo secuencial, el del algoritmo paralelo y el speedup resultante. Supón que el coste de las tareas 1, 2 y 3 es de $2n^2$ flops cada una.

```
for (i=0; i<n; i++) {/* Tarea T4 */
      s1=0;
      for (j=0; j<n; j++) s1+=x[i]*y[i];
      for (j=0; j<n; j++) x[i]*=s1;
}
for (i=0; i<n; i++) {/* Tarea T5 */
      s2=0;
      for (j=0; j<n; j++) s2+=y[i]*z[i];
      for (j=0; j<n; j++) z[i]*=s2;
}
/* Tarea T6 */
a=s1/s2;
res=0;
for (i=0; i<n; i++) res+=a*z[i];
return res;
}
```

Costes de T4 y T5: $\sum_{i=0}^{n-1}\left(\sum_{j=0}^{n-1}2+\sum_{j=0}^{n-1}1\right)=\sum_{i=0}^{n-1}(2n+n)=3n^2$ flops

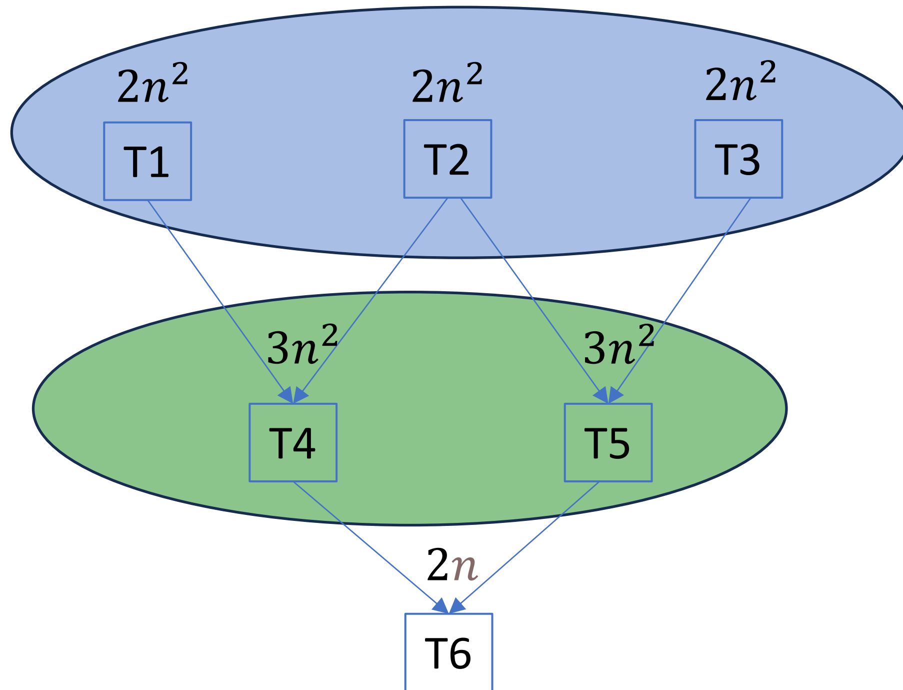
Coste de T6: $2n+1\approx 2n$ flops

c)

Costes de T4 y T5: $3n^2$ flops

Coste de T6 $\approx 2n$ flops

$t(n) \approx 12n^2$ flops



$$t(n) \approx 2n^2 + 2n^2 + 2n^2 + 3n^2 + 3n^2 + 2n \approx 12n^2 \text{ flops}$$

Si $p \geq 3$, entonces

$$t(n,p) \approx 2n^2 + 3n^2 + 2n \approx 5n^2 \text{ flops}$$

$$S(n,p) \approx \frac{12n^2}{5n^2} = 2.4$$

Cuestión 3-10

La siguiente función procesa una serie de transferencias bancarias. Cada transferencia tiene una cuenta origen, una cuenta destino y una cantidad de dinero que se mueve de la cuenta origen a la cuenta destino. La función actualiza la cantidad de dinero de cada cuenta (array saldos) y además devuelve la cantidad máxima que se transfiere en una sola operación:

```
double transferencias(double saldos[], int origenes[],int destinos[],
double cantidades[], int n) {
    int i, i1, i2;
    double dinero, maxtransf=0;
    for (i=0; i<n; i++) {
    /* Procesar transferencia i: La cantidad transferida es
     * cantidades[i], que se mueve de la cuenta origenes[i]
     * a la cuenta destinos[i]. Se actualizan los saldos de
     * ambas cuentas y la cantidad maxima */
        i1 = origenes[i];
        i2 = destinos[i];
        dinero = cantidades[i];
        saldos[i1] -= dinero;
        saldos[i2] += dinero;
        if (dinero>maxtransf) maxtransf = dinero;
    }
    return maxtransf;
}
```

a) Paraleliza la función de forma eficiente con OpenMp

```
double transferencias(double saldos[], int origenes[],
    int destinos[], double cantidades[], int n) {
    int i, i1, i2;
    double dinero, maxtransf=0;
    #pragma omp parallel for private(i1,i2,dinero)
                            reduction(max:maxtransf)
    for (i=0; i<n; i++) {
        i1 = origenes[i];
        i2 = destinos[i];
        dinero = cantidades[i];
        #pragma omp atomic
            saldos[i1] -= dinero;
        #pragma omp atomic
            saldos[i2] += dinero;
        if (dinero>maxtransf) maxtransf = dinero;
    }
    return maxtransf;
}
```