Cuestión 1 (1.2 puntos)
Dada la siguiente función

```
void f(double A[N][N], double B[N][N], double C[N][N], double x[N], double z[N]) {
  int i,j;
  double sumz;

  for (i=0;i<N;i++) {
    sumz = 0;
    for (j=i;j<N;j++)
       C[i][j] = A[i][j] * x[j];
    for (j=0;j<N;j++)
       sumz += B[i][j] * x[j];
    z[i] = sumz;
  }
}
```

0.2 p. (a) Haz una versión paralela basada en la paralelización del bucle externo.

Solución: Justo delante del bucle `i`, incluiremos:

```
#pragma omp parallel for private (j, sumz)
```

0.2 p. (b) Modifica el apartado anterior para que se muestre una única vez el número de hilos que intervienen en la ejecución del bucle paralelo.

Solución:

```
...
double sumz;

#pragma omp parallel
{
  #pragma omp master
  printf("Numero de hilos: %d\n", omp_get_num_threads());
}

#pragma omp parallel for private (j, sumz)
for (i=0;i<N;i++) {
...

// Otra posible solución:

...
```

```
double sumz;

#pragma omp parallel private (j, sumz, id)
{
  #pragma omp master
  printf("Numero de hilos: %d\n", omp_get_num_threads());

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

0.5 p.

(c) Haz una versión paralela basada en la paralelización de los dos bucles internos, usando una sola región paralela. Considera la conveniencia de usar la cláusula `nowait` y, tanto si la usas como si no, justifica por qué.

Solución:

```
for (i=0;i<N;i++) {
  sumz = 0;
  #pragma omp parallel
  {
    #pragma omp for nowait
    for (j=i;j<N;j++)
       C[i][j] = A[i][j] * x[j];
    #pragma omp for reduction(+:sumz)
    for (j=0;j<N;j++)
       sumz += B[i][j] * x[j];
  }
  z[i] = sumz;
}
```

Se usa la cláusula `nowait` en el primer bucle, porque dicho bucle es independiente del segundo.

0.3 p.

(d) Calcula el coste secuencial y el coste paralelo (incluye en ambos casos el desarrollo completo) suponiendo que se paralelizara únicamente el segundo de los bucles internos. Calcula también en el mismo supuesto el Speed-up y la Eficiencia.

Solución:

$$t(N) = \sum_{i=0}^{N-1}(\sum_{j=i}^{N-1} 1 + \sum_{j=0}^{N-1} 2) \approx \sum_{i=0}^{N-1}(N - i + 2N) \approx 3N^2 - \frac{N^2}{2} = \frac{5N^2}{2} flops$$

$$t(N,p) = \sum_{i=0}^{N-1}(\sum_{j=i}^{N-1} 1 + \sum_{j=0}^{\frac{N}{p}-1} 2) \approx \sum_{i=0}^{N-1}(N - i + \frac{2N}{p}) \approx N^2 - \frac{N^2}{2} + \frac{2N^2}{p} = \frac{N^2}{2} + \frac{2N^2}{p} = \frac{p+4}{2p}N^2 flops$$

$$S(N,p) = \frac{\frac{5}{2}N^2}{\frac{p+4}{2p}N^2} = \frac{5p}{p+4}$$

$$E(N,p) = \frac{\frac{5p}{p+4}}{p} = \frac{5}{p+4}$$

Cuestión 2 (1.1 puntos)

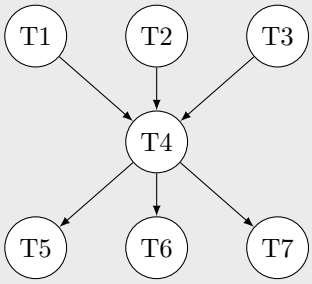
Dada la siguiente función, donde `Image` es un tipo de datos predefinido:

```
void transform(int n, Image im1, Image im2, float weights[3], float factor)
{
  weights[0] = channel_r(im1,im2,n);      /* Tarea T1, coste 5*n^2 flops */
  weights[1] = channel_g(im1,im2,n);      /* Tarea T2, coste 5*n^2 flops */
  weights[2] = channel_b(im1,im2,n);      /* Tarea T3, coste 5*n^2 flops */
  factor *= combine(weights);             /* Tarea T4; coste 12 flops */
  weights[0] += adjust_r(im2,n,factor);   /* Tarea T5, coste n^2 flops */
  weights[1] += adjust_g(im2,n,factor);   /* Tarea T6, coste n^2 flops */
  weights[2] += adjust_b(im2,n,factor);   /* Tarea T7, coste n^2 flops */
}
```

Ninguna de las funciones modifica sus argumentos.

0.3 p. (a) Dibuja el grafo de dependencias de datos entre las tareas.

Solución:



0.5 p. (b) Implementa una versión paralela mediante OpenMP utilizando una sola región paralela.

Solución:

```
void transform_par(int n, Image im1, Image im2, float weights[3], float factor)
{
  #pragma omp parallel
  {
    #pragma omp sections
    {
      #pragma omp section
      weights[0] = channel_r(im1,im2,n);
      #pragma omp section
      weights[1] = channel_g(im1,im2,n);
      #pragma omp section
      weights[2] = channel_b(im1,im2,n);
    }
    #pragma omp single
    {
      factor *= combine(weights);
    }
    #pragma omp sections
    {
      #pragma omp section
      weights[0] += adjust_r(im2,n,factor);
      #pragma omp section
```

```
          weights[1] += adjust_g(im2,n,factor);
          #pragma omp section
          weights[2] += adjust_b(im2,n,factor);
        }
      }
    }
```

0.3 p. (c) Obtén el speedup y la eficiencia de la versión paralela del apartado anterior suponiendo que se ejecuta con 4 hilos en un computador con 4 procesadores (núcleos).

Solución: Tiempo de ejecución secuencial:

$$t(n) = 5n^2 + 5n^2 + 5n^2 + 12 + n^2 + n^2 + n^2 \approx 18n^2 \text{ flops}$$

Tiempo de ejecución paralelo para $p = 4$:

$$t(n,p) = 5n^2 + 12 + n^2 \approx 6n^2 \text{ flops}$$

Speedup:

$$S(n,p) = \frac{18n^2}{6n^2} = 3$$

Eficiencia:

$$E(n,p) = \frac{3}{4} = 0{,}75$$

Cuestión 3 (1.2 puntos)

La siguiente función actualiza una matriz A sumándole los n valores del vector *vals* (no tiene ningún cero) en las posiciones dadas por los vectores *rows* y *cols* que pueden tener valores repetidos.

```
void update( int n,int rows[],int cols[],double vals[], double A[M][N] )
{ int i,j,k, row_max,col_max, cp = 0, cn = 0;
  double x, max = -1e6;

  for ( k = 0 ; k < n ; k++ ) {

    i = rows[k]; j = cols[k]; x = vals[k];

    if ( x > 0 ) cp++; else cn++;

    A[i][j] += x;

    if ( x > max ) {
      max = x; row_max = i; col_max = j;
    }

  }

  printf("%d actualizaciones positivas y %d negativas.\n",cp,cn);
  printf("La mayor actualización ha sido de %.1f en la fila %d columna %d.\n",
   max, row_max, col_max );
}
```

0.8 p. (a) Paraleliza la función usando OpenMP.

Solución:

```
void update( int n,int rows[],int cols[],double vals[], double A[M][N] )
{ int i,j,k, row_max,col_max, cp = 0, cn = 0;
  double x, max = -1e6;

  #pragma omp parallel for private(i,j,x) reduction(+:cp,cn)
  for ( k = 0 ; k < n ; k++ ) {

    i = rows[k]; j = cols[k]; x = vals[k];

    if ( x > 0 ) cp++; else cn++;

    #pragma omp atomic
    A[i][j] += x;

    if ( x > max )
    #pragma omp critical
    if ( x > max ) {
      max = x; row_max = i; col_max = j;
    }

  }

  printf("%d actualizaciones positivas y %d negativas.\n",cp,cn);
  printf("La mayor actualización ha sido de %.1f en la fila %d columna %d.\n",
   max, row_max, col_max );
}
```

0.4 p.

(b) Modifica la paralelización del apartado anterior para que se muestre por pantalla el identificador del hilo que ha realizado más actualizaciones sobre la matriz A y cuántas han sido.

Solución:

```
#include <omp.h>
void update( int n,int rows[],int cols[],double vals[], double A[M][N] )
{ int i,j,k, row_max,col_max, cp = 0, cn = 0, id, m = -1, c;
  double x, max = -1e6;

  #pragma omp parallel private(c)
  { c = 0;
    #pragma omp for private(i,j,x) reduction(+:cp,cn) nowait
    for ( k = 0 ; k < n ; k++ ) {

      i = rows[k]; j = cols[k]; x = vals[k];

      c++;
      if ( x > 0 ) cp++; else cn++;

      #pragma omp atomic
      A[i][j] += x;

      if ( x > max )
```

```
        #pragma omp critical
        if ( x > max ) {
          max = x; row_max = i; col_max = j;
        }

      }
      #pragma omp critical
      if ( c > m ) { m = c; id = omp_get_thread_num(); }
    }

    printf("%d actualizaciones positivas y %d negativas.\n",cp,cn);
    printf("La mayor actualización ha sido de %.1f en la fila %d columna %d.\n",
     max, row_max, col_max );

    printf("El hilo %d es el que más actualizaciones ha realizado (%d).\n",id,m);
  }
```