Qüestió 1 (1 punt)

Donat el següent codi:

```
double funcio( int n, double v[], double w[], double *a ) {
  int i, j, c = 0; double b, f = 0, e;
  for (i=0; i<n; i++) {
    e = 0;
    for (j=i+1; j<n; j++) {
      b = sqrt(v[i]*w[j]);
      if ( b > e ) e = b;
      c++;
    }
    v[i] = e;
    if ( e > f ) f = e;
  }
  *a = f;
  return c;
}
```

0.3 p. (a) Realitza una implementació paral·lela del bucle més extern.

Solució: Bastaria amb introduir la següent directiva just abans del primer bucle:

```
#pragma omp parallel for private(e,j,b) reduction(max:f) reduction(+:c)
```

0.3 p. (b) Realitza una implementació paral·lela del bucle més intern.

Solució: Bastaria amb introduir la següent directiva just abans del segon bucle:

```
#pragma omp parallel for private(b) reduction(max:e) reduction(+:c)
```

0.4 p. (c) Calcula el cost computacional (en flops) de la versió original seqüencial, suposant que la funció `sqrt` té un cost de 3 Flops. Tenint en compte el cost d'una sola iteració del bucle `i`, argumenta si hi hauria bon equilibri de càrrega en cas d'utilitzar la planificació `schedule(static)` en l'apartat a. Repeteix l'argumentació en cas d'usar la mateixa planificació en l'apartat b.

Solució:

Cost seqüencial:

$$t(n) = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} 4 \approx \sum_{i=0}^{n-1} (4n - 4i) \approx 4n^2 - 4\frac{n^2}{2} = 2n^2 \text{ flops.}$$

El cost d'una iteració del bucle i és aproximadament $4(n-i)$. Donat que depén de i, el cost de cada iteració es diferent; en concret, les primeres iteracions són les més costoses i les últimes les menys costoses. La planificació "schedule(static)" donarà lloc a desequilibri de càrrega, donat que al fil 0 li tocarà el bloc d'iteracions més costoses, mentres que a l'ultim fil li tocarà el bloc de les menys costoses.

En el cas del bucle j, el cost d'una iteració es 4 flops. Donat que totes les iteracions costen el mateix, la planificació "schedule(static)" no produirà desequilibri de càrrega.

Qüestió 2 (1.4 punts)

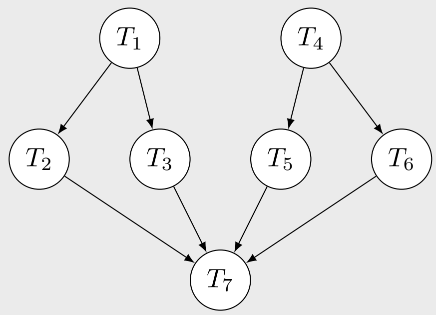
Donat el següent programa, on la funció `generar` modifica l'argument que se li subministra y té un cost computacional de $6N^2$ Flops.

```
float fprocessa(float A[N][N], float factor) {
  int i,j; float resultat=0.0;
  for (i=0;i<N;i++)
    for (j=0;j<N;j++)
      resultat +=A[i][j]/(factor+j);
```

```
    return resultat;
  }
  void processament() {
    float A[N][N], B[N][N], n1, n2, n3, n4;
    generar(A);           // T1
    n1=fprocessa(A,1.1);  // T2
    n2=fprocessa(A,1.2);  // T3
    generar(B);           // T4
    n3=fprocessa(B,1.1);  // T5
    n4=fprocessa(B,1.2);  // T6
    printf("Resultat: %f\n", n1+n2+n3+n4); // T7
  }
```

0.4 p. (a) Determina el graf de dependències de la funció `processament`, obtenint un camí crític i la seua longitud, així com el grau mitja i el grau màxim de concurrència.

Solució:



Un camí crític seria $T_1 \rightarrow T_2 \rightarrow T_7$. La funció `fprocessa` té un cost de $3N^2$ Flops, per tant tenim que la seua longitud es de $9N^2 + 3$. El grau màxim de concurrència és 4 i el grau mitjà de concurrència és $M = \frac{24N^2+3}{9N^2+3} = \frac{24}{9} \approx 2.67$

0.6 p. (b) Realitza una implementació paral·lela eficient, basada en seccions, de la funció `processament`.

Solució:

```
void processament() {
  float A[N][N], B[N][N], n1, n2, n3, n4;

  #pragma omp parallel
   {
    #pragma omp sections
    {
      #pragma omp section
      generar(A);
      #pragma omp section
      generar(B);
    }
    #pragma omp sections
    {
      #pragma omp section
      n1=fprocessa(A,1.1);
      #pragma omp section
      n2=fprocessa(A,1.2);
      #pragma omp section
      n3=fprocessa(B,1.1);
      #pragma omp section
      n4=fprocessa(B,1.2);
    }
  }
  printf("Resultat: %f\n", n1+n2+n3+n4);
}
```

0.4 p. (c) Calcula el cost computacional seqüencial. Calcula el coste paral·lel, el speed-up i l'eficiència en cas d'usar 2 fils y també en cas d'usar 4 fils.

Solució:

$$t(N) = 24N^2 \,\text{Flops}$$

$$t(N,2) = 6N^2 + 3N^2 + 3N^2 + 3 \approx 12N^2 \,\text{Flops}$$

$$Sp(N,2) = \frac{24N^2}{12N^2} = 2$$

$$E(N,2) = \frac{2}{2} = 1$$

$$t(N,4) = 6N^2 + 3N^2 + 3 \approx 9N^2 \,\text{Flops}$$

$$Sp(N,4) = \frac{24N^2}{9N^2} = 2,67$$

$$E(N,4) = \frac{2,66}{4} = 0,67$$

Qüestió 3 (1.1 punts)

El següent programa obté una aproximació del número PI, a partir de la generació de 200000 punts, calculant el número de punts que es trobarien dins o fora d'una circumferència de radi r. A partir d'eixe quocient s'obté una aproximació de l'àrea i per tant del número PI. La funció `aleatori(0,N)` torna un número enter aleatori entre 0 i `N`-1.

```
#define N 20
#define SAMPLES 200000
int main() {
  int i,j, ipos, jpos, imax, jmax, imin, jmin, r, min, max;
  int A[N][N], dins=0, fora=0;
  for (i=0;i<N;i++)
    for (j=0;j<N;j++)
        A[i][j]= 0;
  for (i=0;i<SAMPLES;i++) {
    ipos = aleatori(0,N);
    jpos = aleatori(0,N);
    A[ipos][jpos]++;
  }
  min = A[0][0]; max = A[0][0]; r = N/2;
  for (i=0;i<N;i++) {
    for (j=0;j<N;j++) {
       if ((r-i)*(r-i)+(r-j)*(r-j)<r*r)
         dins+=A[i][j];
       else
         fora+=A[i][j];

       if (min>A[i][j]) {
         min=A[i][j];
         imin = i;
         jmin = j;
       }
       if (max<A[i][j]) {
         max=A[i][j];
         imax = i;
         jmax = j;
       }
    }
  }
  printf("d=%d, f=%d\n", dins, fora);
  printf("Pi = %f\n", (4.0*dins)/(dins+fora));
  printf("Max A[%d][%d] = %d \n", imax,jmax,max);
  printf("Min A[%d][%d] = %d \n", imin,jmin,min);
  return 0;
}
```

0.7 p. (a) Realitza una implementació paral·lela mitjançant OpenMP. No fa falta paral·lelitzar el bucle que inicialitza la matriu A i es pot emprar més d'una regió paral·lela.

Solució:

```
#define N 20
#define SAMPLES 200000
int main() {
  int i,j, ipos, jpos, imax, jmax, imin, jmin, r, min, max;
  int A[N][N];
  int dins=0, fora=0;

  for (i=0;i<N;i++)
    for (j=0;j<N;j++)
        A[i][j]= 0;

  #pragma omp parallel for private (ipos,jpos)
  for (i=0;i<SAMPLES;i++) {
    ipos = aleatorio(0,N);
    jpos = aleatorio(0,N);
    #pragma omp atomic
    A[ipos][jpos]++;
  }
  min = A[0][0];
  max = A[0][0];
  r = N/2;
  #pragma omp parallel for private (j) reduction (+:dins, fora)
  for (i=0;i<N;i++) {
    for (j=0;j<N;j++) {
       if ((r-i)*(r-i)+(r-j)*(r-j)<r*r)
         dins+=A[i][j];
       else
         fora+=A[i][j];

       if (min>A[i][j])
         #pragma omp critical (min)
         if (min>A[i][j]) {
           min=A[i][j];
           imin = i;
           jmin = j;
         }
       if (max<A[i][j])
         #pragma omp critical (max)
         if (max<A[i][j]) {
           max=A[i][j];
           imax = i;
           jmax = j;
         }
    }
  }
  printf("d=%d, f=%d\n", dins, fora);
  printf("Pi = %f\n", (4.0*dins)/(dins+fora));
  printf("Max A[%d][%d] = %d \n", imax,jmax,max);
  printf("Min A[%d][%d] = %d \n", imin,jmin,min);
  return 0;
}
```

0.4 p. (b) Modifica el programa per a que es mostre el valor de les variables `dins` i `fora` que ha calculat cada fil.

Solució:

```
  ...
```

```
r = N/2;
/* Fins la línia anterior, mateix codi que en apartat a */
#pragma omp parallel private (j) reduction (+:dins, fora)
{
   #pragma omp for
   for (i=0;i<N;i++) {
     /* Dins del bucle, mateix codi que en apartat a */
     ...
   }
   printf("Valors de dins i fora per al fil %d: %d y %d\n",
         omp_get_thread_num(), dins, fora);
}
/* A partir d'ací, mateix codi que en programa original */
printf("d=%d, f=%d\n", dins, fora);
...
```