Computación Paralela
Grado en Ingeniería Informática (ETSInf)
Curso 2024-25 ◇ Examen final 27/1/25 ◇ Bloque OpenMP ◇ Duración: 1h 30m

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Cuestión 1 (1 punto)
Dado el siguiente código:

```
double funcion( 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) Realiza una implementación paralela del bucle más externo.

Solución: Bastaría con introducir la siguiente directiva justo antes del primer bucle:

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

0.3 p. (b) Realiza una implementación paralela del bucle más interno.

Solución: Bastaría con introducir la siguiente directiva justo antes del segundo bucle:

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

0.4 p. (c) Calcula el coste computacional (en flops) de la versión original secuencial, suponiendo que la función sqrt tiene un coste de 3 Flops. Teniendo en cuenta el coste de una sola iteración del bucle `i`, argumenta si habría buen equilibrio de carga en caso de utilizar la planificación `schedule(static)` en el apartado a. Repite la argumentación en caso de que se usase la misma planificación en el apartado b.

Solución:
Coste secuencial:

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

El coste de una iteración del bucle i es aproximadamente $4(n-i)$. Puesto que depende de i, el coste de cada iteración es diferente; en concreto, las primeras iteraciones son las más costosas y las últimas las menos costosas. La planificación "schedule(static)" dará lugar a desequilibrio de carga, puesto que al hilo 0 le tocará el bloque de iteraciones más costosas, mientras que al último hilo le tocará el bloque de las menos costosas.
En el caso del bucle j, el coste de una iteración es 4 flops. Puesto que todas las iteraciones cuestan lo mismo, la planificación "schedule(static)" no producirá desequilibrio de carga.

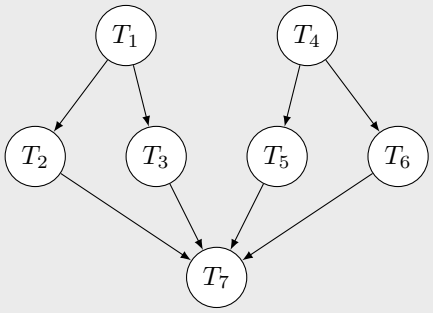
Cuestión 2 (1.4 puntos)

Dado el siguiente programa, en el que la función `generar` modifica el argumento que se le suministra y tiene un coste computacional de $6N^2$ Flops.

```
float fprocesa(float A[N][N], float factor) {
  int i,j; float resultado=0.0;
  for (i=0;i<N;i++)
    for (j=0;j<N;j++)
      resultado +=A[i][j]/(factor+j);
  return resultado;
}
void procesamiento() {
  float A[N][N], B[N][N], n1, n2, n3, n4;
  generar(A);          // T1
  n1=fprocesa(A,1.1);  // T2
  n2=fprocesa(A,1.2);  // T3
  generar(B);          // T4
  n3=fprocesa(B,1.1);  // T5
  n4=fprocesa(B,1.2);  // T6
  printf("Resultado: %f\n", n1+n2+n3+n4); // T7
}
```

0.4 p. (a) Determina el grafo de dependencias de la función `procesamiento`, obteniendo un camino crítico y su longitud, así como el grado medio y el máximo de concurrencia.

Solución:



Un camino crítico sería $T_1 \rightarrow T_2 \rightarrow T_7$. La función `fprocesa` tiene un coste de $3N^2$ Flops, por lo tanto obtenemos que su longitud es de $9N^2 + 3$. El grado máximo de concurrencia es 4 y el grado medio de concurrencia es $M = \frac{24N^2+3}{9N^2+3} \approx 2{,}67$

0.6 p. (b) Realiza una implementación paralela eficiente de la función `procesamiento` basada en secciones.

Solución:

```
void procesamiento() {
  float A[N][N], B[N][N];
  float 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=fprocesa(A,1.1);
        #pragma omp section
        n2=fprocesa(A,1.2);
        #pragma omp section
        n3=fprocesa(B,1.1);
        #pragma omp section
        n4=fprocesa(B,1.2);
      }
    }
    printf("Resultado: %f\n", n1+n2+n3+n4);
  }
```

0.4 p.

(c) Calcula el coste computacional secuencial. Calcula el coste paralelo, el speed-up y la eficiencia en caso de usar 2 hilos y también en caso de usar 4 hilos.

Solución:

$$t(N) = 24N^2 Flops$$

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

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

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

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

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

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

Cuestión 3 (1.1 puntos)

El siguiente programa obtiene una aproximación del número PI, a partir de la generación de 200000 puntos calculando el número de puntos que se encontrarían dentro o fuera de una circunferencia de radio r. A partir de ese cociente se obtiene una aproximación del área y por tanto del número PI. La función `aleatorio(0,N)` devuelve un número entero aleatorio entre 0 y 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], dentro=0, fuera=0;

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

```
  for (i=0;i<SAMPLES;i++) {
    ipos = aleatorio(0,N);
    jpos = aleatorio(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)
         dentro+=A[i][j];
       else
         fuera+=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", dentro, fuera);
  printf("Pi = %f\n", (4.0*dentro)/(dentro+fuera));
  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) Realiza una implementación paralela mediante OpenMP. No hace falta paralelizar el bucle que inicializa la matriz A y se pueden usar varias regiones paralelas.

Solución:

```
#define N 20
#define SAMPLES 200000
int main() {
  int i,j, ipos, jpos, imax, jmax, imin, jmin, r;
  int min, max;
  int A[N][N];
  int dentro=0, fuera=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 (+:dentro, fuera)
    for (i=0;i<N;i++) {
      for (j=0;j<N;j++) {
         if ((r-i)*(r-i)+(r-j)*(r-j)<r*r)
           dentro+=A[i][j];
         else
           fuera+=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", dentro, fuera);
    printf("Pi = %f\n", (4.0*dentro)/(dentro+fuera));
    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 para que se muestre el valor de las variables `dentro` y `fuera` que ha calculado cada hilo.

Solución:

```
    ...
    r = N/2;
  /* Hasta la línea anterior, mismo código que en apartado a */
    #pragma omp parallel private (j) reduction (+:dentro, fuera)
    {
       #pragma omp for
       for (i=0;i<N;i++) {
         /* Dentro del bucle, mismo código que en apartado a */
```

```
        ...
      }
      printf("Valores de dentro y fuera para el hilo %d: %d y %d\n",
            omp_get_thread_num(), dentro, fuera);
  }
/* A partir de aquí, mismo código que en programa original */
  printf("d=%d, f=%d\n", dentro, fuera);
  ...
```