Computación Paralela
Grado en Ingeniería Informática (ETSInf)
Curso 2022-23 ◇ Examen final 5/2/24 ◇ Bloque OpenMP ◇ Duración: 1h 30m

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Cuestión 1 (1.1 puntos)
Dado el siguiente código:

```
double f(double A[N][N], double B[N][N], double C[N][N], double t) {
  int i,j,k,c=0;
  double f=0, s=0, m, aux;
  for (i=0; i<N; i++) {
    m=0;
    for (j=0; j<i; j++) {
      aux=0;
      for (k=0; k<N; k++)
        aux += A[i][k]*B[k][j]*B[k][j];
      C[i][j] = aux;
      if (aux>m) m = aux;
      if (aux>t) c++;
      f += aux*aux;
    }
    s += m;
  }
  return (s+f)/c;
}
```

0.35 p. (a) Haz una versión paralela basada en la paralelización del bucle `i`.

Solución: Se añadiría la siguiente directiva justo antes del bucle.

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

0.45 p. (b) Haz una versión paralela basada en la paralelización del bucle `j`.

Solución: Se añadiría la siguiente directiva justo antes del bucle.

```
#pragma omp parallel for private(aux,k) reduction(max:m) reduction(+:c,f)
```

0.3 p. (c) Calcula el tiempo de ejecución secuencial en flops, detallando los pasos. Recuerda que las comparaciones entre números reales, no aportan ningún flop.

Solución:

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

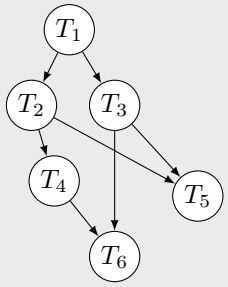
Cuestión 2 (1.2 puntos)
Dada la siguiente función, donde `n` es una constante predefinida, suponemos que las matrices `A` y `B` han sido rellenadas previamente, y además:

- La función `processcol(A,i,x)` modifica la columna `i` de la matriz `A` a partir de cierto valor `x`. Su coste es $2n$ flops.
- La función del sistema `fabs` devuelve el valor absoluto de un número en coma flotante y se puede considerar que tiene un coste de 1 flop.

```
double myfun(double A[n][n], double B[n][n], double C[n][n], double D[n][n]) {
  int i,j;
  double alpha=0.0,beta=1.0;
  for (i=0;i<n;i++) alpha += fabs(A[i][i]-B[i][i]);
  for (i=0;i<n;i++) processcol(A,i,alpha);
  for (i=0;i<n;i++) processcol(B,i,alpha);
  for (i=0;i<n;i++) beta *= A[i][n-i-1];
  for (i=0;i<n;i++) {
    for (j=0;j<n;j++) {
      C[i][j] = A[i][j]+0.5*B[i][j];
    }
  }
  for (i=0;i<n;i++) {
    for (j=0;j<n;j++) {
      D[i][j] = beta*B[i][j];
    }
  }
}
```

0.3 p. (a) Dibuja el grafo de dependencias de datos entre las tareas, suponiendo que hay 6 tareas correspondientes a cada uno de los bucles `i`.

Solución:



0.6 p. (b) Implementa una versión paralela, basada en el grafo, mediante OpenMP utilizando una sola región paralela y usando paralelismo de tareas. Ten en cuenta los costes de cada una de las tareas para tratar de reducir el tiempo de ejecución de la función paralela.

Solución: La tarea T_1 no es concurrente con ninguna otra, por lo que se puede hacer fuera de la región paralela. En cuanto a las otras tareas, el grafo nos ofrece varias implementaciones posibles. Para determinar, cuál es la más adecuada, hay que tener en cuenta el coste de las tareas:

T_1	$3n$
T_2	$2n^2$
T_3	$2n^2$
T_4	n
T_5	$2n^2$
T_6	n^2

La mejor implementación usando la construcción `sections` sería agrupando las tareas T_4 y T_6.

```
double myfun_par(double A[n][n], double B[n][n], double C[n][n], double D[n][n]) {
  int i,j;
```

```
double alpha=0.0,beta=1.0;
for (i=0;i<n;i++) alpha += fabs(A[i][i]-B[i][i]);   /* T1 */
#pragma omp parallel private(i,j)
{
  #pragma omp sections
  {
    #pragma omp section
    for (i=0;i<n;i++) processcol(A,i,alpha);   /* T2 */
    #pragma omp section
    for (i=0;i<n;i++) processcol(B,i,alpha);   /* T3 */
  }
  #pragma omp sections
  {
    #pragma omp section
    {
      for (i=0;i<n;i++) beta *= A[i][n-i-1];   /* T4 */
      for (i=0;i<n;i++) {                      /* T6 */
        for (j=0;j<n;j++) {
          D[i][j] = beta*B[i][j];
        }
      }
    }
    #pragma omp section
    {
      for (i=0;i<n;i++) {                      /* T5 */
        for (j=0;j<n;j++) {
          C[i][j] = A[i][j]+0.5*B[i][j];
        }
      }
    }
  }
}
```

0.3 p. (c) Obtén el grado medio de concurrencia del grafo

Solución: Teniendo en cuenta los costes obtenidos en el apartado anterior, el coste secuencial es:

$$t(n) = 3n + 2n^2 + 2n^2 + n + 2n^2 + n^2 = 7n^2 + 4n \approx 7n^2 \text{ flops}$$

El camino crítico es $T_1 - T_3 - T_5$, cuya longitud es:

$$L = 3n + 2n^2 + 2n^2 = 4n^2 + 3n \approx 4n^2 \text{ flops}$$

Por tanto, el grado medio de concurrencia es:

$$M = \frac{t(n)}{L} = \frac{7n^2}{4n^2} = \frac{7}{4}$$

Cuestión 3 (1.2 puntos)
Dada la siguiente función

```
int fun(int v[], int n) {
  int i, max, sum, ind;
  int count[100];

  for (i=0;i<100;i++)
    count[i]= 0;

  for (i=0;i<n;i++)
     count[v[i]%100]++;

  sum = 0;
  for (i=0;i<100;i++)
     sum += count[i];
  sum /= 100;

  max = count[0];
  ind = 0;
  for (i=1;i<100;i++)
     if (count[i]>sum)
       if (count[i]>max) {
          max = count[i];
          ind = i;
       }
  return ind;

}
```

1 p. (a) Realizar una versión paralela eficiente mediante OpenMP. Nota: No es necesario utilizar una única región paralela.

Solución:

```
int funpar(int v[], int n) {
  int i, max, ind, sum=0;
  int count[100];

  #pragma omp parallel for
  for (i=0;i<100;i++)
    count[i]= 0;

  #pragma omp parallel for
  for (i=0;i<n;i++)
     #pragma omp atomic
     count[v[i]%100]++;

  sum = 0;
  #pragma omp parallel for reduction(+:sum)
  for (i=0;i<100;i++)
     sum += count[i];
  sum /= 100;
```

```
    max = count[0];
    ind = 0;
    #pragma omp parallel for
    for (i=1;i<100;i++)
       if (count[i]>sum)
         if (count[i]>max)
           #pragma omp critical
           if (count[i]>max) {
              max = count[i];
              ind = i;
           }
    return ind;

}
```

0.2 p.

(b) Realiza una nueva versión paralela, lo más eficiente posible, si la parte final de la función (cálculo de max e ind) se modificara de acuerdo con el siguiente fragmento de código:

```
...
max = count[0];
for (i=1;i<100;i++)
  if (count[i]>sum)
     if (count[i]>max)
        max = count[i];
return max;
```

Solución:

```
...
max = count[0];
#pragma omp parallel for reduction(max:max)
for (i=1;i<100;i++)
  if (count[i]>sum)
    if (count[i]>max)
        max = count[i];
return max;
```