Computación Paralela
Grado en Ingeniería Informática (ETSInf)
Curso 2024-25 ◇ Examen parcial 30/10/24 ◇ Bloque OpenMP ◇ Duración: 1h 30m

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Cuestión 1 (1 punto)

Dado el siguiente código, donde `N` y `M` son constantes enteras:

```
double func(double A[][N], double B[][N], double w[]) {
  int i,j,k;
  double x,wt,wk,pxc,val=1000;
  for (j=0; j<N; j++) {
    pxc=1.2;
    for (i=M; i<M+N; i++) {
      x=0.5;
      wt=0.5;
      for (k=-M; k<=M; k++) {
        wk = w[k+M];
        x += A[i+k][j]*wk;
        wt += wk;
      }
      B[i][j] = x/wt;
      pxc *= B[i][j];
    }
    if (pxc<val) val=pxc;
  }
  return val;
}
```

0.2 p. (a) Haz una versión paralela basada en la paralelización del bucle más interno (`k`).

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

```
#pragma omp parallel for private(wk) reduction(+:x,wt)
```

0.2 p. (b) Haz una versión paralela basada en la paralelización del bucle intermedio (`i`).

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

```
#pragma omp parallel for private(x,wt,k,wk) reduction(*:pxc)
```

0.2 p. (c) Haz una versión paralela basada en la paralelización del bucle más externo (`j`).

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

```
#pragma omp parallel for private(pxc,i,x,wt,k,wk) reduction(min:val)
```

0.3 p. (d) Calcula el tiempo de ejecución secuencial en flops, detallando los pasos.

Solución:

$$t = \sum_{j=0}^{N-1} \sum_{i=M}^{M+N-1} \left(2 + \sum_{k=-M}^{M} 3 \right) \approx \sum_{j=0}^{N-1} \sum_{i=M}^{M+N-1} 6M = \sum_{j=0}^{N-1} 6MN = 6MN^2 \text{ flops.}$$

0.1 p. (e) Supongamos que se ha medido el tiempo de ejecución de manera experimental, obteniéndose un tiempo de 40 segundos con un procesador y 10 segundos con 5 procesadores. Indicar el speedup y la eficiencia correspondientes.

Solución:

$$S = \frac{40}{10} = 4, \quad E = \frac{4}{5} = 0{,}8.$$

Cuestión 2 (1.3 puntos)

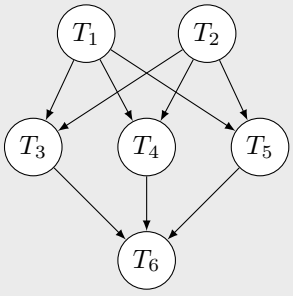
Dada la siguiente función, donde `n` es una constante predefinida, suponemos que las matrices `A`, `B` y `C` han sido rellenadas previamente, y además:

- La función `processcol(M,i,x)` modifica la columna `i` de la matriz `M` 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.

```
void computemat(double A[n][n], double B[n][n], double C[n][n], double D[n][n]) {
  int i,j;
  double alpha=0.0,beta=0.0;
  for (i=0;i<n;i++) alpha += fabs(A[i][i]-B[i][i]);
  for (i=0;i<n;i++) beta += (A[i][i]+B[i][i])/2.0 - C[i][i];
  for (i=0;i<n;i++) processcol(A,i,alpha);
  for (i=0;i<n;i++) processcol(B,i,beta);
  for (i=0;i<n;i++) processcol(C,i,alpha*beta);
  for (i=0;i<n;i++) {
    for (j=0;j<n;j++) {
      D[i][j] = A[i][j]+0.5*B[i][j]-C[j][i];
    }
  }
}
```

0.4 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`. Indica cuál es el camino crítico y calcula el grado medio de concurrencia del grafo.

Solución: Además de las dependencias de flujo, existen anti-dependencias debidas a que T_3, T_4 y T_5 modifican las matrices, que son entrada de T_1 y T_2.



Para obtener el grado medio de concurrencia, necesitamos calcular los costes de cada tarea:

T_1	$3n$	T_2	$4n$	T_3	$2n^2$	T_4	$2n^2$	T_5	$2n^2$	T_6	$3n^2$

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

$$t(n) = 3n + 4n + 2n^2 + 2n^2 + 2n^2 + 3n^2 = 7n + 9n^2 \approx 9n^2 \text{ flops.}$$

El camino crítico es $T_2 - T_4 - T_6$ (o también $T_2 - T_5 - T_6$, o $T_2 - T_3 - T_6$), cuya longitud es:

$$L = 4n + 2n^2 + 3n^2 = 4n + 5n^2 \approx 5n^2 \text{ flops.}$$

Por tanto, el grado medio de concurrencia es:

$$M = \frac{t(n)}{L} = \frac{9n^2}{5n^2} = \frac{9}{5} = 1,8.$$

0.5 p.

(b) Implementa una versión paralela OpenMP con una sola región paralela, basada en el paralelismo de tareas.

Solución: La tarea T_6 no es concurrente con ninguna otra, por lo que se puede hacer fuera de la región paralela. Para el resto de tareas utilizamos dos construcciones `sections`, con 2 y 3 tareas concurrentes, respectivamente. Las variables `alpha` y `beta` son compartidas, ya que no hay posibilidad de que hilos distintos lean y escriban simultáneamente en esas variables. La variable `i` es necesario hacerla privada explícitamente porque no estamos usando la directiva `for`.

```
void computemat(double A[n][n], double B[n][n], double C[n][n], double D[n][n]) {
  int i,j;
  double alpha=0.0,beta=0.0;
  #pragma omp parallel private(i)
  {
    #pragma omp sections
    {
      #pragma omp section
      for (i=0;i<n;i++) alpha += fabs(A[i][i]-B[i][i]);           /* T1 */
      #pragma omp section
      for (i=0;i<n;i++) beta += (A[i][i]+B[i][i])/2.0 - C[i][i];  /* T2 */
    }
    #pragma omp sections
    {
      #pragma omp section
      for (i=0;i<n;i++) processcol(A,i,alpha);       /* T3 */
      #pragma omp section
      for (i=0;i<n;i++) processcol(B,i,beta);        /* T4 */
      #pragma omp section
      for (i=0;i<n;i++) processcol(C,i,alpha*beta);  /* T5 */
    }
  }
  for (i=0;i<n;i++) {                      /* T6 */
    for (j=0;j<n;j++) {
      D[i][j] = A[i][j]+0.5*B[i][j]-C[j][i];
    }
  }
}
```

0.4 p.

(c) Implementa una versión paralela OpenMP con una sola región paralela, basada en el paralelismo de bucles. Cuando sea posible, realiza optimizaciones tales como eliminar las barreras innecesarias.

Solución: En este caso sí incluimos la tarea 6 dentro de la región paralela.

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

```
      double alpha=0.0,beta=0.0;
      #pragma omp parallel
      {
        #pragma omp for reduction(+:alpha) nowait
        for (i=0;i<n;i++) alpha += fabs(A[i][i]-B[i][i]);
        #pragma omp for reduction(+:beta)
        for (i=0;i<n;i++) beta += (A[i][i]+B[i][i])/2.0 - C[i][i];
        #pragma omp for nowait
        for (i=0;i<n;i++) processcol(A,i,alpha);
        #pragma omp for nowait
        for (i=0;i<n;i++) processcol(B,i,beta);
        #pragma omp for
        for (i=0;i<n;i++) processcol(C,i,alpha*beta);
        #pragma omp for private(j)
        for (i=0;i<n;i++) {
          for (j=0;j<n;j++) {
            D[i][j] = A[i][j]+0.5*B[i][j]-C[j][i];
          }
        }
      }
    }
```

Para determinar qué barreras se pueden eliminar o no, basta con tener en cuenta las dependencias de tareas mostradas en el grafo del apartado a). En el primer bucle también es posible eliminar la barrera, a pesar de tener la cláusula de reducción, ya que la variable `alpha` no se usa hasta después de la barrera del segundo bucle.

Cuestión 3 (1.2 puntos)

Mediante la siguiente función, la DGT gestiona las infracciones cometidas por un conjunto de conductores. Para ello, la función recibe como dato de entrada los vectores `Conductores`, `PtosPerder` y `Cuantias`, de `NI` componentes, que almacenan respectivamente el identificador del conductor sancionado y los puntos y la cuantía económica que conllevan cada una de las `NI` infracciones cometidas.

A partir de dichos vectores, la función actualiza los vectores `PtosConductores` y `nSancionesConductores` con los puntos que le quedarán a cada conductor tras la sanción y con el número de infracciones que ha cometido a lo largo del tiempo. Adicionalmente, la función completa el vector `ConductoresSinPtos`, en el cual se guardan los identificadores de los conductores que hayan perdido todos sus puntos. La longitud de dicho vector se devuelve como dato de salida. Finalmente, la función obtiene y muestra por pantalla la mayor sanción económica y la suma del dinero total recaudado en las infracciones.

```
int procesa_multas(int Conductores[],int PtosPerder[],float Cuantias[],
                   int PtosConductores[],int nSancionesConductores[],
                   int ConductoresSinPtos[]) {
  int i,conductor,puntos,cuantia,nCondSinPtos=0;
  float CuantiaMax=0,TotalRecaudado=0;
  for (i=0;i<NI;i++) {
     conductor=Conductores[i];
     puntos=PtosPerder[i];
     cuantia=Cuantias[i];
     TotalRecaudado+=cuantia;
     nSancionesConductores[conductor]++;
     if (PtosConductores[conductor]>0) {
        PtosConductores[conductor]-=puntos;
```

```
         if (PtosConductores[conductor]<=0) {
            PtosConductores[conductor]=0;
            ConductoresSinPtos[nCondSinPtos]=conductor;
            nCondSinPtos++;
         }
      }
      if (cuantia>CuantiaMax)
         CuantiaMax=cuantia;
   }
   printf("Total recaudado: %.2f euros\n",TotalRecaudado);
   printf("Cuantia maxima: %.2f euros\n",CuantiaMax);
   return nCondSinPtos;
}
```

0.8 p. (a) Paralelizar la función mediante OpenMP de la forma más eficiente.

Solución:

```
int procesa_multas(int Conductores[],int PtosPerder[],float Cuantias[],
                   int PtosConductores[],int nSancionesConductores[],
                   int ConductoresSinPtos[]) {
  int i,conductor,puntos,cuantia,nCondSinPtos=0;
  float CuantiaMax=0,TotalRecaudado=0;
  #pragma omp parallel for private(conductor,puntos,cuantia)
              reduction(+:TotalRecaudado) reduction(max:CuantiaMax)
  for (i=0;i<NI;i++) {
     conductor=Conductores[i];
     puntos=PtosPerder[i];
     cuantia=Cuantias[i];
     TotalRecaudado+=cuantia;
     #pragma omp atomic
     nSancionesConductores[conductor]++;
     if (PtosConductores[conductor]>0) {
        #pragma omp critical
        if (PtosConductores[conductor]>0) {
           PtosConductores[conductor]-=puntos;
           if (PtosConductores[conductor]<=0) {
              PtosConductores[conductor]=0;
              ConductoresSinPtos[nCondSinPtos]=conductor;
              nCondSinPtos++;
           }
        }
     }
     if (cuantia>CuantiaMax)
        CuantiaMax=cuantia;
  }
  printf("Total recaudado: %.2f euros\n",TotalRecaudado);
  printf("Cuantia maxima: %.2f euros\n",CuantiaMax);
  return nCondSinPtos;
}
```

0.4 p. (b) Paralelizar de nuevo el bucle, pero esta vez sin utilizar la directiva `for` de OpenMP (es decir, haciendo un reparto explícito de las iteraciones entre los hilos).

Solución:

```
int procesa_multas(int Conductores[],int PtosPerder[],float Cuantias[],
                   int PtosConductores[],int nSancionesConductores[],
                   int ConductoresSinPtos[]) {
  int i,conductor,puntos,cuantia,nCondSinPtos=0,myid,nThreads;
  float CuantiaMax=0,TotalRecaudado=0;
  #pragma omp parallel private(conductor,puntos,cuantia,myid,i)
                  reduction(+:TotalRecaudado) reduction(max:CuantiaMax)
  {
    myid=omp_get_thread_num();
    nThreads=omp_get_num_threads();
    for (i=myid;i<NI;i+=nThreads) {
          ...
             }
          }
       }
       if (cuantia>CuantiaMax)
          CuantiaMax=cuantia;
    }
  }
  printf("Total recaudado: %.2f euros\n",TotalRecaudado);
  printf("Cuantia maxima: %.2f euros\n",CuantiaMax);
  return nCondSinPtos;
}
```