Metodología de la Programación Paralela 2018-2019
Prueba práctica, 24 enero 2019

Estructura de la prueba:

- Entre 16:00 y 17:30, se realizará una prueba escrita respondiendo las cuestiones que se plantean a continuación, pudiendo utilizar material bibliográfico pero sin trabajar con el ordenador. Se entregarán las soluciones antes de empezar la prueba sobre el ordenador. Cada alumno se puede quedar con esquemas de las soluciones que proponen.

- De 17:30 a 19:15, los alumnos trabajarán resolviendo las cuestiones sobre el ordenador, pudiendo acceder a los recursos de internet que consideren, pero sin copiar. Se tendrán en cuenta sólo soluciones que implementen las soluciones propuestas por el alumno en la primera parte. El sistema se cerrará automáticamente a las 19:15.

- Antes de las 19:30 se entregará un documento escrito en el que se indicarán los cambios realizados sobre la versión presentada en la parte escrita y los números de envío en los que se implementan. Para cada cuestión hay que indicar un único número de envío que será el que se evalúe. Para las cuestiones que no se tenga envíos aceptados, el alumno puede indicar el envío que considere que tiene más cercano a una solución correcta.

Se puede trabajar en el laboratorio, pero hay que hacer los envíos finales a calisto.inf.um.es.
Se trabajará con el concurso 2019Enero.
El profesor dejará en un anuncio en el aula virtual los enunciados de los problemas junto con los esquemas y los ficheros con las soluciones secuenciales.
Se muestra la puntuación máxima de cada cuestión. Para cada una de ellas, la mitad de la puntuación corresponde a la parte escrita y la otra a la parte sobre ordenador.
La suma de la puntuación de todas las cuestiones es 11, pero la puntuación máxima que se puede obtener es 10. Esta puntuación se transformará a una puntuación sobre 3 o 4 según se siga la evaluación continua o la no continua.

PROBLEMA 1: Conteo de distancias a elementos iguales en una matriz.
Dada una matriz de caracteres, A, de dimensión $N \times N$, se obtiene otra matriz de enteros D de las mismas dimensiones. En cada posición (i, j) de D se almacena la menor distancia en A entre la posición (i, j) y otra posición con el mismo carácter $A[i, j]$. La distancia entre dos posiciones $(x1, y1)$ y $(x2, y2)$ es la distancia de Manhatan: $|x2 - x1| + |y2 - y1|$. Cuando no hay otra posición con el mismo valor se almacena el valor cero. Finalmente se suman todos los valores almacenados en D.
Por ejemplo, dada la matriz
aabc
cbab
bcca
accb
las distancias son
1 1 2 3
2 2 2 2
2 1 1 2
3 1 1 2
y el resultado es 28.
Se resuelven varios problemas. Para cada problema la función a paralelizar tiene parámetros:

- Input parameter:

  -int N: número de filas y columnas

  -char $*A$: matriz de caracteres, de dimensión $N \times N$

- Return:

  int suma de los valores almacenados en D y calculados como se ha indicado

La entrada tiene en la primera línea el número de problemas. Para cada problema hay una línea con tres valores: N, la semilla para la generación aleatoria, número de caracteres distintos a incluir en el array.
CUESTIÓN 1-a (2 puntos)
Hacer una implementación paralela con OpenMP que utilice paralelismo de bucles. Hay que justificar qué bucles se paralelizan y cuáles no e indicar la forma en que se paralelizan: schedule, variables privadas... (Problema A en mooshak).
CUESTIÓN 1-b (2 puntos)
Hacer una implementación paralela con OpenMP que utilice tareas. Hay que justificar dónde se lanzan las tareas y cuántas se lanzan (Problema A).
CUESTIÓN 1-c (2 puntos)
Hacer una implementación paralela con MPI (Problema B). Explicar con $N = 7$ y 3 procesos cómo funcionaria con la matriz

abcddcb
cdbbadd
cbdbbaa
bddbbac
dbddacc
adccbdb
dcbbbad

CUESTIÓN 1-d (1.5 puntos)
Hacer una implementación MPI+OpenMP (Problema C).

PROBLEMA 2: Subcadena común más larga entre dos cadenas de caracteres.

Dadas dos cadenas de caracteres, $C1$ y $C2$, con N caracteres cada una, se trata de encontrar la longitud de la subcadena más larga común a las dos cadenas, pudiendo haber huecos en los caracteres seleccionados en las cadenas. Por ejemplo, dadas "abbcb" y "abcab" la longitud es 4, y corresponde a la subcadena "abca", que se encuentra en las dos cadenas en varias formas: "ab-cb" y "abc-b", o "a-bcb" y "abc-b".

Se resuelven varios problemas. Para cada problema la función a paralelizar tiene parámetros:

- Input parameter:

  -int N: número de elementos de cada cadena

  -int $*C1$, $*C2$: las dos cadenas

- Return:

  int longitud de la subcadena común más larga

La entrada tiene en la primera línea el número de problemas. Para cada problema hay una línea con tres valores: N, la semilla para la generación aleatoria, número de caracteres distintos a incluir en las cadenas.
CUESTIÓN 2-a (2 puntos)
Hacer una implementación paralela con MPI (Problema E en mooshak). Idea: se puede utilizar el esquema de paralelismo *pipeline*.
Explicar cómo funcionaría con 3 procesos y las cadenas "acbbcadd" y "cbdcbaca".
CUESTIÓN 2-b (1.5 puntos)
Hacer una implementación paralela con OpenMP (Problema D en mooshak). Idea: se puede adaptar el código anterior de MPI para trabajar en OpenMP.

Se incluyen los códigos secuenciales de los dos problemas:
PROBLEMA 1:

```
/*
  CPP_CONTEST=2019Enero
  CPP_PROBLEM=A
  CPP_LANG=CPP+OPENMP
  CPP_PROCESSES_PER_NODE=calisto 1
*/

#include <stdlib.h>
```

```
#include <omp.h>

void distancia(int n,int row,int column, char *A,int *D)
{
    int distancia=0; //guarda la distancia, si no lo encuentra es 0
    int paso=1; //indica el paso por el que vamos
    int encontrado=0; //indica si se ha encontrado
    char elemento=A[row*n+column]; //elemento con el que vamos a comparar
    int rowact,columnizq,columnder; //en cada paso: fila actual, columnas a
//izquierda y derecha de la que vamos
    while(!encontrado && paso!=(2*n-1)) //hay un numero maximo de pasos.
//Si se encuentra se sale
    {
       for(rowact=row-paso,columnizq=column,columnder=column;
!encontrado && rowact<=row;rowact++,columnizq--,columnder++)
//el paso indica cuantas filas por arriba, la primera vez estamos en la misma
//columna, tanto a izquierda como a derecha
//si se encuentra se acaba, o si se llega a la fila actual
//se va bajando de fila y la columna izquierda se hace una a la izquierda y l
//a derecha una a la derecha
       {
          if((rowact>=0) && ((columnizq>=0 &&
elemento==A[rowact*n+columnizq]) || (columnder<n
&& elemento==A[rowact*n+columnder])))
//si no nos salimos de la matriz por arriba y
//no se sale la columna izquierda y coincide con el elemento, o no se sale
//la columna derecha y coincide con el elemento
          {
            encontrado=1;
            distancia=paso;
          }
       }
      for(rowact=row+1,columnizq=column-paso+1,columnder=column+paso-1;
!encontrado && rowact<=row+paso;rowact++,columnizq++,columnder--)
//se empieza en la fila de abajo, la primera vez estamos paso-1 columnas
//mas a la izquierda y a la derecha
//si se encuentra se acaba, o si se llega a la fila actual mas el paso por el que vamos
//se va bajando de fila y la columna izquierda se hace una a la derecha y
//la derecha una a la izquierda
       {
          if((rowact<n) && ((columnizq>=0 && elemento==A[rowact*n+columnizq])
|| (columnder<n && elemento==A[rowact*n+columnder]))) {
            encontrado=1;
            distancia=paso;
          }
       }
       paso++;
    }
    D[row*n+column]=distancia;
}

int sumar(int n,int *m)
//suma todos los elementos del array
{
  int s=0;
  for(int i=0;i<n;i++)
    s+=m[i];
```

```
  return s;
}

int  sec(int n,char *A)
{
  int *D=(int*) calloc(sizeof(int),n*n);
  int result;
  for (int i = 0; i <n; i++) {
     for(int j=0;j<n;j++) {
        distancia(n,i,j,A,D); //calcula la distancia para cada elemento
     }
  }
  result=sumar(n*n,D);
  delete[] D;
  return result;
}
```

PROBLEMA 2:

```
/*
  CPP_CONTEST=2019Enero
  CPP_PROBLEM=D
  CPP_LANG=CPP+OPENMP
  CPP_PROCESSES_PER_NODE=calisto 1
*/
#include <stdlib.h>
#include <omp.h>

int  sec(int n,char *C1,char *C2)
{
  int dim=n+1; //fila y columna 0 para casos base
  int resul;
  int *T=(int*) calloc(sizeof(int),dim*dim); //tabla para programación dinámica

  for(int i=0;i<dim;i++) //caso base fila
     T[i]=0;
  for(int i=1;i<dim;i++) //caso base columna
     T[i*dim]=0;
  for (int i = 1; i <dim; i++) {
     for(int j=1;j<dim;j++) {
//máximo de la fila y columna anterior
        int maximo=(T[(i-1)*dim+j]>T[i*dim+j-1]?T[(i-1)*dim+j]:T[i*dim+j-1]);
//si los elementos de las cadenas coinciden y añadiendo 1 al valor en la diagonal
//anterior es mayor que el máximo, se actualiza
        if(C1[i-1]==C2[j-1] && T[(i-1)*dim+j-1]+1>maximo)
          maximo=T[(i-1)*dim+j-1]+1;
        T[i*dim+j]=maximo;
     }
  }
//la longitud de la cadena más larga es el último valor en la tabla de programación dinámica
  resul=T[dim*dim-1];
  delete[] T;
  return resul;
}
```

Solución parte escrita

CUESTIÓN 1-a

Se puede incluir

`#pragma omp parallel for`

para paralelizar el primer bucle de `sec`. Hay que añadir que i es privada (lo hace por defecto).

El trabajo que se realiza para cada elemento (i y j) tiene distinto coste ya que en `distancia` se acaba antes dependiendo de cómo de alejado esté el elemento igual más cercano. Por tanto, haremos una asignación dimámica con tamaño de reparto 1 (cláusula `schedule(dynamic,1)`).

Cada hilo actualiza en D el elemento (i, j), y como se asignan a hilos distintos valores de i distintos, los hilos escriben en zonas separadas y no hace falta asegurar exclusión mutua.

Se puede paralelizar la función `sumar` de la misma forma, ahora añadiendo la cláusula `reduction(+:s)`. De todas formas, esta función tendrá menos coste que `distancia`, por lo que tampoco es imprescindible paralelizarla.

CUESTIÓN 1-b

Utilizamos el esquema general de lanzamiento de tareas desde un único hilo dentro de una región paralela, con una tarea por cada fila de la matriz

```
#pragma omp parallel
{
    #pragma omp single
    for(int i=0;i < n;i++)
   {
      #pragma omp task firstprivate(i)
      for(int j=0;j < n;j++)
        distancia...
   }//for
}//parallel
```

No es necesario el `taskwait` pues se sincronizan al acabar el `parallel`.

CUESTIÓN 1-c

Se hace broadcast de n del proceso 0 a los demás.

Los procesos distintos del cero tienen que reservar espacio para A.

Se hace broadcast de A con raíz el proceso cero.

El trabajo a realizar por cada proceso puede ser con

```
for(i=nodo;i < n;i+=np)
```

Si los bloques de filas con que trabajan queremos que sean contiguos sería con n/np filas para cada proceso, salvo los que tienen identificador menor que n módulo el número de procesos, a los que les corresponde una fila más. Aunque la distribución cíclica balancea mejor el trabajo, se ha programado con distribución por bloques contiguos. Cada proceso calcula su inicio y final teniendo en cuenta el número de filas que les corresponde a los procesos anteriores y a él mismo. Si el número de filas que le corresponde es *rowloc*, reserva D de tamaño $rowloc * n$. El trabajo a realizar en cada proceso es

```
for (int i = inicio; i <final; i++)
{
   for(int j=0;j<n;j++)
   {
      distancia(n,i,j,A,i-inicio,D);
   }
}
```

donde se añade un índice a `distancia` para que cada proceso trabaje en D desde la fila 0.

Cada proceso calcula las sumas de su D local, y se acumulan en el proceso 0 con una reducción con suma.

En el ejemplo los datos

abcddcb

cdbbadd
cbdbbaa
bddbbac
dbddacc
adccbdb
dcbbbad

se duplican en los tres procesos, el proceso 0 calcula las distancias de las tres primeras filas, el 1 de las 2 siguientes y el 2 de las 2 últimas.

Los valores de D en los distintos procesos son:

En el proceso 0:
5 2 3 1 1 3 4
1 2 1 1 2 1 1
1 2 1 1 1 1 1
En el 1:
2 2 1 1 1 1 1
2 2 1 1 2 1 1
En el 2:
1 2 1 1 1 2 2
2 2 1 1 1 3 2

CUESTIÓN 1-d

Basta con combinar la versión MPI con la primera de OpenMP. Sólo hay que añadir la cláusula

```
#pragma omp parallel for private(i) schedule(dynamic,1)
```

antes del `for` del i en `sec`.

CUESTIÓN 2-a

Utilizamos un esquema pipeline en el que dividimos las columnas de la tabla de programación dinámica entre los procesos. Además, en cada proceso habrá una columna inicial (la cero) que en el proceso cero tendrá los casos base y en el resto de procesos servirá para recibir los datos obtenidos por los procesos anteriores. Así, en el ejemplo, la distribución de datos será:

	proc. 0				proc. 1				proc. 2		
	0	c	b	d	0	c	b	a	0	c	a
0	0	0	0	0	0	0	0	0	0	0	0
a	0										
c	0										
b	0										
b	0										
c	0										
a	0										
d	0										
d	0										

Se han incluido los casos base, inicializados a 0.

Se hace un broadcast de n del proceso cero a los demás. Los que no son el 0 reservan espacio para $C1$ y se hace broadcast de $C1$.

Los procesos distintos de cero reservan espacio para un $C2$ local con n/np elementos, y un elemento más para los procesos cuyo número es menor que n módulo np.

Se hace un `Scatterv` para distribuir los trozos de $C2$ desde el proceso 0 a los demás. Para eso el proceso 0 tiene que calcular los tamaños que le corresponden a cada proceso.

En la computación, todos los procesos hacen n pasos (los que corresponden al primer bucle de `sec`), pero hay tres tipos de procesos, que en cada una de las filas (i) hacen:

El proceso 0:
Calcula los datos correspondientes a la fila i.
Envía el último dato de la fila al proceso 1.
El proceso $np-1$:
Recibe en la primera posición de la fila el dato que le manda el proceso $np-2$.
Calcula los datos correspondientes a la fila i.

El resto de procesos, con identificador p:
Recibe en la primera posición de la fila el dato que le manda el proceso $p-1$.
Calcula los datos correspondientes a la fila i.
Envía el último dato de la fila al proceso $p+1$.
De esta forma, en el ejemplo, en la primera fila primero trabaja el proceso 0, y las tablas quedan:

	proc. 0				proc. 1				proc. 2		
	0	c	b	d	0	c	b	a	0	c	a
0	0	0	0	0	0	0	0	0	0	0	0
a	0	0	0	0	0						
c	0										
b	0										
b	0										
c	0										
a	0										
d	0										
d	0										

Una vez que el proceso 1 ha recibido del cero, puede empezar a trabajar en la fila 1 mientras que el proceso 0 trabaja en la 2:

	proc. 0				proc. 1				proc. 2		
	0	c	b	d	0	c	b	a	0	c	a
0	0	0	0	0	0	0	0	0	0	0	0
a	0	0	0	0	0	0	0	1	1		
c	0	1	1	1	1						
b	0										
b	0										
c	0										
a	0										
d	0										
d	0										

Y ya trabajan al mismo tiempo los tres procesos, obteniéndose:

	proc. 0				proc. 1				proc. 2		
	0	c	b	d	0	c	b	a	0	c	a
0	0	0	0	0	0	0	0	0	0	0	0
a	0	0	0	0	0	0	0	1	1	1	1
c	0	1	1	1	1	1	1	1	1		
b	0	1	2	2	2						
b	0										
c	0										
a	0										
d	0										
d	0										

Y finalmente:

	proc. 0				proc. 1				proc. 2		
	0	c	b	d	0	c	b	a	0	c	a
0	0	0	0	0	0	0	0	0	0	0	0
a	0	0	0	0	0	0	0	1	1	1	1
c	0	1	1	1	1	1	1	1	1	2	2
b	0	1	2	2	2	2	2	2	2	2	2
b	0	1	2	2	2	2	3	3	3	3	3
c	0	1	2	2	2	3	3	3	3	4	4
a	0	1	2	2	2	3	3	4	4	4	5
d	0	1	1	3	3	3	3	4	4	4	5
d	0	1	1	3	3	3	3	4	4	4	5

Al final, el proceso $np-1$ manda al 0 la solución, que está en la última fila y columna.

CUESTIÓN 2-b

No se puede utilizar paralelismo de bucles pues los datos de una fila dependen de la fila anterior, y los de

una columa de la columna anterior.

Una alternativa es adaptar el programa MPI a trabajo con hilos. Se lanzan *np* hilos al principio y cada hilo calcula su parte de la tabla de programación dinámica. Dado que estamos en memoria compartida la tabla no se duplica sino que calculamos qué columnas actualiza cada hilo, de forma similar a como calculamos donde empieza el trabajo de cada proceso en el esquema MPI. No van a ser necesarias comunicaciones, pero inicializamos la tabla a -1 y cuando una posición tenga un valor distinto de -1 es porque se ha calculado. En el ejemplo, la tabla tras inicializar los casos bases, sería:

	proc. 0				proc. 1			proc. 2	
	0	c	b	d	c	b	a	c	a
0	0	0	0	0	0	0	0	0	0
a	0	-1	-1	-1	-1	-1	-1	-1	-1
c	0	-1	-1	-1	-1	-1	-1	-1	-1
b	0	-1	-1	-1	-1	-1	-1	-1	-1
b	0	-1	-1	-1	-1	-1	-1	-1	-1
c	0	-1	-1	-1	-1	-1	-1	-1	-1
a	0	-1	-1	-1	-1	-1	-1	-1	-1
d	0	-1	-1	-1	-1	-1	-1	-1	-1
d	0	-1	-1	-1	-1	-1	-1	-1	-1

Y tras los pasos 1 y 2:

	proc. 0				proc. 1			proc. 2	
	0	c	b	d	c	b	a	c	a
0	0	0	0	0	0	0	0	0	0
a	0	0	0	0	-1	-1	-1	-1	-1
c	0	-1	-1	-1	-1	-1	-1	-1	-1
b	0	-1	-1	-1	-1	-1	-1	-1	-1
b	0	-1	-1	-1	-1	-1	-1	-1	-1
c	0	-1	-1	-1	-1	-1	-1	-1	-1
a	0	-1	-1	-1	-1	-1	-1	-1	-1
d	0	-1	-1	-1	-1	-1	-1	-1	-1
d	0	-1	-1	-1	-1	-1	-1	-1	-1

	proc. 0				proc. 1			proc. 2	
	0	c	b	d	c	b	a	c	a
0	0	0	0	0	0	0	0	0	0
a	0	0	0	0	0	0	1	-1	-1
c	0	1	1	1	-1	-1	-1	-1	-1
b	0	-1	-1	-1	-1	-1	-1	-1	-1
b	0	-1	-1	-1	-1	-1	-1	-1	-1
c	0	-1	-1	-1	-1	-1	-1	-1	-1
a	0	-1	-1	-1	-1	-1	-1	-1	-1
d	0	-1	-1	-1	-1	-1	-1	-1	-1
d	0	-1	-1	-1	-1	-1	-1	-1	-1

Las comunicaciones entre procesos en el cálculo de cada fila en la implementación MPI se sustituyen por accesos a la tabla:

Los envíos son ahora escribir en la parte correspondiente de la tabla, con lo que se sustituye el valor inicial de -1. Sólo hará falta exclusión mutua en la lectura.

Las recepciones son esperas activas con acceso en exclusión mutua a la última columna que tiene que actualizar el hilo anterior. Cuando no hay un -1 en esa posición es que el hilo anterior ya ha acabado su trabajo en esa fila, y se puede seguir con las siguientes columnas.

Solución parte ordenador

CUESTIÓN 1-a

El envío en Mooshak es el 2, con un tiempo de 1156 milisegundos, y un speed-up de aproximadamente 4, lo que, teniendo en cuenta que `calisto` tiene 4 cores, es satisfactorio.

CUESTIÓN 1-b

El envío en Mooshak es el 3, con un tiempo de 1069 milisegundos, y un speed-up de aproximadamente 4, lo que, teniendo en cuenta que `calisto` tiene 4 cores, es satisfactorio.

CUESTIÓN 1-c

El envío en Mooshak es el 5, con un tiempo de 1346 milisegundos, y un speed-up de 3.19. Teniendo en cuenta el coste adicional de las comunicaciones y que no se utiliza una distribución por bloques en vez de cíclica, que balancearía más el trabajo, el speed-up es satisfactorio.

CUESTIÓN 1-d

El envío con dos procesos y dos hilos en cada proceso es el 7, con un tiempo de 1154 milisegundos, y un speed-up de 3.99.

CUESTIÓN 2-a

El envío en Mooshak es el 11, con un tiempo de 1244 milisegundos, y un speed-up de 3.38. Las comunicaciones hacen que no lleguemos a un speed-up de 4, pero como el coste de la computación es cuadrátrico y el de las comunicaciones lineal, se consigue una aceleración importante.

CUESTIÓN 2-b

El envío con dos procesos y dos hilos en cada proceso es el 9, con un tiempo de 3520 milisegundos, y un speed-up de 1.28. Las prestaciones son mucho peores que con MPI, lo que puede estar causado por los accesos en exclusión mutua, que se podrían sustituir por un array de llaves.