

Metodología de la Programación Paralela 2017-2018  
Práctica 2 de Esquemas Algorítmicos  
Evaluación CONTINUA  
martes 12 de diciembre 2017, puntuación sobre 0.5

Esta práctica se realizará como un ensayo para el examen de programación el 15 de enero. Se realizará de forma individual, y con estructura similar a la que se usará en el examen:

- Durante la primera hora, entre 17:45 y 18:45, se realizará una prueba escrita, en la que los alumnos trabajarán 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 18:45 a 19:45, 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 19:45.
- Antes de las 20:00 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.

El profesor se pondrá en contacto con los alumnos para acordar la revisión de prácticas, que será entre el 15 y 21 de diciembre. Se puede trabajar en el laboratorio, pero hay que hacer los envíos finales al cluster Heterosolar a través de Mooshak. Si hay problemas con el uso de Heterosolar se puede realizar los envíos a calisto.inf.um.es.

Se trabajará con el problema de “filas que caen” del concurso 2017EuroPar, que en Heterosolar es el F (versión OpenMP), G (MPI) y H (MPI+OpenMP), y en calisto el D (versión OpenMP), E (MPI) y F (MPI+OpenMP).  
CUESTIÓN 1 (0.1)

Hacer una implementación que explote el paralelismo de hilos en memoria compartida con OpenMP.

CUESTIÓN 2 (0.1)

Hacer una implementación que explote el paralelismo de hilos en memoria compartida con OpenMP usando el paradigma *pipeline*.

CUESTIÓN 3 (0.1)

Hacer una implementación que explote el paralelismo de paso de mensajes con MPI.

CUESTIÓN 4 (0.1)

Hacer una implementación que explote el paralelismo de paso de mensajes con MPI siguiendo el paradigma *pipeline*.

CUESTIÓN 5 (0.1)

Hay que hacer una implementación MPI+OpenMP.

Se incluyen el enunciado del problema y el código secuencial que se proporciona:

**Problema Matrix with falling rows**

In a matrix  $A$  of size  $N \times M$  the rows fall with each element adding its value to its neighbors (in vertical and diagonal) in the row below. Initially, the first row falls, modifying the second row, the second row modifies the third, etc. After the falling of the first row finishes, the second row starts its falling, next the third row, etc.

For example, for a matrix  $3 \times 4$ :

```
0 1 2 1
0 2 -1 -2
1 2 1 -2
```

the successive steps are

```
0 1 2 1 0 1 2 1 0 1 2 1
1 5 3 1 1 5 3 1 1 5 3 1
1 2 1 -2 7 11 10 2 13 20 19 6
```

But in our problem the values are doubles and are maintained between -1 and 1.

The input contains in the first line the number of problems to be solved. For each problem there is a line with four values: two for the sizes ( $N$  and  $M$ ) of the matrix; the seed for the random generation of the values of the matrix between -1 and 1; the gap used when writing the results. A number of problems is solved. For each problem the function to parallelize has:

- Input parameters:
  - int  $N$ : number of rows
  - int  $M$ : number of columns
- Input-output parameter:
  - double  $*A$ : the matrix  $N \times M$
- The MPI version has parallelism parameters:
  - int  $node$ : identification of the MPI process
  - int  $np$ : total number of MPI processes

### sec.cpp

```

/*
  CPP_CONTEST=2017Final
  CPP_PROBLEM=F
  CPP_LANG=C++
  CPP_PROCESSES_PER_NODE=jupiter 1
*/

#include <stdlib.h>
#include <omp.h>

void update(int n,double *vo,double *vd)
{
    vd[0]+=vo[0]+vo[1];
    vd[0]-=(int) vd[0];
    for(int i=1;i<n-1;i++)
    {
        vd[i]+=vo[i-1]+vo[i]+vo[i+1];
        vd[i]-=(int) vd[i];
    }
    vd[n-1]+=vo[n-2]+vo[n-1];
    vd[n-1]-=(int) vd[n-1];
}

void sec(int n,int m,double *A)
{
    for (int i = 0; i <=n-2; i++)
        for (int j = i+1; j < n; j++) {
            update(m,&A[(j-1)*m],&A[j*m]);
        }
}

```

## SOLUCIONES

### Parte escrita

#### CUESTIÓN 1

Vamos a analizar dónde se puede incluir paralelismo de bucles con OpenMP, empezando por los bucles más externos para intentar hacer paralelismo del mayor grano posible.

No se puede paralelizar ninguno de los dos bucles de la función `sec`, ya que los valores de un paso dependen de los de los pasos anteriores.

La opción que queda es paralelizar dentro de la función `update`, incluyendo `#pragma omp parallel for private(i)` antes del bucle en `i`.

### CUESTIÓN 3

Respondemos a la CUESTIÓN 3 antes que a la 2 pues una forma básica de introducir paralelismo MPI es paralelizando cada llamada a `update`, tal como hacemos en la cuestión 1.

Inicialmente se divide la  $A$  por columnas, asignando a cada proceso  $\frac{M}{p}$  ( $p$  es el número de procesos) columnas de la matriz. A los procesos cuyo identificador  $i$  cumpla que  $i < M\%p$  se les asignará una columna más.

En la actualización de cada fila cada proceso actualiza un bloque de columnas, para lo que necesita elementos de la fila anterior. Los elementos de las columnas asignadas al proceso no requieren de comunicaciones, pero en la frontera necesitarán un elemento del proceso a su izquierda y otro a su derecha. Por tanto, cada proceso usará dos variables (`izq` y `der`) para almacenar esos valores. Los procesos de los extremos sólo utilizan una de las variables. En la función `update` lo que hace cada proceso  $P_i$ :

Si  $i = 0$

enviar último dato de fila anterior a  $P_1$

recibir en `der` de  $P_1$

actualizar fila //para el último elemento se usa `der`

en otro caso si  $i = p - 1$

enviar primer dato de fila anterior a  $P_{p-2}$

recibir en `izq` de  $P_{p-2}$

actualizar fila //para el primer elemento se usa `izq`

en otro caso si  $i = p - 1$

enviar primer dato de fila anterior a  $P_{i-1}$

enviar último dato de fila anterior a  $P_{i+1}$

recibir en `izq` de  $P_{i-1}$

recibir en `der` de  $P_{i+1}$

actualizar fila //para el primer elemento se usa `izq` y para el último `der`

### CUESTIÓN 5

La combinación MPI+OpenMP es obvia. Se paraleliza con MPI como la cuestión 3, y la actualización de cada fila se paraleliza con OpenMP como en la cuestión 1, pero en este caso cada proceso actualiza un bloque de la fila.

### CUESTIÓN 4

Respondemos a la cuestión 4 antes que a la dos pues se van a hacer de forma similar. Se asigna un bloque contiguo de  $\frac{N}{p}$  filas (una más los primeros) a cada proceso (MPI) o hilo (OpenMP).

Un proceso  $P_i$  recibe filas del proceso anterior. Como van tratándose filas desde la 0 en adelante, cada proceso recibirá la fila anterior a la suya cuando esta fila quede actualizada con valores que van cayendo de cada una de las anteriores. Así, el proceso  $P_i$  hará  $(i - 1)\frac{N}{p}$  pasos recibiendo la última fila del proceso  $P_{i-1}$  y tratará sus filas a partir de la fila recibida. Después empezará el tratamiento de sus filas. Cada vez que un proceso (salvo el  $p - 1$ ) trate su última fila, la enviará al proceso  $P_{i+1}$ . Un esquema algorítmico puede ser:

En cada  $P_i$ ,  $i = 0, \dots, p - 1$ :

para  $i = 1, \dots, \text{filas anteriores}$

recibir *fila* de  $P_{i-1}$

actualizar primera fila utilizando *fila*

para  $j = 1, \dots, \text{numero filas} - 1$

actualizar fila  $j$  usando la  $j - 1$

si  $i \neq p - 1$

enviar última fila a  $P_{i+1}$

para  $i = 0, \dots, \text{numero filas} - 1$

para  $j = i + 1, \dots, \text{numero filas} - 1$

actualizar fila  $j$  usando la  $j - 1$

si  $i \neq p - 1$

enviar última fila a  $P_{i+1}$

acumular resultados en  $P_0$

### CUESTIÓN 2

El trabajo que hacen los procesos en la cuestión 4 lo hacen en esta hilos. Se ponen en marcha los hilos con `#pragma omp parallel` y dentro de la región paralela cada hilo obtiene su número  $i$ . En el esquema algorítmico anterior se sustituyen los envíos y recepciones por accesos en exclusión mutua a vectores que guardan la información de si un hilo a tratado su última fila, y una copia de ese fila, para que la tome el siguiente proceso. El primer vector es  $nfila$  (0 a  $NUMHILOS-2$ , pues el último hilo no actualiza filas para otro hilo), y el segundo es  $fila$ , de tamaño  $M * (NUMHILOS - 1)$  para tener espacio para los últimos vectores de todos los hilos. El esquema algorítmico se modifica:

$nfila \leftarrow -1$  //se inicializa el array  $nfila$  indicando que no se ha actualizado ninguna de las filas que tienen que leer los hilos

```
# pragma omp parallel //se ponen en marcha los hilos
{
    hilo=omp_get_thread_num()
    para  $i = 1, \dots, \text{filas anteriores}$ 
        en exclusión mutua acceder a  $nfila[hilo-1]$  hasta que valga  $i$ , ponerlo a -1, copiar la fila correspondiente
a  $fila$ 
        actualizar primera fila utilizando  $fila$ 
        para  $j = 1, \dots, \text{numero filas} - 1$ 
            actualizar fila  $j$  usando la  $j - 1$ 
        si  $i \neq p - 1$ 
            en exclusión mutua acceder a  $nfila[hilo]$  hasta que valga  $-1$ , ponerlo a  $i$ 
    para  $i = 0, \dots, \text{numero filas} - 1$ 
        para  $j = i + 1, \dots, \text{numero filas} - 1$ 
            actualizar fila  $j$  usando la  $j - 1$ 
        si  $i \neq p - 1$ 
            en exclusión mutua acceder a  $nfila[hilo]$  hasta que valga  $-1$ , ponerlo a  $i$ 
}
```

### Parte con el ordenador

Se usa la cuenta Domingo en HETEROSOLAR. Se acompañan los códigos.

#### CUESTIÓN 1

El tiempo secuencial es 15800. El envío 65 da tiempo 18160. Es peor que el tiempo secuencial, pues al paralelizar el cálculo de filas, la computación es de grano fino, por lo que no compensa el uso de paralelismo. Se han utilizado 4 hilos, por lo que se podría probar con otros valores, pero los resultados serán similares.

#### CUESTIÓN 3 y 5

Los envíos 67 a 70 corresponden a las versiones MPI y MPI+OpenMP, variando el número de procesos que se ponen en marcha en marte y mercurio (1-1 o 2-2) y el número de hilos dentro de cada proceso (1 o 6). Todos los tiempos son similares y mayores a los obtenidos con OpenMP, lo que es debido al alto coste de las comunicaciones y la frecuencia con que se hacen, al haber una por cada actualización de una fila. Este problema se puede intentar paliar con el uso de *pipeline*.

#### CUESTIÓN 4

El envío 72, con tiempo 11238, corresponde a 3 procesos en marte y 3 en mercurio. Se reduce el tiempo con el uso del esquema *pipeline*. El envío 73, con tiempo 12557, corresponde a 6 procesos en marte. Aunque en este caso las comunicaciones pueden ser más rápidas al estar todos los procesos en el mismo nodo, se obtiene un tiempo ligeramente mejor usando los dos nodos, lo que se puede deber a un mejor uso de la memoria.

#### CUESTIÓN 2

El envío 76 se hace con 12. El tiempo es 4171, mucho mejor que con *pipeline* usando MPI, al no haber comunicaciones en este caso. En el envío 77 se usan 24 hilos, y el tiempo se reduce a 2818, y en el 78 se usan 36, dando un tiempo 2706. Vemos que usar más hilos que cores nos permite un mejor uso de los cores disponibles, pues el sistema balancea la asignación a los cores de los hilos con distinto volumen de computación.