ARQUITECTURA DE SISTEMAS DISTRIBUIDOS
3º GRADO EN ING. INFORMÁTICA. TEC. INFORMÁTICAS
PRÁCTICA 6. OPENMP.

1. OBJETIVOS Y PREPARACIÓN.

En la actualidad los multiprocesadores en un solo chip (denominados habitualmente procesadores de múltiple núcleo) son los predominantes en el mercado. Si queremos aprovechar las posibilidades de estos procesadores es necesario particionar nuestro código en diversos hilos de forma que puedan ejecutarse simultáneamente tanto hilos como procesadores tengamos. Con el fin de facilitar este proceso se han introducido herramientas que permiten realizar esta partición de manera sencilla sin tenerse que preocupar de crear y destruir los hilos o de gestionar las comunicaciones entre ellos de forma explícita. Dentro de estas herramientas una de las más populares es el entorno openMP que hoy en día está soportado en la mayoría de multiprocesadores con memoria compartida. En la familia 80x86 openMP están soportados los compiladores de C++ de Microsoft e Intel (Windows y Linux) y otros. La referencia de openMP podemos encontrarla en http://openmp.org/wp/.

OpenMP utiliza para generar el código paralelo directivas y funciones insertadas en el código C++ (o Fortran). En C las directivas usan el formato:

#pragma omp directiva [comando[(parámetros)], [comando[(parámetros)]…

Tanto las directivas como las funciones no son tenidas en cuenta si:

- no habilitamos la compatibilidad openMP en el compilador. En Visual Studio la opción para habilitar está compatibilidad se encuentra en: *Propiedades de Configuración → C/C++ → Idioma → Compatibilidad con openMP*.

- Y se inserta *#include <omp.h>*

2. REALIZACIÓN DE LA PRÁCTICA.

EJERCICIO 1. Abre la solución **p6**. Dentro de ella elige el proyecto **ej1** para que sea el proyecto inicial y abre dentro de él, el fichero **principal_ej1.cpp**, que contiene este código:

```
using namespace std;

#include <iostream>
#include <omp.h>

int main()
{
  cout << "Hola, mundo!" << endl;
}
```

a) Lo más sencillo que se le puede pedir a OpenMP es que cree diferentes hilos y ejecute en todos ellos un mismo código. Cambia la línea *cout …* en *main()* por esta otra:

```
#pragma omp parallel
{
  cout << "Hola, mundo!" << endl;
}
```

Si el sistema en el que se está ejecutando este programa contiene dos o más procesadores (lógicos o físicos), se habrán creado tantos hilos de ejecución paralela como procesadores existan, así que deberían aparecer tantos mensajes "Hola, mundo!" como hilos se hayan creado.

Si sólo aparece un mensaje, y nuestro sistema es multi-core, probablemente signifique que nuestro compilador no soporta directivas OpenMP, o bien que éstas no han sido habilitadas (ver punto 1 de la práctica)

b) Se puede forzar un número de hilos determinado usando la siguiente función del API de OpenMP: `omp_set_num_threads (numhilos);`

Cambia el código para crear 4, 8 ó 16 hilos.

```
int main()
{
  omp_set_num_threads(4);  // crea 4 hilos para cada pragma omp parallel que se encuentre
  #pragma omp parallel
  {
    cout << "Hola, mundo!" << endl;
  }
}
```

Tanto en este caso como en el anterior, es más que posible que los mensajes en pantalla aparezcan "pisados" unos con otros. Esto es porque las múltples llamadas a *cout* se ejecutan a la vez. Lo solucionaremos en un instante.

c) Habitualmente nos interesa que cada uno de los hilos que se crea sea consciente de que es un hilo dentro de un conjunto de hilos que se ejecuta en paralelo. Para ello hay dos funciones del API de OpenMP:

`int omp_get_thread_num()` : esta función devuelve el número de hilo en el que se encuentra.

`int omp_get_num_threads()` : esta función devuelve el número de hilos que se están ejecutando en paralelo en este momento.

Cambia el código para que el mensaje que se muestra por pantalla indique qué hilo se está ejecutando y cuántos hay. Es decir, que aparezcan líneas con un texto tal como: `Hola, mundo! Soy el hilo __ de __`

d) Si dentro de los hilos hay una cierta sección del código que no es reetrante, como puede ser la escritura en la consola, podemos usar cerrojos (locks) para crear secciones críticas, de forma que nos aseguremos de que sólo un hilo está ejecutando esa región. OpenMP dispone de una API para manejar cerrojos de forma sencilla. Consiste en los siguientes tipos y funciones:

`omp_lock_t` : para cada cerrojo que necesitemos, declararemos una variable que sea de este tipo.

Ej: `omp_lock_t cerrojo;`

`omp_init_lock (omp_lock_t *lock);` Se llamará a esta función indicando la dirección de una variable de tipo `omp_lock_t` para inicializar ese cerrojo. Los cerrojos se inicializan al estado "disponible". Esta operación se hace sólo una vez durante la vida útil del cerrojo y fuera de los hilos en paralelo.

`omp_set_lock (omp_lock_t *lock);` Cuando uno de los hilos necesite entrar en una región crítica, llamará a esta función. Si el cerrojo está disponible, la función termina sin esperar, y lo marca como no disponible. Si el cerrojo no estuviera disponible, el hilo se queda esperando hasta que lo esté.

`omp_unset_lock (omp_lock_t *lock);` Cuando un hilo termine su región crítica, llamará a esta función para marcar el cerrojo como disponible. De esa forma otro hilo tendrá la oportunidad de entrar en la región crítica.

`omp_destroy_lock (omp_lock_t *lock);` Se llamará a esta función la dirección de una variable de tipo `omp_lock_t` para destruir e invalidar ese cerrojo. Una vez destruido no puede volver a usarse salvo que se vuelva a inicializar. Esta operación se realizará fuera de los hilos en paralelo.

Cambia el código para que cree 16 hilos de ejecución, y usa un cerrojo para encerrar en una región crítica la ejecución de la orden *cout* para imprimir en pantalla. Comprueba que aparecen los 16 mensajes correspondientes, sin pisarse unos a otros.

e) Busca información sobre la directiva `#pragma critical` y úsala para simplificar el programa obtenido en **d)**

EJERCICIO 2. Ahora vamos a usar el proyecto **ej2**, en la misma Solución. Activa **ej2** y márcalo como proyecto de inicio. Abre dentro de él, el fichero **principal_ej2.cpp** . En él hay dos funciones, que usaremos, la primera para los apartados a) b) y c); y la segunda para el apartado d). Comenta la que no vayas a usar, y descomenta la que sí.

a) Queremos paralelizar este bucle. Es decir, que varias iteraciones del mismo se ejecuten en diferentes hilos. Para ello se ha añadido una directiva `#pragma omp parallel` de forma que el bloque que contiene el bucle se ejecute concurrentemente en varios hilos. Queremos que se ejecute el mismo número de iteraciones, pero cada una de ellas (o un grupo de ellas) en hilos diferentes. Prueba primero el programa comentando el `#pragma` y comprueba que el número de iteraciones que produce es el esperado. Descomenta el `#pragma` para que surta efecto ¿Qué resultado tenemos ahora? (ignora el que los resultados en pantalla se pisen, o mejor aún, usa cerrojos o `#pragma critical` para evitar que esto se produzca)

b) Usa las funciones `omp_get_thread_num()` y `omp_get_num_threads()` para particionar el bucle de la siguiente forma: el bucle da *N* vueltas. Hay *T* hilos, donde *T* es el valor devuelto por `omp_get_num_threads()`. Supondremos que el número de vueltas *N* es divisible entre el número de hilos *T*, o sea, *N % T = 0*. Entonces, para cada hilo *t* (*0* ≤ *t* < *T*), el valor inicial del bucle *for* será *t*(N/T)* y el valor final (al cual no debe llegar el bucle), *(t+1)*(N/T)*. Comprueba que el número de iteraciones es el correcto tanto si el *pragma* existe como si no.

c) Usando `#pragma omp parallel for` simplifica el programa del apartado b)

d) El bucle mostrado en esta función calcula la suma de los *N* primeros términos de *0+1+2+3+…+(N-1)*. Aplicando los `#pragma` adecuados, distribuye el cálculo de la suma en distintos hilos. ¿Qué resultados se observan? ¿Cómo se puede arreglar?

EJERCICIO 3. Activa el proyecto **ej3**, márcalo como proyecto de inicio y dentro de él, abre el fichero **principal_ej3.cpp**

Este programa cronometra el tiempo de ejecución de la función que se ponga en el sitio indicado. Se pide escribir una serie de funciones usando las directivas de OpenMP, y cronometrarlas. Para cada función hay que cronometrar su tiempo de ejecución dos veces: una comentando todos los `#pragma omp` que haya insertado en la función, y otra con todas las directivas activadas. Con los datos obtenidos, rellene la tabla de la última página de este boletín.

La funciones son:

```
void saxpy (int n);
```

Calcula *Y = a*x + Y*, donde el escalar *a*, y los vectores *X*, *Y* de *n* elementos, están ya definidos como variables globales.

```
float dotproduct (int n);
```

Calcula y devuelve el producto escalar de los vectores de *n* elementos *X* e *Y*, ya definidos. El producto escalar de dos vectores se calcula como: $\overline{X} \cdot \overline{Y} = \sum_{i=0}^{n-1} (x_i \times y_i)$. En la implementación de esta función debe usarse una variable acumulador.

```
int countzeros (int n);
```

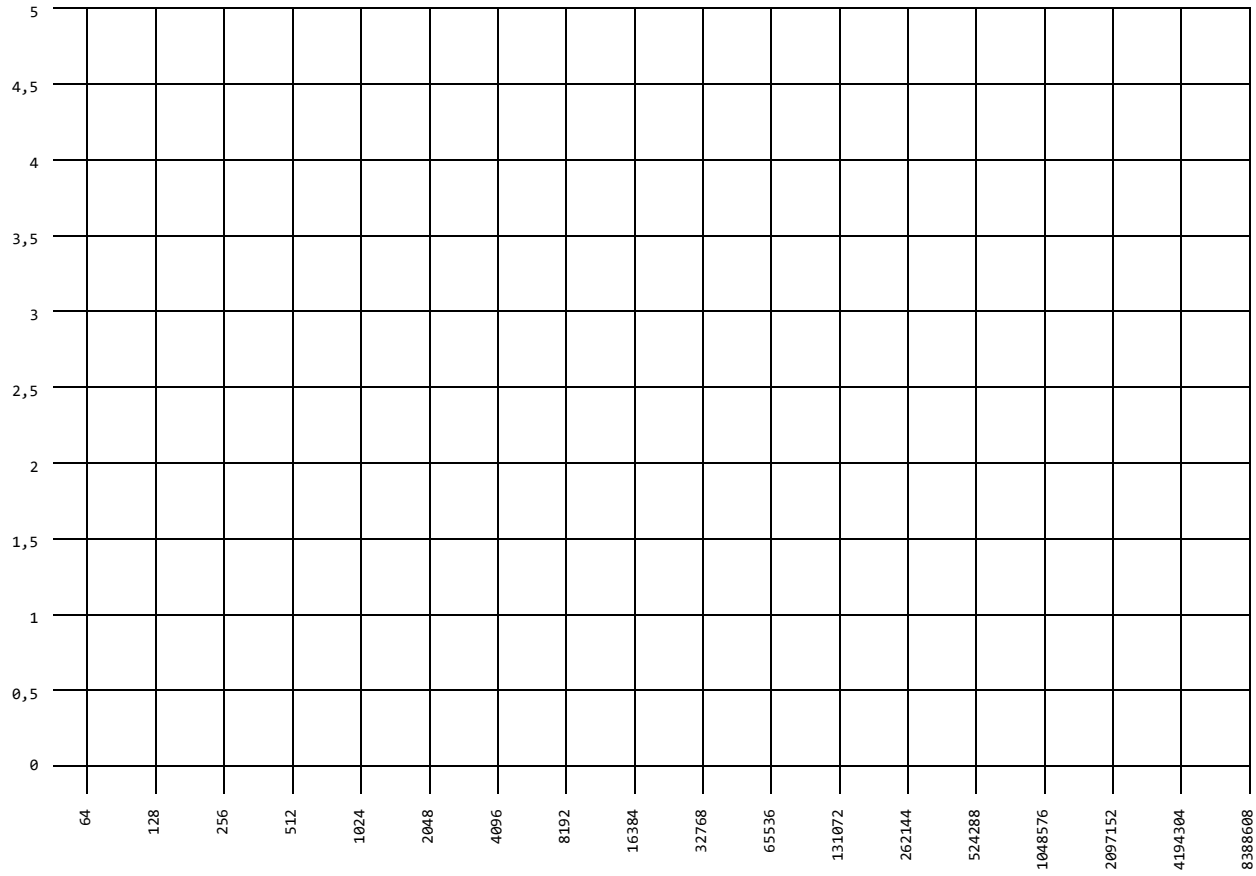
Cuenta y devuelve el número de elementos del vector *Z* que tienen valor 0. El vector *Z* ya ha sido declarado e inicializado como variable global con un total de *n* elementos. En la implementación de esta función debe usarse una sentencia *if* dentro del cuerpo del bucle.

HOJA DE RESULTADOS

Alumno/a: _____ Grupo: _____

N	SAXPY			DOTPRODUCT			COUNTZEROS		
	TsinOMP	*TconOMP*	*Speedup*	*TsinOMP*	*TconOMP*	*Speedup*	*TsinOMP*	*TconOMP*	*Speedup*
64									
32768									
8388608									
Anotar para qué valor de N se produce el mejor resultado (mayor speedup) para cada algoritmo, y los resultados obtenidos									

En cada ejecución del ejercicio 3 se genera un fichero llamado *results.csv* , que puede abrirse directamente en Excel. Para cada algoritmo, recoja los resultados con y sin OpenMP en una hoja Excel (puede usar la que ofrecemos como plantilla), calcule la aceleración (speedup) obtenida con la versión con OpenMP respecto de la versión sin OpenMP, y saque la gráfica para todos los valores de N. Replíquela en el espacio adjunto:



(conteste en el resto de la hoja, o por la parte trasera)

a) A la vista de los resultados del gráfico, ¿es el *speedup* uniforme para todos los valores de N? ¿A qué se puede achacar las discrepancias que hay en los extremos?

b) Para una misma máquina y un mismo valor de N, ¿es el *speedup* conseguido el mismo para los tres algoritmos? ¿A qué se puede achacar las diferencias de *speedup* entre cada uno?