ARQUTECTURA DE SISTEMAS DISTRIBUIDOS. 3º. PRÁCTICA 8b. PARALELIZACIÓN USANDO OPENMP Y MPI DE UN MÉTODO DE RESOLUCIÓN DE LA ECUACIÓN DEL CALOR (FOURIER).

1. OBJETIVOS Y PREPARACIÓN.

En esta práctica vamos a estudiar la optimización de código científico mediante el uso de OpenMP y MPI.

EXPLICACIÓN DE LA APLICACIÓN

Vamos a trabajar con una versión simplificada de una aplicación real que se usa como benchmark en aplicaciones científicas y de multiprocesadores: la resolución de la ecuación del calor. EL calor se transmite de los puntos calientes hacia los fríos, y la solución de la ecuación de Fourier se puede aproximar así: asignar a la temperatura de cada punto, la media aritmética de la temperatura los vecinos.
Para simplificar vamos a suponer que hay una columna izquierda a cierta temperatura máxima `T_MAX` y una columna derecha a cierta temperatura mínima `T_MIN`. De esa forma, el calor siempre fluye de izquierda a derecha y por igual. Por ello, todos los valores de temperatura para las diversas filas deben salir iguales.

2. REALIZACIÓN DE LA PRÁCTICA.

ENTENDIENDO EL CÓDIGO Y TENIENDO EN CUENTA DETALLES DE LA MÁQUINA Y COMPILADOR

En esta prueba, se va a suponer que la matriz de temperaturas `temp_matrix` siempre será cuadrada

Cuando los errores en el chequeo sean grandes (Total quadratic error), la prueba no sirve y se debe corregir.

Dado que el compilador a veces inserta instrucciones vectoriales (multimedia) y otras veces no sin un criterio definido, lo mejor es que nunca las use, de manera que todas las pruebas se harán sin ellas. Para ello, recuerde seleccionar la opción:
Propiedades del proyecto -> C/C++ -> generación de código -> Sin instrucciones mejoradas (Not enhanced instructions)

Debe usar la configuración x86 del menú superior del Visual Studio (NO USAR x64), debido a que hay un fichero de ensamblador de la arquitectura x86 en este proyecto (el de medir los tiempos de ejecución; fichero QueryPerformanceTiming_rdtsc.cpp).

Además el código se ejecuta para diferentes tamaños de matriz, es decir, diferentes tamaños *n* x *n*. Hay dos constantes que (el alumno puede cambiar) para el mayor y menor tamaño de la matriz que se va a probar.

```
#define  MAX_RANGE (512)
#define  MIN_RANGE (16)
```

El número de niveles de caché, su tamaño y el tamaño de la línea (bloque) tendrán una influencia importante en el rendimiento de las pruebas que haremos en esta práctica.

Tener en cuenta que si sólo se usase un core, habrá una sola caché (de las 4 que salen indicadas como 4x32, 4x256). Pero si se paraleliza p ej. en 4 cores, se usarán las 4 cachés.

Asegúrese de que las opciones de energía no estén configuradas para el modo de "Ahorro de energía", ya que en este modo la velocidad de la CPU es variable. Puede cambiarse el modo de energía desde la linea de comandos con la orden powercfg.cpl (Pulse Windows - R, y teclee powercfg.cpl).

El alumno deberá:

- entender cómo está organizado el esquema del código que se proporciona con este enunciado.
- Además, calcular para qué tamaños de la matriz `temp_matrix`, ésta ocupa completamente las cachés (L1, L2,…). Tales tamaños implicarán los primeros fallos de caché..

OBTENIENDO RESULTADOS Y JUSTIFICANDOLOS

El alumno deber paralelizar en Openmp y MPI el trozo indicado con "`//@ STUDENTS MUST WRITE HERE THE PARALLEL VERSION`"

Para MPI, se pide que lo haga con comunicaciones uno a uno (MPI_Send/MPI_Receive) y también con colectivas (MPI_Scatter, MPI_Reduce, etc.).

En vez de declarar e inicializar en el maestro una matriz grande y luego enviar sus trozos a los esclavos, es mejor que los cálculos y reserva de memoria se hagan en los esclavos, para evitar comunicaciones innecesarias que reducen enormemente el rendimiento. Si fuera necesario, se puede reservar memoria de forma dinámica (usando *malloc)*. Por ejemplo, si hubiera *p* esclavos, para reservar espacio en cada esclavo para una matriz, hacer (notar que *A_partial* es un puntero):

```
//malloc:  dinamically allocate memory for each slave:
n_bar = current_intervals / p;
A_partial = (double(*)[current_intervals]) malloc(n_bar * current_intervals * sizeof(double));
```

Después, debe ir realizando pruebas y anotando tiempos y aceleraciones para diferentes rangos *n*, números de procesos y de hilos, unicast frente a multicast, intentando justificar los valores. Por ejemplo:

- Anotar tiempos en función del tamaño de la matriz, para Visual Studio (Windows) y para mpicxx (Linux).
- Paralelizar con openmp el código. Cuidado sobre todo con las variables que se escriben. Anotar tiempos, hallar las aceleraciones para:
  - diferentes rangos *n*, (pensar en el efecto de las cachés según sus tamaños)
  - números de hilos,
- Justificar los resultados. Recordar activar las optimizaciones para favorecer código rápido (flag "Optimización" en `/Ox`) (flag "tamaño o velocidad" en `/Ot`) y las de openmp.
- Paralelizar con MPI el código y proceder similarmente: anotar tiempos, hallar las aceleraciones para diferentes rangos, números de procesos, justificar los resultados.
- Probar finalmente una mezcla de hilos y de procesos en un PC y en varios (ver **apéndice** para lanzar openmpi en varios PCs de la ETSII)

Pistas:

- Compilar el fichero principal_11b.cpp así:
  mpicxx mpiprincipal_11b.cpp –O3 -lm -fopenmp -o mpiprincipal_11b.out
  -O3 es para optimización total ; usar –O0 –g para depurar

- Ha de tenerse cuidado con donde se sitúan los `#pragma omp parallel for ,` puesto que no todos son bucles totalmente paralelos.
- Tenga en cuenta que los vectores y/o matrices se deben enviar a diferentes procesos, por eso tal vez haya que crear matrices nombradas como: *temp_matrix_global*, *temp_matrix_global*, *etc.*
- Recuerde que tras cada fila del bucle externo debe recolectar las partes de la matriz.
- Puede usar comunicaciones colectivas por comodidad: *MPI_Scatter(), MPI_Gather(), MPI_AllGather(), etc.*