ARQUITECTURA DE SISTEMAS DISTRIBUIDOS. 3º. PRÁCTICA 8a. PARALELIZACIÓN USANDO OPENMP Y MPI DE LA MEDICIÓN DE CALIDAD DE NÚMEROS PSEUDOALEATORIOS.

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

En muchas aplicaciones se necesita generar un conjunto de números aleatorios "buenos", es decir, uniformemente distribuidos entre una serie de intervalos ("slices"). Para demostrar la calidad de los números aleatorios se puede crear el histograma de un gran conjunto de números generados. EL histograma contiene la frecuencia de aparición de cada intervalo (o veces que se repite un valor dentro de tal intervalo).
Vamos a realizar diferentes tests (N_TESTS) para diferentes valores de las semillas ("seed"), que es el valor de inicialización de la secuencia de números pseudoaleatorios. De forma que se recogerán un histograma por cada test (por cada semilla) y por tanto se requiere la matriz: `unsigned int hist[N_TESTS][N_SLICES].` Después se probará con otro número de intervalos: cambiaremos la cantidad de intervalos desde `MIN_N_SLICES a MAX_N_SLICES`.

> NOTA: en matemáticas se suelen empezar numerando los subíndices por 1, pero en lenguaje C por 0; de manera que las coordenadas *(x,y)* de los intervalos serían desde (0,0) hasta inclusive, (1,1) exclusive.

2. REALIZACIÓN DE LA PRÁCTICA.

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

El esquema de nuestro código para cada test:

1. simplemente pedir un número aleatorio (con la función rand() u otra que el alumno quiera probar),
2. calcular en que "slice" está tal número, ya que la función rand() devuelve un número grande (0x7fff, es decir, (2^15) - 1) y sólo nos interesa un número pequeño de slices,
3. e incrementar el elemento del histograma: `hist[test][slice] ++;`

Se proporciona ya escrita una versión secuencial del test de de números aleatorios.

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)

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 el histograma para otros valores de "slices", tamaños diferentes.
- Puede incluso probar con otro generador de números aleatorios.

OBTENIENDO RESULTADOS Y JUSTIFICANDOLOS

El alumno deber paralelizar en Openmp y MPI el trozo indicado con:

```
//@  STUDENTS MUST WRITE HERE THE PARALLEL VERSION"

//@  STUDENTS MUST WRITE HERE THE MESSAGES TO COLLECT DATA
```

de la función `par_histogram ()`

Al final del programa siempre debe imprimirse (ya está preparado) la diferencia entre lo devuelto por `par_histogram ()` *,y por* `seq_histogram ()`. Cuando esta diferencia sea grande, mayor que 0.01, el código de `par_histogram ()` está mal, y se debe corregir.

Para MPI, se pide que lo haga con comunicaciones uno a uno (MPI_Send/MPI_Receive) y, si se puede, también con colectivas (MPI_Scatter, MPI_Reduce, etc.).
Además, hay que evitar el tiempo perdido en el envío de matrices.

En vez de declarar y enviar matrices, sus cálculos deben hacerse en las funciones: `seq_histogram` (), `par_histogram` ().
Se ha escrito en la plantilla del código la reserva de memoria de forma dinámica (usando *malloc)* para los valores del histograma. Por ejemplo, si hubiera *p* esclavos, para reservar espacio en cada uno de ellos para una matriz, hacer (notar que *hist_partial* es un puntero):

```
//malloc:  dinamically allocate memory for each slave:
int n_test_per_process  = N_TESTS / p; //@
hist_partial = (unsigned long (*)[range]) malloc(n_test_per_process*range * sizeof(ElementType)); //@
```

La forma de partir `hist[][]` que parece ser más fácil de paralelizar es la del siguiente esquema; sabiendo que se va a ejecutar solamente hasta la columna `range=8` de un total de `N_SLICES=16`: (para 2 procesos/hilos). Notar que un simple *MPI_Gather()* no funcionaría al no ser correlativos los elementos para `range < N_SLICES.`

Test								range								N_SLICES
0																
1																
2																
3																

La parte azul la computaría un proceso/hilo y la roja otro.
Para usar la semilla de números aleatorios tenga en cuenta que cada proceso debe buscar el índice correcto. Algo como :

```
srand(seeds[test_del_proceso_actual + n_test_por_proceso * my_rank]);
```

Evidentemente para la versión secuencial esto no afecta en nada (pues no hay comunicaciones), pero para la paralela, sí. Por ejemplo, así se evitaría hacer un MPI_Scatter de la matriz hist[][]. Así los tiempos de comunicación (envío) son mucho menores que los tiempos de computación pura.

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

- Anotar tiempos en función de `current_slices`, 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 `current_slices`, (pensar en el efecto de las cachés según sus tamaños)
  - números de hilos o procesos,
- 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. NOTA: Para MPI se han añadido mensajes por consola (usando la constante DEBUG_PRINT) y se han valorado las constantes con números pequeños para evitar muchos mensajes en la consola. Cuando el programa MPI funcione, debería aumentarlos. Estos:
  - #define N_REPETITIONS 1
  - #define MAX_N_SLICES (MIN_N_SLICES*1) // max number of slices in which the original random numbers are divided
  - #define N_ITERATIONS (100*(long)MAX_N_SLICES) //NUMBER of tested random numbers //big number to execute a considerable time .
  - #define N_TESTS 2
- 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, si se pudiera)

Pistas:

- Comprobar si está activo openmp en Visual Studio, y que
- Compilar el fichero mpi_Pract8a_histogram.cpp así:
  mpicxx mpi_Pract8a_histogram.cpp –O3 -lm -fopenmp -o mpi_Pract8a_histogram.out
  -O3 es para optimización total ; usar –O0 para depurar
- 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: *values_global, etc. y values_local , etc*.
- Recuerde cómo proceder con acumuladores en OpenMP y en MPI

- Podrían usarse comunicaciones colectivas por comodidad: *MPI_Scatter()*, *MPI_Gather()*, *MPI_AllGather()*, *etc.* Pero se ha de tener muy claro como es el reparto de las matrices.