UNIVERSIDAD DE GRANADA

Arquitectura y computación de altas prestaciones (2022-2023)
Grado en Ingeniería Informática

Práctica 3

Realizado por:
Hlib Bukhtieiev

Ejercicio 1

Se pide encontrar el máximo en un array de double generado aleatoriamente. Lo primero era encontrar una forma de generar los valores aleatorios double – he usado srand(time(NULL)) y una función que genera valores en coma flotante en un rango. Lo segundo, había que hacer que cada hebra tuviera su propia porción del array en su pila. La solución fue reservar memoria en el heap porque es una zona común para todos los TCBs de un mismo proceso y precalcular los intervalos de trabajo para cada hebra.

Por último, como nos piden llegar a una solución correcta pero sin recurrir a mecanismos implícitos de sincronización, reservamos otro array con las mismas entradas que hebras se creen y cada una de éstas almacenará su resultado en la entrada correspondiente. Una vez terminada la ejecución de todas las hebras, el hilo principal del proceso será responsable de encontrar el máximo de los máximos.

Ejercicio 2

El segundo ejercicio es exactamente el mismo, con la excepción de que hay que usar un cerrojo mutex. El planteamiento del problema es idéntico, solamente ahora las hebras comparten el puntero a una variable en el heap que va a ser el máximo del array.

```
struct arguments{
    double* arr;
    double* res;
    int id;
    int low_int;
    int upp_int;
    pthread_mutex_t *lock;
};
```

Cuando una hebra termina de encontrar el máximo del subarray que le corresponde, escribe en la variable en el heap, comprobando previamente si el valor que va a escribir es mayor que el que está ya almacenado. Al existir condiciones de carrera hemos de usar un mutex.

```
pthread_mutex_lock(info->lock);
*info->res = (*info->res > max) ? *info->res : max;
pthread_mutex_unlock(info->lock);
```

Además, había que tener en cuenta un caso especial a la hora de comparar si el valor almacenado en el heap es mayor o menor que el que vamos a escribir – la primera vez. No es un problema porque se definieron los límites dentro de los cuales se generan los números en coma flotante, por tanto, simplemente inicializamos la variable en el heap al mínimo.

```
#define MIN -100
#define MAX 100
```

Ejercicio 3

Hemos de encontrar la distancia de Jaccard de dos conjuntos que se define por la división del cardinal de la intersección entre el cardinal de la unión de dos conjuntos.

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

El problema en este caso se reducía a aplicar el paralelismo a la intersección, porque por la propiedad de los conjuntos el cardinal de la unión de dos conjunto se calcula como |A| + |B| - |A ∩ B|. La idea que he implementado se reduce a darle a cada hebra unos intervalos del array 1 y computar el número de ocurrencias que hay de ese subarray en el segundo array. Como tenemos una variable compartida entre todas las hebras en el heap, de nuevo hay que usar un cerrojo mutex. A partir de ahí calculamos el coeficiente.

Lo que interesa analizar en este caso es la complejidad computacional. Ésta del algoritmo secuencial es O(a*n²) para tamaños de vectores iguales. En la versión paralela la complejidad sigue siendo la misma, sin embargo, conforme se incrementa la cantidad de hebras empleadas, la constante 'a' se hace más pequeña y el algoritmo se hace más lineal.

Lo que ocurre es que se puede disminuir la constante computacional aún más ajustando un poco los datos de entrada del algoritmo. ¿Para qué queremos que para una porción pequeña del primer array se recorra el segundo array entero? Suponiendo que los dos arrays están ordenados, se puede acotar el intervalo del segundo array que se envía a procesar en el algoritmo de intersección. Es decir, por cada hebra solamente se procesarán los elementos del array 2 tales que sean mayores que el contenido del límite inferior del array1 y menores que el contenido del límite superior del array1. Por ejemplo:

0	1	2	3	4	5	6	7	8	9

y

0	2	4	6	8	10	12	14	16	18

Suponiendo que hay dos hebras que van a trabajar con la intersección, la hebra 0 trabaja con los elementos 0-4 del primer array y 0-2 del segundo array, la hebra 1 trabaja con el resto de datos. Con los datos inventados que tenemos el algoritmo no va a mostrar su máximo potencial, pero en condiciones reales con conjuntos de datos enormes, que estadísticamente tienden a aproximarse, la paralelización va a ser óptima.

Con esta implementación puede darse el caso de que una hebra ni siquiera se lance – al detectarse que para un intervalo de datos del array1 no existen ocurrencias en el array2 – de esta forma se ahorraría la creación del TCB y reserva de recursos. Sin embargo, nos puede dar problemas a la hora de hacer el join, así que hay que crear un array con el mismo número de entradas que hebras solicitadas por el usuario, para hacer el control de que hebras se han lanzado y que no. Tambíen se podría intentar asignar las hebras que no fueron lanzadas a intervalos del array que sí se procesarán,

pero creo que sería demasiado enrevesado y en condiciones reales, no se va a dar este caso muy frecuentemente y la ganancia de velocidad tampoco merecería la pena.

```
Introduzca el tamaño del primer array: 2345678

Introduzca el tamaño del segundo array: 4567

Introduzca el numero de hebras a utilizar: 16

Paralelo : el coeficiente de Jaccard es 0.001947, tiempo consumido: 0.723979
Secuencial : el coeficiente de Jaccard es 0.001947, tiempo consumido : 9.378968
```

Ejercicio 4

Planteamiento

He decidido paralelizar el cálculo de PI por área. La función que se les envía a las hebras no ha sufrido modificación salvo que ahora no se devuelve el valor con return, sino que suma en exclusión mutua a una variable compartida entre hebras de un proceso en el heap – exactamente igual que en el ejercicio 2 y 3. Después se hace MPI_Reduce para sumar el valor que han computado todos los procesos en uno.

Solamente hubo una cosa a plantearse: cómo repartir los intervalos entre procesos y luego entre hebras. Reparto de intervalos entre procesos he hecho exactamente igual que en la práctica 1, luego calculo la diferencia del intervalo superior e inferior por cada proceso y llamo a la misma función para que ahora las hebras obtengan sus intervalos.

```
void getIntervals(size_t parallel_ent, int arr_size, int id, int *low_int, int *upp_int)
{
    size_t remainder = arr_size % parallel_ent,
           per_ent = arr_size / parallel_ent,
           lsize = per_ent + (id < remainder);

    int mod = (id > remainder) ? remainder : id;
    *low_int = per_ent*id + mod;
    *upp_int = *low_int + lsize;
}
```

Ahora para que cada hebra tenga los límites bien establecidos (respecto al número de intervalos global entre todos los procesos) falta sumarles el intervalos inicial del proceso.

```
for (int i = 0; i < NUM_THREADS; i++){
    info[i].intervals = intervals;
    info[i].lock = &lock;
    info[i].part_pi = part_pi;
    getIntervals(NUM_THREADS, interval_size, i, &info[i].l_interval, &info[i].u_interval);
    info[i].l_interval += l_limit;
    info[i].u_interval += l_limit;
    pthread_create(&threads[i], NULL, (void *) piRectangles, (void *) &info[i]);
}
```

Medición de tiempos

En la práctica se pide usar hasta 16 hilos y para ello se propone usar atcgrid, pero al ejecutar lscpu veo que no me va a hacer falta (bien!) :

```
CPU(s):                          16
On-line CPU(s) list:             0-15
Thread(s) per core:              2
```

Voy a hacer las pruebas con 2 hebras por proceso para el número de intervalos máximo que puede caber en el int (2147483647). Realmente no hay mucha diferencia entre usar procesos y hebras en linux porque TCB y PCB son lo mismo en linux, así que podría usar cualquier combinación de hebras por proceso.

Procesadores	Tiempo
2	3.415890
4	1.777749
8	1.107258
16	0.697324