ARQO

Memoria Práctica 4

Roberto Martín Alonso

Diego Forte Jara

Pareja 11

ÍNDICE

Aclaraciones previas:

La práctica actual se va a desarrollar haciendo uso del subsistema de Windows para Linux (WSL), con las siguientes especificaciones:

```
(base) tugfa123@AmogOS:~$ lscpu
Architecture:            x86_64
  CPU op-mode(s):        32-bit, 64-bit
  Address sizes:         48 bits physical, 48 bits virtual
  Byte Order:            Little Endian
CPU(s):                  16
  On-line CPU(s) list:   0-15
Vendor ID:               AuthenticAMD
  Model name:            AMD Ryzen 7 3700X 8-Core Processor
    CPU family:          23
    Model:               113
    Thread(s) per core:  2
    Core(s) per socket:  8
    Socket(s):           1
    Stepping:            0
    BogoMIPS:            8400.03
    Flags:               fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse ss
                         e2 ht syscall nx mmxext fxsr_opt pdpe1gb rdtscp lm constant_tsc rep_good nopl tsc_reliable nons
                         top_tsc cpuid extd_apicid pni pclmulqdq ssse3 fma cx16 sse4_1 sse4_2 movbe popcnt aes xsave avx
                          f16c rdrand hypervisor lahf_lm cmp_legacy svm cr8_legacy abm sse4a misalignsse 3dnowprefetch o
                         svw topoext perfctr_core ssbd ibpb stibp vmmcall fsgsbase bmi1 avx2 smep bmi2 rdseed adx smap c
                         lflushopt clwb sha_ni xsaveopt xsavec xgetbv1 clzero xsaveerptr arat npt nrip_save tsc_scale vm
                         cb_clean flushbyasid decodeassists pausefilter pfthreshold v_vmsave_vmload umip rdpid
Virtualization features:
  Virtualization:        AMD-V
  Hypervisor vendor:     Microsoft
  Virtualization type:   full
Caches (sum of all):
  L1d:                   256 KiB (8 instances)
  L1i:                   256 KiB (8 instances)
  L2:                    4 MiB (8 instances)
  L3:                    16 MiB (1 instance)
Vulnerabilities:
  Gather data sampling:  Not affected
  Itlb multihit:         Not affected
  L1tf:                  Not affected
  Mds:                   Not affected
  Meltdown:              Not affected
  Mmio stale data:       Not affected
  Retbleed:              Mitigation; untrained return thunk; SMT enabled with STIBP protection
  Spec rstack overflow:  Mitigation; safe RET
  Spec store bypass:     Mitigation; Speculative Store Bypass disabled via prctl and seccomp
  Spectre v1:            Mitigation; usercopy/swapgs barriers and __user pointer sanitization
  Spectre v2:            Mitigation; Retpolines, IBPB conditional, STIBP always-on, RSB filling, PBRSB-eIBRS Not affecte
                         d
  Srbds:                 Not affected
  Tsx async abort:       Not affected
```

Ejercicio 0: Información sobre la topología del sistema

Tras ejecutar el comando “cat /proc/cpuinfo” por terminal y volcarlo al fichero de texto “cpuinfoE0.txt” se obtienen los siguientes resultados:

```
processor   : 0
vendor_id   : AuthenticAMD
cpu family  : 23
model       : 113
model name  : AMD Ryzen 7 3700X 8-Core Processor
stepping    : 0
microcode   : 0xffffffff
cpu MHz     : 4200.016
cache size  : 512 KB
physical id : 0
siblings    : 16
core id     : 0
cpu cores   : 8
```

Aquí se puede apreciar que el número de cores físicos es 8 (campo cpu cores), el número de cores lógicos es 16 (campo siblings) y su frecuencia es 4200.16MHz (campo cpu MHz). Podemos concluir que el hyperthreading está activado en el sistema debido a que el número de cores lógicos es mayor que el de cores físicos.

NOTA: En la carpeta E0 de la entrega se adjunta el fichero “cpuinfoE0.txt” donde se puede ver toda la información completa tras ejecutar el comando “cat /proc/cpuinfo”

Ejercicio 1: Programas básicos de OpenMP

1.1-

El sistema en el que se realiza la práctica dispone de 16 cores lógicos (16 hilos máximo). Se prueba a lanzar el programa “omp1.c” con 32 hilos y el programa funciona, por lo que es posible lanzar más hilos que cores lógicos.

Con respecto a si tiene sentido hacerlo, en general no debido a que el sistema operativo debe gestionar el cambio de contexto entre hilos, lo que produce sobrecarga en el sistema, pero en situaciones donde se produzcan un gran número de bloqueos si puede resultar beneficioso debido a que mientras que haya hilos bloqueados puede haber otros trabajando.

1.2-

El número de hilos a utilizar depende de la tarea que se quiera realizar. Se podría comenzar usando el número máximo de hilos del sistema (6 para el ordenador del laboratorio y 16 hilos para el ordenador propio) y posteriormente experimentar con más hilos y medir el rendimiento para ver si hay mejoras.

1.3-

Se modifica el programa “omp1.c” tal y como se pide en el enunciado y se ejecuta. A partir de esta ejecución se deduce que la prioridad del número de hilos es, de forma descendente:

nº hilos cláusula -> nº hilos con función -> nº hilos variable de entorno

NOTA: Si no se indica número de hilos mediante cláusula o función, OpenMP utiliza el valor de la variable de entorno OMP_NUM_THREADS, por lo que para que se ejecute con el valor de dicha variable de entorno no se le asigna número de hilos en la ejecución con el valor de la variable de entorno

1.4-

Se ejecuta el programa “omp2”:

```
(base) tugfa123@AmogOS:~/Practicas/Practicas_ARQO/P4/src$ ./omp2
Inicio: a = 1,    b = 2,   c = 3
        &a = 0x7ffc7df4c764,x   &b = 0x7ffc7df4c768,    &c = 0x7ffc7df4c76c

[Hilo 0]-1: a = 2135301520,      b = 2,   c = 3
[Hilo 0]         &a = 0x7ffc7df4c700,    &b = 0x7ffc7df4c768,    &c = 0x7ffc7df4c6fc
[Hilo 0]-2: a = 237629711,       b = 4,   c = 237629699
[Hilo 3]-1: a = 0,       b = 2,   c = 3
[Hilo 3]         &a = 0x7fcd26f95d80,    &b = 0x7ffc7df4c768,    &c = 0x7fcd26f95d7c
[Hilo 3]-2: a = 21,      b = 6,   c = 3
[Hilo 1]-1: a = 0,       b = 2,   c = 3
[Hilo 1]         &a = 0x7fcd27f97d80,    &b = 0x7ffc7df4c768,    &c = 0x7fcd27f97d7c
[Hilo 1]-2: a = 27,      b = 8,   c = 3
[Hilo 2]-1: a = 0,       b = 2,   c = 3
[Hilo 2]         &a = 0x7fcd27796d80,    &b = 0x7ffc7df4c768,    &c = 0x7fcd27796d7c
[Hilo 2]-2: a = 33,      b = 10,          c = 3

Fin: a = 1,      b = 10,          c = 3
        &a = 0x7ffc7df4c764,    &b = 0x7ffc7df4c768,    &c = 0x7ffc7df4c76c
```

Cuando se declara una variable privada en OpenMP ésta no se comparte con el resto de hilos, como es el caso de “a”, donde cada hilo tiene dicha variable (sin inicializar) pero cada uno tiene un valor distinto para ella.

1.5-

Cuando comienza la ejecución paralela, se hace una copia independiente para cada hilo de las variables privadas. Dichas copias están sin inicializar, por lo que contienen valores indefinidos.

1.6-

Cuando finaliza la región paralela el valor de esta variable privada no se mantiene. Es similar a cuando se declaran variables locales en una función y esta retorna.

1.7-

En el caso de las variables públicas si conservan su valor al finalizar la región paralela debido a que forman parte de la memoria original que se ha compartido entre los hilos al iniciar la región paralela. Es similar a cuando se le pasa por referencia una variable a una función y esta retorna

Ejercicio 2: Paralelizar el producto escalar

2.1-

El resultado del programa “pescalar_serie” es el tamaño “M” del vector que se le pase como argumento a la función “generateVectorOne”, que crea un vector de tamaño “M” y lo inicializa a “1s”. Es decir, si los vectores son de tamaño 1000, el resultado del producto escalar del programa será 1000.

2.2-

El resultado que arroja el programa “pescalar_par1” no es correcto. Esto es debido a que como la variable “sum” es compartida, todos los hilos acceden a ella y se producen condiciones de carrera, por lo que en cada ejecución muestra un resultado distinto.

2.3-

Se crea el programa “pescalar_par2” para obtener el resultado correcto. Esta condición de carrera puede resolverse mediante ambas directivas, los cambios realizados son los siguientes:

```
    #pragma omp parallel for
for(k=0;k<M;k++)
{
    // #pragma omp critical
    // {
    //  sum = sum + A[k]*B[k];
    // }

    #pragma omp atomic
    sum = sum + A[k]*B[k];
}
```

En este caso la solución elegida es el pragma atomic, ya que es solo una línea la que se quiere proteger. En caso de que hubiese más instrucciones susceptibles a sufrir condiciones de carrera se usaría el pragma critical para proteger todo el bloque entre llaves.

2.4-

Se crea el programa “pescalar_par3” para obtener el resultado correcto mediante el uso del pragma “omp parallel for reduction”. Los cambios realizados son:

```
    #pragma omp parallel for reduction (+ : sum)
for(k=0;k<M;k++)
{
    sum = sum + A[k]*B[k];
}
```

En este caso la opción elegida sería este último pragma ya que es el que tiene mayor afinidad con el problema que queremos resolver.

2.5 (2.6)-

Tras realizar varias ejecuciones del programa del producto escalar con distintos tamaños de vectores se encuentra un posible valor umbral de vector, el cual es 2750000. Para considerarlo válido se ejecuta el programa del producto escalar para tamaños de vector:

$$\text{ceil}(0.8 \times 2750000) = 2200000,\ \text{ceil}(1.2 \times 2750000) = 3300000$$

```
(base) tugfa123@AmogOS:~/Practicas/Practicas_ARQO/P4/src/exe$ ./pescalar_par4 2200000
Tamanio del vector = 2200000
EJECUCION EN SERIE
Resultado: 2200000.000000
Tiempo: 0.005990
(base) tugfa123@AmogOS:~/Practicas/Practicas_ARQO/P4/src/exe$ ./pescalar_par4 2200000 p
Tamanio del vector = 2200000
PARALELIZACION: Se han lanzado 16 hilos.
Resultado: 2200000.000000
Tiempo: 0.006192
(base) tugfa123@AmogOS:~/Practicas/Practicas_ARQO/P4/src/exe$ ./pescalar_par4 3300000
Tamanio del vector = 3300000
EJECUCION EN SERIE
Resultado: 3300000.000000
Tiempo: 0.008815
(base) tugfa123@AmogOS:~/Practicas/Practicas_ARQO/P4/src/exe$ ./pescalar_par4 3300000 p
Tamanio del vector = 3300000
PARALELIZACION: Se han lanzado 16 hilos.
Resultado: 3300000.000000
Tiempo: 0.008458
```

Se comprueba que para ceil(0.8 x 2750000) el tiempo del programa en serie es menor que el del programa en paralelo y para ceil(1.2 x 2750000) = 3300000 el tiempo del programa en paralelo es menor que el del programa en serie.

A continuación, se comprueba la estimación del umbral con tamaños por encima y por debajo del óptimo:

```
(base) tugfa123@AmogOS:~/Practicas/Practicas_ARQO/P4/src/exe$ ./pescalar_par4 4000000
Tamanio del vector = 4000000
EJECUCION EN SERIE
Resultado: 4000000.000000
Tiempo: 0.010709
(base) tugfa123@AmogOS:~/Practicas/Practicas_ARQO/P4/src/exe$ ./pescalar_par4 4000000 p
Tamanio del vector = 4000000
PARALELIZACION: Se han lanzado 16 hilos.
Resultado: 4000000.000000
Tiempo: 0.007107
(base) tugfa123@AmogOS:~/Practicas/Practicas_ARQO/P4/src/exe$ ./pescalar_par4 1500000
Tamanio del vector = 1500000
EJECUCION EN SERIE
Resultado: 1500000.000000
Tiempo: 0.004190
(base) tugfa123@AmogOS:~/Practicas/Practicas_ARQO/P4/src/exe$ ./pescalar_par4 1500000 p
Tamanio del vector = 1500000
PARALELIZACION: Se han lanzado 16 hilos.
Resultado: 1500000.000000
Tiempo: 0.007180
```

Como puede apreciarse, por encima del umbral el tiempo de la ejecución paralela es menor al tiempo de la ejecución en serie y por debajo del umbral el tiempo de la ejecución en serie es menor al tiempo de la ejecución en paralelo.

Ejercicio 3: Paralelizar el producto escalar

Tiempos de ejecución (s)				
Versión\nhilos	1	2	3	4
Serie	45,098091	45,098091	45,098091	45,098091
Paralela-bucle1	47,703853	80,219340	77,515044	84,054902
Paralela-bucle2	48,639896	24,923234	17,541910	14,180161
Paralela-bucle3	45,712199	21,847603	14,316261	10,480383

Aceleración				
Version\nhilos	1	2	3	4
Serie	1			
Paralela-bucle1	0,945376278	0,56218477	0,58179792	0,53653136
Paralela-bucle2	0,92718313	1,8094799	2,57087689	3,18036523
Paralela-bucle3	0,986565774	2,06421231	3,15013054	4,3030957

Tablas obtenidas con una dimensión de matriz N = 1800.

Los datos de tiempos han sido obtenidos mediante la ejecución de un script creado para este propósito y la aceleración ha sido calculada en una hoja de Excel. Ambos ficheros (E3.xlsx y ej3.sh) se adjuntan en los ficheros entregados

3.1-

La versión que obtiene el peor rendimiento es la paralela del bucle 1 con 4 hilos. Esto se debe a que consume más tiempo en gestionar todo lo referente a los hilos que en computar los resultados. Como se abren 4 hilos en el bucle interno, en total los hilos que se abren son 1000x1000x4, es decir 4000000 hilos.

La versión que obtiene el mejor rendimiento es la paralela del bucle 3 con 4 hilos. Esto se debe a que se abren y se cierran menos hilos durante toda la ejecución del programa, por lo que se pierde menos tiempo en la gestión de los hilos. Como los hilos se abren en el bucle externo, el total de hilos abiertos es 4.

3.2-

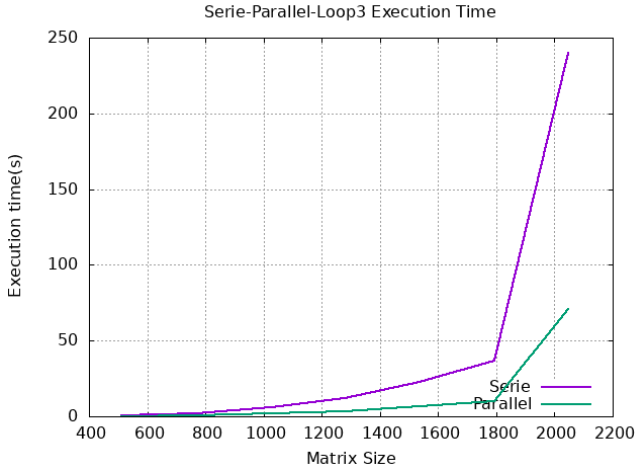
Depende del algoritmo a implementas será preferible la paralelización de grano fino o de grano grueso.

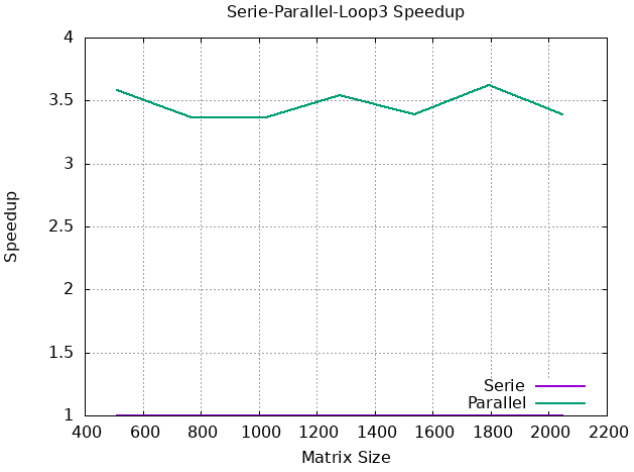
Si el algoritmo no necesita demasiada coordinación entre hilos (como es el caso de la multiplicación de matrices) es preferible utilizar paralelismo de grano grueso, ya que así se reduce el coste de lanzar hilos (overhead) y cada hilo realiza una cantidad significativa de trabajo.

En cambio, si se necesita coordinación constante entre hilos (como en el caso de la suma de elementos de dos arrays en uno nuevo) será preferible utilizar paralelismo de grano fino, aumentando el coste de lanzar hilos.

3.3-

Se crea un script para automatizar la toma de tiempos (ej3.3.sh) y crear las gráficas con gnuplot, obteniéndose los siguientes resultados:

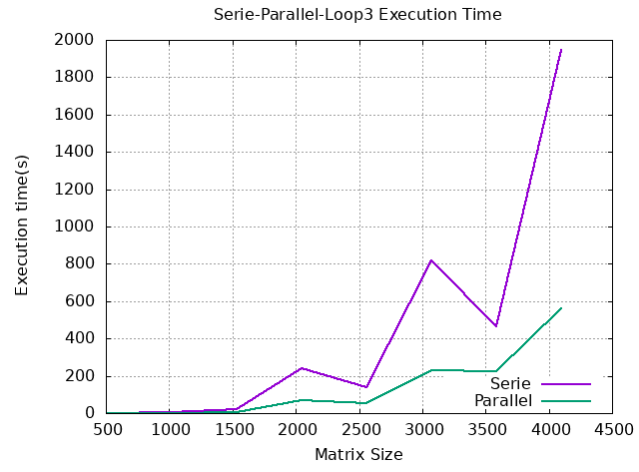


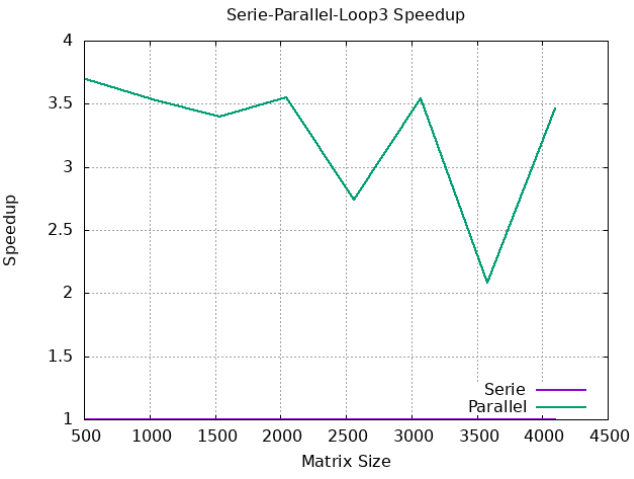


En la gráfica del tiempo de ejecución se observa como el tiempo de ejecución del programa en serie aumenta mas rápido que el tiempo de ejecución del programa paralelo a medida que se va aumentando el tamaño N de la matriz.

En la gráfica de la aceleración se observa como la aceleración del programa paralelo con respecto al programa en serie está estable a medida que se aumenta el tamaño N de la matriz.

Como no se obtiene una gráfica cuya aceleración disminuya con el incremento del tamaño de la matriz se repiten las mediciones con tamaños mayores de N (N inicial = 512, N final = 4096 y pasos de 512), resultando:





Ahora, a pesar de los picos que presenta la nueva gráfica de aceleración, se puede apreciar como esta tiene una tendencia descendente.

Ejercicio 4: Pérdida de rendimiento por el falso compartir (False sharing) en OpenMP

4.1-

En la versión del programa serie, el número de rectángulos que se dan son 100000000, por lo que el valor h (ancho del rectángulo) es de 1/100000000

4.2-

Se ejecuta el programa en serie y todos los paralelos mediante el script ej4.sh, obteniéndose los siguientes resultados:

```
pi_serie
Resultado pi: 3.141593
Tiempo 0.277151
pi_par1
Numero de cores del equipo: 16
Resultado pi: 3.141593
Tiempo 0.254153

pi_par2
Numero de cores del equipo: 16
Resultado pi: 3.141593
Tiempo 0.251406

pi_par3
Numero de cores del equipo: 16
Double size: 8 bytes
Cache line size: 64 bytes => padding: 8 elementos
Resultado pi: 3.141593
Tiempo 0.040708

pi_par4
Numero de cores del equipo: 16
Resultado pi: 3.141593
Tiempo 0.028056

pi_par5
Resultado pi: 3.141593
Tiempo 0.033620

pi_par6
Numero de cores del equipo: 16
Resultado pi: 3.141593
Tiempo 0.233791

pi_par7
Resultado pi: 3.141593
Tiempo 0.027823
```

Pasando los datos a tabla y calculando su aceleración, se obtiene:

Tiempos de ejecución (s)								
Datos\Versión	serie	pi_par1	pi_par2	pi_par3	pi_par4	pi_par5	pi_par6	pi_par7
Tiempos (s)	0,277151	0,254153	0,251406	0,040708	0,028056	0,033620	0,233791	0,027823
Aceleración	1,000000	1,090489	1,102404	6,808269	9,878493	8,243635	1,185465	9,961219

Como puede verse en la salida de las ejecuciones, los resultados de todos los programas son correctos

4.3-

No tiene sentido declarar la variable sum como privada ya que esta no se modifica, sino que se modifican los datos a los que apunta el puntero. Si una variable de tipo puntero se declara como privada, esta se encuentra disponible en cada hilo pero no está inicializada, por lo que se debería usar firstprivate para obtener su valor original y que sea privada a cada hilo.

4.4-

En pi_par5 los hilos van sumando sus resultados a la variable pi, la cual está protegida la directiva critical.

En pi_par1 se crea un array sum que tiene nº de elementos = nº de cores del equipo donde se ejecuta y cada hilo va sumando a su elemento del array denotado por su numero de hilo y después de terminar la región paralela se suman todos los elementos de ese array a la variable pi.

En pi_par3 es parecido a pi_par1 pero el array sum que tiene nº de elementos = nº de cores del equipo * nº de doubles que caben en el tamaño de la línea de caché (padding) y al cual cada hilo accede a sum[tid *padsz]

El false sharing es una situación que se produce cuando los hilos acceden a distintas variables pero estas se encuentran en un mismo bloque de caché, por lo que cuando un hilo actualiza una variable en el bloque caché y otro quiere acceder a otra variable distinta pero en el mismo bloque, este bloque debe escribirse en memoria antes de utilizarse.

En pi_par3 se obtiene el tamaño de la línea de caché para crear un array donde cada elemento que actualiza cada hilo esté en un bloque caché distinto, por lo que no se produce false sharing.

4.5-

Al utilizar la directiva critical la región de código afectada solo puede ser accedida por un hilo a la vez, evitando así las condiciones de carrera.

Al restringir el acceso, el tiempo de ejecución aumenta ya que cada hilo tiene que esperar a que se libere la zona crítica para poder acceder a ella.

4.6-

Al ejecutarse pi_par6 se observa que es de las ejecuciones que tarda mas tiempo en comparación de pi_par3, por ejemplo. Esto se debe a que por la forma en la que está diseñado el array en pi_par6 sufre los efectos de false sharing, aumentando el tiempo de ejecución y en cambio pi_par3 no se ve afectado.

4.7-

La versión óptima según los tiempos de ejecución y memoria utilizada es pi_par7 debido a que es el programa con menor tiempo de ejecución, no necesita reservar memoria adicional gracias al uso de la directiva reduction y el uso esta directiva se ajusta mas al problema a resolver.

Ejercicio 5: Optimización de programas de cálculo

5.0-

El programa entregado recibe de argumentos una lista de nombres de imágenes, y mediante un bucle for detecta los bordes de la imagen primero pasándola a escala de grises, después obteniendo los bordes de la imagen con ruido y por último aplicándola una reducción de ruido para obtener la imagen final

5.1-

El bucle más externo puede ser el óptimo para ser paralelizado si se quiere hacer una paralelización de grano grueso, es decir, si lo que se busca es que cada hilo trabaje en una imagen distinta. Si solo se va a trabajar con una imagen o un grupo muy reducido de imágenes lo óptimo sería hacer una paralelización de grano fino donde los hilos trabajan en una misma imagen.

5.1a-

Se ejecuta el programa con más hilos que imágenes a procesar mediante la directiva omp parallel for, obteniéndose:

```
(base) tugfa123@AmogOS:~/Practicas/Practicas_ARQO/P4/src/E5$ ./ej5.sh
make: Entering directory '/home/tugfa123/Practicas/Practicas_ARQO/P4/src'
rm -f exe/*
make: Leaving directory '/home/tugfa123/Practicas/Practicas_ARQO/P4/src'
make: Entering directory '/home/tugfa123/Practicas/Practicas_ARQO/P4/src'
gcc -g -Wall -D_GNU_SOURCE -fopenmp -std=gnu99 -Iutils/  -o exe/edgeDetector E5/edgeDetector.c -lgomp -lm
make: Leaving directory '/home/tugfa123/Practicas/Practicas_ARQO/P4/src'
edgeDetector
Numero de cores del equipo: 16
Soy el hilo 0
[info] Processing ../E5/img/8k.jpg
Soy el hilo 1
[info] Processing ../E5/img/SD.jpg
[info] ../E5/img/SD: width=640, height=360, nchannels=3
[info] Using gaussian denoising...
Tiempo: 0.029577
[info] ../E5/img/8k: width=7680, height=4320, nchannels=3
[info] Using gaussian denoising...
Tiempo: 5.398147
```

Como se puede observar, aunque se establece que se lancen 16 hilos, como solo se le pasan dos imágenes como argumento se abren solo 2 hilos, uno para cada imagen en lugar de los 16.

5.1b-

Para procesar una única imagen de gran tamaño esta opción no es la adecuada debido a que solo habría un hilo trabajando en dicha imagen. Cada hilo consume AxBxCx4, donde A el número de píxeles de ancho de la imagen, B es el número de píxeles de alto de la imagen, C es el tamaño en bytes de cada pixel (8 bits por canal = 24 bits para RBG = 3 bytes) y el 4 final debido a que se reserva memoria para la imagen original, para la de escala de grises, para la de detección de bordes con ruido y para la de detección de bordes sin ruido.

5.2-

5.2a-

Se encuentran los bucles con acceso subóptimo a los datos y se corrigen en el nuevo fichero edgeDetectorParallel:

```
// RGB to grey scale
int r, g, b;
// for (int i = 0; i < width; i++) //OLD
for (int j = 0; j < height; j++)
{
    // for (int j = 0; j < height; j++) //OLD
    for (int i = 0; i < width; i++)
    {
        getRGB(rgb_image, width, height, 4, i, j, &r, &g, &b);
        grey_image[j * width + i] = (int)(0.2989 * r + 0.5870 * g + 0.1140 * b);
    }
}
```

```
#define PIXEL_GREY(x, y) (grey_image[(x) + (y)*width])
        // for (int i = 1; i < width - 1; i++) //OLD
        for (int j = 1; j < height - 1; j++)
        {
            // for (int j = 1; j < height - 1; j++) //OLD
            for (int i = 1; i < width - 1; i++)
            {
                int x = i - 1;
                int y = j - 1;
                float a = (PIXEL_GREY(i - 1, j - 1) + PIXEL_GREY(i - 1, j) * 2 + PIXEL_GREY(i - 1, j + 1) -
                           (PIXEL_GREY(i + 1, j - 1) + PIXEL_GREY(i + 1, j) * 2 + PIXEL_GREY(i + 1, j + 1)));
                float b = (PIXEL_GREY(i - 1, j - 1) + PIXEL_GREY(i, j - 1) * 2 + PIXEL_GREY(i + 1, j - 1) -
                           (PIXEL_GREY(i - 1, j + 1) + PIXEL_GREY(i - 1, j + 1) * 2 + PIXEL_GREY(i - 1, j + 1)));

                edges[x + y * width_edges] = sqrt(a * a + b * b);
            }
        }
```

```
printf("[info] Using median denoising...\n");
// for (int i = radius; i < width_edges - radius; i++) //OLD
for (int j = radius; j < height_edges - radius; j++)
{
    // for (int j = radius; j < height_edges - radius; j++) //OLD
    for (int i = radius; i < width_edges - radius; i++)

    {
        y = j - radius;
        x = i - radius;
        k = 0;

        for (int p1 = i - radius; p1 <= i + radius; p1++)
        {
            for (int p2 = j - radius; p2 <= j + radius; p2++)
            {
                array[k++] = PIXEL_EDGES(p1, p2);
            }
        }
        qsort(array, (2 * radius + 1) * (2 * radius + 1), 1, comparate);
        edges_denoised[x + y * width_denoised] = array[(2 * radius + 1) * (2 * radius + 1) / 2]>50?255:0;
    }
}
```

```
printf("[info] Using gaussian denoising...\n");
float* kernel = gaussian_kernel(2*radius+1, 1.0);
double sum = 0;
// for (int i = radius; i < width_edges - radius; i++) //OLD
for (int j = radius; j < height_edges - radius; j++)
{
    // for (int j = radius; j < height_edges - radius; j++) //OLD
    for (int i = radius; i < width_edges - radius; i++)
    {
        x = i - radius;
        y = j - radius;
        sum = 0;
        for (int p1 = 0; p1 <= 2 * radius; p1++)
        {
            for (int p2 = 0; p2 <= 2 * radius; p2++)
            {
                if (kernel[p1+p2*(2*radius+1)]>1)
                    printf("%f, %d, %d\n", kernel[p1+p2*(2*radius+1)], p1, p2);
                sum += kernel[p1+p2*(2*radius+1)] * PIXEL_EDGES(i-radius+p1, j-radius+p2);
            }
        }
        edges_denoised[x + y * width_denoised] = sum>50?255:0;
    }
}
```

Posteriormente se ejecuta y se comprueba que las imágenes obtenidas son idénticas a las que se producían con el bucle original. También se observa que para imágenes de gran tamaño (la imagen 8K) los tiempos de ejecución se ven reducidos.

5.2b-

El orden de acceso a los datos era subóptimo debido a que en lugar de acceder a los datos del array secuencialmente (posición 0, 1, 2…) se accedían de la forma “posición 0*width, 1*width, 2*width…” lo que provocaba que no se aprovechase el principio de localidad espacial de los datos y aumentase la cantidad de fallos de caché, lo que acababa resultando en tiempos de ejecución mayores.

5.3-

5.3a-

Se realizan las paralelizaciones en los bucles mas externos de edgeDetectorParallel haciendo privadas las variables necesarias y usando directivas como reduction donde se requiera quedando los bucles:

```
int r, g, b;
// for (int i = 0; i < width; i++) //OLD
int j;
#pragma omp parallel for private (r, g, b)
for (j = 0; j < height; j++)
{
    // for (int j = 0; j < height; j++) //OLD
    for (int i = 0; i < width; i ++)
    {
        getRGB(rgb_image, width, height, 4, i, j, &r, &g, &b);
        grey_image[j * width + i] = (int)(0.2989 * r + 0.5870 * g + 0.1140 * b);
    }
}
```

```
        // Sobel edge detection
#define PIXEL_GREY(x, y) (grey_image[(x) + (y)*width])
        #pragma omp parallel for
        // for (int i = 1; i < width - 1; i++) //OLD
        for (j = 1; j < height - 1; j++)
        {
            // for (int j = 1; j < height - 1; j++) //OLD
            for (int i = 1; i < width - 1; i++)
            {
                int x = i - 1;
                int y = j - 1;
                float a = (PIXEL_GREY(i - 1, j - 1) + PIXEL_GREY(i - 1, j) * 2 + PIXEL_GREY(i - 1, j + 1) -
                           (PIXEL_GREY(i + 1, j - 1) + PIXEL_GREY(i + 1, j) * 2 + PIXEL_GREY(i + 1, j + 1)));
                float b = (PIXEL_GREY(i - 1, j - 1) + PIXEL_GREY(i, j - 1) * 2 + PIXEL_GREY(i + 1, j - 1) -
                           (PIXEL_GREY(i - 1, j + 1) + PIXEL_GREY(i - 1, j + 1) * 2 + PIXEL_GREY(i - 1, j + 1)));

                edges[x + y * width_edges] = sqrt(a * a + b * b);
            }
        }
```

```
printf("[info] Using median denoising...\n");
// for (int i = radius; i < width_edges - radius; i++) //OLD
#pragma omp parallel for private (x, y, k)
for (int j = radius; j < height_edges - radius; j++)
{
    // for (int j = radius; j < height_edges - radius; j++) //OLD
    for (int i = radius; i < width_edges - radius; i++)

    {
        y = j - radius;
        x = i - radius;
        k = 0;

        for (int p1 = i - radius; p1 <= i + radius; p1++)
        {
            for (int p2 = j - radius; p2 <= j + radius; p2++)
            {
                array[k++] = PIXEL_EDGES(p1, p2);
            }
        }
        qsort(array, (2 * radius + 1) * (2 * radius + 1), 1, comparate);
        edges_denoised[x + y * width_denoised] = array[(2 * radius + 1) * (2 * radius + 1) / 2]>50?255:0;
    }
}
```

```
printf("[info] Using gaussian denoising...\n");
float* kernel = gaussian_kernel(2*radius+1, 1.0);
double sum = 0;
// for (int i = radius; i < width_edges - radius; i++) //OLD
#pragma omp parallel for reduction (+ : sum) private (x, y)
for (int j = radius; j < height_edges - radius; j++)
{
    // for (int j = radius; j < height_edges - radius; j++) //OLD
    for (int i = radius; i < width_edges - radius; i++)
    {
        x = i - radius;
        y = j - radius;
        sum = 0;
        for (int p1 = 0; p1 <= 2 * radius; p1++)
        {
            for (int p2 = 0; p2 <= 2 * radius; p2++)
            {
                if (kernel[p1+p2*(2*radius+1)]>1)
                    printf("%f, %d, %d\n", kernel[p1+p2*(2*radius+1)], p1, p2);
                sum += kernel[p1+p2*(2*radius+1)] * PIXEL_EDGES(i-radius+p1, j-radius+p2);
            }
        }
        edges_denoised[x + y * width_denoised] = sum>50?255:0;
    }
}
```

5.3b-

Los bucles que deberían tener mejor rendimiento al ser paralelizados deberían ser los externos, debido a que como se ha comprobado anteriormente en la práctica, se abren menos hilos en total y trabajan de forma mas eficiente, haciendo que el tiempo de ejecución se vea reducido ya que se ahorra tiempo al gestionar menos hilos.

5.4-

Para este apartado se crea un script (ej5.4.dat) que automatiza la tarea de la toma de tiempos y los exporta a un fichero .dat, posteriormente estos datos se pasan a una hoja de Excel y se obtienen los siguientes resultados:

	SD	HD	FHD	4K	8K
Tiempo serie	0,028695818	0,109735636	0,246258182	1,187596909	4,90861709
Tiempo paralelo	0,010671182	0,023896364	0,035755455	0,123505909	0,479786
Aceleración	2,68909467	4,592147916	6,887289932	9,615709229	10,2308469
FPS Serie	34,84828335	9,112809965	4,060778784	0,842036547	0,20372337
FPS Paralelo	93,710333	41,84737129	27,96776083	8,096778592	2,08426257

Como se puede observar, tanto 4K como 8K no llegarían a la tasa de FPS necesaria para un vídeo. FHD se queda en aproximadamente 28 FPS, que para las películas de cine tradicionales es totalmente válido ya que éstas van a 24 FPS.