



## Cátedra de Sistemas Operativos II

### Trabajo Práctico N° II

Piñero, Tomás Santiago  
14 de Mayo de 2020



# Índice

<b>Índice</b>	<b>1</b>
<b>1. Introducción</b>	<b>2</b>
1.1. Objetivo . . . . .	2
1.2. Definiciones, Acrónimos y Abreviaturas . . . . .	2
<b>2. Descripción general</b>	<b>2</b>
2.1. Restricciones . . . . .	2
2.2. Esquema del proyecto . . . . .	2
2.3. Requisitos futuros . . . . .	3
<b>3. Diseño de solución</b>	<b>3</b>
3.1. Profilers . . . . .	3
3.1.1. perf . . . . .	3
3.1.2. gprof . . . . .	4
3.1.3. Valgrind . . . . .	5
3.1.4. V-Tune . . . . .	5
3.2. Makefile . . . . .	6
<b>4. Implementación y resultados</b>	<b>6</b>
4.1. Aplicación de los filtros . . . . .	6
4.1.1. Filtro lineal . . . . .	7
4.1.2. Filtro de desenfoque . . . . .	7
4.2. Resultados . . . . .	8
4.2.1. Filtrado . . . . .	8
4.2.2. Medición de tiempos . . . . .	10
4.2.3. Análisis de los tiempos . . . . .	13
<b>5. Conclusiones</b>	<b>13</b>
<b>Referencias</b>	<b>13</b>



# 1. Introducción

## 1.1. Objetivo

El objetivo del trabajo es diseñar una solución que utilice el paradigma de memoria compartida mediante *OpenMP* para realizar el procesamiento de una imagen *BMP*.

## 1.2. Definiciones, Acrónimos y Abreviaturas

- **BMP:** *Bitmap*. Formato de imagen capaz de almacenar imágenes bidimensionales tanto en color como monocromático.[1]
- **API:** *Application Programming Interface*. Es un conjunto de funciones ofrecidas para ser utilizadas por otro software.
- **Nodo:** computadora individual que forma parte de un *cluster*.
- **Cluster:** conjunto de nodos unidos entre sí que se comportan como una única computadora.
- **OpenMP:** *Open Multi-Processing*. Es una API para programación multiprocesos de memoria compartida.[2]

# 2. Descripción general

## 2.1. Restricciones

A las restricciones dadas (ver `Enunciado.pdf`) se le agregaron las siguientes:

1. La imagen a filtrar debe estar ubicada en la carpeta `imgs`.
2. Para ejecutar el programa se debe ingresar desde la carpeta `bin`:

```
./main 'nombre_imagen' 'radio' 'cantidad_hilos'
```

## 2.2. Esquema del proyecto

El proyecto está dividido en las siguientes carpetas:

- **bin:** contiene el archivo ejecutable.
- **inc:** contiene los *headers* utilizados por el código fuente:
  - `simple bmp.h`: header para la carga, lectura y escritura de las imágenes *BMP*.
- **imgs:** contiene la imagen a la que se aplican los filtros y el resultado de su aplicación.



- **src:** contiene los códigos fuente.
  - *main.c*: realiza la aplicación de los filtros a la imagen de entrada.
  - *simple\_bmp.c*: contiene la implementación de las funciones de *simple\_bmp.h*.

## 2.3. Requisitos futuros

- Utilizar alguno de los métodos conocidos para el tratamiento de los bordes de la imagen.
- Soportar más de dos filtros de imagen.

## 3. Diseño de solución

Como se mencionó en la sección anterior, se organizó el proyecto en varias carpetas (ver [2.2](#)).

Luego, se decidió por separar la aplicación de los filtros en dos funciones distintas para una mejor legibilidad del código. Por lo tanto, se creará una función que determine la posición del píxel respecto al centro de la imagen y en función del resultado aplique el filtro correspondiente.

En el caso de que el píxel corresponda a alguno de los bordes de la imagen, no será modificado.

Los parámetros correspondientes a los filtros se establecen en el momento de compilación, para evitar una pérdida de tiempo en la obtención de los parámetros en el momento de la ejecución. Los parámetros con los que se realizaron las ejecuciones son los siguientes:

- Brillo (*L*): 1.8
- Contraste (*K*): 15
- Radio (*R*): 500 [px]
- Tamaño de kernel (*SIZE\_K*): 45

### 3.1. Profilers

Un *profiler* es un *software* que realiza un análisis sobre el comportamiento de un determinado programa mediante la información obtenida durante la ejecución del mismo. Este tipo de herramienta es utilizado para depurar y optimizar los algoritmos utilizados en el programa.

A continuación se muestran las herramientas de *profiling* para analizar la *performance* del programa realizado.

#### 3.1.1. perf

Es una herramienta disponible para *Linux* a partir de la versión de kernel 2.6.31 (2009).

Soporta contadores de rendimiento tanto de *hardware* como *software*, puntos de rastreo y sondas dinámicas.

Algunos de las opciones de ejecución son las siguientes:

- *stat*: mide el recuento total de eventos.
- *record*: mide y guarda los datos de muestreo.
- *report*: analiza archivo generado por registro perf. Puede generar un *profile* plano o un grafo de llamadas.
- *sched*: seguimiento/medición de las acciones y latencias del *scheduler*.

Para ejecutar *perf* se debe realizar lo siguiente:

```
$ sudo perf stat ./main 'nombre_imagen' 'radio' 'cantidad_hilos'
```

```
Performance counter stats for './main mont.bmp 1500 8':  
      53.158,42 msec task-clock          #    1,618 CPUs utilized  
        41.069    context-switches       #    0,773 K/sec  
         555    cpu-migrations          #    0,010 K/sec  
        35.313    page-faults           #    0,664 K/sec  
 154.782.027.541    cycles            #    2,912 GHz  
 211.783.368.082    instructions       #    1,37  insn per cycle  
   1.055.034.838    branches          # 19,847 M/sec  
     10.446.192    branch-misses      #    0,99% of all branches  
  
 32,844590069 seconds time elapsed  
 52,618617000 seconds user  
  0,638835000 seconds sys
```

Figura 1: Resultado de *perf stat*.

### 3.1.2. gprof

Es la mejora de *perf*. Para poder utilizarla, el programa a analizar debe compilarse con la opción -pg, que genera archivo con un grafo de llamadas (*default gmon.out*). Este archivo es leído por *gprof* y calcula los tiempos de cada función utilizada.

Para hacer un análisis con *gprof* se deben ejecutar los siguientes comandos:

```
$ ./main.o 'nombre_imagen' 'radio' 'cantidad_hilos'  
$ gprof ./main.o gmon.out
```

%	cumulative	self	self	total	
time	seconds	seconds	calls	us/call	us/call
99.61	42.29	42.29	23204652	1.82	1.82
0.42	42.47	0.18			apply_filters
0.00	42.47	0.00	1	0.00	0.00
0.00	42.47	0.00	1	0.00	0.00

blur\_filter  
apply\_filters  
sbmp\_initialize\_bmp  
sbmp\_load\_bmp

Figura 2: Salida plana de *gprof*.

Véase que la Figura 2 indica que el 99.61 % del tiempo de ejecución del programa pertenece a la función ‘*blur\_filter*’ (ver 3), por lo que se utilizará *OpenMP* en la función ‘*apply\_filters*’.

### 3.1.3. Valgrind

Es un conjunto de herramientas que se utilizan para depurar problemas de memoria y rendimiento de un programa. Algunas de estas herramientas son:

- *memcheck*: realiza un seguimiento del uso de la memoria.
- *cachegrind*: mide el rendimiento de la caché durante la ejecución.
- *helgrind*: detecta las condiciones de carrera en un código multihilo.

Con *Valgrind*, el programa a *testear* se ejecuta entre cinco y veinte veces más lento, por lo que es ralmente importante tenerlo en cuenta al momento de su uso.

```
==27029== I refs: 292,217,441,341
==27029== I1 misses: 2,041
==27029== LLi misses: 1,991
==27029== I1 miss rate: 0.00%
==27029== LLi miss rate: 0.00%
==27029==
==27029== D refs: 79,344,943,784 (78,111,658,188 rd + 1,233,285,596 wr)
==27029== D1 misses: 410,410,241 ( 394,223,154 rd + 16,187,087 wr)
==27029== LLd misses: 22,687,177 ( 19,308,473 rd + 3,378,704 wr)
==27029== D1 miss rate: 0.5% ( 0.5% + 1.3% )
==27029== LLd miss rate: 0.0% ( 0.0% + 0.3% )
==27029==
==27029== LL refs: 410,412,282 ( 394,225,195 rd + 16,187,087 wr)
==27029== LL misses: 22,689,168 ( 19,310,464 rd + 3,378,704 wr)
==27029== LL miss rate: 0.0% ( 0.0% + 0.3% )
```

Figura 3: Resultado de *valgrind* con la herramienta *cachegrind*.

### 3.1.4. V-Tune

Es desarrollado por Intel. Puede utilizarse con una interfaz gráfica o por medio de la línea de comandos. Los comandos soportados pueden verse en el siguiente enlace: *user guide*.

```
Elapsed Time: 15.459s
CPU Time: 46.860s
Effective Time: 46.860s
Idle: 0.025s
Poor: 17.952s
Ok: 0s
Ideal: 28.883s
Over: 0s
Spin Time: 0s
Overhead Time: 0s
Total Thread Count: 8
Paused Time: 0s

Top Hotspots
Function      Module      CPU Time
-----
blur_filter    main        46.501s
apply_filters  main        0.219s

Effective Physical Core Utilization: 77.5% (1.549 out of 2)
Effective Logical Core Utilization: 76.2% (3.047 out of 4)
```

Figura 4: Resultado de *vtune* con la herramienta *hotspots*.



### 3.2. *Makefile*

Las recetas disponibles en el *Makefile* son las siguientes:

1. *all*: genera el ejecutable del sistema.
2. *check*: corre *cppcheck* sobre el directorio **src**.
3. *doc*: genera la documentación del código en HTML utilizando *doxygen* en la carpeta llamada **Doc** y la abre en el navegador *Firefox*.<sup>1</sup>
4. *clean*: elimina los directorios **bin** y **Doc**.

## 4. Implementación y resultados

En las siguientes subsecciones se explicarán las funciones encargadas de aplicar los filtros y luego los resultados obtenidos con distintas configuraciones.

Para reducir los tiempos se declararon ambas imágenes (de entrada y salida) a nivel global, evitando una sobrecarga en la pila y logrando un acceso más rápido a estas dos variables[3].

### 4.1. Aplicación de los filtros

Código 1: Función ‘*apply\_filters*’.

```
1 int32_t distancia = 0;
2 int32_t distancia_y = 0;
3 int32_t radio_cuadrado = r*r;
4
5 uint16_t **kernel = calloc(SIZE_K, sizeof (int *));
6
7 for(int k = 0; k < SIZE_K; k++)
8     kernel[k] = calloc(SIZE_K, sizeof (uint16_t));
9
10 kernel_setup(kernel, SIZE_K);
11
12 uint16_t norm = get_norm(kernel, SIZE_K);
13 int32_t offset = (int32_t) ((SIZE_K-1) / 2);
14
15 double start = omp_get_wtime();
16
17 #pragma omp parallel num_threads(threads)
18 #pragma omp for schedule(dynamic)
19 for (int32_t y = offset; y < in.info.image_height-offset; y++)
20 {
21     distancia_y = (y - c.pos_y)*(y - c.pos_y);
22
23     for (int32_t x = offset; x < in.info.image_width-offset; x++)
24     {
25         distancia = (x - c.pos_x)*(x - c.pos_x) + distancia_y;
```

<sup>1</sup>Si no se encuentra instalado, se sugiere cambiar el navegador en la receta.

```
27 if(distancia <= radio_cuadrado) /* Inside circle */
28     out.data[y][x] = lineal_filter(in.data[y][x]);
29 else /* Outside circle */
30     out.data[y][x] = blur_filter(x, y, kernel, norm);
31 }
32 }
33
34 printf("\n-- Tiempo transcurrido (%d hilos): %f --\n\n", threads,
35     omp_get_wtime() - start);
```

El Código 1 muestra la implementación de la función que aplica ambos filtros en la imagen. Dicha función recorre la imagen por filas y determina si el píxel está o no dentro de la circunferencia de radio especificado. En caso de estar dentro, se aplica el filtro lineal, sino se aplica el de desenfoque.

#### 4.1.1. Filtro lineal

Código 2: Función ‘*lineal\_filter*’.

```
1 int32_t r, g, b;
2 r = pixel.red;
3 g = pixel.green;
4 b = pixel.blue;
5
6 r = (int32_t) (r * K) + L;
7 r = normalize_pixel_color(r);
8
9 g = (int32_t) (g * K) + L;
10 g = normalize_pixel_color(g);
11
12 b = (int32_t) (b * K) + L;
13 b = normalize_pixel_color(b);
14
15 pixel = (sbmp_raw_data) { (uint8_t) b, (uint8_t) g, (uint8_t) r};
16
17 return pixel;
```

El Código 2 muestra la implementación de la función encargada de aplicar el filtro lineal dentro de la circunferencia dada. Esta función toma cada uno de los colores del píxel, los modifica según los valores de brillo y contraste dados, y devuelve el píxel modificado.

#### 4.1.2. Filtro de desenfoque

Código 3: Función ‘*blur\_filter*’.

```
1 int32_t sum_r = 0;
2 int32_t sum_g = 0;
3 int32_t sum_b = 0;
4 int32_t offset = (int32_t) ((SIZE_K-1) / 2);
5
6 for (int32_t m = -offset; m <= offset; m++)
7 {
8     for (int32_t n = -offset; n <= offset; n++)
9     {
```

```
10     sum_r += (int32_t) (kernel[m+offset][n+offset] * in.data[y+m][x+n].  
11         red);  
12     sum_g += (int32_t) (kernel[m+offset][n+offset] * in.data[y+m][x+n].  
13         green);  
14     sum_b += (int32_t) (kernel[m+offset][n+offset] * in.data[y+m][x+n].  
15         blue);  
16 }  
17 sum_r = (int32_t) (sum_r / norm);  
18 sum_g = (int32_t) (sum_g / norm);  
19 sum_b = (int32_t) (sum_b / norm);  
20  
21 return (sbmp_raw_data) {(uint8_t) sum_b, (uint8_t) sum_g, (uint8_t) sum_r};
```

El Código 3 muestra la implementación de la función encargada de aplicar el filtro de desenfoque fuera de la circunferencia dada. Simplemente realiza la convolución entre la matriz kernel y la imagen de entrada, una vez por cada color, los normaliza y los devuelve en el nuevo píxel.

## 4.2. Resultados

### 4.2.1. Filtrado

A continuación se muestran los resultados de la modificación de la imagen dada y dos a elección del estudiante con los parámetros mencionados la sección 3.



Figura 5: Imagen de entrada.

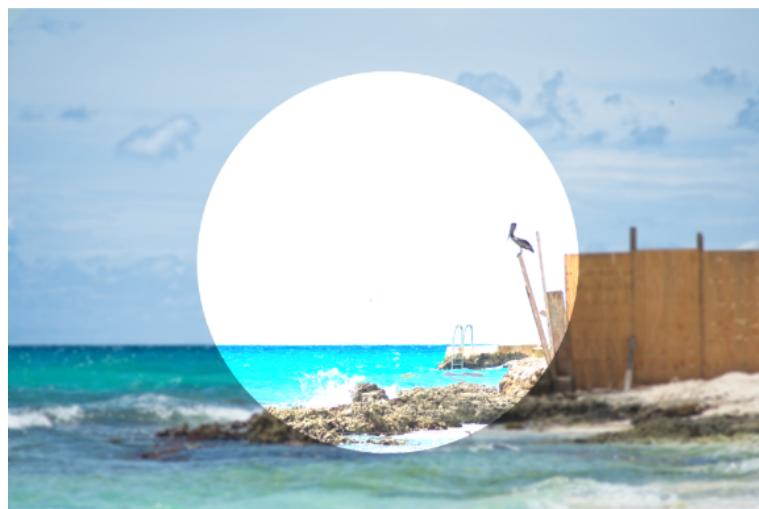


Figura 6: Imagen de salida.



Figura 7: Imagen de entrada.



Figura 8: Imagen de salida.



Figura 9: Imagen de entrada.



Figura 10: Imagen de salida.

#### 4.2.2. Medición de tiempos

Para medir los tiempos de ejecución se realizó un *script* en *perl*.

Código 4: *Script* de ejecución.

```
1 #!/usr/bin/perl
2
3 use Time::HiRes qw[time];
4
5 $M = 1;
6 $muestras = $M;
7 $pot = 0;
8 $hilos = 1;
9 $max_hilos = 16;
10 $radio = 100;
11
12 while($hilos <= $max_hilos)
13 {
14     while($muestras > 0)
15     {
16         $cmd = "./main base.bmp $radio $hilos";
17         system($cmd);
```

```
18 $muestras--;
19 }
20 $muestras = $M;
21 $pot++;
22 $hilos = 2**$pot;
23 }
24
25 print("DONE.\n");
```

El Código 4 muestra dicho *script*, que realiza una determinada cantidad de muestras para una cantidad de hilos que varía en potencias de 2 hasta alcanzar una cantidad total de 64.

Este *script* se ejecuta tanto en la PC local como en un nodo del *cluster* de la Facultad, *Wayra*. Dicho nodo tiene un procesador Intel Xeon E5-2630 v4<sup>2</sup>, con las siguientes características:

- *CPU(s)*: 20
- *Thread(s) per core*: 2
- *Core(s) per socket*: 10
- *Socket(s)*: 1
- *CPU MHz*: 1200.221
- *L1d cache*: 320 KiB
- *L1i cache*: 320 KiB
- *L2 cache*: 2.5 MiB
- *L3 cache*: 25 MiB

La PC local tiene un Intel Core i7-7500U con estas características:

- *CPU(s)*: 4
- *Thread(s) per core*: 2
- *Core(s) per socket*: 2
- *Socket(s)*: 1
- *CPU MHz*: 523.441
- *L1d cache*: 32K
- *L1i cache*: 32K
- *L2 cache*: 256K
- *L3 cache*: 4096K

Se probó en ambos lados con y sin el *flag* de compilación `-march=native`. Esta bandera de compilación indica al compilador que habilite todos los subconjuntos de instrucciones soportadas por la arquitectura nativa, lo cual produce un código optimizado, pero únicamente para la máquina en la que se está compilando el programa.

<sup>2</sup>Para información detallada del procesador, dirigirse aquí.

Seguidamente se muestran tablas con los resultados de las ejecuciones con y sin la bandera tanto en el nodo como en la PC local. Estas tablas contienen la cantidad de hilos con que se ejecutó el programa, el tiempo que duró su ejecución y un factor de escala, que indica cómo va escalando el tiempo de acuerdo a la cantidad de hilos utilizados.

Cantidad de hilos	Tiempo [s]	Scale Factor
1	67.74	–
2	36.26	1.87
4	19.98	3.39
8	11.69	5.79
16	9.81	6.91
20	9.10	7.44
32	9.14	7.41
64	8.95	7.57

Tabla 1: Duración del programa **sin** el *flag* en el *cluster*.

Cantidad de hilos	Tiempo [s]	Scale Factor
1	47.76	–
2	25.14	1.90
4	14.08	3.39
8	8.14	5.87
16	6.48	7.37
20	5.90	8.09
32	5.95	8.03
64	6.01	7.95

Tabla 2: Duración del programa **con** el *flag* en el *cluster*.

Cantidad de hilos	Tiempo [s]	Scale Factor
1	72.15	–
2	47.34	1.52
4	45.76	1.58
8	41.10	1.76
16	41.07	1.76
32	40.82	1.77
64	41.72	1.73

Tabla 3: Duración del programa **sin** el *flag* en la PC.

Cantidad de hilos	Tiempo [s]	Scale Factor
1	41.54	–
2	23.66	1.75
4	19.93	2.08
8	19.83	2.09
16	19.86	2.09
32	19.90	2.09
64	19.87	2.09

Tabla 4: Duración del programa **con** el *flag* en la PC.

#### 4.2.3. Análisis de los tiempos

Viendo los resultados de las tablas anteriores, se nota claramente que al utilizar la bandera de compilación **-march=native**, mejoran notablemente los tiempos de ejecución y el factor de escala, ya que en las tablas **2** y **4** los factores de escala se comportan de manera adecuada, es decir, las ejecuciones con una cantidad de hilos mayor a los físicos, causan tiempos de ejecución mayores debido a que se realizan más ciclos de reloj a causa de los cambios de contexto y sincronización de los caché.

En las tablas **1** y **3**, sin embargo, lo dicho anteriormente no ocurre. En el caso del nodo cuando se realiza una ejecución con 64 hilos, el factor de escala vuelve a aumentar en lugar de disminuir, mientras que en el caso de la PC, cuando llega a 32 ejecuciones el factor aumenta y en la ejecución siguiente vuelve a disminuir. Claramente la optimización hecha por **-march=native** realiza una corrección de este comportamiento.

Cabe mencionar que antes de estas ejecuciones se realizaron pruebas con distintas banderas de compilación, como **-O2** y **-O1**, que aumentaron el tiempo de ejecución del programa. Debido a esto, el programa ejecutado en el cluster es el que mejor rendimiento obtuvo en la PC local.

## 5. Conclusiones

Los requerimientos pedidos por la cátedra fueron cumplidos con éxito: el programa realiza la aplicación de los filtros dentro y fuera de la circunferencia del radio pasado como parámetro mediante parallelización con *OpenMP*.

## Referencias

- [1] Wikipedia, *BMP file format*,  
[Link to Wikipedia](#)
- [2] *OpenMP*,  
[Link to OpenMP](#)
- [3] Nora Sandler: *C compiler, Part 10*,  
[Link to Global variables](#)



UNC



FCEFyN

[4] *Wikipedia: perf,*

[Link to perf](#)

[5] *Linux Manual: gprof,*

[Link to gprof](#)

[6] *Valgrind,*

[Link to Valgrind](#)