PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN

IIC2523 Sistemas Distribuidos (II/2017)

Tarea 1

Raimundo Herrera (rjherrera@uc.cl)

1 Solución

Para esta tarea, se implementó un código muy simple en `C`. El flujo general del programa corresponde a abrir los archivos de imagen y texto pertinentes, y pasarlos a las matrices correspondientes. Luego, se itera (1) sobre la cantidad de aplicaciones del filtro que se desean realizar, (2) sobre la cantidad de filas de la imagen, (3) sobre la cantidad de columnas de la imagen, (4) por la cantidad de filas del kernel filtro y (5) por la cantidad de columnas del kernel o filtro.

La sección paralelizada corresponde a la parte externa del segundo `for`, es decir, antes de empezar a iterar sobre las filas de la imagen.
En líneas generales el la sección importante del código se ve así, donde se omiten asignaciones y cálculos para solo entender la estructura.

```
for (int it = 0; it < n_iters; it++){
    // code
    #pragma omp parallel for num_threads(n_threads) collapse(2)
    for (int row = 0; row < height; row++) {
        for (int col = 0; col < width; col++) {
            // code
            for (int i = row - margin_rows; i < row + margin_rows + 1; i++){
                for (int j = col - margin_cols; j < col + margin_cols + 1; j++){
                    // code
                }
            }
            // code
        }
    }
}
```

Como se observa, se utilizó la instrucción *pragma* para paralelizar con *OpenMP* el programa.

Para que la ejecución sea secuencial basta con que el argumento de `n_threads` sea 1. Se incluye el `collapse(2)` para que ambos `for`, el de filas y de columnas, se colapsen en uno solo antes de la división.

Como se explica en el `README`, para ejecutar el programa se optó por la siguiente estructura:

```
./filter <input.png> <kernel.txt> <output.png> <iteraciones> <n_threads>
```

Que, muy brevemente, implica que para ejecutar el programa hay que ejecutar el comando filter con los siguientes argumentos: imagen de entrada en *png*, filtro o kernel en *txt*, la imagen de salida, la cantidad de iteraciones o aplicaciones del filtro, número de threads.

2 Experimentos

Para la parte experimental de la tarea la configuración se describe muy explícitamente en el archivo `script.sh` de mi directorio en el cluster. Esta es:

- 2 imagenes:
  - lena.png (512×512)
  - fruit.png (1024×732)
- 2 filtros:
  - blur.txt (3×3)

$$\begin{bmatrix} 0.0625 & 0.125 & 0.0625 \\ 0.125 & 0.25 & 0.125 \\ 0.0625 & 0.125 & 0.0625 \end{bmatrix}$$

  - double.txt (5×5)

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

- Cantidad de iteraciones: 25,50
- Número de threads: 1,2,4,8,16

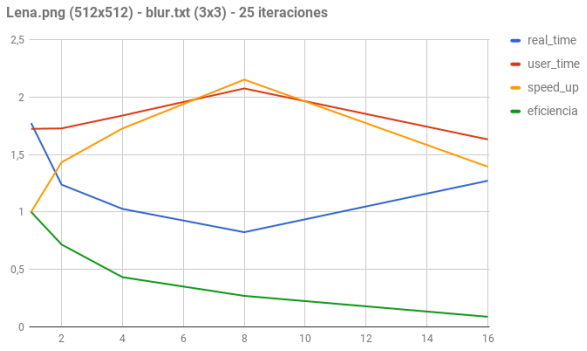
En total, por lo tanto, se realizaron 40 ejecuciones para las pruebas de rendimiento. 20 por imagen según las diferentes configuraciones de parámetros.

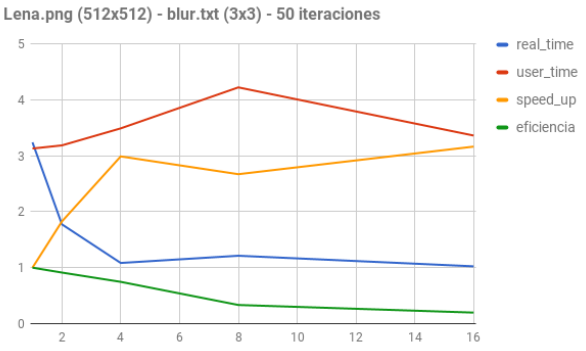
La máquina utilizada es una de las del cluster Grima, específicamente Titan. Tiene una CPU *Intel(R) Xeon(R) CPU X5647 @ 2.93GHz*, la cual, según el comando `lscpu` tiene 8 cores. Por el lado de la memoria, según el comando `more /proc/meminfo`, el nodo utilizado posee alrededor de 96GB de RAM (98997284kB).

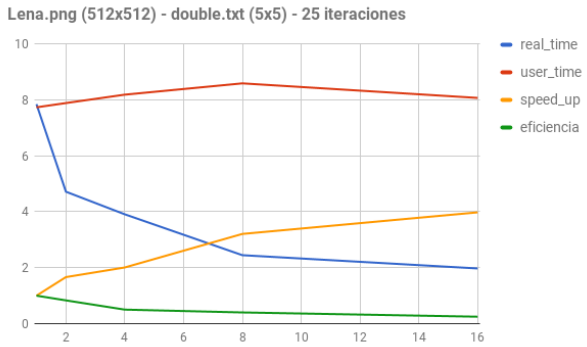
3 Rendimiento y observaciones

Para esta sección se incluyen ciertos gráficos para ilustrar de mejor manera.

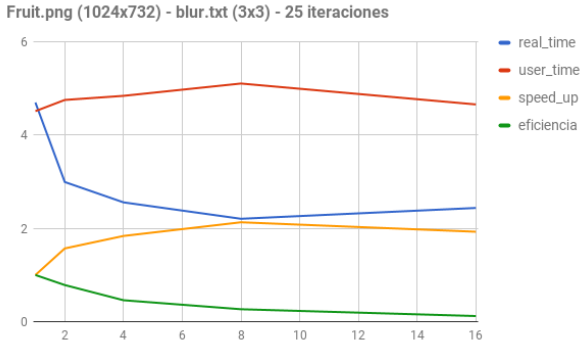
Se incluyen 4 gráficos por imagen, primero para *lena.png* las distintas configuraciones.

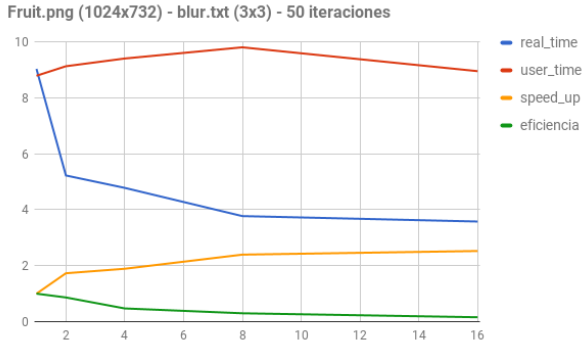


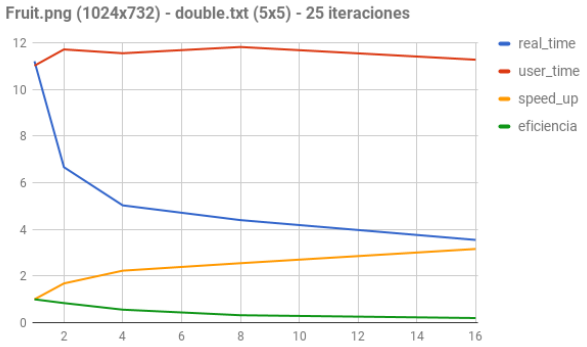


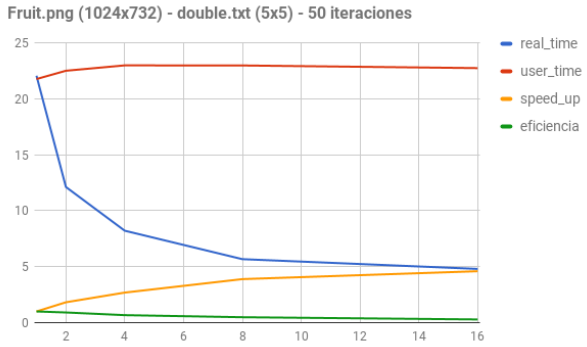


Ahora para *fruit.png* las distintas configuraciones.









Todos los ejes horizontales representan el número de cores y los verticales cumplen doble función, muestran tiempo para real time y user time, y son adimensionales para las otras medidas, sin embargo, los valores son esos.
Es posible observar en los gráficos con menos iteraciones que se comportan de acuerdo a lo esperado, tanto en el gráfico con 25 iteraciones de blur para la primera como para la segunda imagen, se observa que el peak de rendimiento ocurre en 8 cores, que justo concuerda con la cantidad de cores de la maquina.

Este comportamiento se ve reflejado en una menor *real time*, la cual refleja que cuando menor es el tiempo y mayor el *speedup* es en ese peak.
Al cambiar, en general la cantidad de cores, para todos los experimentos, el rendimiento aumenta, pero no siempre es claro que descienda o se mantenga una vez superados los 8 físicos, sino que a veces se observan comportamientos anómalos, como en el ejemplo de lena, con el filtro double y las 25 iteraciones. Esto se puede deber a que la ejecución es más intensiva, y en este caso el *scheduler* asigna más recursos a el programa, de modo que al llegar a 8 threads sigue sin usar todos los recursos disponibles de el PC.

Como se observa también, el análisis es similar para 50 iteraciones en todos los casos. Se ve que es un caso

más intensivo de uso de CPU y por lo tanto se ve esa anomalía descrita.

Se puede ver también que el tamaño del filtro afecta sustancialmente a el rendimiento, siendo que al mantener constante lo demás se aumenta casi al doble los tiempos de ejecución entre un filtro y otro.

Lo mismo se observa con el tamaño de la imagen, se incluyó una imagen que tiene 2,85 veces más pixeles que lena, y el rendimiento se afectó casi disminuyendo a la mitad en todos los casos, lo que muestra su significancia.

En cuanto al speedup, se hace evidente que es de cierto modo asintótico, al menos en los casos que se da la anomalía descrita, y aumenta de manera pseudo-exponencial en los casos ideales, llegando a su peak en 8. Esto tiene total sentido considerando los cores físicos. Sin embargo, se puede concluir de la máquina, que los context switching de cores son muy poco costosos ya que el rendimiento es muy bueno aún con más cores virtuales que físicos.

Por último señalar que en todos los casos, por obvio que suene el rendimiento mejoró sustancialmente en su forma paralela a la secuencial y se mostró empíricamente lo importante de la paralelización y distribución.