Trabajo práctico 0: Infraestructura básica

Alejandro García Marra, *Padrón Nro. 91.516*
alemarra@gmail.com
Sebastián Javier Bogado, *Padrón Nro. 91.707*
sebastian.j.bogado@gmail.com
Grupo Nro. 0 - 2do. Cuatrimestre de 2012
66.20 Organización de Computadoras
Facultad de Ingeniería, Universidad de Buenos Aires

Resumen

El presente trabajo busca crear un programa que permita el ordenamiento de archivos a través de dos implementaciones distintas, una utilizando el algoritmo Quicksort y la otra el algoritmo Stooge sort.
Sobre este programa, luego, se realizarán una serie de mediciones con el fin de determinar los desempeños relativos de cada implementación y las posibles mejoras a realizar. Para esto haremos uso de los programas **time** y **gprof**.

1. Introducción

Muchas veces tanto para programas recién terminados, como para aquellos que llevan un tiempo en funcionamiento, se desconoce realmente qué partes del programa insumen la mayor cantidad de recursos, sean estos de tiempo, carga de cpu, etc. Poseer esta información se torna en algo crítico cuando se busca realizar una mejora de performance en dicho programa. Sería poco útil intentar optimizar a ciegas, por no decir inútil.
Haremos uso entonces de dos herramientas distintas, el profiling del código (por medio de *gprof*) y la medición de los tiempos de ejecución (por medio de *time*).

- **Profiling**: El profiling permite aprender donde el programa pasa la mayor parte de su tiempo, y cuales funciones llaman a otras mientras se ejecuta. Esta informacion puede mostrar qué piezas del programa son mas lentas de lo esperado, convirtiéndolas en candidatas para su reescritura en la etapa de optimización.
  También puede ayudarnos a descubrir cuales funciones son llamadas más o menos veces delo esperado, pudiendo así encontrar nuevos bugs (aunque el descubrimiento de bugs no es el fin principal de esta etapa)

  El profiler utiliza información recolectada en tiempo de ejecución, por lo que puede ser utilizado en programas demasiado grandes o complejos, donde un análisis por lectura de fuentes sería impracticable.
  Como consecuencia del análisis durante la ejecución, los datos con los que se corra el programa afectaran el resultado del profiler. Es decir, distintos datos de entrada pueden provocar distintas ramas de ejecución, dando po resultado que, por ejemplo, no se llamen algunas funciones.

- **Medición de Tiempos**: Permite conocer con precisión los tiempos de ejecución de un programa, discriminados entre tiempos de systema, de usario, tiempos totales, etc., así como también conocer los porcentajes para cada parte del programa, cantidad de entradas, y muchas otras opciones. La combinación con una herramienta de profiling permite exactitud a la hora de conocer la forma en que se ejecuta el programa bajo estudio, permitiendo optimizar únicamente las partes críticas del ciclo de ejecución.

2. Métodos de ordenamiento

2.1. Quicksort

El Quicksort es un algoritmo re pulenta

2.2. Stooge sort

El Stooge sort es un algoritmo de ordenamiento recursivo muy ineficiente, de complejidad $O(n^{\log 3/\log 1{,}5})$.
Realiza el intercambio entre el primer y último de los elementos si es el primero es mayor (en ordenamiento ascendente). Luego, si el conjunto está compuesto por al menos tres valores, entonces aplica Stooge sort a los primeros dos tercios, luego a los dos últimos tercios, y finalmente repite con los primeros dos tercios.

3. Flujo del programa

El programa comienza verificando el primer parámetro recibido. Si no existe o no es reconocido, sucede el comportamiento por defecto, que es el mensaje de ayuda, información sobre el uso. Con esto, termina la ejecución. Otro camino rápido en la aplicación es al solicitar la versión, con el parámetro `-V` o `--version`.

Si el primer parámetro solicitó un algoritmo de ordenamiento (`-q` o `--quick` para Quicksort y `-s` o `--stooge` para el Stooge sort), entra en el corazón del programa: la función `void sort(unsigned n, char* files[], line** (*sort_func)(line** , unsigned ))`:

- `n`: la cantidad de archivos sobre los que se aplicará el ordenamiento. Si es cero, se tomará la entrada estándar.
- `files`: los nombres de los archivos
- `sort_func`: puntero a la función de ordenamiento seleccionada anteriormente con el primer parámetro

En el archivo `line.h` se encuentra definido el tipo homónimo, desde `typedef struct line line`. Esto fue necesario porque en un archivo podría aparecer un byte nulo antes de el byte 10, aquél tomado como fin de línea. Como las cadenas en C terminan en un byte nulo, todas las funciones estándar de la librería `<string.h>` resultaron inservibles porque necesitábamos que fuera otro el indicador de fin.

Volviendo a la función `sort`, se arma el conjunto de líneas a ser reordeanadas recolectándolas de a un archivo por vez. Si no hubiera archivos, se trabaja con la entrada estándar con la misma función (`parseLineas`, que se detallará en breve).

Armado el conjunto, se llama a la función de ordenamiento desde su puntero pasándole como parámetros el vector de líneas y la cantidad de elementos. Luego se imprimen por salda estándar y se libera la memoria reservada, todo esto con funciones propias para el tipo `line`.

La lectura de cada archivo sucede en la función `unsigned parseLineas( line** *pLinea, unsigned lineas, FILE* stream)`:

- `pLinea`: puntero a un arreglo de punteros a `line`
- `lineas`: cantidad de elementos del arreglo
- `stream`: archivo desde donde se llevarán a cabo las lecturas
- `return`: cantidad de elementos en el arreglo después del proceso

Esta función carga en un buffer el archivo y lo recorre en busca del caracter de fin de línea. Por cada uno encontrado, se aloca espacio para un `line`, mediante sus métodos dedicados, y se agrega al vector. Si el archivo no termina naturalmente como una línea, se agrega al final el `EOL`. Retorna con la cantidad de líneas cargadas (las acumuladas y las nuevas del archivo actual).

4. Mediciones

4.1. Valores Obtenidos

En la tabla 1 se presentan las mediciones realizadas con **time** sobre ambos algoritmos de ordenamiento y con archivos de distintos tamaños.

Además de los archivos indicados en el enunciado, fueron agregadas mediciones sobre archivos con tamaños arbitrarios, mayores, con el fin de mostrar de mejor manera las diferencias entre algoritmos.

		Quicksort			Stooge sort		
		Ordenado	Invertido	Aleatorio	Ordenado	Invertido	Aleatorio
1kb	real*	0.00	0.00	0.00	0.00	0.00	0.00
	user*	0.00	0.00	0.00	0.00	0.00	0.00
	sys*	0.00	0.00	0.00	0.00	0.00	0.00
8kb	real	0.00	0.00	0.00	0.02	0.02	0.01
	user	0.00	0.00	0.00	0.01	0.01	0.01
	sys	0.00	0.00	0.00	0.00	0.00	0.00
16kb	real	0.00	0.00	0.00	0.00	0.02	0.02
	user	0.00	0.00	0.00	0.00	0.01	0.02
	sys	0.00	0.00	0.00	0.00	0.00	0.00
32kb	real	0.00	0.00	0.00	0.17	0.17	0.17
	user	0.00	0.00	0.00	0.17	0.17	0.17
	sys	0.00	0.00	0.00	0.00	0.00	0.00
64kb	real	0.00	0.00	0.00	1.44	1.44	1.44
	user	0.00	0.00	0.00	1.44	1.43	1.44
	sys	0.00	0.00	0.00	0.00	0.00	0.00
1024kb	real	0.04	0.03	0.03	>1500	>1500	>1500
	user	0.03	0.02	0.03	>1500	>1500	>1500
	sys	0.00	0.00	0.00	0.00	0.00	0.00

Cuadro 1: Resultados comando Time

* Referencia:

- real: %e, tiempo total real usado por el proceso.
- user: %U, total de segundos-CPU usados por el proceso directamente.
- sys : %S, total de segundos-CPU utilizados por el systema en nombre del proceso.

4.2. Análisis de los datos

La marcada diferencia entre la complejidad de los algoritmos se refleja en muestras tan chicas como la de 8kb. A partir de ahí, el Stooge sort ya hace suficiente uso del procesador como para ser notado por time, mientras que el Quicksort hace lo propio recién en la muestra más grande, de 1024kb. En este caso, el Stooge sort se tornó intolerable.

En la figura 1 se muestra el gráfico del tiempo insumido por el Quicksort para las distintas muestras.
En la figura 2 se muestra el gráfico del tiempo insumido por el Stooge sort para las distintas muestras, a excepción de la de 1024kb, porque demanda una escala que haría inapreciable la situación de las otras muestras.

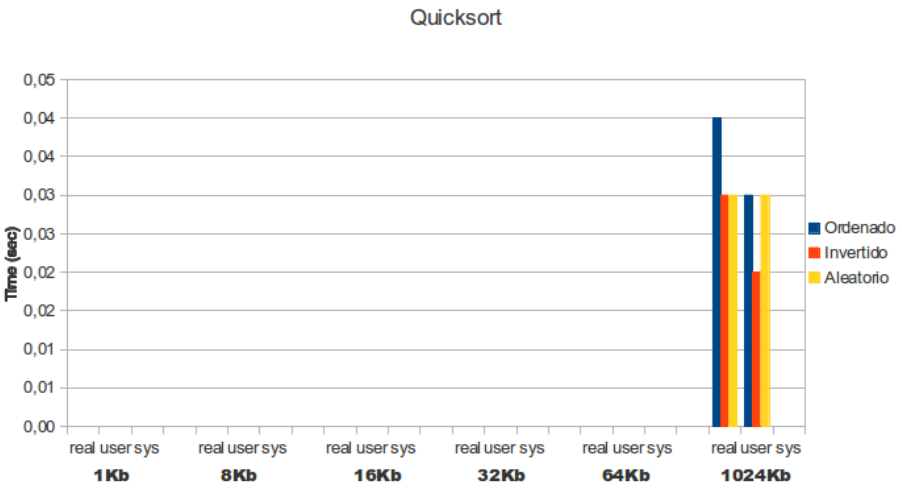

Figura 1: Tiempo tomado para distintas muestras del Quicksort.

Al momento de calcular el speedup de Quicksort contra Stooge sort, con la razón entre los tiempos de cada uno, las cifras obtenidas no lo permiten. El primero no arroja resultados apreciables por **time** en ninguna instancia sin contar la última, donde el Stooge sort es inmanejable.

Esto es porque la complejidad del Quicksort es, en promedio, $O(n \log(n))$, mientras que el Stooge sort es de $O(n^{2,7})$. Entonces, el speedup entre ambos algoritmos tiende a infinito exponencialmente, según crece el tamaño de la muestra.

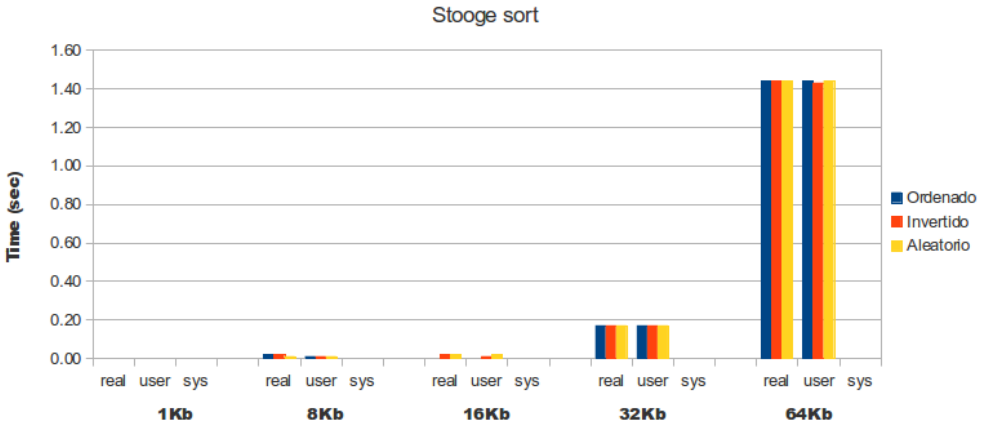

Figura 2: Tiempo tomado para distintas muestras del Stooge sort.

5. Profiling

Como se pudo apreciar en la tabla de tiempos de la página anterior, el algoritmo de quicksort resulta extremadamente veloz para tamaños de archivos relativamente grandes. Decidimos, entonces, forzar un poco más al programa y hacer el profiling sobre un archivo desordenado de 32mb. Además, para que las funciones mismas de **gprof** tuvieran una incidencia despreciable en la prueba. Realizar esta misma prueba sobre el algoritmo Stooge sort, sería impracticable, ya que los tiempos demandados serían demasiado grandes.

Un punto interesante a destacar son llamadas a métodos de ordenamiento: como era de esperarse, no se realizan llamadas al método stoogesort, lo cual confirma que el algoritmo no es utilizado.

Por otra parte, para el método quicksort_r que implementa dicho algoritmo, tenemos cientos de miles de llamados. Esto era de esperarse debido a las dimensiones del archivo y la naturaleza recursiva del algoritmo utilizado.

La tabla 2 es la salida del **gprof**, en particular aquella conocida como "flat profile". Muestra el uso de las funciones invocadas, ordenadas de mayor a menor según el porcentaje de tiempo de ejecución de cada una.

Se aprecia en la tabla que la función particionar ocupa el 91,87 % del tiempo de ejecución de programa, convirtiéndola en la candidata para mejorar.

Para calcular el speedup máximo, usamos la fórmula:

$$speeduptotal = 1/(1 - f + f/SM)$$

$f = 0,9187$; SM tiende a infinito por ser arbitrariamente mejorable

$$Speedupmaximo = 1/0,08123 = 12,3$$

%	cumulative	self		self	total	
time	seconds	seconds	calls	s/call	s/call	name
91.87	1.53	1.53	448399	0.00	0.00	particionar
2.40	1.57	0.04	448801	0.00	0.00	cargarBuffer
1.80	1.60	0.03	3565398	0.00	0.00	swap
1.80	1.63	0.03	1	0.03	0.07	parseLineas
1.80	1.66	0.03	1	0.03	1.67	sort
0.60	1.67	0.01	1	0.01	1.57	quickSort_r
0.00	1.67	0.00	1	0.00	0.00	check_param
0.00	1.67	0.00	1	0.00	1.57	quick_sort

Cuadro 2: Ejemplo de tabla.

6. Comandos de ejecución y corridas de prueba

Comandos de Compilación:

- make all: genera el programa, modo release"
- make debug: genera el programa con flags para debugging
- make asm: genera los archivos Assemblies, sin generar el programa
- make gprof: genera el programa con flags para **gprof**
- make clean: remueve los archivos generados

Comandos de Ejecución:

- tp0 [OPTIONS] [file...]
- <stdout> | tp0 [OPTIONS]
- -h, –help
- -V, –version
- -q, –quicksort
- -s, –stoogesort

7. Conclusiones

El presente trabajo, a modo de introducción, nos inicializó en las herramientas que usaremos en los trabajos siguientes.

Pudimos manejar la máquina virtual MIPS e interpretar un archivo de profiling, como si estuviésemos optimizando un programa real.

Respecto de las velocidades observadas, podemos confirmar lo esperado a partir del análsis teórico. El algoritmo quicksort fue varios ordenes de magnitud más veloz que el stoogesort, al punto de poder procesar archivos que con el segundo no serían viables.

Referencias

[1] J. L. Hennessy and D. A. Patterson, "Computer Architecture. A Quantitative Approach," 3ra Edición, Morgan Kaufmann Publishers, 2000.