Trabajo práctico 1: Conjunto de instrucciones MIPS

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 del algoritmo Stooge sort, aproximando a un comportamiento minimalista del comando *sort*.

ESTO VA????: No sé, lo discutimos luego
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*).

1.1. Stooge sort

El Stooge sort es un algoritmo de ordenamiento recursivo muy ineficiente, de complejidad $1\left(n^{\log 3/\log 1{,}5}\right)$.
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.

2. 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`.

En una ejecución normal, el primer parámetro comienza la lista de archivos a procesar.

Una vez en 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 stoogesort pasándole como parámetros el vector de líneas, y dos enteros que representan el principio y el final del vector. Luego se imprimen por salida estándar y se libera la memoria reservada.

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

- `pLinea`: puntero a un arreglo de punteros a char
- `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 va cargando en un buffer cada línea, retornándolas de a una. Si el archivo no termina naturalmente como una línea, se agrega al final el `EOL`.

3. Mediciones

3.1. Valores Obtenidos

Acá iría el stooge.c vs stooge.S
En la tabla 1 se presentan las mediciones realizadas con **time** sobre ambas versiones del algoritmo y con archivos de distintos tamaños.

		Stooge sort		
		Ordenado	Invertido	Aleatorio
3*1kb	real*	0.00	0.00	0.00
	user*	0.00	0.00	0.00
	sys*	0.00	0.00	0.00
3*8kb	real	0.02	0.02	0.01
	user	0.01	0.01	0.01
	sys	0.00	0.00	0.00
3*16kb	real	0.00	0.02	0.02
	user	0.00	0.01	0.02
	sys	0.00	0.00	0.00
3*32kb	real	0.17	0.17	0.17
	user	0.17	0.17	0.17
	sys	0.00	0.00	0.00
3*64kb	real	1.44	1.44	1.44
	user	1.44	1.43	1.44
	sys	0.00	0.00	0.00
3*1024kb	real	>1500	>1500	>1500
	user	>1500	>1500	>1500
	sys	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.

3.2. Análisis de los datos

Calculo que el resultado va a ser que el algoritmo es re choto y por eso sigue siendo lento, aunque un poco seguro mejoró en Assembly

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 **??** se muestra el gráfico del tiempo insumido por el Quicksort para las distintas muestras.
En la figura 1 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 Stooge sort.

Un speed up del .S vs .c podría ir acá
BLASFJASGAJDFKGSDFGKSDFGADFg 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, $1\big(n\log(n)\big)$, mientras que el Stooge sort es de $1\big(n^{2,7}\big)$. Entonces, el speedup entre ambos algoritmos tiende a infinito exponencialmente, según crece el tamaño de la muestra.
ÁSDFJAKSGLADMGKMASDFGKADFGADFG

4. 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.

%	cumulative	self		self	total	
time	seconds	seconds	calls	s/call	s/call	name

Cuadro 2: dfasdfasdfasdfasdfasdfasdfasdfasfasdfasdfasdfasdfasdfasergaerg

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

Para calcular el speedup máximo, usamos la fórmula:

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

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

$$Speedupmaximo = 1/0 = 0$$

5. 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 cdebug: genera el programa con flags para debugging, pero utilizando el stoogesort.c
- make gprof: genera el programa con flags para **gprof**
- make clean: remueve los archivos generados

Comandos de Ejecución:

- tp1 [file...]
- <stdout> | tp1

6. Conclusiones

Había dicho el Niño que no pongamos cosas del estilo .ªprendimos a usar MIPS blah blah", sino orientado a .ª pesar de que lo escribimos en el lenguaje de programación más óptimo existente, los tiempos siguen siendo una cagada porque el algoritmo es una cagada, y no vale la pena el esfuerzo". Como diciendo "no te rompas el orto en Assembly si no vas a ganar mucho"
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.