Optimización de programas en Perl usando el profiler NYTProf para analizar su código fuente

Resumen

Escribir código puede ser una tarea sencilla en principio, pero hacer que este sea eficiente sí que puede ser una tarea bastante más complicada y tediosa. En el siguiente texto se va a hacer una demostración de que usando un profiler para analizar la ejecución de un programa se pueden detectar los cuellos de botella que se forman en la ejecución del código, viendo las llamadas a las funciones, el tiempo que está dentro de cada una, y más variables de interés. Y así intentar mejorar el código que produce estos cuellos de botella con lo que vamos a mejorar el tiempo de ejecución del mismo. En este caso, el lenguaje seleccionado es Perl, el profiler es NYTProf, y lo que se va a hacer es analizar el código de un programa con el profiler, mejorarlo, y luego volver a analizarlo, para ver si de verdad los cambios han tenido un efecto positivo en el tiempo que tarda el programa en ejecutarse. Tras esto se presentan los resultados obtenidos después de la optimización con la ayuda del profiler comparados con la ejecución del código inicial.

1.Introducción

1.1. ¿Qué es Perl?

Perl es un lenguaje de programación creado por Larry Wall, que salió a la luz por primera vez, el 18 de diciembre de 1987 con su versión 1.0. En un principio el lenguaje fue creado como un "pegamento" para el sistema Unix, pero si vamos un poco más adelante en el tiempo podemos ver fácilmente que Perl actualmente es un lenguaje de programación de propósito general con un entorno completo de desarrollo incluyendo debuggers, profilers, etc. y, además, uno de los entornos de programación más portables disponible, pudiendo usarse sin modificaciones en el código tanto en Unix/Linux como en Windows e incluso en Mac OS.

Completando lo dicho anteriormente, en Programming Perl [1] se dice que *"Perl is designed to make the easy jobs easy, without making the hard jobs impossible."*. Lo que quiere decir con esto se explica más adelante en un [1], diciendo *"¿Y cuáles son esos "trabajos fáciles" que deberían ser fáciles? Los que haces cada día, por supuesto. Quieres un lenguaje que haga fácil manipular números y textos, archivos y directorios, ordenadores y redes, y especialmente programas. Debería ser fácil ejecutar programas externos y escanear sus salidas en busca de información interesante. Debería ser fácil enviar esa información a otros programas que pudieran hacer cosas especiales con ella. Debería ser fácil desarrollar, modificar y depurar tus propios programas también. Y, por supuesto, debería ser fácil compilar y ejecutar tus programas, y hacerlo de manera portable, en cualquier sistema operativo moderno. Perl hace todo eso y mucho más."*. Como vemos es un lenguaje que sirve para todo, aunque actualmente para lo que más se utiliza es para la programación de sistemas y para el desarrollo web.

Por último, comentar que dispone de una comunidad muy activa de desarrolladores al ser un lenguaje de código abierto distribuido bajo la licencia GNU General Public License, y que todos sus módulos (o la inmensa mayoría) además de código en Perl se puede encontrar en CPAN (Comprehensive Perl Archive Network), el cual es referenciado en [1] diciendo *"…the most powerful thing about Perl is not Perl itself, but CPAN (Comprehensive Perl Archive Network), which contains myriads of modules that accomplish many different tasks that you don't have to know how to do. You just have to download it…"*. Y si entramos en CPAN [2] veremos que en su página de inicio pone "*The Comprehensive Perl Archive Network (CPAN) currently has 175,437 Perl modules in 34,637 distributions, written by 12,921 authors, mirrored on 250 servers. The archive has been online since October 1995 and is constantly growing."*. Con lo que se reafirma lo dicho anteriormente sobre que tiene una comunidad muy activa.

1.2. ¿Qué es un profiler y para qué sirve?

Los profilers son herramientas de análisis de aplicaciones que sirven para estudiar el comportamiento de los programas de manera dinámica, de forma que podemos saber el tiempo de ejecución de las funciones que utiliza, y así ver donde se forman los cuellos de botella, para poder solucionarlos, optimizando de esta manera el consumo de recursos o la velocidad de ejecución.

Podemos encontrar una definición más exacta en Measuring computer performance: a practitioner's guide [3] que dice *"A profile provides an overall view of the execution behavior of an application program. More specifically, it is a measurement of how much time, or the fraction of the total time, the system spends in certain states. A profile of a program can be useful for showing how much time the program spends executing each of its various subroutines, for instance. This type of information is often used by a programmer to identify those portions of the program that consume the largest fraction of the total execution time. Once the largest time consumers have been identified, they can, one assumes, be enhance to thereby improve performance."*.

1.3. Algunos profilers para Perl

El principal profiler es Devel::DProf, DProf es un profiler que esta ya obsoleto, y que será eliminado en una futura versión de Perl. La funcionalidad principal de DProf consiste en averiguar que subrutina de un script está usando más tiempo y que rutinas la llaman. Para ejecutar este profiler se usa el comando "perl -d:DProf test.pl". El archivo de salida que se crea tiene el nombre tmon.out, y recoge todos los tiempos de cada rutina y subrutinas. Además, tiene otro comando "dprofpp" para mostrar las 15 rutinas que más tiempo han usado.

Devel::SmallProf es otro profiler recomendado por Perl como podemos ver en [1] *"CPAN also holds Devel::SmallProf, which reports the time spent in each line of your program"*, su principal funcionalidad es analizar el tiempo que usa cada línea del programa. Se usa para tener un menor impacto en el uso de memoria y que tenga un formato fácil de entender. El formato que tiene lo podemos ver en [2] y es "<num> <time> <ctime> <line>:<text>", donde <num> es el número de ejecuciones que ha tenido, <time> que es el tiempo de ejecución, <ctime> es el tiempo que se ha estado usando la CPU y <line> y <text> son el número de línea y el trozo de código de esa línea.

También hay varios profiler que estudian el rendimiento, pero de funcionalidades específicas como Cache::Profile, que mide el rendimiento de la memoria cache usada. Cache::Profile se usa para medir el tiempo de uso de cache y aciertos/fallos, y con ello poder elegir la relevancia de la cache y escoger entre varias caches. Este profiler no debe ser usado como un benchmark, ya que los datos que muestra no tienen del todo veracidad, porque al usar el profiler incrementa el uso de cache un poco.

Como último ejemplo tenemos Devel::NYTProf que es una mejora de Devel::FastProf. NYTProf es un profiler potente, rápido y con una gran cantidad de características. Con el que se pueden medir una gran variedad funcionalidades de un script. Además, a la hora de mostrar el estudio realizado crea un informe en HTML. También añade un paquete de pruebas (test suite) que te dan un conjunto de condiciones con el que se puede comprobar la validez de la aplicación, sistema o una característica de este.

2. NYTProf

NYTProf es uno de los mejores profilers de los que dispone actualmente Perl, como ya se ha mencionado anteriormente en la introducción, debido a la gran cantidad de funcionalidades que incluye, las cuales podemos encontrar en [2] siendo las siguientes:

"Performs per-line statement profiling for fine detail
Performs per-subroutine statement profiling for overview
Performs per-opcode profiling for slow perl builtins
Performs per-block statement profiling (the first profiler to do so)

Accounts correctly for time spent after calls return
Performs inclusive and exclusive timing of subroutines
Subroutine times are per calling location (a powerful feature)
Can profile compile-time activity, just run-time, or just END time
Uses novel techniques for efficient profiling
Sub-microsecond (100ns) resolution on supported systems
Very fast - the fastest statement and subroutine profilers for perl
Handles applications that fork, with no performance cost
Immune from noise caused by profiling overheads and I/O
Program being profiled can stop/start the profiler
Generates richly annotated and cross-linked html reports
Captures source code, including string evals, for stable results
Trivial to use with mod_perl - add one line to httpd.conf
Includes an extensive test suite
Tested on very large codebases"

Además, su instalación es extremadamente sencilla, tan solo hay que ejecutar *cpanm Devel::NYTProf* y se instalará la última versión disponible en CPAN.

Por último, antes de comenzar con el estudio sobre la capacidad del profiler para detectar cuellos de botella, decir que NYTProf son dos profiler en uno. Uno es un profiler de sentencias que es invocado cuando perl pasa de una sentencia a otra, y el otro es un profiler de subrutinas que es invocado cuando perl llama a una o retorna de una. También señalizar que NYTProf no funciona con programas que utilizan hebras, con versiones de Perl anteriores a la 5.10.1 puede que alguna de sus funcionalidades falle, y si se tiene activada la variable de entorno *use_db_sub* las subrutinas "lvalue" no son analizadas.

Ya mencionadas sus características, pasamos a una demostración más a fondo y práctica de su funcionamiento.

2.1. Definición del sistema y especificación de los objetivos a conseguir

En primer lugar, vamos a definir las especificaciones del hardware que vamos a utilizar. El programa que vamos a optimizar se ejecutará en un portátil HP-G62, y como podemos ver en la *Ilustración 1* obtenida en cpu-world[4], el sistema tiene un procesador Intel® Core™ i3 M 350 de 2,27 GHz con 2 núcleos y cada núcleo tiene 2 hebras.

Intel Core i3 Mobile i3-350M SLBPK			
Part number:	CP80617004161AC	Comment:	
Measured Frequency:	2260 MHz	Submitted by:	CPU-World

General information	
Vendor:	GenuineIntel
Processor name (BIOS):	Intel(R) Core(TM) i3 CPU M 350 @ 2.27GHz
Cores:	2
Logical processors:	4
Processor type:	Original OEM Processor
CPUID signature:	20652
Family:	6 (06h)
Model:	37 (025h)
Stepping:	2 (02h)
TLB/Cache details:	64-byte Prefetching Data TLB0: 2-MB or 4-MB pages, 4-way set associative, 32 entries Data TLB: 4-KB Pages, 4-way set associative, 64 entries Instruction TLB: 2-MB or 4-MB pages, fully associative, 7 entries Instruction TLB: 4-KB pages, 4-way set associative, 64 entries Shared 2nd-level TLB: 4 KB pages, 4-way set associative, 512 entries

Ilustración 1. Características de la CPU obtenidas en cpu-world

En segundo lugar, el programa que hemos diseñado, para hacer la demostración de la utilidad del profiler, se encarga de encriptar un número que se le pasa al programa junto con un archivo el cual contiene una serie de números separados por ";", los cuales se utilizan para cifrar el número introducido. El método que se utiliza para encriptar realiza la siguiente serie de pasos:

1. En el primer paso recorre la lista de números que hay en el fichero (a partir de este momento llamaremos *vector* a esta serie de números ya que es donde están almacenados) empezando por el primero y avanzando de uno en uno. De modo que en cada iteración el número seleccionado va a actuar de pivote y, entonces, sumará al número que queremos encriptar cada valor del *vector* partiendo del pivote en adelante (incluyendo al pivote entre los números que se suman). Como se muestra en el siguiente código:

```
sub primera_parte_encriptacion{
        my($num, @vec) = @_;
        for(my $i=0; $i<@vec; $i++){
                for(my $j=$i; $j<@vec; $j++){
                        $num = suma($num, $vec[$j]);
                }
        }
        return $num;
}
```

2. En el segundo paso vuelve a recorrer el *vector* de uno en uno, modificando ahora el número que queremos encriptar (el cual ya está modificado debido al primer paso) de distinta manera, dependiendo de si la posición en el *vector* es el primer elemento del *vector*, si es múltiplo de dos, si es múltiplo de tres, si es múltiplo de siete o si no cumple ninguna de las anteriores condiciones.

```
sub segunda_parte_encriptacion{
        my($num, @vec) = @_;
        for(my $i=0;$i<@vec;$i++){
                if($i == 0){
                        $num = div_truncada($num, $vec[$i]);
                }
                elsif($i != 0 && ($i%7)==0){
                        $num = suma($num, $vec[$i]);
                }
                elsif($i != 0 && ($i%3)==0){
                        $num = resta($num, $vec[$i]);
                }
                elsif($i != 0 && ($i%2)==0){
                        $num = suma($num, $vec[$i]);
                }
                else{
                        $num = resta($num, $vec[$i]);
                }
        }
        return $num;
}
```

3. Finalmente, ordena los elementos del *vector* de menor a mayor con el algoritmo de ordenación burbuja, y multiplica el número que hay en la posición central menos uno del *vector*, al número a encriptar que ya ha sido modificado por los dos pasos mencionados anteriormente.

```
sub ordenacion_burbuja{
    my(@vec) = @_;
        for(my $i=0; $i<@vec; $i++){
                for(my $j=$i; $j<@vec; $j++){
                        if($vec[$i] > $vec[$j]){
                                my $temp = $vec[$j];
                                $vec[$j] = $vec[$i];
                                $vec[$i] = $temp;
                        }
```

```
                }
            }
            return @vec;
}
sub tercera_parte_encriptacion{
            my($num, @vec) = @_;
            @vec = ordenacion_burbuja(@vec);
            my $n = scalar(@vec);
            return (mull($num, $vec[resta(div_truncada($n, 2), 1)]));
}
```

El código completo del programa lo podemos ver en el *Anexo 1*, y como se puede apreciar, está implementado de una manera bastante ineficiente tanto en las funciones descritas anteriormente, como en el hecho de que para hacer operaciones básicas llama a funciones para que estas devuelvan el resultado de realizar la operación.

Por lo tanto, lo que queremos conseguir, es optimizar el código para que se ejecute lo más rápido posible, un minuto para 20000 elementos en el *vector* sería suficientemente bueno, ya que son simplemente operaciones con números.

De modo que el objetivo del experimento va a ser ver si con la ayuda de NYTProf para analizar el código y ver los cuellos de botella, podemos conseguir optimizar el programa, para que este se ejecute en un minuto o menos para un *vector* con 20000 elementos.

2.2. Analizando el código con NYTProf

Para empezar, vamos a estudiar la ejecución del código más detenida y detalladamente con el profiler NYTProf. Para ello lo primero que vamos a hacer es ejecutar el programa como se ejecutaría normalmente, diciendo que el número que se va a encriptar es el 20, lo cual no es muy relevante, y el archivo que tiene los números, va contener 20000 números, ya que es nuestro caso de estudio. Estos números han sido generados de manera aleatoria entre 0 y 1000000, pero de aquí en adelante siempre se utilizará este fichero con los mismos números, para que las mediciones no se vean afectadas por un cambio en los datos. Además, añadimos la orden time en la llamada a la ejecución del programa, para ver cuánto tiempo tarda en ejecutarse sin la instrumentación que añade el profiler, y después volvemos a ejecutar el programa, pero esta vez sí que le decimos que use el profiler, para ello ejecutamos la orden de la siguiente manera *time perl -d:NYTProf programa.pl número_encriptar archivo_con_números*, esto nos creará un archivo llamado *nytprof.out* en la carpeta que se haya ejecutado la orden anterior, mediante el cual podremos cargar el código HTML en un navegador, para ver el resultado del profiler.

En la *Ilustración 2*, podemos ver en la terminal de la derecha el tiempo que tarda en ejecutarse sin el profiler, y en la de la izquierda vemos el tiempo utilizando el profiler. Como se puede apreciar, NYTProf supone una carga bastante mayor a la hora de ejecutar el programa, ya que va realizando mediciones del tiempo que está en cada subrutina, y las veces que se ejecuta, entre otras cosas que vamos a ver más detalladamente más adelante.

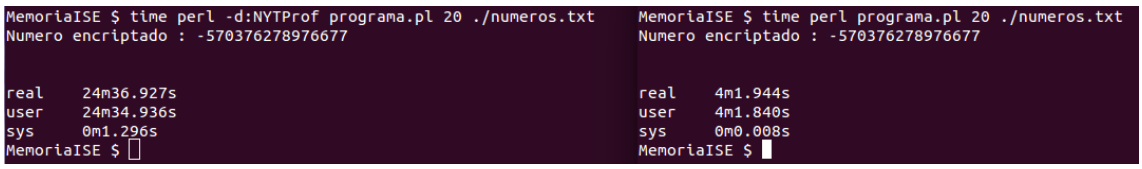


Ilustración 2. Tiempos de ejecución del programa con y sin el profiler

Una vez tenemos el archivo *nytprof.out*, para abrirlo usamos la orden *nytprofhtml --open*, la cual empezará a leer el archivo y acto seguido nos abrirá una ventana del navegador con los resultados del profiler (la parte de leer el archivo *nytprof.out* también puede tardar bastante rato, proporcionalmente a lo que haya tardado en ejecutarse el programa con la opción del profiler).

2.2.1. Flame Graph

Al completar de leer el archivo, se nos mostrará una ventana, con una gráfica similar a la que se muestra en la *Ilustración 3*, en la que vemos las funciones que más tiempo han tardado, y cuánto tiempo han tardado. Además, se muestran jerárquicamente, de manera que la función inferior llama a la que tiene encima. Por lo demás, se nos advierte de que los colores y la posición en el eje de la x no tienen ningún sentido, más allá de distinguir una función de otra.

Si miramos un poco más detalladamente el Flame Graph de la *Ilustración 3*, el cual está incluido en el *Anexo 2* más grande, podemos ver que el eje de la y representa la pila de llamadas a funciones, por lo que las funciones en la parte superior del gráfico son las que se está ejecutando, y así las funciones que hay debajo son las que han llamado a las que hay encima. En el eje x simplemente están ordenadas las funciones alfabéticamente, no quiere decir que el orden de ejecución sea de izquierda a derecha, para lo que nos sirve es para ver el largo de la caja, que simboliza la cantidad de tiempo que se está ejecutando dicha función. Finalmente, los colores solo distinguen una función de otra como se ha dicho anteriormente, y como dato curioso, le dan nombre al gráfico, ya que simulan una llama, en referencia a que estas funciones son las que hacen que la CPU esté caliente por estar en funcionamiento.

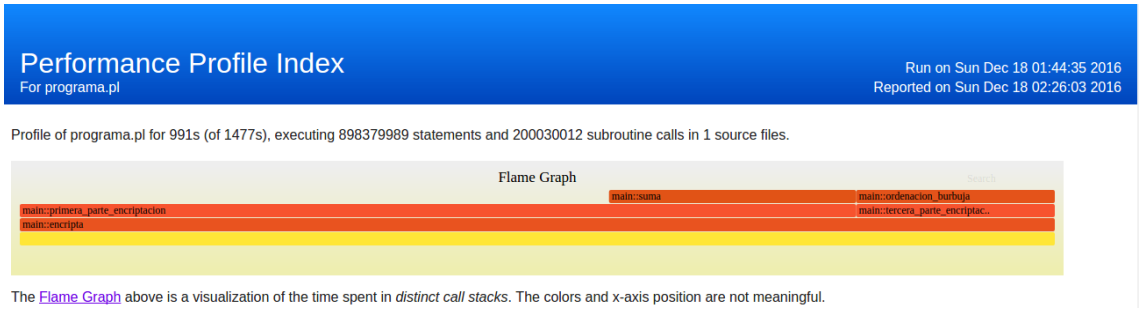


Ilustración 3. Flame Graph del código generado por el profiler

Además, el Flame Graph, contiene una información más exacta si ponemos el cursor encima de cualquiera de las cajas, de manera que nos dirá el tiempo de ejecución que ha tardado en microsegundos dicha subrutina, y el porcentaje del tiempo total que implican esos microsegundos. También si pinchamos en la caja, nos llevará a otra ventana en la que se nos muestra el código fuente de la subrutina, analizado línea por línea.

2.2.2. Tabla con el top 15 de subrutinas que más tiempo han empleado

A parte del Flame Graph, en la misma ventana, vemos que hay una tabla con el top 15 de las subrutinas que más tiempo del programa consumen, la cual se muestra en la *Ilustración 4*. En ella podemos observar diferente filas y columnas. Cada fila hace referencia a una subrutina del código, y cada columna a un parámetro de medición. La primera columna (*Calls*) muestra el número de llamadas que se han hecho a dicha subrutina. La segunda columna (*P*) hace referencia al número de líneas diferentes en el código en las que se llama a dicha subrutina. La tercera columna (*F*) dice el número de archivos diferentes en los que se llaman a esa función. En nuestro caso al estar todo el programa implementado en un único fichero, todas las funciones están en el mismo archivo, por lo que F siempre será 1 como se muestra en la *Ilustración 4*. La cuarta columna (*Exclusive Time*) es el tiempo de uso de la función en el programa sin contar las llamadas a subrutinas que esta hace internamente. La quinta columna (*Inclusive Time*) es la suma de la cuarta columna y del tiempo de uso de las subrutinas que llama la función en su ejecución. La última columna contiene el nombre de la subrutina (si es propia del lenguaje le añade opcode al final del nombre), y como se puede ver cada subrutina tiene un hipervínculo de forma que si pulsamos sobre ella nos mostrará el trozo de código que corresponde a la subrutina estando este analizado línea a línea.

Estos hipervínculos y su posibilidad de saltar de una subrutina a otra y ver su código analizado, es una de las características principales de NYTProf y por la cual es muy útil, cómodo y práctico.

Como podemos observar en la *Ilustración 4*, la tabla que se muestra está ordenada por la columna de tiempo exclusivo de mayor a menor. Así vemos que la subrutina que más tiempo a estado en uso, es *primera_parte_encriptacion* por lo que nos centraremos en reducirla. También se puede observar que la función suma usa una gran cantidad de tiempo, pero esto se debe al elevado número de veces que es llamada. Por último, podemos observar que *ordenacion_burbuja* también requiere de una elevada cantidad de tiempo, eso se debe a que el algoritmo de ordenación por burbuja tiene un orden de eficiencia de $O(n^2)$ siendo bastante ineficiente, por lo que si se aumenta el número de valores que se le pasan al programa para encriptar el número, este cada vez va a durar más tiempo, lógicamente, pero su tiempo va crecer de manera cuadrática lo cual puede provocar un cuello de botella, haciendo que el programa pierda la mayor parte de su tiempo en él.

Además, también podemos ver, que el tiempo exclusivo de *tercera_parte_encriptacion* y de *encripta*, es bastante grande, lo cual es debido a que internamente, *tercera_parte_encriptacion* llama a *ordenacion_burbuja*, y *encripta* internamente llama a las tres funciones que realizan la encriptación.

Subroutines

Calls	P	F	Exclusive Time	Inclusive Time	Subroutine
1	1	1	564s	801s	main::primera_parte_encriptacion
200018571	2	1	236s	236s	main::suma
1	1	1	190s	190s	main::ordenacion_burbuja
1	1	1	81.7ms	108ms	main::segunda_parte_encriptacion
1	1	1	24.0ms	24.0ms	main::CORE:readline (opcode)
11429	3	1	14.6ms	14.6ms	main::resta
1	1	1	9.67ms	190s	main::tercera_parte_encriptacion
1	1	1	3.80ms	991s	main::encripta
1	1	1	722µs	722µs	main::CORE:print (opcode)
2	2	1	36µs	36µs	main::div_truncada
1	1	1	32µs	32µs	main::CORE:open (opcode)
1	1	1	29µs	29µs	main::CORE:close (opcode)
1	1	1	5µs	5µs	main::mull

Ilustración 4. Tabla con las 15 funciones que más tiempo utilizan en el programa

2.2.3. Grafo de llamadas entre subrutinas

NYTProf también nos permite visualizar un grafo que muestra las llamadas entre las distintas subrutinas, el cual podemos ver en la *Ilustración 5*.

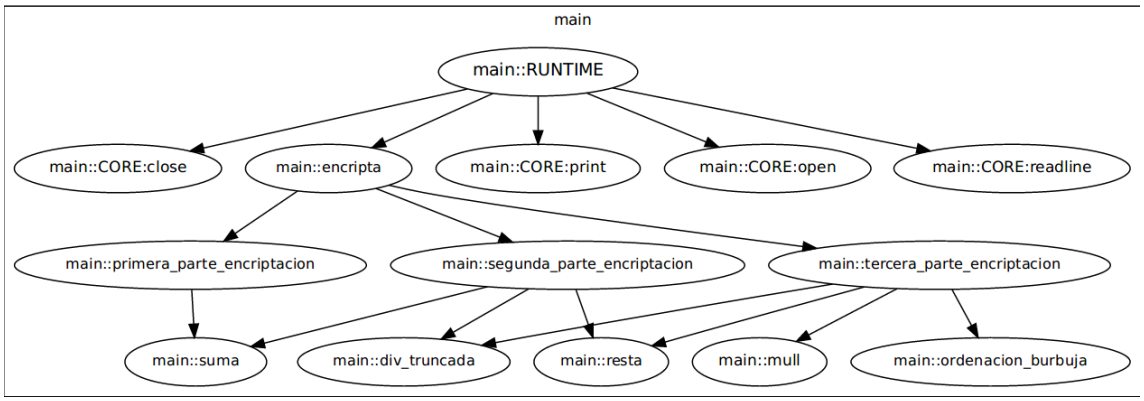


Ilustración 5. Grafo de llamadas entre subrutinas

2.3. Detección de cuellos de botella

Mirando la tabla de la *Ilustración 4*, vemos que hay dos cuellos de botella, uno situado en *primera_parte_encriptacion* y el otro en *ordenacion_burbuja*.

Si pulsamos sobre el hipervínculo de *primera_parte_encriptacion*, nos muestra lo que vemos en la *Ilustración 6*, donde podemos observar que la mayoría del tiempo se dedica a llamar a la función *suma* dentro de un bucle for anidado, por lo que, si conseguimos reducir el número de iteraciones del bucle for, optimizando el algoritmo que se emplea en la subrutina *primera_parte_encriptacion* también reduciremos el número de llamadas a la función *suma*.

Al pulsar sobre el hipervínculo de *ordenacion_burbuja*, nos enseña el contenido de la *Ilustración 7*, en la que el uso del tiempo mayoritariamente viene dado por el bucles for anidado, como en el caso anterior, que van recorriendo el vector una tras otra vez, dándonos un orden de eficiencia de $O(n^2)$. Por lo que si encontramos un algoritmo con mejor eficiencia podremos solucionar el cuello de botella.

21					`# spent 801s (564+236) within main::primera_parte_encriptacion which was called:`
					`#  once (564s+236s) by main::encripta at line 81`
					`sub primera_parte_encriptacion{`

22	1	2.88ms			`  my($num, @vec) = @_;`
23	1	251s	200010000	236s	`  for(my $i=0; $i<@vec; $i++){` `  # spent  236s making 200010000 calls to main::suma, avg 1µs/call`
24					`  for(my $j=$i; $j<@vec; $j++){`
25					`  $num = suma($num, $vec[$j]);`
26					`  }`
27					`  }`
28					
29	1	931µs			`  return $num;`
30					`}`

Ilustración 6. Código analizado de la subrutina primera_parte_encriptacion

55					`# spent 190s within main::ordenacion_burbuja which was called:`
					`#  once (190s+0s) by main::tercera_parte_encriptacion at line 73`
					`sub ordenacion_burbuja{`

56	1	2.86ms			`  my(@vec) = @_;`
57					
58	1	14.2ms			`  for(my $i=0; $i<@vec; $i++){`
59	20000	72.6s			`  for(my $j=$i; $j<@vec; $j++){`
60	200010000	57.7s			`  if($vec[$i] > $vec[$j]){`
61	99421415	17.2s			`  my $temp = $vec[$j];`
62	99421415	21.3s			`  $vec[$j] = $vec[$i];`
63	99421415	21.4s			`  $vec[$i] = $temp;`
64					`  }`
65					`  }`
66					`  }`
67					
68	1	3.78ms			`  return @vec;`
69					`}`
70					

71					`# spent 190s (9.67ms+190) within main::tercera_parte_encriptacion which was called:`
					`#  once (9.67ms+190s) by main::encripta at line 83`
					`sub tercera_parte_encriptacion{`

72	1	2.27ms			`  my($num, @vec) = @_;`
73	1	6.12ms	1	190s	`  @vec = ordenacion_burbuja(@vec);` `  # spent  190s making 1 call to main::ordenacion_burbuja`
74	1	4µs			`  my $n = scalar(@vec);`

75	1	1.01ms	3	30µs	`  return (mull($num, $vec[resta(div_truncada($n, 2), 1)]));`
					`  # spent  20µs making 1 call to main::div_truncada`
					`  # spent  5µs making 1 call to main::mull`
					`  # spent  5µs making 1 call to main::resta`

76					`}`

Ilustración 7. Código analizado de la subrutina tercera_parte_encriptacion y ordenacion_burbuja

Además de lo dicho anteriormente, también añadir que cada columna de la *Ilustración* 6 y de la *Ilustración* 7 describe un parámetro de medición. La primera columna sería el número de línea del fichero en el que está situado el código. La segunda es el número de veces que se declara esa línea en tiempo de ejecución. La tercera columna muestra el tiempo total que se está en esa línea. La cuarta columna dice el número de subrutinas que se llaman en esa línea. La quinta columna muestra el tiempo dentro de la subrutina o subrutinas llamada en esa línea, y, por último, la sexta columna muestra el código de la línea y en caso de que haya llamadas a una o varias subrutinas en esa línea, dice el tiempo total empleado haciendo el número de llamadas que se hayan hecho a cada subrutina.

Decir también que en la última columna las zonas donde pone *main::nombre_función* son hipervínculos a la zona del código donde empieza la función mencionada, y en las zonas donde pone *line número*, ocurre exactamente lo mismo, si pinchas sobre *line número* te lleva a esa línea del código. Esto es uno de los puntos fuertes de NYTProf, ya que permite moverse de una manera mucho más cómoda por el código y comprobar que hace cada subrutina a la que se llama en el momento en el que se llama.

3. Optimización del código

Ya detectados los cuellos de botella del programa, miramos a ver si tienen alguna solución.

Estudiando el algoritmo que hace *primera_parte_encriptación* vemos que en realidad no es necesario hacer un bucle for anidado recorriendo el vector por cada avance del primer bucle. Si no que se trata de sumarle a la variable *$num* el valor que hay en la posición del vector, multiplicando este valor por su posición en el vector más uno. De esta manera, podemos reducir el tiempo de esta subrutina a O(n), y con ello, debería estar solucionado el cuello de botella. El código resultante sería:

```
sub primera_parte_encriptacion{
            my($num, @vec) = @_;
            for(my $i=0; $i<@vec; $i++){
                $num = suma($num, mull($vec[$i],suma($i,1)));
            }
            return $num;
}
```

Para el segundo cuello de botella, el cual es la ordenación por burbuja, se puede realizar una primera mejora aprovechando las funcionalidades de Perl para intercambiar dos valores, lo cual se puede hacer en una línea, y sin necesidad de crear una tercera variable intermedia, pero aun así esto no nos mejoraría la eficiencia del algoritmo, por lo que seguiría tardando un tiempo muy similar.

De modo que, para solucionar este segundo cuello de botella, lo mejor es cambiar el algoritmo ordenación a otro que tenga una mejor eficiencia. Como Perl incluye una función sort para ordenar, definida en el lenguaje, es preferible usarla especificando que use el algoritmo mergesort que tiene un orden de eficiencia de O(nlog(n)), también se podría elegir quicksort, pero su orden de eficiencia en el peor caso es cuadrático, por lo que es mejor usar mergesort. Así el código de esta parte quedaría de la siguiente manera:

```
sub tercera_parte_encriptacion{
    my($num, @vec) = @_;
       @vec = sort { $a <=> $b } @vec;
       my $n = scalar(@vec);
       return (mull($num, $vec[resta(div_truncada($n, 2), 1)]));
}
```

También hay que añadirle al principio del fichero `use sort '_mergesort';`

Tras optimizar estas dos secciones del código (el código completo del programa optimizado se puede encontrar en el *Anexo 3*), la cuales forman los dos grandes cuellos de botella del programa. Este debería funcionar mucho más rápido. Vamos a comprobarlo en la siguiente sección.

4. Comprobación de la optimización con NYTProf

Una vez corregido el código, tenemos que comprobar que estás correcciones tienen el efecto deseado, para ello volvemos a ejecutar el programa con la orden time, sin y con el profiler, y, pasándole exactamente los mismos parámetros que se habían utilizado al ejecutar el código antes de optimizarlo.

En la *Ilustración 8*, se puede ver a la derecha la ejecución del programa sin el uso del profiler, y a la izquierda la ejecución añadiendo la instrumentación del profiler. Como se puede apreciar, los tiempos de ejecución en ambos casos se han reducido notablemente si lo comparamos con el resultado que dio en la *Ilustración 2*, mientras que el resultado de encriptar el número sigue siendo el mismo. Por lo que podemos asumir que la optimización del código se ha realizado correctamente.

Aunque con la ejecución del programa con la orden time ya se pueda apreciar bien que se ha reducido el tiempo que tarda en ejecutarse considerablemente, vamos a comprobar con el profiler las mejoras de tiempo de cada subrutina y comprobar cuáles son ahora los cuellos de botella.

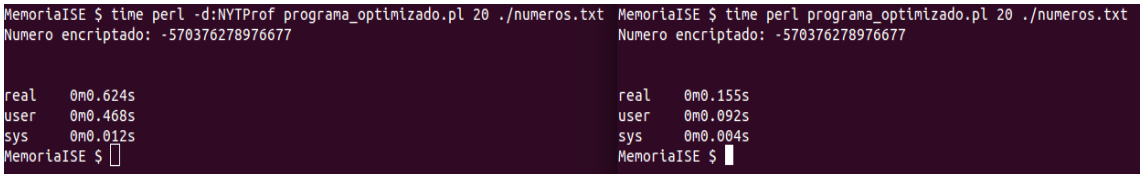


Ilustración 8. Tiempos de ejecución del programa con el código optimizado

Al ver el informe que se nos genera en HTML, lo primero que nos encontramos es que el Flame Graph, mostrado en la *Ilustración 9*, ha cambiado en comparación al de la *Ilustración 3*, y ahora se muestran más subrutinas, ya que algunas como la multiplicación han pasado a tener más uso, y, por lo tanto, suponen un mayor tiempo en la ejecución del código. También podemos ver que la subrutina *primera_parte_encriptacion* sigue siendo la que más tiempo consume, pero en cambio, podemos ver que ahora *segunda_parte_encriptacion* requiere de más tiempo que *tercera_parte_encriptacion*. Para ver el Flame Graph más grande, se puede consultar en el *Anexo 4*.

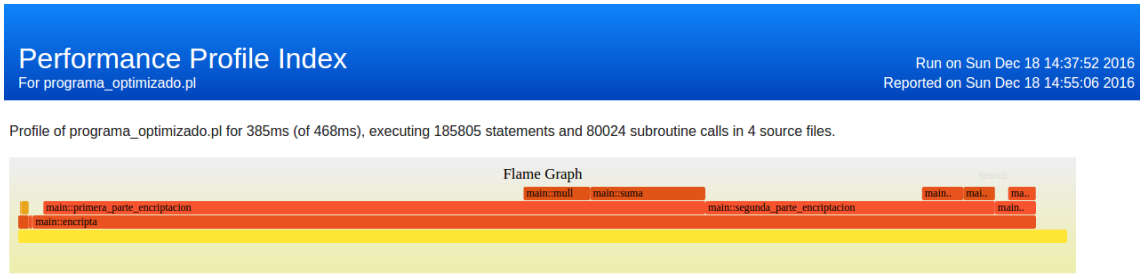


Ilustración 9. Flame Graph del código optimizado generado por el profiler

Para analizar la mejora más detalladamente podemos ver la tabla que nos muestra el top 15 de subrutinas más utilizadas y sus tiempos de ejecución, la cual es mostrada en la *Ilustración 10*, y si la comparamos con la de la *Ilustración 4*, veremos que hay una mejora muy significativa.

En primer lugar, lo primero que apreciamos en la *Ilustración 10* es que todos los tiempos de ejecución de las subrutinas se han reducido a milisegundos, ya no hay ninguna que requiera de medirla en segundos. Esto se ve especialmente bien, en las que más impacto tenían, las cuales eran *primera_parte_encriptacion* que utilizaba un bucle anidado con un coste en tiempo excesivamente alto y *tercera_parte_encriptacion* que utilizaba la ordenación por burbuja que ahora ha sido cambiada a la ordenación mergesort, la cual está en la subrutina propia del lenguaje *sort*.

Lo siguiente que podemos observar es que el número de llamadas a *suma* se ha reducido de 200018571 a 48571, a cambio de aumentar el número de llamadas a *mull* de 1 a 20001. Pero se ve claramente, que esté cambio ha sido muy positivo, ya que si sumamos los *inclusive time* de *suma* y *mull* de la *Ilustración 10*, veremos que no se aproxima ni remotamente a los 236 segundos que se tardaban antes, solamente en la función *suma*.

Por último, observamos que ahora *segunda_parte_encriptacion* es la segunda subrutina que más tiempo consume, y podríamos analizar su comportamiento de manera individual, pero como su tiempo realmente es pequeño, y el tiempo de ejecución del programa ya cumple con el objetivo definido, podemos dejar de optimizar en este punto.

Subroutines

Calls	P	F	Exclusive Time	Inclusive Time	Subroutine
1	1	1	176ms	243ms	main::primera_parte_encriptacion
1	1	1	79.6ms	106ms	main::segunda_parte_encriptacion
48571	2	1	53.9ms	53.9ms	main::suma
20001	2	1	24.5ms	24.5ms	main::mull
11429	3	1	15.2ms	15.2ms	main::resta
1	1	1	10.2ms	10.2ms	main::CORE:sort (opcode)
1	1	1	4.86ms	15.0ms	main::tercera_parte_encriptacion
1	1	1	3.94ms	369ms	main::encripta
1	1	1	2.67ms	2.71ms	sort::BEGIN@48
1	1	1	1.10ms	1.10ms	main::CORE:print (opcode)
1	1	1	655µs	3.85ms	main::BEGIN@1
1	1	1	419µs	470µs	sort::BEGIN@13
1	1	1	296µs	296µs	main::CORE:readline (opcode)
2	2	1	31µs	31µs	main::div_truncada
1	1	1	24µs	24µs	strict::CORE:regcomp (opcode)

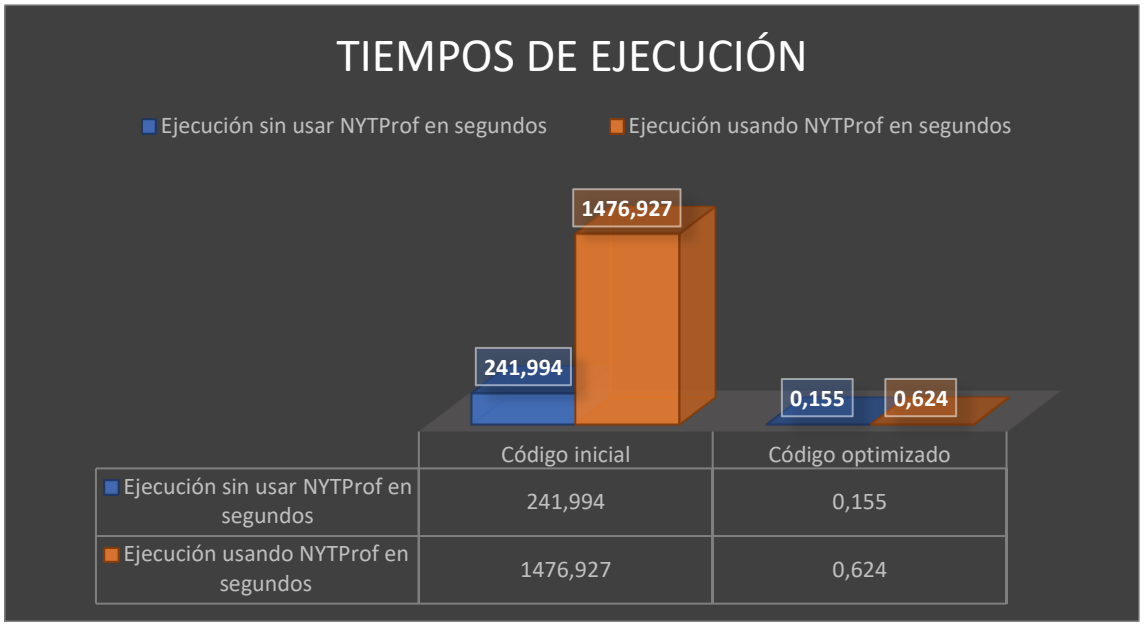
Ilustración 10. Tabla con las 15 funciones que más tiempo utilizan en el programa optimizado

5. Resultados del estudio

Tras obtener los resultados del análisis del profiler, y ver que el programa se ejecuta en menos de un minuto para 20000 elementos en el vector, podemos afirmar que el objetivo del estudio ha sido cumplido con éxito.

En la siguiente gráfica podemos ver la comparativa entre los tiempos reales de ejecución del programa antes de analizar su ejecución con NYTProf, y después de optimizar el código tras haberlo estudiado con el análisis de rendimiento proporcionado por el profiler.

Por lo resultados obtenidos podemos observar que el tiempo de ejecución del código inicial es aproximadamente 1560 veces mayor que el tiempo de ejecución del código optimizado, hablando en referencia a la ejecución del programa sin usar NYTProf que es como se va usar realmente.



6. Conclusiones

Como se ha demostrado en este estudio, el uso de un profiler, puede ayudar a mejorar los tiempos de ejecución de un programa considerablemente.

Con la ayuda de NYTProf se ha podido analizar la ejecución del código, viendo el número de llamadas a cada función, y lo que ha sido más importante, viendo el tiempo que se está en cada función desde que empieza a ejecutarse hasta que termina, contando y sin contar el tiempo que tarda en llamadas que haga a otras funciones internamente. Así se han podido detectar donde estaban los dos grandes cuellos de botella que tenía el programa, y, además, se ha podido analizar cada uno de ellos de manera individual, viendo un análisis del código de dichas funciones línea a línea.

El único problema de NYTProf, pero que es común a la mayoría de los profilers, es que como tiene que instrumentar el código para poder hacer las mediciones, hace que el programa tarde más en ejecutarse, pero si vemos los resultados obtenidos, parece razonable emplear un tanto más de tiempo para poder luego analizar bien el código, ya que así se van a encontrar rápidamente los cuellos de botella, y se podrá optimizar el código para mejorar su tiempo de ejecución notablemente.

Por lo tanto, podemos afirmar que NYTProf tiene una gran utilidad práctica para mejorar el rendimiento de los programas. Ya que nos permite visualizar el análisis del código tanto a nivel de funciones, como línea a línea.

7. Bibliografía

[1] Wall, L., Christiansen, T., Orwant, J., & D Foy, B. (2012). *Programming Perl* (4th ed.). O'Reilly.

[2] *The Comprehensive Perl Archive Network - www.cpan.org*. http://www.cpan.org/

[3] Lilja, D. (2000). *Measuring computer performance* (1st ed.). Cambridge University Press.

[4] CPU World, Especificaciones Intel Core i3-350M http://www.cpu-world.com/sspec/SL/SLBPK.html

Anexo 1. Código del programa a optimizar

```
sub suma{
    my($num1, $num2) = @_;
    return ($num1 + $num2);
}

sub resta{
    my($num1, $num2) = @_;
    return ($num1 - $num2);
}

sub mull{
    my($num1, $num2) = @_;
    return ($num1 * $num2);
}

sub div_truncada{
    my($num1, $num2) = @_;
    return (int($num1 / $num2));
}

sub primera_parte_encriptacion{
            my($num, @vec) = @_;
            for(my $i=0; $i<@vec; $i++){
                for(my $j=$i; $j<@vec; $j++){
                    $num = suma($num, $vec[$j]);
                }
            }
            return $num;
}

sub segunda_parte_encriptacion{
    my($num, @vec) = @_;
    for(my $i=0;$i<@vec;$i++){
        if($i == 0){
            $num = div_truncada($num, $vec[$i]);
        }
        elsif($i != 0 && ($i%7)==0){
            $num = suma($num, $vec[$i]);
        }
        elsif($i != 0 && ($i%3)==0){
            $num = resta($num, $vec[$i]);
        }
        elsif($i != 0 && ($i%2)==0){
            $num = suma($num, $vec[$i]);
        }
        else{
            $num = resta($num, $vec[$i]);
        }
    }

    return $num;
}

sub ordenacion_burbuja{
   my(@vec) = @_;
      for(my $i=0; $i<@vec; $i++){
          for(my $j=$i; $j<@vec; $j++){
                  if($vec[$i] > $vec[$j]){
              my $temp = $vec[$j];
                      $vec[$j] = $vec[$i];
                      $vec[$i] = $temp;
                  }
```

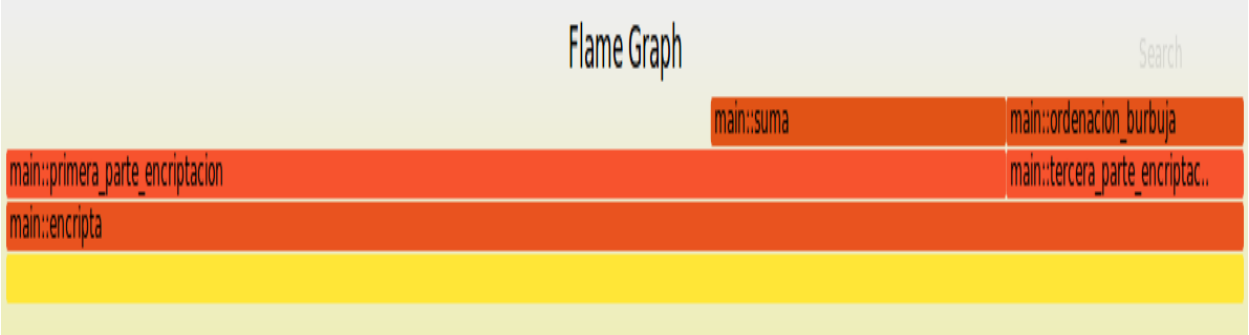
```
                }
            }
        return @vec;
}

sub tercera_parte_encriptacion{
    my($num, @vec) = @_;
        @vec = ordenacion_burbuja(@vec);
        my $n = scalar(@vec);
        return (mull($num, $vec[resta(div_truncada($n, 2), 1)]));
}

sub encripta {
    my($num,@vec) = @_;
    $num = primera_parte_encriptacion($num, @vec);
    $num = segunda_parte_encriptacion($num, @vec);
    $num = tercera_parte_encriptacion($num, @vec);
    return ($num);
}

my $numero = $ARGV[0];
open(ARCHIVO, '<', $ARGV[1]);
my $string = <ARCHIVO>;
my @vector = split(';', $string);
close(ARCHIVO);
print "Numero encriptado: " . encripta($numero,@vector) . "\n\n";
```

Anexo 2. Flame Graph del programa sin optimizar



Anexo 3. Código del programa optimizado

```
use sort '_mergesort';

sub suma{
    my($num1, $num2) = @_;
    return ($num1 + $num2);
}

sub resta{
    my($num1, $num2) = @_;
    return ($num1 - $num2);
}

sub mull{
    my($num1, $num2) = @_;
    return ($num1 * $num2);
}

sub div_truncada{
    my($num1, $num2) = @_;
```

```
    return (int($num1 / $num2));
}

sub primera_parte_encriptacion{
            my($num, @vec) = @_;
            for(my $i=0; $i<@vec; $i++){
                $num = suma($num, mull($vec[$i],suma($i,1)));
            }
            return $num;
}

sub segunda_parte_encriptacion{
    my($num, @vec) = @_;
    for(my $i=0;$i<@vec;$i++){
        if($i == 0){
            $num = div_truncada($num, $vec[$i]);
        }
        elsif($i != 0 && ($i%7)==0){
            $num = suma($num, $vec[$i]);
        }
        elsif($i != 0 && ($i%3)==0){
            $num = resta($num, $vec[$i]);
        }
        elsif($i != 0 && ($i%2)==0){
            $num = suma($num, $vec[$i]);
        }
        else{
            $num = resta($num, $vec[$i]);
        }
    }

    return $num;
}

sub tercera_parte_encriptacion{
    my($num, @vec) = @_;
       @vec = sort { $a <=> $b } @vec;
       my $n = scalar(@vec);
       return (mull($num, $vec[resta(div_truncada($n, 2), 1)]));
}


sub encripta {
    my($num,@vec) = @_;
    $num = primera_parte_encriptacion($num, @vec);
    $num = segunda_parte_encriptacion($num, @vec);
    $num = tercera_parte_encriptacion($num, @vec);
    return ($num);
}

my $numero = $ARGV[0];
open(ARCHIVO, '<', $ARGV[1]);
my $string = <ARCHIVO>;
my @vector = split(';', $string);
close(ARCHIVO);
print "Numero encriptado: " . encripta($numero,@vector) . "\n\n";
```

Anexo 4. Flame Graph del programa optimizado