



Optimización de scripts en bash para procesadores de varios núcleos

PENSAMIENTO PARALELO

No hace falta ningún enigma numérico para poder disfrutar de las maravillas del procesamiento paralelo. En este artículo describimos algunas técnicas simples para dotar de paralelización a los scripts bash.

POR BERNHARD BABLOK

Si queremos que parte de nuestro software ejecute una tarea de forma paralela, el primer reto consiste en dividir esa tarea en subtareas menores que puedan ser ejecutadas simultáneamente por la máquina. Hay librerías, como *OpenMP*, que ayudan al

programador a conseguir este tipo de paralelización.

Los scripts en bash no se suelen usar para manejar problemas numéricos, por lo que la mayoría de los programadores no llegan a plantearse dotarlos de paralelización. El venerable intérprete de bash, sin

embargo, se usa para trabajos que sí derivan en procesos potencialmente paralelos. Por ejemplo, los scripts en bash se suelen utilizar para procesar varios archivos de un modo específico.

El Listado 1 muestra una función del intérprete que procesa todos los argumen-

¿Tiene Sentido?

Antes de empezar a paralelizar scripts en bash a diestro y siniestro, conviene preguntarse si es razonable y posible. Sorprendentemente, esta pregunta tiene fácil respuesta. El comando *sar u P ALL 1 0*, que es parte del paquete *sysstat*, puede ayudarnos a responder. Para llevar a cabo la prueba, iniciamos el script bash desde un segundo intérprete. El comando *sar* muestra la carga de cada una de las CPUs de la máquina (Figura 1). Además del valor *%idle*, el valor *%iowait* también es interesante, indicándonos si el procesamiento se ha detenido debido a que el sistema está en espera de E/S o por alguna otra razón. Los valores de *sar* nos lo ponen más fácil: La paralelización solamente merece la

pena si algún procesador está parado mientras otros trabajan sin descanso (como se muestra en la Figura 1). En esta categoría se encuentran aplicaciones como las de conversión de imagen y música, que generan una considerable carga de CPU, o las de análisis de registros, que hacen uso de complejas expresiones regulares. Los procesos asociados a la E/S no son buenos candidatos. Aunque se pueda paralelizar la copia de 200 archivos, esta estrategia no supondrá un ahorro significativo de tiempo cuando el cuello de botella no es la CPU, sino el disco. Normalmente, cuando los pasos a llevar a cabo durante el procesamiento dependen los unos de los otros, o cuando el orden de éstos es impor-

tante, no queda más remedio que seguir un orden secuencial. Puede que nos viniera bien un algoritmo distinto, pero el método mostrado en este artículo no supone una ventaja significativa. Además, un administrador también debe saber que no siempre conviene elevar al máximo la carga de una máquina. Cuando se ha de continuar con los quehaceres diarios (correos, Internet, edición de texto, etc.) al mismo tiempo que se ejecutan tareas con un alto nivel de carga de CPU, la mejor opción suele ser el procesamiento secuencial en segundo plano, que sólo ocupa un núcleo, en vez de usar una alternativa que, aunque muy veloz, se adueñe del sistema entero.

Listado 1: Procesamiento en Serie

```
01 doSerie11() {
02 local item
03 for item in "$@" do
04 hazAlgo "$item"
05 done
06 }
```

tos del script uno a uno y pasa los resultados a un programa (*hazAlgo*). A partir de este escenario podemos deducir las ventajas de la aplicación de las técnicas de procesamiento paralelo.

Fuerza Bruta

Con unos cambios mínimos sobre el código del Listado 1, se obtiene la alternativa mostrada en el Listado 2, que hace uso del procesamiento paralelo. El Listado 2 inicia un proceso distinto por cada argumento. En la línea 6, el script espera a que todos sus procesos hijo terminen. Este método puede llegar a causarnos ciertos problemas: Si el sistema está sobrecargado por culpa de un número excesivamente alto de procesos, su sobrecarga se verá acentuada debido a la cantidad de cambios de contexto. En entornos con cantidades limitadas de memoria, la máquina también puede ralentizarse muchísimo por culpa de la continua alternancia entre los distintos procesos.

Listado 2: Procesamiento Paralelo Masivo

```
01 doMassiveParallel() {
02 local item
03 for item in "$@" do
04 hazAlgo "$item" &
05 done
06 wait
07 }
```

Listado 2a: Paralelo Entrada Serie

```
01 doMassiveParallel2() {
02 local item
03 while read item do
04 hazAlgo "$item" &
05 done
06 wait
07 }
08
09 creadorDeElementos |
doMassiveParallel
```

Bajo ciertas circunstancias, sin embargo, el simple uso de la paralelización puede resultar altamente beneficioso.

El Listado 2a es una variación del Listado 2. En este caso los argumentos no se conocen desde un primer momento; en lugar de eso, hay un proceso independiente (creadorDeElementos) que los crea secuencialmente (como podría ser el caso de un *find* ejecutándose sobre un sistema de archivos de gran volumen). Si las tasas de generación y de procesamiento, que dependen del número de procesadores disponibles, son muy similares, no experimentaremos una sobrecarga del sistema. De no serlo, necesitaríamos una solución más elaborada. El script del Listado 3 distribuye los argumentos dependiendo del número de procesadores (*PMAX*) de los que dispone el sistema. Si el proceso hace un uso intensivo de E/S, lo más lógico sería definir el número de procesadores con un valor superior a *PMAX* para permitir que un proceso trabaje mientras otro espera E/S.

Por desgracia, bash sólo usa arrays unidimensionales, lo que complica ligeramente las construcciones de las líneas 6 y

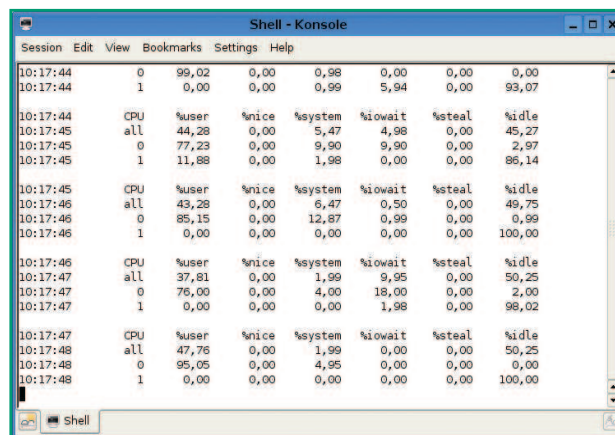


Figura 1: La salida de *sar* muestra la carga de todas las CPUs. La paralelización sólo tiene sentido cuando hay núcleos sobrecargados mientras otros no hacen nada.

13. El script crea una cadena larga por cada proceso, guardando los argumentos del proceso dentro de un elemento del array (líneas 5 a 9). Luego lanza *PMAX* procesos paralelos (líneas 11 a 14). Con la línea 12 se impide un procesamiento vacío (por ejemplo, sólo dos argumentos en una máquina quadcore), mientras que el *eval* de la línea 12 garantiza la correcta interpretación del entrecomillado de la línea 6.

Dispensador Dinámico

El diseño del Listado 3 es ideal cuando el tiempo medio de procesamiento por elemento no se ve afectado por fluctuaciones de consideración. Por desgracia, no siempre podemos basarnos en esto. Por ejemplo, a la hora de convertir pistas, una distribución inadecuada de pistas cortas y largas podría provocar que algu-

Listado 3: Paralelo con Balanceo de Carga

```
01 ${PMAX:=`ls ld /sys/devices/system/cpu/cpu* | wc l`}
02
03 doParallel() {
04 local items item currentProcess=0
05 for item in "$@" do
06 items[$currentProcess]="${items[$currentProcess]} \"$item\""
07 shift
08 let currentProcess=$(( (currentProcess+1)%PMAX ))
09 done
10
11 for (( currentProcess=0 currentProcess<PMAX currentProcess++ ))
do
12 [ n "${items[$currentProcess]}" ] &&
eval doSequentiell ${items[$currentProcess]} &
13 done
14 wait
15 }
```

nos procesos terminasen mucho antes que otros. Otra situación ante la cual el método podría resultar problemático, es la conversión de imágenes de cámaras digitales. Algunas cámaras crean JPGs o miniaturas además de los archivos en bruto. La mitad de los procesos terminarían significativamente más pronto debido a que, siguiendo el diseño de procesamiento, se asignarían todos los archivos en bruto a un proceso y las miniaturas a otro.

El método de procesamiento del Listado 3 también es ineficiente en casos en los que no se conocen de antemano todos los argumentos. Si los argumentos generados a posteriori por el script ocurren de forma secuencial, no tiene sentido esperar a que estén creados todos para distribuirlos entre los distintos procesos.

La solución a estos problemas pasa por usar varios procesos de trabajo junto con un dispensador dinámico. El dispensador acepta las tareas y las distribuye del modo más inteligente posible a los distintos procesos de trabajo. A diferencia de las soluciones mostradas anteriormente, en las que todos los procesos de trabajo

debían conocer sus argumentos antes de empezar, el dispensador se comunica con los procesos una vez que éstos han comenzado.

Los llamados FIFOs o tuberías se emplean en calidad de canales de comunicación. Al comenzar, el dispensador crea una tubería por cada proceso de trabajo y envía las nuevas tareas a ella (Figura 2). Otra tubería, que es compartida por el dispensador y los procesos de trabajo, se usa como canal de retorno. Si algún proceso queda ocioso, envía su ID por la tubería. El dispensador va leyendo de la tubería los IDs después de cada tarea y asigna la siguiente tarea al proceso de trabajo correspondiente.

En el Listado 4 se muestra una implementación del concepto. En las líneas 1 a 4, el programa define una serie de constantes, si éstas no existen ya. Normalmente, el usuario sólo define la variable `_cmd`. La función `dispensaTarea`, en las líneas 54 a 72, es la parte pública de la interfaz. La función comienza creando un directorio temporal para todas las tuberías (línea 55), al que el script se refiere como `controlDir`. El comando

`mkfifo` de la línea 58 se encarga de crear el canal de retorno.

La línea 59 requiere un mayor detenimiento. Aquí, la terminal abre el canal de retorno para escritura y lectura, a pesar de que sólo necesita lectura. El problema radica en que un acceso de sólo lectura bloquearía la llamada al sistema. En la función `iniciaProceso()` se da un problema similar. La función crea una tubería para cada proceso de trabajo (línea 37) y la abre con permisos de lectura y escritura (línea 40).

El `eval` de la línea 40 es necesario porque el analizador de bash procesa el redireccionamiento de entrada en una fase muy temprana (antes de la sustitución de variables). Así se explica que haya que escapar los símbolos de *menor que* y *mayor que*.

El Listado 4 sólo contiene funciones. Se trata de un archivo que otros scripts incluyen para usar la función `dispensaTarea` (ver el Listado 5).

Riesgos

En el script del Listado 4 también hay un par de cosas que merece la pena comentar: Por ejemplo, si hiciésemos un `kill` al proceso padre, quedarían huérfanos los

Listado 4: Dispensador Dinámico

```

01 ${DEBUG:=0}
02 ${_cmd:=echo}
03 ${PMAX:=`ls ld
   /sys/devices/system/cpu/cpu* |
   wc l`}
04 ${FDOFF:=4}
05
06 processWorkItem() {
07     eval $_cmd `"$1"`
08 }
09
10 processWorkItems() {
11     local line workerFifo="$1"
       dispatcherFifo="$2" id="$3" fd
12     exec 3<>"$dispatcherFifo"
13     while ! echo "$id" >&3 do
14         sleep 1
15     done
16     let fd=id+FDOFF
17     while true do
18         read r u $fd line
19         if [ $? ne 0 ] then
20             break
21         fi
22         if [ "$line" = "EOF" ] then
23             break
24         else
25             processWorkItem "$line"
26             while ! echo "$id" >&3 do
27                 sleep 1
28             done
29             fi
30             done
31             rm f "$workerFifo"
32         }
33     }
34     iniciaProceso() {
35         local i fd fifo
36         for (( i=0 i<PMAX ++i )) do
37             workerFifo=
38                 "$controlDir/worker$i"
39             mkfifo "$workerFifo"
40             let fd=i+FDOFF
41             eval exec $fd<\>
42                 "$workerFifo"
43             processWorkItems
44                 "$workerFifo" "$dispatcherFifo"
45                 "$i" &
46             done
47         }
48         detieneProceso() {
49             local i fifo
50             for (( i=0 i<PMAX ++i )) do
51                 fifo="$controlDir/worker$i"
52                 echo "EOF" > "$fifo"
53             done
54         }
55         dispatcherFifo=
56             "$controlDir/dispatcher"
57         mkfifo "$dispatcherFifo"
58         exec 3<>"$dispatcherFifo"
59         iniciaProceso
60         while read r u 0 line do
61             read u 3 idleId
62             echo "$line" >>
63                 "$controlDir/worker$idleId"
64             done
65         detieneProceso
66         rm f "$dispatcherFifo"
67         rm fr "$controlDir"
68     }
69 }
70 }
71 }
72 }

```

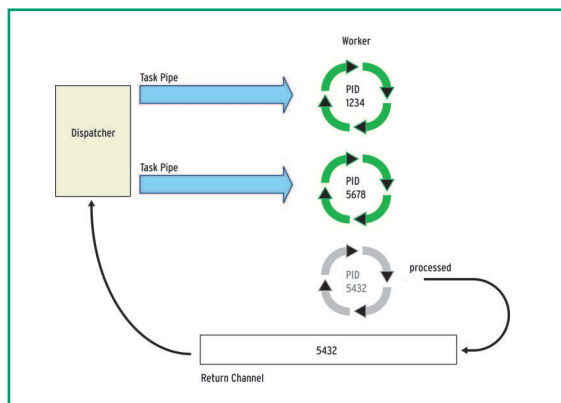


Figura 2: El dispensador y los procesos usan tuberías para comunicarse.

procesos de trabajo (aunque es un problema que se puede evitar definiendo una variable *timeout*). Además, si se tienen más de seis procesos, el script usará descriptores de archivo (números de canal) mayores que 9. Según el manual de bash, hay que tener cuidado con esto, puesto que bash ya podría estar usando estos descriptores internamente. Como solución alternativa, podemos modificar el valor de compensación (*offset*) para los números de canal (línea 4).

Son posibles otras implementaciones. Por ejemplo, el dispensador y los procesos de trabajo podrían emplear archivos para sus comunicaciones. El dispensador escribiría las tareas en archivos específicos para cada proceso de trabajo. Cada proceso comprobaría si hay un archivo para él, procesaría las tareas definidas en el

Listado 5: Dispensador en Funcionamiento

```
01 source workDispatcher
02 doDynamic() { _cmd="hazAlgo" local item for item in "$@"
do echo "$item" done | dispensaTarea }
```

archivo y lo borraría al terminar. El dispensador sabe así que un proceso está ocioso si no se encuentra en el disco el correspondiente archivo.

Normalmente esta solución es ineficiente debido a la necesidad de estar haciendo comprobaciones continuamente.

En el sitio web de Linux Magazine [1] hay disponible una versión ampliada del Listado 4. Esta versión extendida soporta las llamadas a *dispensaTarea* desde la línea de comandos:

```
$ dispensaTarea c "hazAlgo" archivo1
archivo2 [...]
```

La versión extendida incluye además comentarios e interruptores para una depuración opcional que permita al administrador monitorizar el script.

Aplicación a Redes de Máquinas

Aquel que no se conforme con la eficiencia del procesamiento paralelo en una máquina local, puede que quiera aplicar este principio a la red. En este caso, habría

un dispensador de primer nivel que usaría TCP/IP para comunicarse con los dispensadores de segundo nivel de las máquinas remotas. El dispensador de segundo nivel se comunicaría a su vez con los procesos de trabajo locales. Este método sólo es viable en redes seguras, claro está.

Conclusiones

Hemos visto cómo con un par de líneas de código se pueden utilizar las técnicas descritas a lo largo del artículo para dotar de paralelización a scripts del intérprete de comandos. Existen otros lenguajes de scripting que pueden beneficiarse de este método; sin embargo, algunos lenguajes ofrecen alternativas bastante superiores. Por ejemplo, Python usa la clonación explícita de procesos (*os.fork()*) conjuntamente con las tuberías (*os.pipe()*), lo que permite la elaboración de soluciones más eficientes y cercanas a la máquina.

RECURSOS

[1] Código fuente del Dispensador Dinámico: http://www.linux-magazine.es/Magazine/Downloads/50/Bash_paralelo

Ventajas

Hemos usado el método del dispensador dinámico explicado en este artículo en dos programas de pruebas. En el primer escenario, el script convierte 20 archivos RAW a formato TIFF en una máquina Intel Quadcore (Q9450 con una velocidad de reloj de 2.67GHz y una caché L2 de 2x 6MB).

Pasando todos los archivos a *ufrawbatch* al mismo tiempo, el programa tarda 132 segundos (iterando autónomamente sobre los archivos). El dispensador dinámico con *PMAX = 12* y *PMAX = 4* redujo el tiempo de ejecución de 134 segundos a 68 y 35 segundos respectivamente. La eficiencia de este método con cuatro procesadores es de alrededor del 95 por ciento. Dicho de otro modo, el tiempo de ejecución se ve reducido a casi una cuarta parte con el procesamiento paralelo.

La diferencia entre este escenario y la paralelización estática es marginal. El motivo de esta mínima diferencia es

que todos los archivos fuente son de aproximadamente el mismo tamaño, por lo que todos los procesadores comparten una carga similar.

En el segundo escenario se emplea otro método, de alta carga para la CPU, para la conversión de archivos WAV a MP3, aunque en condiciones más adversas esta vez. El script lee y escribe los archivos desde y hacia un servidor NFS con una conexión de 100MB. Algunas observaciones interesantes son que, primero, el método escala perfectamente (con tiempos de ejecución de 207, 107 y 55 segundos; es decir, un 94 por ciento de eficiencia con cuatro procesadores).

Además, la segunda pasada, en la que los archivos fuente se encontraban en la caché del servidor NFS, sólo difería ligeramente comparada con la prueba local. Finalmente, usando cinco procesos de trabajo en vez de cuatro, se obtienen resultados algo mejores.

El efecto de los procesos de trabajo adicionales es evidente cuando se trata de líneas de datos más "irregulares". La clasificación de los archivos WAV, ordenados por tamaño descendente, afectó en mayor medida al rendimiento. Al final del procesamiento sólo un procesador continuaba trabajando en el último archivo, lo que supone un efecto de descompensación sobre el tiempo de ejecución total. Aún se puede optimizar más, pero no son unos resultados malos. En el caso de complejas simulaciones con tiempos de ejecución de varias horas, o días, sí que sería necesario experimentar con otras optimizaciones. El balance de energía de una máquina trabajando a tope es ligeramente mejor que una involucrada en procesamientos secuenciales con un solo núcleo. Eso no quita que se ahorra más energía simplemente apagando la pantalla mientras la máquina está ocupada con trabajos de procesamiento complejo.