

Tarea 1 - Optimización estocástica

Erick Salvador Alvarez Valencia

CIMAT A.C.,
`erick.alvarez@cimat.mx`

Resumen En el presente reporte se hablará sobre la creación de un programa que permite ejecutar instancias de otros programas en paralelo, los cuales estarán dentro del cluster *El Insurgente* de CIMAT. Se hablará sobre la motivación del programa, así como la estructura del mismo y los resultados obtenidos ejecutándolo con un código de ejemplo.

Keywords: Cluster, cómputo paralelo, optimización estocástica

1. Motivación

A lo largo de la materia se realizarán algoritmos evolutivos, los cuales atacarán a problemas grandes y por lo tanto necesitarán mucho poder computacional para poder encontrar soluciones viables que minimicen o maximicen dichos problemas. Para esto anterior se utilizará el cluster con el que cuenta el CIMAT llamado *El Insurgente*.

Una vez desarrollados los algoritmos evolutivos es necesario subirlos y ejecutarlos en el cluster, y para ello un caso de interés puede ser el de ejecutar varias instancias de un mismo programa en diferentes nodos y con diferentes condiciones iniciales. Esto para que nuestro programa pueda explorar de una forma mejor el espacio de soluciones existentes y así poder encontrar mejores soluciones. Ahora esto anterior es una tarea que se debe automatizar debido a que podemos querer correr un gran número de instancias en los diferentes nodos con los que cuenta el cluster, y por ello se desarrolló un programa que automatiza esta tarea de una manera **greedy**.

El programa consiste en un planificador que creará un cierto número de procesos en paralelo, los cuales denominaremos procesos hijo, y dichos procesos se encargarán de ejecutar las instancias del o los programas que le indiquemos al planificador. El proceso padre se encargará de crear un cierto número de hijos (de acuerdo con el número de procesos que le indiquemos al inicio) e ir comprobando que terminen para así crear más hijos, eso hasta que se agoten las tareas pendientes y entonces el programa termina.

2. Estructura del planificador

En esta sección se explicará a grandes rasgos la estructura del programa antes mencionado.

El programa comienza con el proceso padre leyendo dos archivos los cuales cuentan con la información de las tareas a ejecutar y los nodos que usará, hay que destacar que dichas tareas (programas) ya se encuentran compiladas en el cluster y listas para correr. Una vez el programa lee los archivos, crea una cola con la información de las tareas, como no nos importa el orden en que estas son ejecutadas, la cola que se usa es la más básica. De la misma forma crea un arreglo de estructuras las cuales cuenta con la información de los nodos disponibles: nombre, si está o no está en uso, el pid del proceso hijo que está usando esa instancia del nodo. Posteriormente el programa entrará en un ciclo donde el padre se encargará de crear todos los hijos que se correspondan con el número de nodos solicitado en el archivo de tareas, al crear todos los hijos, cada que se crea un proceso hijo el padre hace un *usleep* para evitar crear muchos procesos en poca cantidad de tiempo y que el cluster los detectara como amenaza. El padre se pondrá en modo espera mediante la función *waitpid* en donde la ejecución se pone en pausa hasta que un proceso hijo termina, lo cual le permitirá al padre continuar su ejecución sabiendo que un hijo ha terminado, y de aquí el proceso padre podrá crear otro hijo para que ejecute otra tarea, esto se repite hasta que todos los procesos hijo hayan terminado y no queden tareas disponibles. Por otra parte, cuando se crea un proceso hijo, este sólo se encargará de usar la función *execl* para poder ejecutar un comando por consola, dicho comando se encargará de correr cierto programa en cierto nodo, una vez que el programa ejecutado en consola termina, el nodo hijo también lo hace, avisando al padre dicha terminación mediante el *waitpid*.

No hay que olvidar que una vez que se crea un proceso hijo, este tiene que saber qué tarea ejecutar y en qué nodo hacerlo, para ello utilizamos las ventajas de la copia de memoria hacia los procesos hijos. En general sabemos que cuando se crea un proceso hijo, se generan copias de las variables que ya existían hasta ese momento, la alteración de dichas copias de memoria no se ve reflejada entre procesos, sabiendo esto se crearon dos variables *string* las cuales, previo a la ejecución del *fork* se actualizaban con la siguiente tarea a ejecutar y algún nodo disponible, y con ello los procesos hijos ya podían formar la cadena que le pasarían a *execl* para ejecutarse.

Por último, cabe destacar que se implementaron algunas funciones para hacer *logging* con archivos de texto, esto para evitar mandar salidas por consola y que se saturara la red.

3. Pruebas

Para el momento de probar el planificador se creó un programa que se encargaba de llenar un vector de 100 elementos con números aleatorios y posteriormente sumar dichos elementos, esta tarea se realizaba N veces donde $N \in [100000, 10000000]$.

El programa se ejecutó 1000 veces dentro de 20 procesos del cluster, para lo cual se utilizaron nodos c-1-x. Ahora para generar el archivo de tasks que tuviera la mil tareas se utilizó un programa generador donde eligió N en el rango indicado

anteriormente.

A continuación se mostrarán los resultados de la ejecución del planificador usando 20 procesos del cluster, 10 en el nodo c-1-18 y los otros 10 en el c-1-22.

```
top - 15:33:04 up 85 days, 4:10, 1 user, load average: 8.57, 4.01, 1.57
Tasks: 397 total, 11 running, 386 sleeping, 0 stopped, 0 zombie
%Cpu(s): 40.8 us, 0.1 sy, 0.0 ni, 59.1 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 32910284 total, 198924 free, 455660 used, 32255700 buff/cache
KiB Swap: 93180 total, 53368 free, 39812 used, 31759216 avail Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
13911	user_de+	20	0	13288	1620	1472	R	100.0	0.0	0:07.40	test
13951	user_de+	20	0	13288	1596	1448	R	100.0	0.0	0:05.08	test
13972	user_de+	20	0	13288	1672	1524	R	100.0	0.0	0:04.61	test
14008	user_de+	20	0	13288	1572	1424	R	100.0	0.0	0:03.54	test
13960	user_de+	20	0	13288	1564	1420	R	100.0	0.0	0:04.97	test
13984	user_de+	20	0	13288	1712	1564	R	100.0	0.0	0:04.31	test
13999	user_de+	20	0	13288	1556	1408	R	100.0	0.0	0:03.72	test
14020	user_de+	20	0	13288	1660	1512	R	60.5	0.0	0:01.82	test
14032	user_de+	20	0	13288	1568	1424	R	34.2	0.0	0:01.03	test
14045	user_de+	20	0	13288	1644	1496	R	6.6	0.0	0:00.20	test
13850	user_de+	20	0	41296	3696	2864	R	0.7	0.0	0:00.08	top
8	root	20	0	0	0	0	S	0.3	0.0	37:03.84	rcu_sched
440	root	20	0	109548	52068	51564	S	0.3	0.2	0:59.96	systemd-jo+
861	message+	20	0	43456	3728	2968	S	0.3	0.0	1:11.67	dbus-daemon
896	root	20	0	67824	6004	5264	S	0.3	0.0	0:09.59	sshd
1	root	20	0	55040	6784	5056	S	0.0	0.0	1:15.84	systemd
2	root	20	0	0	0	0	S	0.0	0.0	0:00.26	kthreadd

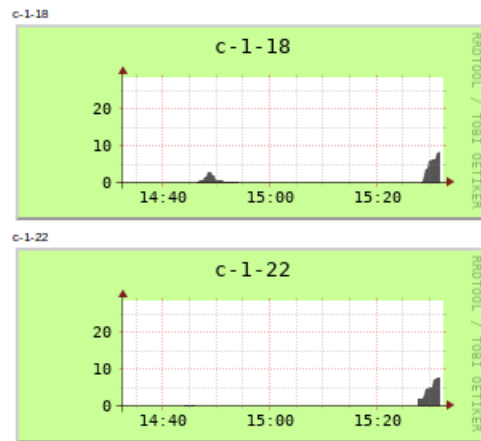
(a) Figura 1. Estado actual del nodo c-1-18 durante la ejecución del planificador.

```
top - 15:33:39 up 85 days, 4:10, 1 user, load average: 8.27, 4.43, 1.81
Tasks: 384 total, 11 running, 373 sleeping, 0 stopped, 0 zombie
%Cpu(s): 40.0 us, 0.2 sy, 0.0 ni, 59.8 id, 0.1 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 32910284 total, 223912 free, 418904 used, 32267468 buff/cache
KiB Swap: 93180 total, 87580 free, 5600 used, 31826576 avail Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
7076	user_de+	20	0	13288	1564	1420	R	100.0	0.0	0:13.18	test
7188	user_de+	20	0	13288	1572	1424	R	100.0	0.0	0:07.56	test
7173	user_de+	20	0	13288	1600	1456	R	100.0	0.0	0:08.06	test
7243	user_de+	20	0	13288	1712	1564	R	100.0	0.0	0:05.53	test
7271	user_de+	20	0	13288	1524	1380	R	100.0	0.0	0:03.99	test
7296	user_de+	20	0	13288	1640	1496	R	48.8	0.0	0:01.47	test
7334	user_de+	20	0	13288	1528	1380	R	19.6	0.0	0:00.59	test
7335	user_de+	20	0	13288	1528	1380	R	19.6	0.0	0:00.59	test
7348	user_de+	20	0	13288	1712	1564	R	12.6	0.0	0:00.38	test
7357	user_de+	20	0	13288	1592	1448	R	11.0	0.0	0:00.33	test
445	root	20	0	121844	56988	56484	S	0.7	0.2	0:49.50	systemd-jo+
1	root	20	0	55392	7276	5168	S	0.3	0.0	1:04.66	systemd
758	message+	20	0	43456	4244	3488	S	0.3	0.0	0:56.15	dbus-daemon
788	root	20	0	44800	5124	4156	S	0.3	0.0	0:30.44	systemd-lo+
7284	user_de+	20	0	41296	3636	2840	R	0.3	0.0	0:00.03	top
2	root	20	0	0	0	0	S	0.0	0.0	0:00.18	kthreadd
3	root	20	0	0	0	0	S	0.0	0.0	0:52.57	ksoftirqd/0

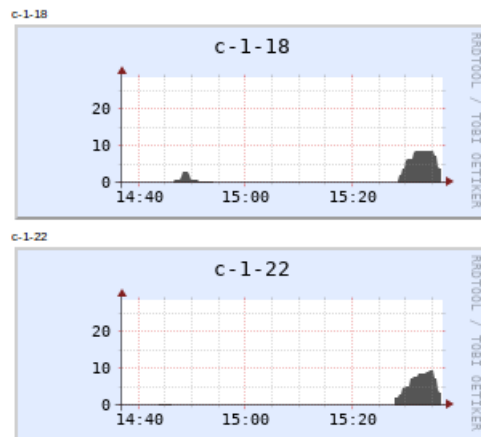
(b) Figura 2. Estado actual del nodo c-1-18 durante la ejecución del planificador.

Podemos ver en las Figuras 1 y 2 que para cada nodo se estaban usando en paralelo 10 de los 24 procesos que tienen por lo que se puede comprobar que el planificador si lanzó los procesos de manera adecuada.



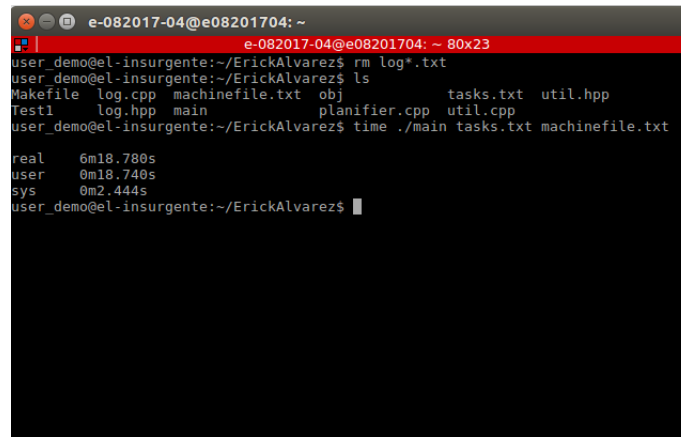
(c) Figura 3. Carga de la CPU de los nodos c-1-18 y c-1-22 obtenida de ganglia.

En la Figura 3 podemos observar que en efecto se veía reflejado el aumento en la carga del CPU para los nodos c-1-18 y c-1-22 en la página de ganglia.



(d) Figura 4. Carga de la CPU de los nodos c-1-18 y c-1-22 obtenida de ganglia.

De la misma forma se puede ver que una vez terminada la ejecución del programa, se liberaron los recursos de los dos nodos.

A terminal window with a red title bar containing the text "e-082017-04@e08201704: ~ 80x23". The terminal shows a user named "user_demo" at a host "el-insurgente" in the directory "~/ErickAlvarez". The user runs "rm log*.txt", then "ls", which lists files: "Makefile", "log.cpp", "machinefile.txt", "obj", "tasks.txt", and "util.hpp". Below these are "Test1", "log.hpp", "main", "planifier.cpp", and "util.cpp". The user then runs "time ./main tasks.txt machinefile.txt". The output shows timing statistics: "real 6m18.780s", "user 0m18.740s", and "sys 0m2.444s".

```
e-082017-04@e08201704: ~
user_demo@el-insurgente:~/ErickAlvarez$ rm log*.txt
user_demo@el-insurgente:~/ErickAlvarez$ ls
Makefile  log.cpp  machinefile.txt  obj      tasks.txt  util.hpp
Test1     log.hpp  main             planifier.cpp  util.cpp
user_demo@el-insurgente:~/ErickAlvarez$ time ./main tasks.txt machinefile.txt

real    6m18.780s
user    0m18.740s
sys     0m2.444s
user_demo@el-insurgente:~/ErickAlvarez$
```

(e) Figura 5. Tiempo que tardó el planificador en ejecutar las 1000 tareas.

En la Figura 5 se muestra que el planificador tardó al rededor de 6 minutos ejecutando las mil tareas en los 20 procesos del cluster, para obtener el tiempo transcurrido se utilizó la función *time* de bash.

Notas y comentarios. Como conclusión tenemos que el planificador funcionó como se esperaba y que se estará usando al rededor del semestre para ejecutar en el cluster los algoritmos evolutivos desarrollados, esto de forma paralela.