Computación de altas prestaciones

Práctica 1

Pedro Sánchez de la Muela Garzón

Laura de Paz Carbajo

Práctica 1 Parte0-S1 : Instalar cluster Rocks utilizando Maquina Virtuales

EJERCICIOS de la práctica1-parte0-sección1 (15%)

Ejercicio 1: Pare y reinicie el cluster de manera ordenada. Enumere los pasos que ha realizado para conseguirlo.

Desde el front end reiniciamos los nodos del clúster:

rocks run host reboot

Ejercicio 2: Suponga que uno de los nodos ha fallado. Elimine el nodo de las bases de datos de rocks y reinstale uno nuevo en su lugar manteniendo el nombre del antiguo, aunque no se mantenga necesariamente su IP. Enumere los pasos que ha realizado para conseguirlo.

Supongamos que ha fallado el nodo 0. Primero lo eliminamos del Sistema de colas:

insert-ethers –remove compute-0-0

rocks list host

rocks sync config

insert-ethers --hostname compute-0-0

Ejercicio 3: Realice un script que compruebe el estado de los nodos de computo y sea capaz de devolver el porcentaje de nodos que están activos en el cluster. Adicionalmente la salida debe indicar para cada nodo, su nombre, su dirección IP y su estado. Considere que un nodo no esta activo si la columna `states` indica `au`.

Usando el script *'script-ej3-p0.py'* con un solo nodo active, obtenemos la siguiente salida:

```
[bigdata@clusterlab2 Desktop]$ python script-ej3-p0.py
Numero de nodos = 1
Numero de nodos no activos = 0
Porcentaje de nodos activos = 1/1


Listado de Nodos
compute-0-0 10.11.12.254 ACTIVO_
```

Ejercicio 4: Realice un script que añada usuarios al cluster rocks y devuelva como salida el nombre y el password de cada usuario. El nombre debe ser usuBDxx, donde xx se incrementa de 01 a 20 y utilice una política de passwords razonable.
¿Cómo puede verificar cuantos usuarios existen en el cluster y que nombre tienen?

No hemos sido capaces de gestionar la creación de usuarios por los problemas que supone el interactuar con el prompt a la hora de introducir nuevas contraseñas. Dejamos nuestro intento en el script *'script-ej4-p0.py'*.

Ejercicio 5: Realice las siguientes pruebas de conexión entre el las MV.
5.1.- Conexión desde el anfitrión al cluster (sería el equivalente a un acceso en remoto);
$ ssh bigdata@192.168.182.100

Realizamos primero la prueba de conexión al cluster a través de ssh desde el host, obteniendo timed out, lo que asociamos a que el host en nuestro caso sea Windows 11, y que probablemente por medidas del firewall no nos deja conectarnos, ya que la VM tiene conexión a internet.
Sin embargo mediante el comando ping comprobamos que tenemos conexión entre la VM y el host.

5.2.- Desde la sesión anterior conectarse a un nodo de computo:
$ ssh compute-0-0

Este apartado tampoco lo hemos podido hacer por lo explicado en el apartado anterior, aunque sí que podemos conectarnos a los nodos desde la VM con lo que sería posible si Windows nos dejase conectarnos por ssh.

5.3.- ¿Qué sudera si intentamos la conexión desde el anfitrión a los nodos de computo?
$ ssh bigdata@10.11.12.252

Esto no sería posible ya que la red 10.12.XX.XX que estamos usando para conectarnos entre la VM y los nodos es local a la VM y por tanto el host no tiene acceso directo, hay que hacerlo a través de la VM como en el apartado anterior.

Práctica 1 - parte0-S2 : Ejecución y Planificación de tareas en cluster Rocks

EJERCICIOS: Práctica 1 – Parte0 – Sección2 (15%)

Ejercicio 6: Con un editor cree un fichero con el siguiente contenido
compute-0-0
compute-0-1
compute-0-2
y denomínelo **maquinasMPI.txt**

Pruebe el siguiente comando:

$ /opt/openmpi/bin/mpirun -np 10 –machinefile maquinasMPI.txt hostname

y explique como varía el resultado respecto al comando:

/opt/openmpi/bin/mpirun -np 10 hostname

Responda a las siguientes preguntas:

- ¿Se ha ejecutado algún proceso en el frontend?
- ¿Cómo se puede ejecutar parte de los procesos en el frontend?
- Varíe el número de procesos e intente deducir el reparto de tareas que se utiliza.

Comprobamos que se puede ejecutar el comando *'/opt/openmpi/bin/mpirun -np 10 hostname'*:

```
[bigdata@clusterlab2 ~]$ /opt/openmpi/bin/mpirun -np 10 hostname
clusterlab2.ii.uam.es
clusterlab2.ii.uam.es
clusterlab2.ii.uam.es
clusterlab2.ii.uam.es
clusterlab2.ii.uam.es
clusterlab2.ii.uam.es
clusterlab2.ii.uam.es
clusterlab2.ii.uam.es
clusterlab2.ii.uam.es
clusterlab2.ii.uam.es
```

Ahora hacemos que desde un archivo se lea en qué máquinas se ha de ejecutar obteniendo el siguiente resultado:

```
[bigdata@clusterlab2 Desktop]$ /opt/openmpi/bin/mpirun -np 10 -machinefile maqui
nasMPI.txt hostname
compute-0-1.local
compute-0-1.local
compute-0-1.local
compute-0-0.local
compute-0-0.local
compute-0-0.local
compute-0-0.local
compute-0-2.local
compute-0-2.local
compute-0-2.local
```

Podemos intuir que hace un balance de carga para que cada nodo tenga que ejecutar el mismo número de procesos. Podemos ver que todos los procesos se han ejecutado en los

nodos de cómputo porque así lo hemos indicado en el fichero, si incluyéramos en dicho fichero *'clusterlab2.ii.uam.es'* algunos de los procesos se ejecutarían en el frontend.

Ejercicio 7: Compile los ejemplos de mpi disponibles en `/opt/mpi-tests/e` indique como funcionan.

El comando *'opt/openmpi/bin/mpirun -np 6 -machinefile maquinasMPI.txt /opt/mpi-tests/bin/mpi-ring'* comprueba el estado de la red que conecta los nodos de cómputo.

```
bigdata@clusterlab2 Desktop]$ /opt/openmpi/bin/mpirun -np 6 -machinefile maquin
asMPI.txt /opt/mpi-tests/bin/mpi-ring
rocess 0 on compute-0-0.local
rocess 2 on compute-0-2.local
rocess 4 on compute-0-1.local
rocess 3 on compute-0-0.local
rocess 1 on compute-0-1.local
rocess 5 on compute-0-2.local
rocess 5 on compute-0-2.local:successfully sent (1048576) bytes to id (0)
rocess 2 on compute-0-2.local:successfully sent (1048576) bytes to id (3)
rocess 0 on compute-0-0.local:successfully sent (1048576) bytes to id (1)
rocess 0 on compute-0-0.local:successfully received (1048576) bytes from id (5)
rocess 1 on compute-0-1.local:successfully sent (1048576) bytes to id (2)
rocess 2 on compute-0-2.local:successfully received (1048576) bytes from id (1)
rocess 5 on compute-0-2.local:successfully received (1048576) bytes from id (4)
rocess 4 on compute-0-1.local:successfully sent (1048576) bytes to id (5)
rocess 1 on compute-0-1.local:successfully received (1048576) bytes from id (0)
rocess 3 on compute-0-0.local:successfully sent (1048576) bytes to id (4)
rocess 3 on compute-0-0.local:successfully received (1048576) bytes from id (2)
rocess 4 on compute-0-1.local:successfully received (1048576) bytes from id (3)
```

Y el comando *'/opt/openmpi/bin/mpirun -np 6 -machinefile maquinasMPI.txt /opt/mpi-tests/bin/mpi-verify'* parece que simplemente verifica que los nodos estén en funcionamiento.

```
[bigdata@clusterlab2 Desktop]$ /opt/openmpi/bin/mpirun -np 6 -machinefile maquinasMPI.txt /opt/mpi-tests/bin/mpi-verify
Process 1 on compute-0-1.local
Process 2 on compute-0-2.local
Process 3 on compute-0-0.local
Process 0 on compute-0-0.local
Process 4 on compute-0-1.local
Process 5 on compute-0-2.local
```

Ejercicio 8: Crear una cola denominada colapares.q basándose en la cola all.q que tenga solo los nodos con nombres compute-0-x, siendo x par. Compruebe su funcionamiento.

Indique los comandos utilizados para su creación y el proceso para comprobar su funcionamiento.

Editamos un archivo de texto para contener la información de nuestra cola personalizada y la añadimos mediante el comando *'qconf -Aq colapares.txt'*

```
qname                 colapares.q
hostlist              compute-0-0,compute-0-2
seq_no                0
load_thresholds       np_load_avg=1.75
suspend_thresholds    NONE
nsuspend              1
suspend_interval      00:05:00
priority              0
min_cpu_interval      00:05:00
processors            UNDEFINED
qtype                 BATCH INTERACTIVE
ckpt_list             NONE
pe_list               make mpich mpi orte
rerun                 FALSE
slots                 1,[compute-0-0.local=1],[compute-0-2.local=1]
```

Ejercicio 9: Cree nuevas colas de acuerdo a criterios que le parezcan significativos, por ejemplo, cola1core para máquinas con un solo procesador y cola2core para máquinas con dos procesadores. Para ello reinstale el nodo 1 con 2 cores y cree una nuevo nodo compute-0-3 con 2 cores. Realice pruebas para comprobar el funcionamiento de las colas creadas.

Indique los comandos utilizados para su creación y el proceso para comprobar su funcionamiento.

Realizamos los mismos pasos que en el ejercicio anterior, pero esta vez solo añadiendo el nodo 'compute-0-1' al que le hemos asignado dos cores en lugar de uno.

```
qname                 cola2core.q
hostlist              compute-0-1
seq_no                0
load_thresholds       np_load_avg=1.75
suspend_thresholds    NONE
nsuspend              1
suspend_interval      00:05:00
priority              0
min_cpu_interval      00:05:00
processors            UNDEFINED
qtype                 BATCH INTERACTIVE
ckpt_list             NONE
pe_list               make mpich mpi orte
rerun                 FALSE
slots                 1,[compute-0-1.local=1]
```

Práctica 1 Parte 1 (35%)
Vector processing and SIMD

You must write a report answering the questions proposed in each exercise, plus the requested files. Submit a zip file through Moodle. Check submission date in Moodle (deadline is until 11:59 pm of that date).

- **Exercise 1:**
  - Identify your CPU model and list the supported SIMD instructions.

Modelo CPU

```
e399596@2-16-2-16:~/UnidadH/CAP_P1$ cat /proc/cpuinfo | grep model
model           : 158
model name      : Intel(R) Core(TM) i5-7500 CPU @ 3.40GHz
```

Instrucciones SIMD soportadas

```
e399596@2-16-2-16:~/UnidadH/CAP_P1$ cat /proc/cpuinfo | grep flags
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts
acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bt
s rep_good nopl xtopology nonstop_tsc cpuid aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx smx est
 tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave
 avx f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb invpcid_single pti ssbd ibrs ibpb stibp tpr
_shadow vnmi flexpriority ept vpid ept_ad fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid mpx rds
eed adx smap clflushopt intel_pt xsaveopt xsavec xgetbv1 xsaves dtherm ida arat pln pts hwp hwp_notify
 hwp_act_window hwp_epp md_clear flush_l1d arch_capabilities
```

  - Provide the GCC version and explain the report you get

Versión de GCC: 11.2.0

Al compilar simple2.c con el comando del enunciado obtenemos el siguiente esquema de vectorización:

```
e399596@2-16-2-16:~/UnidadH/CAP_P1$ gcc -O3 -march=native -fopt-info-vec-optimized -o simple2 simple2.c
simple2.c:26:21: optimized: loop vectorized using 32 byte vectors
simple2.c:19:17: optimized: loop vectorized using 32 byte vectors
```

Esto nos indica que ha sido posible vectorizar los bucles for de las líneas 19 y 26 en vectores de 32 bytes, pero no ha sido posible vectorizar el bucle de la línea 25, que encapsula el de la línea 26.

```
e399596@2-16-2-16:~/UnidadH/CAP_P1$ gcc -O3 -march=native -fno-tree-vectorize -fopt-info-vec-optimized -o simple2_no_vec simple2.c
```

En este siguiente caso (con la flag -fno-tree-vectorize) estamos indicando al compilador que queremos desactivar la vectorización, por lo que obviamente no nos muestra ningún vector vectorizado.

- Explain the main differences between both assembly codes (vectorized and non-vectorized) focused on the SIMD instructions generated by the compiler.

Generamos el assembly code .s con los comandos proporcionados en el enunciado y al echarles un vistazo rápido se aprecia que el que usa vectorización es más largo. Fijándonos más en el código generado y comparando los dos archivos nos damos cuenta que esto debe a que al hacer procesar los datos con vectores se necesita inicializarlos y usar funciones específicas y finalmente combinar los datos de los vectores para obtener un resultado final. En cambio, cuando no se aplica la vectorización, se lleva a cabo una computación más directa que no necesita inicialización ni operaciones específicas de vectores. Aunque el programa sin vectorización tenga menor número de funciones en assembly, tarda más en ejecutarse porque se realizan más iteraciones por bucle.

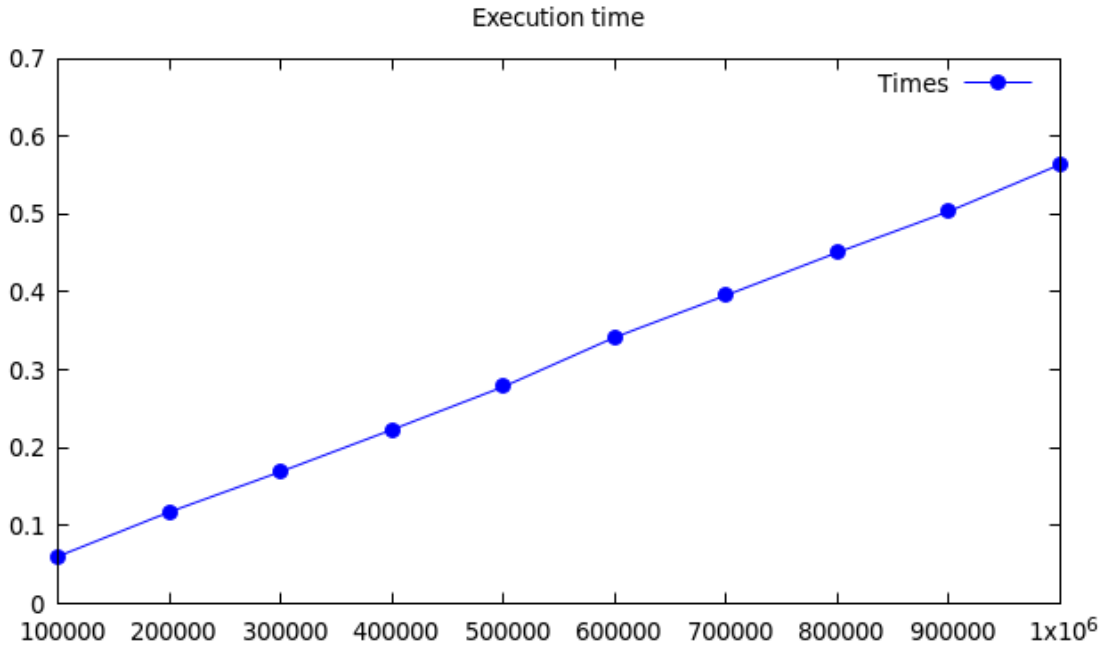
- **Exercise 2:**
  - Provide the source code of *simple2_intrinsics.c* after the vectorization of the loops. Explain how you have carried out the vectorization of the code.

Código adjunto en el .zip entregado.

La primera parte de la vectorización se llevó a cabo siguiendo las instrucciones de la práctica.

Para la segunda parte, inicializamos un vector de tipo *__m256d* con los cuatro primeros elementos de cada vector: *a = {1,2,3,4}* y *b {0,1,2,3}*. Al inicializar los elementos de los vectores de 4 en 4, en lugar de sumar 1 a cada elemento, sumaremos 4, por ello creamos el vector *increment = {4,4,4,4}*, que iremos sumando a los vectores *a* y *b* en cada iteración.

- Compare the execution time for different values of NUMBER_OF_TRIALS: from 100.000 to 1.000.000 in steps of 100.000. Plot the results in a graph. Discuss the results.



Podemos observar que el resultado de la ejecución de *simple2_intrinsics* para diferentes tamaños de la variable *NUMBER_OF_TRIALS* es claramente lineal. Es un resultado que esperábamos obtener ya que aunque estamos aumentando el número de veces que se ejecuta la operación, no estamos alterando el tamaño del array sobre el que se itera para realizar el cálculo *m*a+b*. Así que como es razonable, el tiempo de ejecución de este programa crecerá linealmente con el número de trials.

- **Exercise 3:**
  - The program includes two loops. The first loop (indicated as Loop 0) iterates over the arguments applying the algorithm to each of them. The second loop (indicated as Loop 1) computes the grey scale algorithm. Is this loop optimal to be vectorized? Why?

El bucle 1 no se puede vectorizar ya que el acceso a memoria al iterar con los 2 bucles (width y height) se está llevando a cabo de forma "desordenada", es decir, no se está accediendo a bloques de memoria contiguos. Lo que provoca que le cueste al procesador mucho más procesar las imágenes y el acceso a memoria, cediendo así tiempo de ejecución pero asegurando un resultado más preciso.

  - Provide the source code of the auto-vectorized version of the code. Explain the changes in the code to help the compiler to vectorize the loop.

Código adjunto en el .zip entregado con el nombre *greyscale2.c*.

Como vimos hace 2 años en la asignatura de Arquitectura de ordenadores, un compilador no trabaja como una persona por lo que el orden de ejecución de los bucles que más natural nos puede parecer a nosotros puede no ser el más eficiente. Es lo que ocurre en el programa greyscale.c de Moodle y es necesario cambiar el orden en el que se ejecuta el bucle 1, en el que se procesa cada pixel.

Para ello simplemente cambiamos los índices del bucle exterior y el bucle anidado para que recorre la imagen primero verticalmente (height) y luego horizontalmente (width). Como consecuencia hay que cambiar los índices que se le pasan a la función *getRGB* y cuando se guarda el resultado en la nueva imagen.

De esta manera, al recorrer la imagen se accederá a direcciones de memoria consecutivas, es decir, el programa no tendrá que saltar de bloque en bloque de memoria tan frecuentemente y será posible la autovectorización.

  - Provide the source code after manually vectorizing the code. Explain your solution.

Código adjunto en el .zip entregado con el nombre *greyscale_intrinsics.c*.

Para evitar problemas de ejecución por operaciones no soportadas en ciertos ordenadores, implementaremos la vectorización con AVX de 256 bits en lugar de con operaciones AVX-512.

Procesaremos 4 pixels simultáneamente pero dividiéndolos en 2 vectores para que las operaciones intrinsics puedan manejar su tamaño. Cargamos primero los 2 vectores en variables de tipo __m128i, luego los convertimos a enteros de 32 bits y a continuación a floats para poder procesar la multiplicación y tener los valores preparados para obtener el espectro gris. Multiplicamos a continuación cada uno de los vectores de floats obtenidos por un vector constante donde

hemos guardado los coeficientes. Lo último que necesitamos hacer es sumar cada uno de los resultados obtenidos de la multiplicación de cada pixel y reordenar los pixels. Para ello utilizamos la operación *hadd* y recordamos con la operación *permut* y *shuffle*.

- Fill in a table with time and speedup results compared to the original version and auto-vectorized version for images of different resolutions (SD, HD, FHD, UHD-4k, UHD-8k). You must include a column with the fps at which the program would process. Discuss the results.

	SD	HD	FHD	4k	8k
time original	0.010079	0.021417	0.041485	0.167452	0.898945
time vectorized	0.018995	0.027415	0.041187	0.097297	0.347312
speedup	0.530613	0.781215	1.007235	1.721040	2.588292
speedup %	-54.06	-78.12	0.72	72.10	58.83
fps	52.65	36.48	24.28	10.28	2.88

Hemos obtenido esta tabla a partir de la ejecución de greyscale.c original y el del segundo apartado de este ejercicio, donde el compilador vectoriza el bucle automáticamente. El speedup y los frames per second se han calculado de la siguiente manera:

$$speedup = \frac{time\ original}{time\ auto{-}vectorized} \qquad fps = \frac{1}{time\ auto{-}vectorized}$$

Con estos datos podemos ver que este programa sería suficiente para reproducir un video en calidad SD e incluso HD a tiempo real ya que se consiguen procesar más de 30 imágenes por segundo. Sin embargo, esto no es posible para imágenes de más calidad como FHD, 4k y 8k. En este caso, se tendrían que procesar las imágenes antes de la reproducción para que de tiempo a convertir a blanco y negro las imágenes antes de que esta se tengan que reproducir y no se pare el vídeo. En el caso de las imágenes de 8k, al ser tan pesadas, prácticamente habría que tenerlas todas o casi todas procesadas para poder obtener una reproducción fluida o mejorar mucho el programa *greyscale*.