PRÁCTICA 3

Entrenamiento con ARM: OpenMP y OpenCV

David Martínez García

Alejandro Gea Belda

Parte 1 - Paralelismo en sistemas de memoria centralizada: Introducción a OpenMP

Tarea 1.1: Estudio del API OpenMP

OpenMP es una interfaz de programación de aplicaciones (API) para la programación paralela en diferentes plataformas. La API define únicamente el estándar que hay que seguir para poder utilizar la interfaz. OpenMP está disponible en múltiples plataformas como Microsoft, Windows o Linux. En nuestro caso, su implementación está integrada en cualquier compilador de C y C + + de GNU con una versión 4.2 o superior.

Tarea 1.2: Estudio de OpenMP

1. Explique qué hace el código que se añade a continuación.

El código dado hace un par de operaciones sencillas. En primer lugar define tres vectores de tamaño 100 y asigna a los vectores a y b un valor a cada posición igual al índice del vector. Posteriormente, el programa suma en un bucle los vectores a y b, guardando el resultado en el vector c e imprimiendo por pantalla el resultado obtenido.

```
float a[N], b[N], c[N];

for (i=0; i < N; i++)
    a[i] = b[i] = i * 1.0;
```

```
for (i=0; i<N; i++)
{
    c[i] = a[i] + b[i];
    printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
}
```

En esencia, el programa hace estas operaciones básicas. Sin embargo, se utiliza OpenMP para que la ejecución del código sea simultánea en varios hilos de nuestro equipo.

2. Delimite las distintas regiones OpenMP en las que se está expresando paralelismo.

El uso de OpenMP modifica el código utilizando diferentes instrucciones que se identifican con el tipo #pragma que permiten modificar como se ejecuta nuestro programa. Una vez modificado el código, hay que asegurarse de que se compila añadiendo la opción -fopenmp para que el programa se ejecute correctamente.

La primera instrucción que se utiliza es `#pragma omp parallel shared (a ,b ,c, nthreads, chunk) private (i, tid)` y realiza varias instrucciones. La parte de `#pragma omp parallel` especifica que todo el código que hay dentro de la instrucción se puede ejecutar de forma paralela. La cláusula '**shared**' se utiliza para especificar qué variables van a ser compartidas por los diferentes hilos mientras que la cláusula '**private**' se utiliza para definir qué variables serán privadas en cada hilo. Esto implica que cada hilo

tendrá su propia copia de las variables definidas como private mientras que las variables definidas en shared serán comunes para todos los hilos..

```
#pragma omp parallel shared(a,b,c,nthreads,chunk) private(i,tid)
{
    ...
    ...
    ...
}
```

La siguiente instrucción `omp_get_thread_num()` devuelve el identificador del hilo que invoca la función. Si ese hilo es el número 0, el programa devuelve el número de hilos activos mediante la función `omp_get_num_threads()`.

La segunda instrucción `#pragma omp for schedule(dynamic, chunk)` realiza varias instrucciones. La parte `#pragma omp for` permite que un bucle for se pueda ejecutar mediante varios hilos en paralelo. El fragmento `schedule(dynamic, chunk)` es una opción que se usa para controlar cómo se dividen las iteraciones del bucle entre los hilos. En este caso, la opción '**dynamic**' especifica que las iteraciones se asignan de manera dinámica a los hilos, es decir, que las iteraciones se irán asignando automáticamente a medida que estén disponibles para su uso. El parámetro '**chunk**' especifica cuántas iteraciones se asignan a la vez a cada hilo que, en nuestro caso, son 10 iteraciones por hilo.

```
#pragma omp for schedule(dynamic,chunk)
for (i=0; i<N; i++)
{
    c[i] = a[i] + b[i];
    printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
}
```

3. Explique con detalle qué hace el modificador de OpenMP schedule. ¿Cuáles pueden ser sus argumentos? ¿Qué función tiene la variable chunk en el código? ¿A qué afecta?

El modificador schedule en OpenMP se utiliza para controlar cómo se distribuyen las iteraciones de un bucle entre los hilos en un equipo paralelo. Esta permite controlar la asignación de trabajo a los hilos y, en consecuencia, la forma en que se aprovechan los recursos del equipo. El modificador schedule tiene varios argumentos que te permiten especificar cómo se asignan las iteraciones del bucle a los hilos:

- static: Con `schedule(static, chunk)`, se divide el bucle en bloques de un tamaño fijo llamado "chunk" y se asigna a los hilos en un patrón estático y equitativo. Esto significa que las iteraciones se distribuyen de manera uniforme entre los hilos al principio, y cada hilo obtiene un conjunto de iteraciones contiguas.

- dynamic: Con `schedule(dynamic, chunk)`, las iteraciones se asignan de manera dinámica a los hilos a medida que los hilos terminan su trabajo. Cada hilo obtiene un

conjunto de iteraciones, y cuando un hilo termina su conjunto, recibe otro conjunto de iteraciones disponibles.

- guided: Con `schedule(guided, chunk)`, se asignan inicialmente bloques grandes de iteraciones a los hilos, pero a medida que los hilos completan su trabajo, reciben bloques más pequeños.

- auto: Con `schedule(auto)`, OpenMP permite al compilador determinar la estrategia de planificación más adecuada.

- runtime: Con `schedule(runtime, chunk)`, implica que la política de planificación y el tamaño de "chunk" se determinarán en tiempo de ejecución, generalmente mediante la configuración de variables de entorno o directivas de control específicas proporcionadas por la implementación de OpenMP.

4. ¿Qué función tiene el modificador de OpenMP dynamic en el código?

Como se ha explicado anteriormente, al utilizar el modificador '**dynamic**' en nuestro código hace que el bucle for que se encarga de sumar los vectores a y b se ejecute en hilos diferentes. Estos hilos van recibiendo conjuntos de instrucciones conforme resuelven las instrucciones recibidas anteriormente sin esperar a que todos los hilos terminen de ejecutar sus órdenes.

```
Thread 0 starting...
Thread 0: c[0]= 0.000000
Thread 0: c[1]= 2.000000
Thread 0: c[2]= 4.000000
Thread 0: c[3]= 6.000000
Thread 0: c[4]= 8.000000
Thread 0: c[5]= 10.000000
Thread 0: c[6]= 12.000000
Thread 0: c[7]= 14.000000
Thread 0: c[8]= 16.000000
Thread 0: c[9]= 18.000000
```

```
Thread 1 starting...
Thread 1: c[50]= 100.000000
Thread 1: c[51]= 102.000000
Thread 1: c[52]= 104.000000
Thread 1: c[53]= 106.000000
Thread 1: c[54]= 108.000000
Thread 1: c[55]= 110.000000
Thread 1: c[56]= 112.000000
Thread 1: c[57]= 114.000000
Thread 1: c[58]= 116.000000
Thread 1: c[59]= 118.000000
```

5. Investigue qué pasa si no declara como privadas las variables i y tid

Si no se declaran las variables i y tid como privadas en la región paralela, OpenMP tratará de manera predeterminada a estas variables como compartidas entre todos los hilos, lo que puede llevar a resultados incorrectos o comportamiento inesperado.

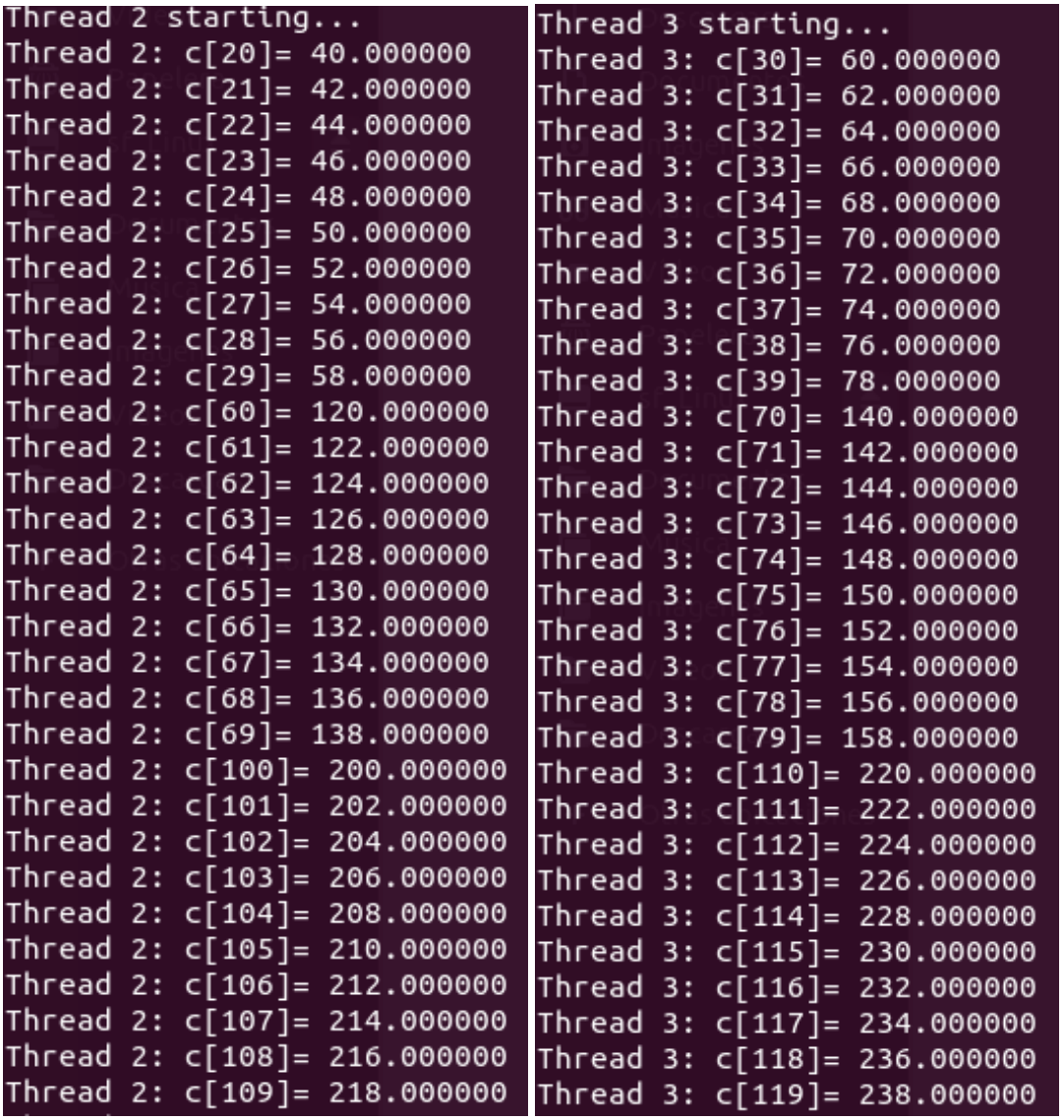
Dentro de una región paralela, OpenMP asume que las variables que no se declaran como privadas son compartidas, lo que significa que todos los hilos tendrán acceso y pueden modificar las mismas variables. Esto puede conducir a problemas de concurrencia, ya que múltiples hilos podrían estar intentando acceder o modificar las mismas variables i y tid al mismo tiempo.

Tarea J1. [Reto JEDI borde exterior]

Para emplear un planificador estático con OpenMP hay que cambiar el parámetro de nuestro pragma vector `#pragma omp for schedule(dynamic, chunk)` por el parámetro '**static**'. Para mejor visualización del código, se ha cambiado el tamaño de los vectores a 120 para que el reparto de la ejecución de nuestro código sea equitativo para los 4 procesadores que se están usando.

```
Thread 0 starting...
Thread 0: c[0]= 0.000000
Thread 0: c[1]= 2.000000
Thread 0: c[2]= 4.000000
Thread 0: c[3]= 6.000000
Thread 0: c[4]= 8.000000
Thread 0: c[5]= 10.000000
Thread 0: c[6]= 12.000000
Thread 0: c[7]= 14.000000
Thread 0: c[8]= 16.000000
Thread 0: c[9]= 18.000000
Thread 0: c[40]= 80.000000
Thread 0: c[41]= 82.000000
Thread 0: c[42]= 84.000000
Thread 0: c[43]= 86.000000
Thread 0: c[44]= 88.000000
Thread 0: c[45]= 90.000000
Thread 0: c[46]= 92.000000
Thread 0: c[47]= 94.000000
Thread 0: c[48]= 96.000000
Thread 0: c[49]= 98.000000
Thread 0: c[80]= 160.000000
Thread 0: c[81]= 162.000000
Thread 0: c[82]= 164.000000
Thread 0: c[83]= 166.000000
Thread 0: c[84]= 168.000000
Thread 0: c[85]= 170.000000
Thread 0: c[86]= 172.000000
Thread 0: c[87]= 174.000000
Thread 0: c[88]= 176.000000
Thread 0: c[89]= 178.000000
```

```
Thread 1 starting...
Thread 1: c[10]= 20.000000
Thread 1: c[11]= 22.000000
Thread 1: c[12]= 24.000000
Thread 1: c[13]= 26.000000
Thread 1: c[14]= 28.000000
Thread 1: c[15]= 30.000000
Thread 1: c[16]= 32.000000
Thread 1: c[17]= 34.000000
Thread 1: c[18]= 36.000000
Thread 1: c[19]= 38.000000
Thread 1: c[50]= 100.000000
Thread 1: c[51]= 102.000000
Thread 1: c[52]= 104.000000
Thread 1: c[53]= 106.000000
Thread 1: c[54]= 108.000000
Thread 1: c[55]= 110.000000
Thread 1: c[56]= 112.000000
Thread 1: c[57]= 114.000000
Thread 1: c[58]= 116.000000
Thread 1: c[59]= 118.000000
Thread 1: c[90]= 180.000000
Thread 1: c[91]= 182.000000
Thread 1: c[92]= 184.000000
Thread 1: c[93]= 186.000000
Thread 1: c[94]= 188.000000
Thread 1: c[95]= 190.000000
Thread 1: c[96]= 192.000000
Thread 1: c[97]= 194.000000
Thread 1: c[98]= 196.000000
Thread 1: c[99]= 198.000000
```



En las imágenes se puede observar como, al utilizar el modificador static, los 4 hilos que están trabajando reciben 10 instrucciones cada uno y hasta que no terminan todos no vuelven a recibir instrucciones. Gracias a este modo de trabajar, podemos saber de antemano qué instrucciones exactas va a recibir cada hilo de nuestro procesador antes de ejecutar el código.

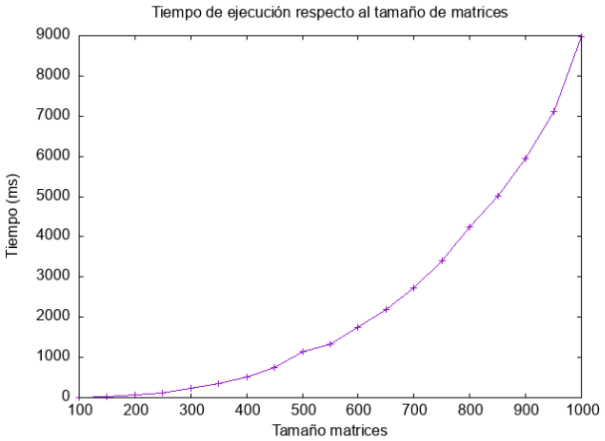
Tarea 1.3: Paralelización de una aplicación con OpenMP

Apartado 0:

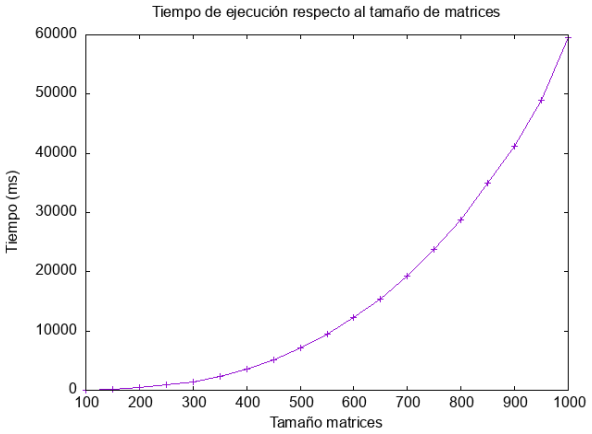
Se ha modificado el programa pedido para que muestre por pantalla dos columnas de datos, en una se mostrará el tamaño de las matrices con las que se realizan las operaciones y en la otra se muestra el tiempo de ejecución del programa. Después se ha utilizado gnuplot para representar la gráfica resultante. Para medir el tiempo se utilizará la función de OpenMP omp_get_wtime(), que devuelve el tiempo de ejecución del programa.

```
alumno@VDIUbuntuEPS2022:~/Escritorio/Empotrados/Practica3$ gcc -fopenmp -o matri
x matrix.c
alumno@VDIUbuntuEPS2022:~/Escritorio/Empotrados/Practica3$ matrix > matrix.steps
alumno@VDIUbuntuEPS2022:~/Escritorio/Empotrados/Practica3$ gnuplot matrix.gpi
```

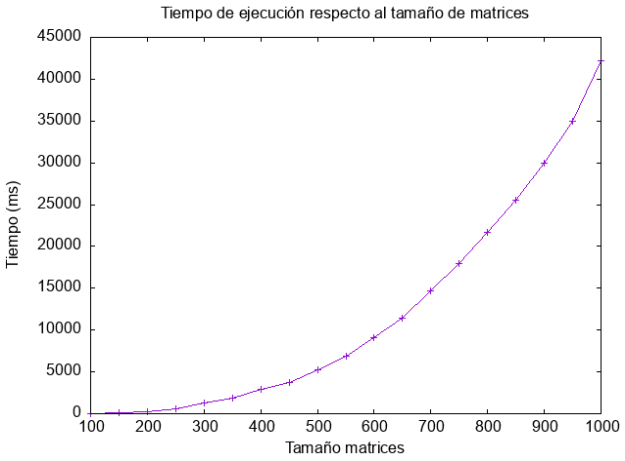
Las gráficas obtenidas para las diferentes arquitecturas son las siguientes:



Arquitectura x86-64



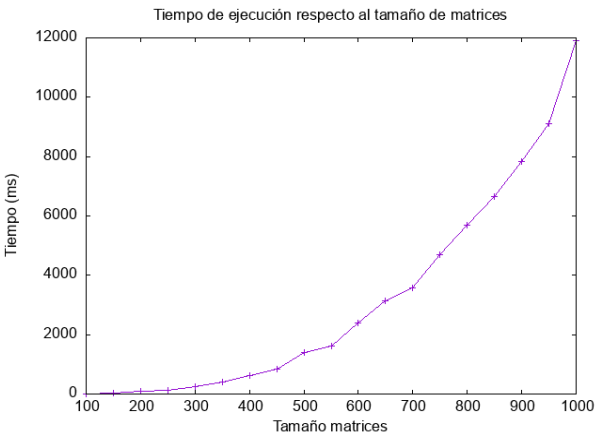
Arquitectura ARM



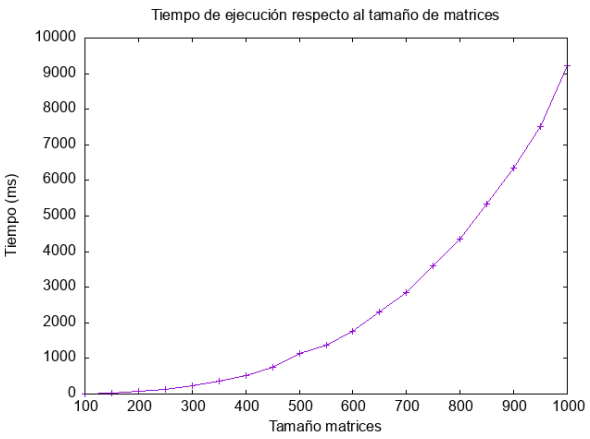
Arquitectura Risc V

Se puede observar que en ARM el programa tarda considerablemente más en ejecutarse para tamaños de matrices grandes. Por otro lado, en risc v el programa tarda menos en ejecutarse que para ARM pero el tiempo para tamaños grandes de matrices sigue siendo muy superior que en una arquitectura intel/AMD.

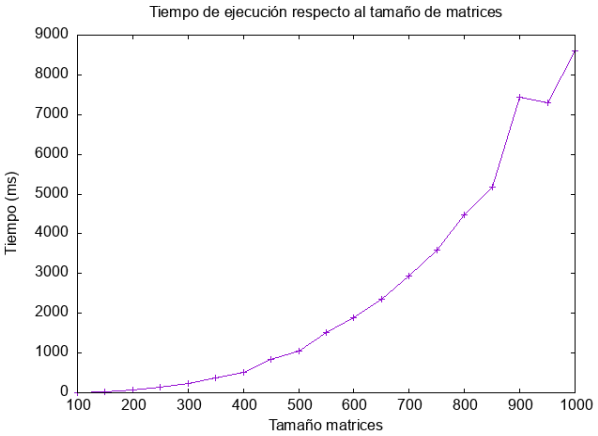
Ahora se va a hacer una comparación temporal en función del tipo de dato que contienen las matrices (int, double, long y char) compilados para arquitectura x86-64.



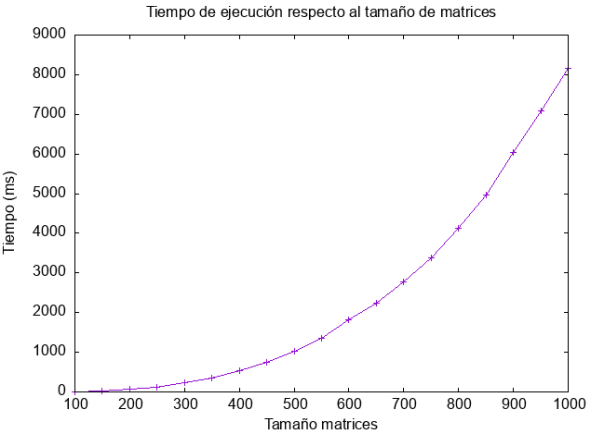
Tipo de dato: double



Tipo de dato: long



Tipo de dato: int



Tipo de dato: char

En las gráficas anteriores se puede observar que el tiempo de ejecución de nuestro programa varía ligeramente respecto al tipo de dato que utilicemos en las matrices. El tipo double es el que más tiempo tarda en ejecutarse, seguido del tipo long, int y, por último, el tipo char. Estas diferencias se aprecian claramente para tamaños altos, siendo de más de 3 segundos entre números definidos como double y enteros para un tamaño de matrices de 1000x1000.

Apartado 1:

Parte 2 -

Tarea 2.1: Acceso a Internet

Al utilizar el comando `ls /sys/class/net` en la terminal obtenemos tres interfaces llamadas docker0, enp0s3 y lo. Para obtener su dirección IP utilizamos el comando `cat /sys/class/net/<interfaz>/address`.

```
alumno@VDIUbuntuEPS2022:/$ ls /sys/class/net
docker0  enp0s3  lo
alumno@VDIUbuntuEPS2022:/$ cat /sys/class/net/docker0/address
02:42:fe:8a:e4:35
alumno@VDIUbuntuEPS2022:/$ cat /sys/class/net/enp0s3/address
08:00:27:06:94:a2
alumno@VDIUbuntuEPS2022:/$ cat /sys/class/net/lo/address
00:00:00:00:00:00
```

Al usar `ifconfig <interfaz>` podemos ver más detalles sobre la interfaz de red seleccionada.

```
alumno@VDIUbuntuEPS2022:/$ ifconfig docker0
docker0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 172.17.0.1  netmask 255.255.0.0  broadcast 172.17.255.255
        ether 02:42:fe:8a:e4:35  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

alumno@VDIUbuntuEPS2022:/$ ifconfig enp0s3
enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.2.15  netmask 255.255.255.0  broadcast 10.0.2.255
        inet6 fe80::e513:a9c8:8111:e7e9  prefixlen 64  scopeid 0x20<link>
        ether 08:00:27:06:94:a2  txqueuelen 1000  (Ethernet)
        RX packets 67098  bytes 101151504 (101.1 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 30286  bytes 1872912 (1.8 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

alumno@VDIUbuntuEPS2022:/$ ifconfig lo
lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Bucle local)
        RX packets 71  bytes 6752 (6.7 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 71  bytes 6752 (6.7 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

La dirección MAC es un identificador único que cada fabricante le asigna a la tarjeta de red de sus dispositivos. Estas tarjetas de red se utilizan principalmente en dispositivos conectados a Internet. Un mismo dispositivo puede tener varias tarjetas de red (WiFi, Ethernet, etc) donde cada tarjeta tiene una dirección MAC diferente.

Las direcciones MAC están formadas por 48 bits binarios, representados generalmente por 12 dígitos hexadecimales agrupados en seis parejas separadas generalmente por dos puntos.

Las direcciones MAC de nuestras direcciones web se pueden ver con el comando `cat /sys/class/net/<interfaz>/address` y son:

- docker0 → 02:42:fe:8a:e4:35
- enp0s3 → 08:00:27:06:94:a2
- lo → 00:00:00:00:00:00

Por otro lado, el parámetro MTU (unidad de transmisión máxima) es el tamaño del mayor paquete permitido que se puede transferir a través de la conexión. Cuanto mayor sea la MTU, mayor cantidad de datos se pueden transferir en un solo paquete.

Para cambiar el MTU podemos utilizar el comando `sudo ifconfig <interfaz> mtu <valor>`. En este caso cambiaremos el valor de la interfaz docker0 a 2000 bytes/paquete. Podemos comprobar que se ha cambiado correctamente utilizando los comandos anteriores.

```
alumno@VDIUbuntuEPS2022:/$ sudo ifconfig docker0 mtu 2000
[sudo] contraseña para alumno:
alumno@VDIUbuntuEPS2022:/$ ifconfig docker0
docker0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 2000
        inet 172.17.0.1  netmask 255.255.0.0  broadcast 172.17.255.255
        ether 02:42:bd:a2:05:f5  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Tarea 2.2: Exploración del sistema Linux

Apartado a)

El directorio /proc es una especie de “ventana” que nos proporciona información sobre qué sucede en el kernel en un momento dado. Si ejecutamos el comando ls /proc obtenemos por pantalla bastantes datos. La gran mayoría de estos datos son números, esto se debe a que son la ID de ciertos procesos y sus subdirectorios contienen información específica de cada proceso. A continuación se muestra una parte de los contenidos del directorio /proc.

```
alumno@VDIUbuntuEPS2022:/$ ls /proc
1     1152   1428    24      3918  4468  5411  6684        crypto         net
10    1154   1451    245166  3919  4480  5415  7           devices        pagetypeinfo
100   116    1454    248076  398   4482  5425  8           diskstats      partitions
1002  1166   1460    248090  3992  4484  5441  805         dma            pressure
1007  1167   1478    248091  4     4492  5471  8444        driver         schedstat
1015  117    147831  25      40    4497  548   91          dynamic_debug  scsi
102   1171   15      251371  4001  4584  5523  92          execdomains    self
```

Para ver el nombre que tienen los procesos asociados a esas ID podemos utilizar el comando ps –pid=<id_proceso>:

```
alumno@VDIUbuntuEPS2022:/$ ps --pid=1152
    PID TTY          TIME CMD
   1152 ?        00:00:01 run-cupsd
```

Apartado b)

Para obtener las características completas de nuestro procesador podemos utilizar varios comandos. Si queremos usar la información que tenemos en el directorio /proc existe el comando cat /proc/cpuinfo o podemos utilizar el comando lscpu que nos proporciona una salida similar. El comando anterior nos mostrará la misma información para cada uno de los cores de nuestro procesador. La información para uno de ellos es:

```
alumno@VDIUbuntuEPS2022:/$ cat proc/cpuinfo
processor       : 0
vendor_id       : AuthenticAMD
cpu family      : 25
model           : 80
model name      : AMD Ryzen 7 5800H with Radeon Graphics
stepping        : 0
cpu MHz         : 3193.962
cache size      : 512 KB
physical id     : 0
siblings        : 4
core id         : 0
cpu cores       : 4
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 16
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ht
 syscall nx mmxext fxsr_opt rdtscp lm constant_tsc rep_good nopl nonstop_tsc cpuid extd_apicid pni pclmulqdq ssse3 cx1
6 sse4_1 sse4_2 x2apic movbe popcnt aes xsave avx rdrand hypervisor lahf_lm cmp_legacy cr8_legacy abm sse4a misalignss
e 3dnowprefetch vmmcall fsgsbase bmi1 avx2 bmi2 invpcid rdseed clflushopt arat
bugs            : fxsave_leak sysret_ss_attrs null_seg spectre_v1 spectre_v2
bogomips        : 6387.92
TLB size        : 2560 4K pages
clflush size    : 64
cache_alignment : 64
address sizes   : 48 bits physical, 48 bits virtual
power management:
```

En el apartado “flags” podemos ver todas las extensiones al repertorio ISA de nuestro procesador. En nuestro caso, la CPU sí que contiene extensiones vectoriales. Algunas de ellas son mmx, sse o avx y sus derivadas. Estas extensiones permiten utilizar registros de 64, 128 y 256 bits (con mmx, sse y avx respectivamente) para realizar varias operaciones de forma simultánea.

Para ver el número de procesadores que tiene nuestra CPU podemos utilizar el comando cat /proc/cpuinfo | grep processor. Podemos observar que tenemos 4 procesadores, que son los que hemos asignado a nuestra máquina virtual.

```
alumno@VDIUbuntuEPS2022:/$ cat /proc/cpuinfo | grep processor
processor       : 0
processor       : 1
processor       : 2
processor       : 3
```

Apartado c)

Con el comando grep flags /proc/cpuinfo obtenemos la información de las extensiones de instrucción específicas que admiten diferentes tipos de operaciones que posee nuestro procesador. En este caso, para que nuestro procesador pueda ejecutar el cálculo de un CRC de 32 bits tiene que tener las extensiones 'crc32' o 'sse'.

```
alumno@VDIUbuntuEPS2022:/$ grep flags /proc/cpuinfo
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush
 mmx fxsr sse sse2 ht syscall nx mmxext fxsr_opt rdtscp lm constant_tsc rep_good nopl nonstop_tsc
 cpuid extd_apicid pni pclmulqdq ssse3 cx16 sse4_1 sse4_2 x2apic movbe popcnt aes xsave avx rdran
d hypervisor lahf_lm cmp_legacy cr8_legacy abm sse4a misalignsse 3dnowprefetch vmmcall fsgsbase b
mi1 avx2 bmi2 invpcid rdseed clflushopt arat
```

(Falta cosa)

Apartado d)

Utilizando el directorio /proc podemos poner el comando cat proc/uptime que nos devolverá el tiempo en segundos que lleva el ordenador encendido y el tiempo donde la CPU no ha estado realizando tareas de procesamiento.

```
alumno@VDIUbuntuEPS2022:/$ cat proc/uptime
2937.20 9409.90
```

También podemos utilizar el comando uptime directamente, lo que nos da el tiempo actual, el tiempo de ejecución del ordenador, el número de usuarios que hay conectados y la carga media de la CPU en los últimos 1, 5 y 15 minutos.

```
alumno@VDIUbuntuEPS2022:/$ uptime
 14:28:53 up 49 min,  1 user,  load average: 1,25, 1,35, 1,39
```

Apartado e)

Para obtener la versión de Linux y de GNU a través del directorio /proc podemos utilizar el comando cat /proc/version.

```
alumno@VDIUbuntuEPS2022:/$ cat  /proc/version
Linux version 5.14.0-1042-oem (buildd@lcy02-amd64-044) (gcc (Ubuntu 9.4.0-1ubunt
u1~20.04.1) 9.4.0, GNU ld (GNU Binutils for Ubuntu) 2.34) #47-Ubuntu SMP Fri Jun
 3 18:17:11 UTC 2022
```

Otra manera de obtener esta información sin utilizar el directorio /proc es escribiendo los comandos uname -r y gcc --version. Estos comandos nos proporcionan la versión de Linux y de GNU.

```
alumno@VDIUbuntuEPS2022:/$ uname -r
5.14.0-1042-oem
alumno@VDIUbuntuEPS2022:/$ gcc --version
gcc (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0
Copyright (C) 2019 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Apartado f)

Para ver todos los sistemas de archivos que soporta nuestro sistema podemos utilizar el comando cat /proc/filesystems.

El sistema de archivos bdef permite controlar los archivos de dispositivos que están almacenados en el directorio /dev.

El sistema de archivos configfs se encarga de configurar y administrar los módulos y dispositivos del sistema. Los archivos de configfs existen de forma independiente en el dispositivo, es decir, no son necesarios que estén cargados.

Los sistemas de archivos ext2, ext3 y ext4 permiten al sistema trabajar con archivos de gran tamaño. Su versión más moderna, ext4, permite trabajar con archivos de hasta 2^{50} bytes.

(Faltan 2)

Apartado g)

Si ponemos el comando cat /proc/stat obtenemos lo siguiente:

```
alumno@VDIUbuntuEPS2022:/$ cat /proc/stat
cpu  386049 11749 203069 3173696 2333 0 2189 0 0 0
cpu0 102882 2886 54889 782745 409 0 122 0 0 0
cpu1 107313 3105 55661 773952 355 0 394 0 0 0
cpu2 84815 2964 47797 814110 841 0 1221 0 0 0
cpu3 91038 2792 44721 802886 726 0 450 0 0 0
intr 24112788 37 13820 0 0 0 0 0 0 0 0 0 0 20902 0 0 10171 0 0 0 38067 118433 419529 57 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0
ctxt 33854959
btime 1699956568
processes 2529242
procs_running 2
procs_blocked 0
softirq 15885587 9 1571088 1901 38053 410531 0 11380 4943471 0 8909154
```

El comando mpstat da cosas

```
alumno@VDIUbuntuEPS2022:/$ mpstat
Linux 5.14.0-1042-oem (VDIUbuntuEPS2022)        14/11/23        _x86_64_        (4 CPU)

13:57:21     CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest  %gnice   %idle
13:57:21     all   10,13    0,37    5,36    0,07    0,00    0,06    0,00    0,00    0,00   84,01
```