Computación Concurrente, Paralela y Distribuida, Práctica 1

Gestión de recursos y repaso de programación

Versión: 29 de enero de 2026

Contexto

Esta primera práctica servirá como introducción a varios conceptos de *gestión y análisis de recursos* en un entorno Linux, algo que nos será muy útil en las siguientes prácticas para labores de análisis y diagnóstico de los programas que estamos ejecutando, ayudándonos así a comprender su comportamiento en términos de rendimiento y a medir el efecto del paralelismo.

De igual forma, también veremos varios conceptos previos de *programación*, tanto del paradigma imperativo como del paradigma *funcional*, que serán repetidos luego muchas veces en las siguientes prácticas.

Finalmente, usaremos esta práctica también para adaptarnos al *uso del Contenedor Docker* para la ejecución de programas y comandos, recordando que el profesorado solo atenderá dudas o problemas de las ejecuciones en dicho entorno.

Objetivos

De forma más detallada, los conceptos de *gestión de recursos* a comprender en esta práctica son:

- **Identificación del hardware y recursos** disponibles en un equipo, algo que nos servirá para conocer el entorno de ejecución de forma general, y para saber los potenciales límites de rendimiento a la hora de ejecutar códigos en particular.
- **Análisis** de consumo de **recursos** en tiempo real, algo que nos servirá para saber el progreso del programa en algunos escenarios y en ocasiones, detectar anomalías o problemas.
- **Medición de tiempos** de ejecución de comandos y programas, algo fundamental a la hora de cuantificar ganancias del efecto de la paralelización, y también para conocer el comportamiento y progreso del programa según necesitemos.

Y los aspectos estudiados de *programación* son:

- **Estilo de programación** que se usará en el resto de prácticas, considerando que se repetirá y usará de forma constante, y puede ser determinante a la hora de entender el código así el objetivo del programa.
- **Programación funcional** básica en Python, teniendo en cuenta que este paradigma es extensamente utilizado en las técnicas y librerías de paralelización por motivos técnicos y de diseño.

Índice

Índice de Tareas

1. Conceptos de análisis de recursos

La gestión de recursos es importante en el ámbito de la ingeniería de computadores, y todavía más importante cuando se trata del procesamiento de datos. Para poder tener un procesamiento de datos que sea eficiente, sobre todo si se busca aplicar técnicas de paralelismo para mejorar sus tiempos de ejecución, es fundamental primero conocer los recursos hardware de los que se disponen. Esto es debido a que a menudo estos recursos hardware, especialmente la CPU, serán los que determinarán el *límite de paralelismo* que tendremos en un equipo, o en un clúster de equipos.

En esencia, este límite de paralelismo lo determinará el punto en el que antes se produzca *un cuello de botella* que impida alcanzar una mejora del rendimiento. En nuestro escenario actual y en numerosas ocasiones cuando se procesan datos, este cuello de botella estará determinado por el número de núcleos disponibles. Por ejemplo, si nuestro contenedor tiene asignados 4 núcleos, nuestro límite de paralelismo será, la mayoría de las veces, de un factor de 4 (4x). A lo largo de la asignatura veremos no obstante como se pueden alcanzar límites de paralelismo derivados de otros factores (e.g., limitación software, límites de memoria...).

1.1. Identificación del Hardware

Primero, vamos a proceder a la identificación del hardware del que disponemos en nuestro equipo local (portátil), más específicamente, a su procesador. Ejecuta el siguiente comando en el contenedor (*los comandos van precedidos del símbolo '$'*), deberías ver una salida similar a la mostrada a continuación:

```
$ cat /proc/cpuinfo
```

```
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 78
model name      : Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz
stepping        : 3
microcode       : 0xf0
cpu MHz         : 699.982
cache size      : 4096 KB
physical id     : 0
siblings        : 4
core id         : 0
cpu cores       : 2
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 22
wp              : yes
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 s
yscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf pni pclmulqdq dtes6
4 monitor ds_cpl vmx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdr
and lahf_lm abm 3dnowprefetch cpuid_fault epb pti ssbd ibrs ibpb stibp tpr_shadow flexpriority ept vpid ept_ad fsgsbase tsc_adjust bmi1 avx
2 smep bmi2 erms invpcid mpx rdseed adx smap clflushopt intel_pt xsaveopt xsavec xgetbv1 xsaves dtherm ida arat pln pts hwp hwp_notify hwp_
act_window hwp_epp vnmi md_clear flush_l1d arch_capabilities
vmx flags       : vnmi preemption_timer invvpid ept_x_only ept_ad ept_1gb flexpriority tsc_offset vtpr mtf vapic ept vpid unrestricted_gues
t ple pml
bugs            : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l1tf mds swapgs itlb_multihit srbds mmio_stale_data retbleed gds
bogomips        : 5199.98
clflush size    : 64
cache_alignment : 64
address sizes   : 39 bits physical, 48 bits virtual
power management:

processor       : 1
vendor_id       : GenuineIntel
cpu family      : 6
model           : 78
model name      : Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz
stepping        : 3
microcode       : 0xf0
cpu MHz         : 700.022
```

Figura 1: Salida del comando para mostrar información de la CPU

Con este comando podemos obtener información sobre el procesador instalado, su velocidad de reloj y lo más importante, el número de núcleos. También es interesante considerar el campo de los flags, donde podemos comprobar las capacidades del procesador (e.g., instrucciones SSE), así como el campo de bugs hardware, donde podríamos encontrar algunos relacionados con el rendimiento (e.g., spectre_v2).

De igual forma, con el nombre al completo del modelo de procesador, en este caso 'Intel Core i7-6700K', podemos buscar más información técnica en:

- La página web del fabricante (link para este ejemplo)
- Otras webs especializadas como TechPowerUp o CpuBenchmark
- Herramientas de IA como Copilot (verifica la fuente de sus datos)

Tarea 1 – Buscando las especificaciones del procesador

Busca las especificaciones de tu procesador, del resultado de esta búsqueda deberías anotar:

- Nombre del modelo al completo
- Número de cores y de hilos (*threads*)
- Frecuencia de reloj base y turbo (*boost*)
- El Thermal Design Power (TDP)

Otra información que puede ser interesante de considerar, si está disponible, sería:

- Litografía usada (e.g., 14nm, 7nm)
- Tamaño de las caches L1, L2 y L3
- Tipología de los cores (e.g., eficientes, potentes)

Hardware bugs: Los bugs o errores, al igual que en el software, también existen en el hardware. Estos bugs se pueden producir debido a fallos en el diseño interno de la arquitectura del procesador y es posible que se puedan aprovechar por parte de programas. Por desgracia, a diferencia del software, los bugs hardware no tienen arreglo y la única solución es un 'parche' mediante software, a menudo en el núcleo del SO, algo que a veces tiene un impacto en el rendimiento. Explicación de los bugs Spectre y Meltdown aquí.

TDP: El TDP, o *Thermal Design Power*, es una medida en vatios (*Watts*) usada para tener una estimación del consumo eléctrico y de la disipación térmica del procesador. Por norma general, a más núcleos, mayor TDP, aunque con los procesadores nuevos también se busca reducir o al menos mantener el TDP en unos niveles aceptables para la disipación térmica disponible (muy baja en un móvil, baja en un portátil, media en un ordenador de mesa, alta en un servidor). En el caso de que la temperatura del procesador de un dispositivo supere un umbral considerado como segura, se producirá un thermal throttling, un mecanismo de protección que buscará bajarla reduciendo la velocidad del procesador, lo que implicará un menor rendimiento.

Información: Los hilos, o *threads*, cuando se hace referencia a ellos en un contexto hardware, se refieren a la tecnología de Simultaneous Multithreading (SMT) de los procesadores. Los hilos hardware son 'núcleos virtuales'. No tienen la capacidad de un núcleo físico real, pero en algunos escenarios sí que pueden comportarse como tales. Se comentará más en detalle en teoría.

Vamos a proceder ahora a la identificación de la memoria disponible. Esto se puede hacer también con un comando dentro del entorno del contenedor. Ejecuta el siguiente comando, deberías ver una salida similar a la mostrada. Con esta salida tenemos la cantidad de memoria disponible en el sistema, medida en KiB (kibibyte). Con una división entre (1024*1024) podemos tener la memoria en GiB (gibibyte), en este caso serían aproximadamente 3.84 GiB. También podemos ver la cantidad de memoria libre, así como la disponible y la usada para cachés del sistema.

```
$ cat /proc/meminfo
```

```
MemTotal:        8026452 kB
MemFree:         2175008 kB
MemAvailable:    4511132 kB
Buffers:          125924 kB
Cached:          2777720 kB
SwapCached:           56 kB
Active:          3843380 kB
Inactive:        1201136 kB
Active(anon):    2543672 kB
Inactive(anon):     5176 kB
Active(file):    1299708 kB
Inactive(file):  1195960 kB
Unevictable:      281036 kB
Mlocked:           17052 kB
SwapTotal:       8388604 kB
SwapFree:        8388348 kB
Zswap:                 0 kB
Zswapped:              0 kB
Dirty:             18112 kB
Writeback:             0 kB
AnonPages:       2421912 kB
Mapped:           791576 kB
Shmem:            402008 kB
KReclaimable:     129192 kB
Slab:             351392 kB
SReclaimable:     129192 kB
SUnreclaim:       222200 kB
KernelStack:       13120 kB
PageTables:        49116 kB
SecPageTables:         0 kB
NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:          0 kB
CommitLimit:    12401828 kB
Committed_AS:   19123136 kB
VmallocTotal:   34359738367 kB
VmallocUsed:       47940 kB
VmallocChunk:          0 kB
Percpu:             3808 kB
```

Figura 2: Salida del comando para mostrar información del subsistema de memoria

Tarea 2 – Análisis de recursos del procesador

A fin de conocer los recursos 'reales' del equipo, realiza un análisis de los recursos disponibles en tu sistema nativo (el SO del portátil, **fuera** del contenedor).

- En Windows puedes hacer eso con la utilidad de 'Información del Sistema'.
- En Linux puedes hacer eso con el propio comando ***'lshw'***, con permisos de superusuario.
- En Mac puedes hacer eso, ¿con el comando ***'lshw'*** también?

Puedes también buscar el modelo de portátil en la página del fabricante, o en general en cualquier página de Internet con información fiable.

Con la información de los recursos con los que cuenta tu portátil o equipo, podrás re-ajustar mejor si es necesario los recursos que quieres asignar al contenedor. Por norma general, **NO es aconsejable asignarle más de la mitad de la memoria disponible**, a partir de ese punto es posible que se consuma demasiada memoria (la del SO nativo + la del contenedor) y el equipo entre en paginación (i.e., vaya lento o se 'congele'). Por otro lado, en lo que respecta a los **núcleos de procesamiento**, por motivo técnicos relacionados con las arquitecturas híbridas actuales

de los procesador, se recomienda asignar solamente **la mitad**, aunque no debería haber mayor inconveniente con asignarlos todos. En este último caso, es posible que el equipo se 'congele' temporalmente durante ejecuciones muy intensas que usen todos los núcleos. **Sigue siempre las recomendaciones del profesor** indicadas en los códigos para las ejecuciones y la configuración de la carga.

1.2. Análisis de consumo de recursos

Una vez que ya conocemos los recursos de los que disponemos y sabemos los que le podemos dar al contenedor, podremos ejecutar programas conociendo las limitaciones y los motivos por los que en algunos casos pueden alcanzarse límites en la mejora de los tiempos, pueden tardar demasiado, o directamente pueden ocasionar un fallo por falta de memoria.

De todas formas, una labor a realizar de forma paralela a la ejecución experimental de los códigos en las siguientes prácticas, será la labor de analizar el consumo de recursos. Esta tarea será especialmente importante para comprender la evolución en la ejecución de un código, así como para realizar en algunos casos labores de '*debugging*' para aquellos escenarios en los que pueda haber *locks* software o funcionamientos del código anómalos.

La información de recursos consumidos, así como de los procesos en el sistema, puede ser obtenida fácilmente en tiempo real con diversos comandos como ***'atop'*** (manual online) o ***'htop'*** (manual online). Si ejecutas ***'atop'***, deberías ver algo similar a lo que se muestra en la Figura 3.

```
$ atop
```

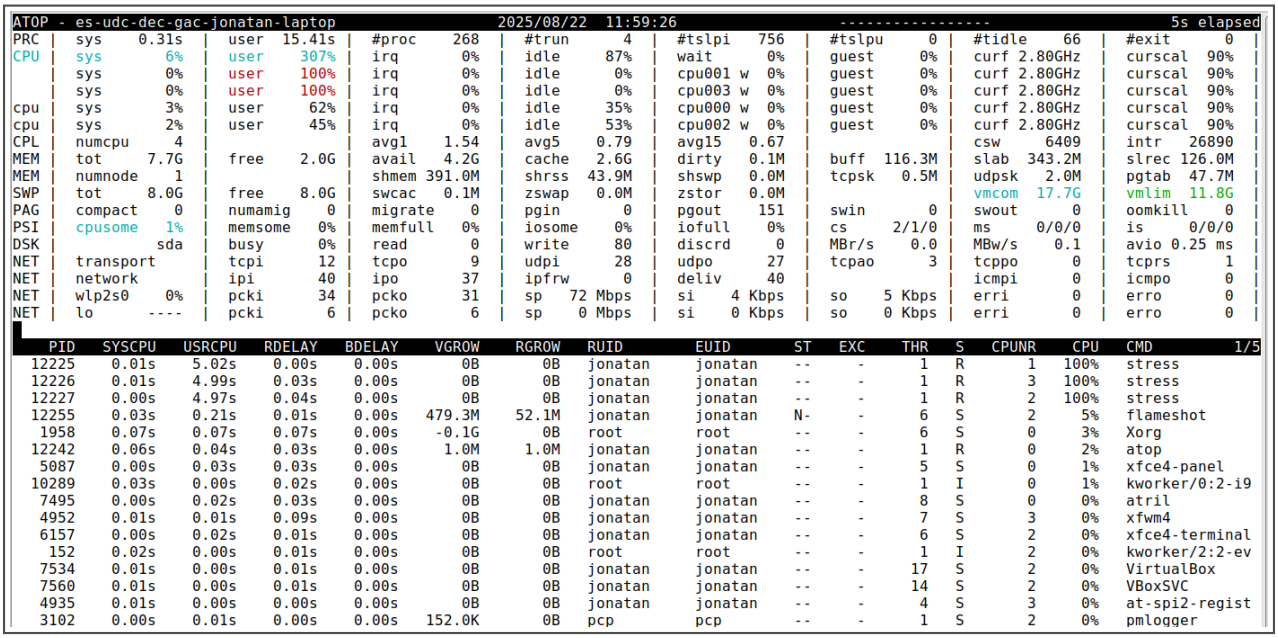


Figura 3: Captura de pantalla mostrando la salida de atop en ejecución

Este programa acepta varias opciones de configuración. Si introduces:

- 'i5', cambiarás el *intervalo* de refresco de la interfaz a 5 segundos
- 'p', podrás ver información agrupada por tipo de proceso (e.g., programas python)
- 'c', podrás ver información de los diferentes procesos individualmente (commands)
- 'g', podrás ver información de recursos de los procesos individuales
- 'a', podrás cambiar entre mostrar los recursos de todos los **procesos** del sistema o solo los activos
- 'f', podrás cambiar entre mostrar los recursos de todos los **recursos** del sistema o solo los activos

- 'k', seguido de un PID, podrás matar un proceso en particular
- 'q', para salir

Tarea 3 – Monitorización consumo de recursos (idle)

Intenta comprender la salida del comando ***'atop'***. Deberías poder averiguar, usando su salida, la siguiente información:

- Consumo global de CPU (todos los núcleos)
- Consumo individual en cada núcleo
- Memoria disponible y libre
- Consumo de memoria en intercambio *(swap)*
- Número de procesos presentes en el sistema
- Número de hilos en ejecución
- Frecuencia de ejecución actual
- Listado de procesos o comandos ordenados por consumo de CPU
- Consumo de recursos por cada proceso (e.g., CPU, memoria)

Por si te sirve de ayuda, aquí tienes una descripción de algunos campos expuestos por el comando:

- **CPU** → Consumo de recursos de CPU global
- **cpu** → Consumo de recursos de por núcleo
- **MEM** → Consumo de recursos de memoria
- **SWP** → Información de paginación (swap)
- **#proc** → Número de procesos
- **#trun** → Número de hilos en ejecución
- **curf** → Frecuencia actualmente usada por el procesador y núcleos
- **numcpu** → Número de CPUs
- **SYSCPU** → Consumo de CPU en modo sistema por proceso o programa
- **USRCPU** → Consumo de CPU en modo usuario por proceso o programa
- **VSIZE** → Consumo de memoria virtual por proceso o programa
- **RSIZE** → Consumo real de memoria por proceso o programa
- **CPUNR** → núcleo usado para la ejecución por el proceso

Atención: Si no encuentras alguno de los campos previamente descritos en ***'atop'***, prueba a redimensionar la ventana para que tenga más espacio. Hay programas como ***'atop'*** que son 'conscientes' del tamaño de ventana que tienen, incluso de cuando esta cambia, y que se adaptan a ello.

De igual forma, puedes ejecutar ***'htop'*** como una opción más simple, pero también más rápida. Deberías ver algo como lo observado en la Figura 4.

Con estos comandos además es posible enviar señales a procesos en particular, en especial señales de muerte, útiles en caso de programas que hayan podido caer en un bloqueo (e.g., *deadlock*).

```
$ htop
```

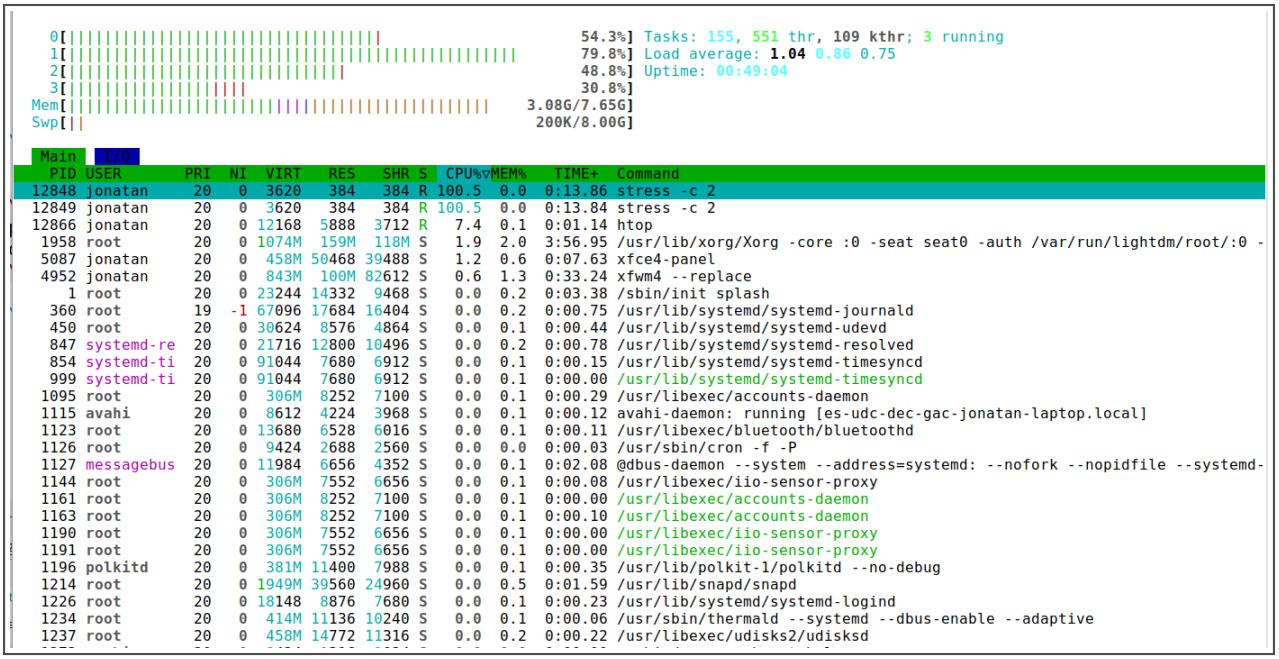


Figura 4: Captura de pantalla mostrando la salida de htop en ejecución

Información: El comando ***'atop'***, al igual que ***'htop'*** y otros muchos comandos, usa una librería llamada ***'ncurses'***, la cual permite mostrar información usando una interfaz más 'gráfica' que una simple línea de comandos. Además, esta interfaz permite interactuar mediante selección con desplazamientos e incluso, si hay disponible una interfaz gráfica con ratón, usando selección con el puntero. Aunque sea menos visual que una interfaz gráfica al uso, su consumo de recursos es mucho menor y por ello es muy utilizada en entornos donde no hay una interfaz de usuario o no se requiere mucha complejidad. **A menudo estos programas usan la letra 'q' (quit) para salir.**

Por último en lo que respecta a análisis de recursos, vamos a proceder a estresar al sistema para poder ver el efecto de dicho estrés en los programas anteriormente descritos. Para realizar el estrés vamos a utilizar el comando ***'stress'*** y ***'stress-ng'***. Primero procederemos a estresar la CPU, lanzando 2 procesos que utilicen la CPU todo lo que puedan. Ejecuta:

```
$ stress -c 2
```

Para matar este comando en ejecución, pulsa simultáneamente 'Ctrl + C'.

A continuación, podemos crear un uso de memoria artificial mediante el comando ***'stress-ng'***. Ejecuta:

```
$ stress-ng -vm 3 --vm-bytes 512M --vm-keep --vm-populate
```

Tarea 4 – Monitorización consumo de recursos (stressed)

Mientras se ejecutan las pruebas de estrés, tanto de CPU como de memoria, lanza los programas de análisis de recursos (e.g., htop) y observa cómo reportan el uso.

Información: Tanto en esta práctica como en las siguientes será necesario realizar acciones 'simultáneas' en varios terminales que están conectados al contenedor. Esto lo podemos abriendo más conexiones directas en otras terminales.

Tarea 5 – Enviando señales

Lanza el comando de stress con 3 procesos y a continuación, en otra terminal usando ***'atop'*** o ***'htop'***, mata uno de los procesos. Con ***'htop'*** puedes realizar esta tarea de forma más sencilla, con ***'atop'*** requerirás escribir el PID. En ambos casos, la señal a enviar podría ser la #15 (SIGTERM), que pide la finalización ordenada al propio proceso (señal enmascarable o ignorable por el proceso, sobre todo si está 'atascado'), o la #9 (SIGKILL), que pide al SO que elimine al proceso de forma inmediata (no ignorable por el proceso).

- ¿Qué ocurre?
- ¿Qué se muestra por la salida de la terminal donde se ejecutaba el comando?

1.3. Medición de tiempos de ejecución

Para finalizar en lo que respecta a la gestión de recursos, se puede mencionar una manera simple de medir el tiempo de ejecución de un programa. Para ello usaremos el comando ***'time'***, que ejecuta el comando que se le indica a continuación realizando una medición básica de tiempos. Ejecuta el siguiente comando, deberías ver una salida similar a la mostrada en la Figura 5, salvo los tiempos.

El resultado de la ejecución de este programa se divide entre la salida del comando a ejecutar realmente, en este caso ***'ls -lash -R /'***, y la salida del comando ***'time'***. Dejando de lado la primera parte de la salida, que en este caso no nos interesa, podemos analizar la salida del comando time. Dicha salida se divide en 3 mediciones:

- tiempo 'real': *Tiempo de pared* (Wall clock), el tiempo que nosotros realmente hemos 'sentido' pasar
- tiempo 'user': *Tiempo de CPU en modo Usuario* (código de programa ejecutado por el usuario)
- tiempo 'sys': *Tiempo de CPU en modo Kernel* (llamadas a librerías del SO)

Como se puede apreciar, la suma de los tiempos 'user' y 'sys' **no resultan** en el tiempo 'real', eso es porque la suma de los tiempos de 'user' y 'sys' son el tiempo total de ejecución en el procesador, pero el comando también pudo pasar tiempo 'fuera' del procesador, en espera o bloqueado. Este último tiempo sería la diferencia de tiempos.

Información: Los comandos ***'stress'*** y ***'stress-ng'*** son unos comandos usados para generar una carga artificial en un sistema, tanto para 'testear' el procesador, la memoria, como otros dispositivos como los discos. Son comandos útiles cuando se quiere asegurar que un sistema es estable, como por ejemplo después de cambiar parámetros hardware (e.g., overclocking o underclocking), para probar una correcta monitorización y alerta en los sistemas, o para generar una carga que permita medir un consumo energético. El estrés de CPU que estamos realizando pretende ocupar con computación todo el tiempo posible, buscando no tener ningún tiempo de bloqueo ni ninguna parada.

```
$ time ls -lash -R /
```

```
/var/snap:
total 8.0K
4.0K drwxr-xr-x  2 root root 4.0K Aug 20  2024 .
4.0K drwxr-xr-x 13 root root 4.0K Nov 22  2024 ..

/var/spool:
total 12K
4.0K drwxr-xr-x  3 root root 4.0K Nov 22  2024 .
4.0K drwxr-xr-x 13 root root 4.0K Nov 22  2024 ..
4.0K drwxr-xr-x  3 root root 4.0K Nov 22  2024 cron
   0 lrwxrwxrwx  1 root root    7 Aug 27  2024 mail -> ../mail

/var/spool/cron:
total 12K
4.0K drwxr-xr-x 3 root root    4.0K Nov 22  2024 .
4.0K drwxr-xr-x 3 root root    4.0K Nov 22  2024 ..
4.0K drwx-wx--T 2 root crontab 4.0K Mar 31  2024 crontabs
ls: cannot open directory '/var/spool/cron/crontabs': Permission denied

/var/tmp:
total 20K
4.0K drwxrwxrwt  5 root root 4.0K Sep  3 15:47 .
4.0K drwxr-xr-x 13 root root 4.0K Nov 22  2024 ..
4.0K drwx------  3 root root 4.0K Sep  3 17:46 systemd-private-b71c5fb3319444d7a17f3e75be42efda-polkit.service-qIXpul
4.0K drwx------  3 root root 4.0K Sep  3 17:46 systemd-private-b71c5fb3319444d7a17f3e75be42efda-systemd-logind.service-29PoTQ
4.0K drwx------  3 root root 4.0K Sep  3 17:46 systemd-private-b71c5fb3319444d7a17f3e75be42efda-systemd-resolved.service-Ow5F9j
ls: cannot open directory '/var/tmp/systemd-private-b71c5fb3319444d7a17f3e75be42efda-polkit.service-qIXpul': Permission denied
ls: cannot open directory '/var/tmp/systemd-private-b71c5fb3319444d7a17f3e75be42efda-systemd-logind.service-29PoTQ': Permission denied
ls: cannot open directory '/var/tmp/systemd-private-b71c5fb3319444d7a17f3e75be42efda-systemd-resolved.service-Ow5F9j': Permission denied

real    0m11.039s
user    0m1.555s
sys     0m5.477s
alumno@ParalelismoMV:~$
```

Figura 5: Final de la salida del comando time con una búsqueda recursiva del sistema de ficheros

Tarea 6 – Ejecutando comandos de estresado de CPU

- Lanza ahora el siguiente comando, que medirá los tiempos de un estrés de CPU con 2 procesos durante 5 segundos:

  ```
  $ time stress -c 2 -t 5
  ```

  Si analizas la salida de tiempos, verás otra discrepancia al comparar los tiempos de 'user' y 'sys' con 'real', pero diferente a la anterior con el comando 'ls'. El tiempo del 'user' debería ser mayor al de 'real'. ¿A qué crees que se debe esto?

- Puedes ejecutar también el siguiente comando, que lanzará 4 procesos, y comparar:

  ```
  $ time stress -c 4 -t 5
  ```

- Finalmente, ejecuta el siguiente comando, que lanza 100 procesos y los ejecuta 'simultáneamente':

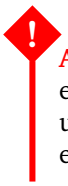
  ```
  $ time stress -c 100 -t 5
  ```

  Si analizas la salida de este último comando, verás que sus resultados no son los esperados si el comportamiento fuese similar a los anteriores comandos, ya que no se mantiene la tendencia. ¿A qué crees que se debe?

2. Conceptos de programación

En lo que respecta a la programación, es interesante destacar que aunque las prácticas no serán intensivas en programación, sí que contarán con muchos códigos en los cuales se repetirán una serie de técnicas y estructuras, a menudo orientadas a 'configurar' el comportamiento del código, implementar y dimensionar la carga de trabajo a procesar, o a estudiar el rendimiento en general.

De igual forma, en la paralelización con Python es extremadamente común ver algunas técnicas y métodos de programación propios de la programación funcional, es decir, en los códigos de las siguientes prácticas trabajaremos a menudo con funciones como argumentos de otras funciones.



Atención: El paradigma de la programación funcional es considerablemente distinto al imperativo o 'clásico', el más comúnmente impartido. Aprovecha esta práctica y las primeras para entender las técnicas básicas más usadas en programación funcional y paralelismo. No obstante, si en algún momento, en alguna práctica, no estás seguro/a de entender algo en lo que respecta a los códigos, no dudes en preguntar al profesor.

2.1. Estilo de programación

A continuación se expondrán algunos segmentos de código que hacen referencia a estructuras muy usadas.

- **Medición de tiempos dentro de un código** usando la librería 'time' (no confundir con el comando time). Guardamos un instante temporal en la variable 'start' (número real, flotante), ejecutamos el código deseado, y volvemos a guardar otro instante temporal en la variable 'end'. La diferencia entre ambas variables es el tiempo transcurrido en segundos. Dado que la escala de segundos a veces puede ser demasiado grande, podemos multiplicar por 1000, teniendo así el tiempo en milisegundos. Finalmente, si en algún momento deseamos tener números enteros para imprimir, ya sean segundos o milisegundos, podemos hacer una conversión al tipo entero (int).

Medición de tiempos intra-código (codigo1.py)

```
15 import time
16
17 start = time.time()
18 for n in range(1, 10000):
19     n*n
20 end = time.time()
21 print("Time in seconds is: " + str(end - start))
22 print("Time in milliseconds is: " + str(int(1000 * (end - start))))
23
24 start = time.time()
25 for n in range(1, 5000):
26     for k in range(1, 5000):
27         x = n*k
28 end = time.time()
29 print("Time in seconds is: " + str(end - start))
30 print("Time in milliseconds is: " + str(int(1000 * (end - start))))
```

- Podemos hacer una **parada de la ejecución de un proceso o hilo 'durmiéndolo'**, algo que nos será útil en numerosas ocasiones. Para hacer esto llamaremos a la función 'time.sleep'. Esta función admite valores mayores que 0, pudiendo ser número enteros (ints) o reales (floats). De esta forma, un argumento de '5', dormiría durante 5 segundos y un argumento de '0.2', dormiría durante 0.2 segundos, o lo que es lo mismo, 200 milisegundos. En los códigos que usaremos será habitual ver la posibilidad de trabajar en segundos (esperas largas) o milisegundos (esperas cortas).

Durmiendo la ejecución (codigo2.py)

```
15 import time
16
17 start = time.time()
18 print("Starting execution")
19 time.sleep(5)
20 print("Finishing execution")
21 end = time.time()
22 print("Time in seconds is: " + str(end - start))
23 print("Time in milliseconds is: " + str(int(1000 * (end - start))))
24
25 start = time.time()
26 print("Starting execution")
27 time.sleep(0.2)
28 print("Finishing execution")
29 end = time.time()
30 print("Time in seconds is: " + str(end - start))
31 print("Time in milliseconds is: " + str(int(1000 * (end - start))))
```

- En algunas ocasiones haremos **uso de números pseudo-aleatorios**, que podemos obtener con la librería 'random'. Podemos pasarle 2 números enteros como argumentos, y nos devolverá un número entero en ese rango.

Usando números aleatorios (codigo3.py)

```
15 import time
16 import random
17
18 start = time.time()
19 sleep_time = random.randint(0, 10)
20 print("Time to sleep (randomly chosen) is : " + str(sleep_time))
21 time.sleep(sleep_time)
22 end = time.time()
23 print("Time in seconds is: " + str(end - start))
24 print("Time in milliseconds is: " + str(int(1000 * (end - start))))
```

- Muchas veces haremos **uso de cadenas (strings)**, ya sea para mostrar información de **debugging**, o para **etiquetar resultados** de ejecuciones al guardarlas en diccionarios (dicts). En todo caso, podemos formatear fácilmente estas cadenas usando la función 'format'. Dentro de la cadena formateada, el primer argumento se emplazará donde se encuentre la subcadena '{0}', el segundo argumento en la subcadena '{1}' y así sucesivamente. Finalmente, los parámetros que sean números reales, se pueden parametrizar para mostrar 'n' número de decimales (en el código, 2 → .2f).

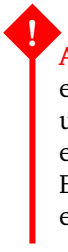
Formateo de cadenas y números (codigo4.py)

```
15 import time
16 import random
17
18 start = time.time()
19 count = random.randint(1000, 10000)
20 print("Going to iterate {0} times".format(count))
21 for n in range(1, count):
22     for k in range(1, count):
23         n*k
24 end = time.time()
25 print("It took {0} seconds, or {1} miliseconds".format(
26     int(end - start),
27     int(1000 * (end - start))))
28 print("It took exactly {0:.2f} seconds".format(end - start))
```

- Por último, es interesante mencionar el uso de **variables de configuración** que serán **claves** para cambiar el comportamiento del código, ya sea en el tamaño de la carga simulada, en el número de recursos desplegados, o ambos objetivos. Estas variables muy a menudo estarán en mayúsculas y pueden ser numéricas, en cuyo caso pueden usar notación científica (e.g., 5.5e3 → 5500) por comodidad. También es destacable algunas variables que usaremos como contadores, especialmente para el control de bucles.

Variables de configuración (codigo5.py)

```
15 import time
16
17 COUNT_INI = int(1e2) # Esto es 100
18 COUNT_MAX = int(1e6) # Esto es 1.000.000
19 COUNT_STEP = int(1e1) # Esto es 10
20
21 count = COUNT_INI
22 while count <= COUNT_MAX:
23     count = count * COUNT_STEP
24     print(count)
```



Atención: El valor de las variables será en muchos casos **fundamental**, ya sea para obtener resultados de experimentación 'interesantes' y comprender lo que está sucediendo, para ajustar la carga computacional a unos límites aceptables (ni acabar muy rápido ni tardar demasiado o no acabar directamente), o para modelar el experimento en general (ya lo veremos). En algunos casos podrá ser importante anotar los valores usados. En todo caso, no es aconsejable exceder 'al alza' los valores orientativos dados por el profesor dado que las ejecuciones pueden ser demasiado intensas y 'bloquear' el contenedor e incluso el equipo nativo.

Tarea 7 – Ejecutando códigos Python generales

Analiza todos los códigos anteriormente especificados (del 1 al 5) y ejecútalos en el contenedor. Puedes ejecutar los programas con el siguiente comando:

```
$ python3 codigo1.py
```

2.2. Programación funcional en Python

La programación funcional es un paradigma de programación que aunque es menos común o menos utilizado que la programación imperativa (clásica), puede ser extremadamente flexible y robusta. En nuestro caso nos centraremos en el uso de funciones como argumentos. Esto nos permitirá trabajar con códigos muy cortos pero altamente flexibles, en los que por ejemplo podremos cambiar el tipo de paralelismo usado con una variable y sin replicar código. Además hay que destacar que el uso de este **paradigma** es prácticamente **exigido por muchas librerías de paralelismo**.

Tarea 8 – Ejecutando códigos Python de programación funcional

Abre el fichero *"codigo6.py"*, analiza el código, ejecútalo y comprende su funcionamiento. Para esto puedes apoyarte en los mensajes informativos que genera el código, así como en la lectura del propio código.

Copyright & Disclaimer