

PRÁCTICA PROFESIONAL SUPERVISADA

TEMA

Estudio de librerías de paralelismo en python
Análisis de rendimiento en Raspberry Pi

LUGAR DE REALIZACIÓN:

Laboratorio de Computación - FCEfyN - UNC.

COORDINADORA PS:

Ing. Rodriguez, Carmen.

SUPERVISOR:

Ing. Algorry, Aldo.

TUTOR:

Ing. Wolfmann, Gustavo.

INTEGRANTE:

Cazajous, Miguel A.	34980294
Sleiman, Mohamad I.	42828207

CARRERA:

Ingeniería en Computación.



Índice

Introducción	5
Marco teórico	6
Python	6
Procesos vs Hilos	6
Paralelismo en Python	8
Multiprocessing vs Multithreading	8
Multiprocessing: Pool vs Process	10
Algoritmo Merge Sort	10
Investigación y pruebas	12
Rendimiento de CPU con algoritmo de ordenamiento Merge Sort	12
Rendimiento de CPU con operaciones matriciales - Multiprocessing vs Multithreading	14
Rendimiento de CPU con Productor - Consumidor utilizando OpenCV - Multiprocessing vs Multithreading	17
Desarrollo	24
Ejecución de pruebas finales	25
Pruebas Matriciales	26
Prueba 1- 2 -3 (Matricial-Elementos):	26
Prueba 4 - 5 - 6 (Matricial-Procesos):	29
Pruebas Merge Sort - Multiprocessing	32
Pruebas 1 - 2 - 3 (Merge Sort - Elementos)	33
Pruebas 4 - 5 - 6 (Merge Sort - Procesos)	35
Pruebas Merge Sort - Threading	38
Pruebas 1 - 2 - 3 (Merge Sort - Elementos)	38
Pruebas 4 - 5 - 6 (Merge Sort - Hilos)	41
Conclusión	45
Anexos	46
Trace-cmd	46
Lttng y Tracecompass	47
Referencias	51
Bibliografía	51



Introducción

Con el avance de las capacidades de hardware de los dispositivos de procesamiento, en particular el incremento de núcleos en los microprocesadores, se hace necesario empezar a pensar a desarrollar aplicaciones para que puedan aprovechar toda esa capacidad y mejorar tiempos de conclusión, ejecutando operaciones simultáneamente en lugar de secuencialmente, evitando así que recursos de hardware queden ociosos durante un tiempo prolongado.

Durante esta práctica supervisada nos planteamos el objetivo de estudiar, implementar y analizar el rendimiento de diferentes aplicaciones explotando el paralelismo haciendo uso de las librerías ofrecidas por Python como multiprocessing y threading sobre Raspberry PI. El rendimiento se analiza en base al funcionamiento y distribución de procesos o hilos en el planificador del kernel observando cómo esto influye en los tiempos de ejecución y desempeño de los programas sobre dicho hardware.

Para la observación de la distribución de procesos en el procesador se utilizan las herramientas trace-cmd, que interactúa con el framework Ftrace del kernel de linux y Lttng que realiza una grabación de los eventos del kernel junto con Trace Compass que permite visualizarlo.

Finalmente analizamos la capacidad de respuesta de la Raspberry PI implementando programas multiprocesos o multihilos sobrecargando el sistema al límite.

Marco teórico

Python

Python es un lenguaje de programación interpretado de tipado dinámico que está pensado para ser simple y conciso, resultando así, generalmente en código más corto que el que se puede realizar en otros lenguajes.

Con interpretado nos referimos a que el código no se compila para obtener un binario en lenguaje máquina como ocurre en lenguajes como C. En Python la traducción a lenguaje máquina se va haciendo gradualmente mientras se ejecuta el código.

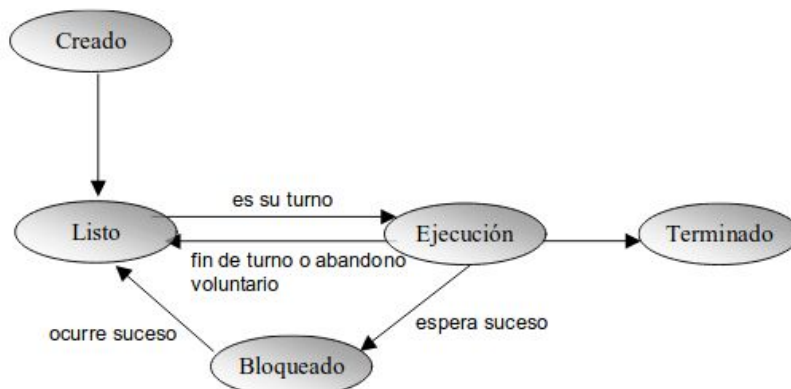
Que sea de tipado dinámico nos dice que la comprobación de tipo se hace en tiempo de ejecución, haciendo de esto un lenguaje más lento que los que hacen uso de tipado estático, además de que es más propenso a errores, pero dándonos un poco más de flexibilidad a la hora de codificar.

Python cuenta con una vasta cantidad de módulos para un sinfín de diversos propósitos, sin mencionar además, que es muy sencillo realizar módulos propios para usarse en proyectos personales.

Para implementar programas que aprovechen el hardware en el que corren ejecutándose en paralelo, Python cuenta con dos módulos ampliamente usados: Multiprocessing y Threading.

Procesos vs Hilos

Para introducir el concepto de proceso vamos a referirnos a la siguiente imagen:



Esta imagen muestra una versión simplificada del ciclo de vida de un proceso, cuando un hilo es creado este pasa al estado Listo, es decir que está en condiciones de hacer uso de la CPU una vez que el planificador de procesos o scheduler lo

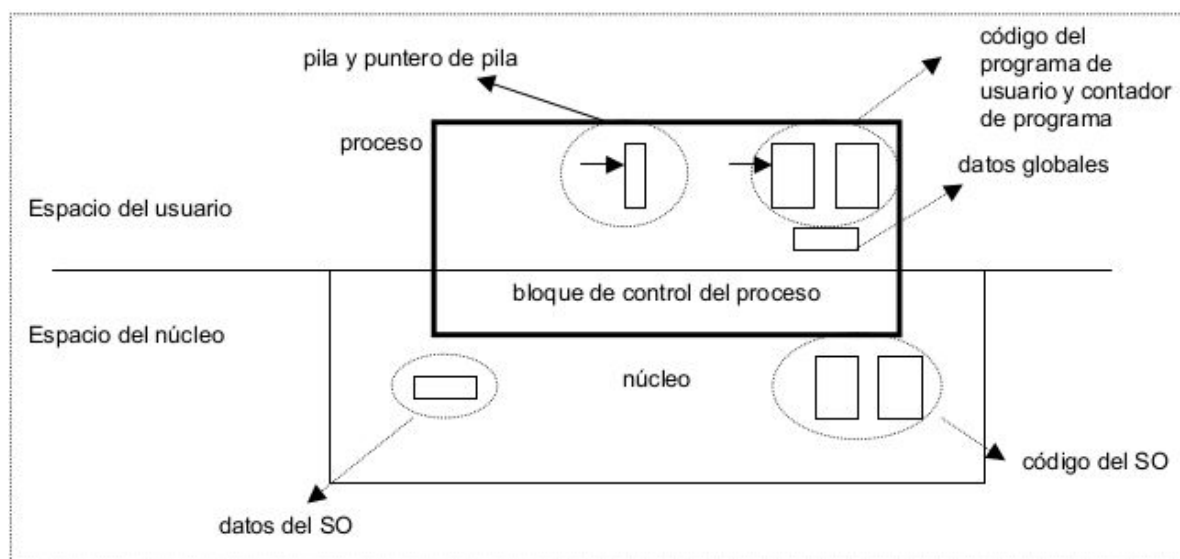
considere adecuado. Cuando el planificador decide darle tiempo de CPU el proceso pasa al estado de ejecución.

En la imagen se observa también que un proceso puede pasar de Ejecución a Listo, esto lo decide el planificador en base a una serie de políticas de planificación como por ejemplo asignación de rodajas de tiempos equitativas a cada proceso, entonces al cumplir con su rodaja de tiempo el proceso es desalojado y pasa al estado Listo. Otra alternativa es que el proceso en Ejecución pase voluntariamente al estado Listo.

Un proceso pasa del estado Ejecución a Bloqueado cuando debe esperar por determinado suceso, como una operación de entrada/salida, la finalización de otro procesos o un bloqueo voluntario. Una vez ocurrido el evento el proceso vuelve al estado de Listo.

El cambio de un estado a otro de un proceso se denomina cambio de contexto.

En un SO la memoria suele estar dividida en un espacio de usuario donde se encuentran la mayoría de la información relativa a los procesos de usuario y un espacio del núcleo donde reside el código y las estructuras de datos propias del SO. En un SO multitarea, la información relativa a un proceso se divide entre los dos espacios, en el espacio de usuario se encuentra la información propia del proceso tales como el código del proceso, contador de programa, variables, pila y puntero a pila, en el espacio del núcleo el SO contiene en bloque de control del proceso (PCB) que contiene la información relativa al proceso para poder administrarlo y realizar cambios de contexto entre procesos.



Cuando el SO tiene más de un proceso en el espacio del núcleo se encuentra el planificador de procesos que será el encargado de realizar cambios de contextos, es decir, poner un nuevo proceso en estado de ejecución para ello necesita recuperar la estructura del proceso y actualizar convenientemente los

registros del procesador para que el nuevo proceso tome el control. Estos cambios de contexto son costosos ya que consumen un tiempo considerable.

Los procesos son entidades pesadas. La estructura del proceso está en la parte del núcleo y cada vez que quiere acceder a ella tiene que realizar una llamada al sistema que consume tiempo extra del procesador.

Dentro de un proceso puede haber varios hilos de ejecución un hilo se puede definir como cada secuencia de control dentro de un proceso que ejecuta sus instrucciones de forma independiente. Un hilo es una entidad más ligera que el proceso, estos comparten la información entre procesos como códigos, datos, etc. Además sin un hilo modifica una variable del proceso, el resto de los hilos verán esa modificación. Los cambios de contexto entre procesos consumen poco tiempo del procesador.

Entonces un sistema multihilo hace referencia a la capacidad de un sistema operativo de dar soporte a múltiples hilos de ejecución en un único proceso. En este proceso los distintos hilos pueden tener un estado de ejecución (ejecutando, listo, etc), contexto de hilo que se almacena cuando no está en ejecución, pila de ejecución, por cada hilo un espacio de almacenamiento para variables locales, acceso a la memoria y recursos de su proceso (compartido con todos los hilos del mismo proceso).

Paralelismo en Python

De la innumerable cantidad de módulos que están disponibles en Python, existen algunos pensados para proporcionar ejecución concurrente y paralela mediante la creación de hilos y procesos respectivamente, que pueden mejorar considerablemente el tiempo de procesamiento y aprovechar de mejor manera los recursos de hardware. Se describen a continuación los módulos del lenguaje Python para lograr paralelización y concurrencia, estas son Multiprocessing y Multithreading respectivamente.

Multiprocessing vs Multithreading

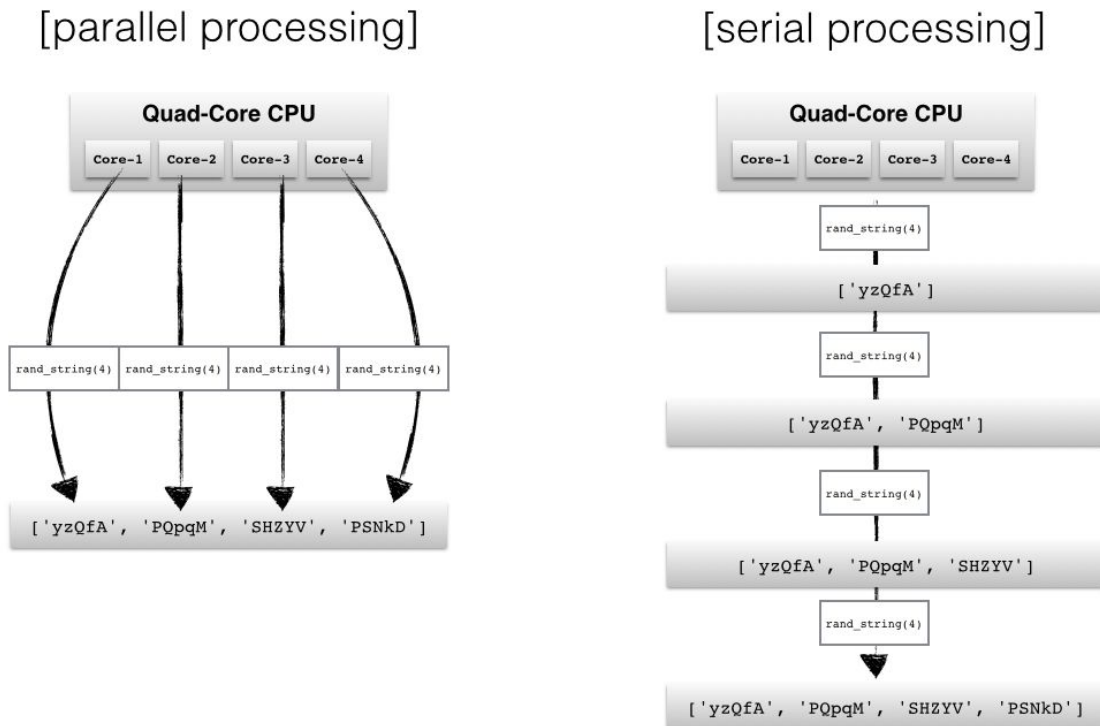
Inicialmente podemos encontrar diferencias entre las dos formas de paralelismo en Python, una de las formas utiliza hilos y la otra procesos. Los hilos son entidades más livianas que los procesos y no necesitan intervención del núcleo ya que comparten el espacio de memoria, no así los procesos, ya que estos últimos se ejecutan utilizando espacios de memoria dedicados para cada proceso.

Los hilos en Python no pueden usarse para implementar paralelismo real, lo cual se verá más adelante en el apartado Investigación y Pruebas, ya que al compartir un mismo espacio de memoria, la implementación más ampliamente difundida del lenguaje, la cual es CPython, mediante la incorporación de GIL (global interpreter locking) previene que varios hilos escriban en ese espacio de memoria al mismo tiempo. De esta manera lo que se consigue al utilizar hilos en Python son aplicaciones concurrentes. Para más detalle sobre GIL véase el anexo.

Para el desarrollo de aplicaciones haciendo uso de hilos, en Python se utiliza el módulo `threading`.

La librería `multiprocessing` permite la creación de procesos que se ejecutan de forma independiente y en espacios de memorias distintos, en base a esto los procesos pueden tener mayor dificultad para comunicarse y/o compartir información, es necesario la implementación de mecanismos de sincronización cuando se desee compartir recursos entre procesos. La ventaja de `multiprocessing` es que hay un GIL (global interpreter locking) independiente en cada proceso, lo que permite que estos se ejecuten en simultáneo, proporcionando un paralelismo real en las aplicaciones que hagan uso de este módulo.

El siguiente esquema¹ muestra la diferencia entre ambos enfoques



¹ "An introduction to parallel programming using ... - Sebastian Raschka."

https://sebastianraschka.com/Articles/2014_multiprocessing.html. Se consultó el 23 oct.. 2018.

Entonces los hilos son fáciles de crear, destruir y consumen poca memoria ya que es compartida, la creación y destrucción de procesos es más lenta además poseen la característica de que en Python los procesos no hacen una copia completa de la memoria, sino que comparten memoria con el padre y sólo cuando tienen que escribir en memoria acuden a un nuevo espacio de memoria.

Multiprocessing: Pool vs Process

El uso de multiprocessing en Python puede hacer uso de dos clases llamadas Pool y Process, las cuales difieren en algunos aspectos.

Process hace uso de las funciones start() y join(), donde la primera inicia los procesos y ejecuta la tarea que se le asigna mediante "target".

Por ejemplo:

```
p=Process(target=func, args=())
```

Luego mediante join() el hilo principal espera la finalización de los iniciados dentro del programa para continuar con su ejecución.

Usando Process nos aseguramos que a cada proceso se le asigna una tarea en concreto.

El uso más común de Pool es un poco diferente, pues el propósito también cambia. En este caso la función o tarea es una, y lo que se busca es dividir la ejecución de esa tarea entre diferentes intervalos de argumentos. Lo anterior mencionado se lleva a cabo usando Pool.map.

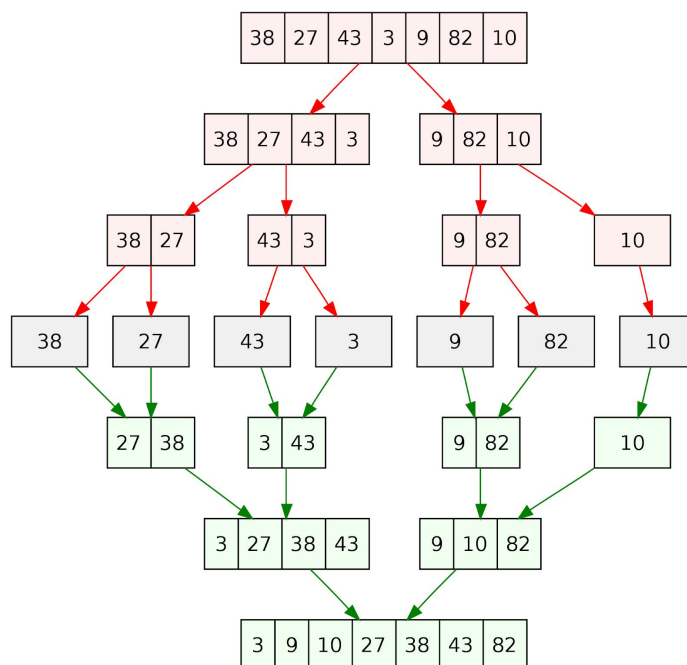
Como se verá más adelante Pool.map es muy útil en la implementación del algoritmo de ordenamiento Merge Sort, en donde la función se ejecuta en paralelo por diferentes procesos corriendo en diferentes cores pero usando diferentes argumentos.

Algoritmo Merge Sort

Merge Sort es uno de los diversos algoritmos de ordenamiento disponibles actualmente, del que se pueden encontrar diversas implementaciones en una gran variedad de lenguajes.

El algoritmo primero realiza una división en bloques de manera recursiva hasta que se obtenga un simple valor, y luego se realiza una unión sucesiva (merge) de esas partes ordenándose entre sí, para dar lugar a un vector del tamaño del original pero con los valores en él ya ordenados.

Tomemos la siguiente imagen² como un ejemplo:



El algoritmo es bastante simple como se puede ver en la imagen. Se toma un arreglo o vector de elementos y se los va dividiendo en mitades, o con un valor de diferencia entre las partes si el vector contiene un número impar de elementos.

Como se comentó anteriormente, se divide recursivamente hasta obtener un par de elementos, esto implica que cada subdivisión es una nueva llamada a la función.

En el momento en que queda el vector dividido en simples elementos se llama a una función merge que va agrupando. Es importante mencionar que cada bloque que recibe la función "merge" ya se encuentra ordenado, y el proceso también es recursivo.

Las características de este algoritmos de ordenamiento hacen que sea muy susceptible a ser implementado explotando el paralelismo logrando una reducción muy notoria en los tiempos de conclusión.

² "File:Merge sort algorithm diagram.svg - Wikimedia Commons." 17 sept.. 2018, https://commons.wikimedia.org/wiki/File:Merge_sort_algorithm_diagram.svg. Se consultó el 26 oct.. 2018.

Investigación y pruebas

La primera tarea a realizar, en principio fue probar las capacidades de paralelización que el lenguaje Python ofrece mediante sus variadas librerías, observando e identificando cuáles de ellas nos permiten obtener una ejecución concurrente y cuáles nos permiten obtener una ejecución paralela. Esto será de mayor importancia más adelante en el análisis de rendimiento en la Raspberry Pi.

Para lo antes mencionado se probaron diferentes proyectos ya desarrollados disponibles en la web, en los cuales se realizaron modificaciones que nos permitiera entender su funcionamiento y observar comportamiento.

Rendimiento de CPU con algoritmo de ordenamiento Merge Sort

Para las pruebas con Merge Sort se utilizó como base un código [\[1\]](#) obtenido desde la web que funciona con Python2.7, se le realizaron algunas modificaciones para que sea posible correrlo en Python3.

El algoritmo se ejecuta de manera procedural y luego de forma paralela para comparar tiempos de ejecución, usando para este último caso el módulo multiprocessing-Pool.

Código (modificado de la fuente): El archivo MergeSort.py que se encuentra en el repositorio [\[2\]](#).

Ejecuciones en notebook y raspberry pi:

Se realizaron una serie de pruebas en una notebook i3-5005u, 2Ghz con 4GB de memoria RAM y se utilizó la herramienta trace-cmd para graficar el uso de los núcleos y visualizar el paralelismo. Las pruebas se realizaron con diferentes cantidades de elementos a ordenar.

100 elementos:

```
Miguel Arch Linux pruebas/MergeSort master python MergeSort.py 100
Se generó una lista aleatoria de 100 elementos
El algoritmo demoró 0.000468 segundos
El algoritmo paralelo demoró 0.019172 segundos
```

1.10⁴ elementos:

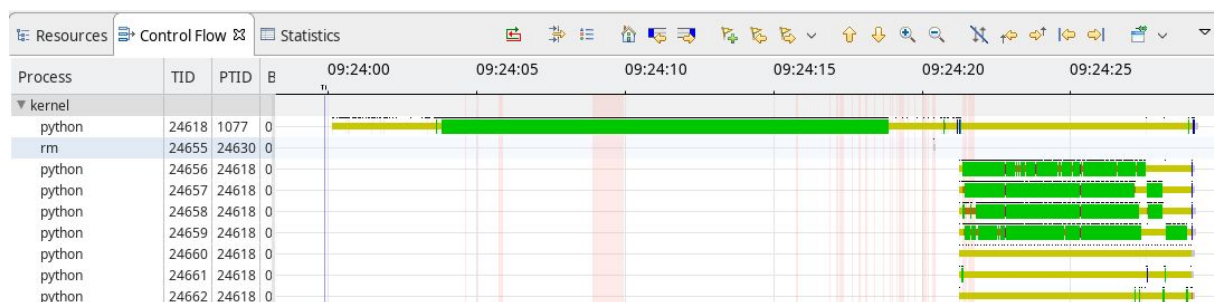
```
Miguel Arch Linux pruebas/MergeSort master python MergeSort.py 10000
Se generó una lista aleatoria de 10000 elementos
El algoritmo demoró 0.082205 segundos
El algoritmo paralelo demoró 0.069072 segundos
```

1.10⁶ elementos:

```
Miguel@Arch Linux ~$ cd pruebas/MergeSort && python MergeSort.py 1000000
Se generó una lista aleatoria de 1000000 elementos
El algoritmo demoró 12.088303 segundos
El algoritmo paralelo demoró 6.767313 segundos
```

Lo que se quiso demostrar en esta serie de ejemplos es que el paralelismo no siempre implica mejora en tiempos de ejecución, y es por esto que en ocasiones puede ser mejor implementar procesos, en otras hilos, o simplemente dejar la ejecución en un solo hilo de ejecución.

A continuación se muestra la captura con lttng y Tracecompass que se realizó para 1.10⁶ elementos.



Se ve claramente cuando empieza el procesamiento en un solo hilo y cuando comienza la ejecución del algoritmo usando paralelismo.

Podemos observar también en la ejecución en paralelo cuando se ejecuta la función merge y que procesador lo hace.



Como se puede ver en la ampliando la imagen existe solapamiento real en la ejecución del programa, demostrando que con el módulo multiprocessing podemos ejecutar en simultáneo.

Luego se realizaron las pruebas en una Raspberry Pi V1.2 con ARMv7 rev 4 de 4 núcleos y 1GB de memoria RAM.

50 elementos:

```
pi@raspberrypi:~/pps/pps/pruebas/MergeSort $ python3.5 MergeSort.py 50
Se generó una lista aleatoria de 50 elementos
El algoritmo demoró 0.001460 segundos
El algoritmo paralelo demoró 0.108044 segundos
```

$5 \cdot 10^3$ elementos:

```
pi@raspberrypi:~/pps/pps/pruebas/MergeSort $ python3.5 MergeSort.py 5000
Se generó una lista aleatoria de 5000 elementos
El algoritmo demoró 0.273277 segundos
El algoritmo paralelo demoró 0.231720 segundos
```

$5 \cdot 10^4$ elementos:

```
pi@raspberrypi:~/pps/pps/pruebas/MergeSort $ python3.5 MergeSort.py 50000
Se generó una lista aleatoria de 50000 elementos
El algoritmo demoró 3.340397 segundos
El algoritmo paralelo demoró 1.244065 segundos
```

Nuevamente observamos cómo a medida que la cantidad de valores aumenta, el paralelismo se hace más efectivo, reduciendo los tiempos de ejecución. La cantidad de valores inferiores a las utilizadas en la notebook es con el propósito de evitar daños en el hardware ya que se nota el calentamiento excesivo que ocurre en la placa al utilizar una cantidad de valores demasiado grandes.

Las imágenes para la ejecución en la Raspberry se obviaron, pues no difieren notoriamente de las hechas para la ejecución en la notebook, ya que ambos dispositivos poseen 4 núcleos. La mayor diferencia existe en la capacidad de procesamiento en cuanto a cantidad de valores se refiere, lo que influye en los tiempos de conclusión.

Rendimiento de CPU con operaciones matriciales - Multiprocessing vs Multithreading

El objetivo de este análisis consiste en comparar el rendimiento entre las librerías de Multiprocessing y Multithreading en base a un programa que hace uso extensivo del CPU, mediante operaciones matriciales.

El programa consiste en generar 4 matrices de números aleatorios cada matriz es tratada por un hilo o proceso distinto el cual realiza la suma de todos los valores de la misma y almacena el resultado en una estructura de datos compartida

(cola). Cuando finaliza el procesamiento de cada hilo o proceso los resultados de las 4 matrices son sumados por el programa principal.

Código implementado con Multiprocessing (process_queue_3.py) y Multithreading (thread_queue_3.py) ambos obtenidos de internet [3] y modificado (que también se encuentra en el repositorio [2]) :

Resultado de ejecución en python3 en una computadora portátil con un procesador Intel Core i5-7200U con 2 núcleos - 4 subprocesos, memoria RAM 8 GB DDR4 2133MHz:

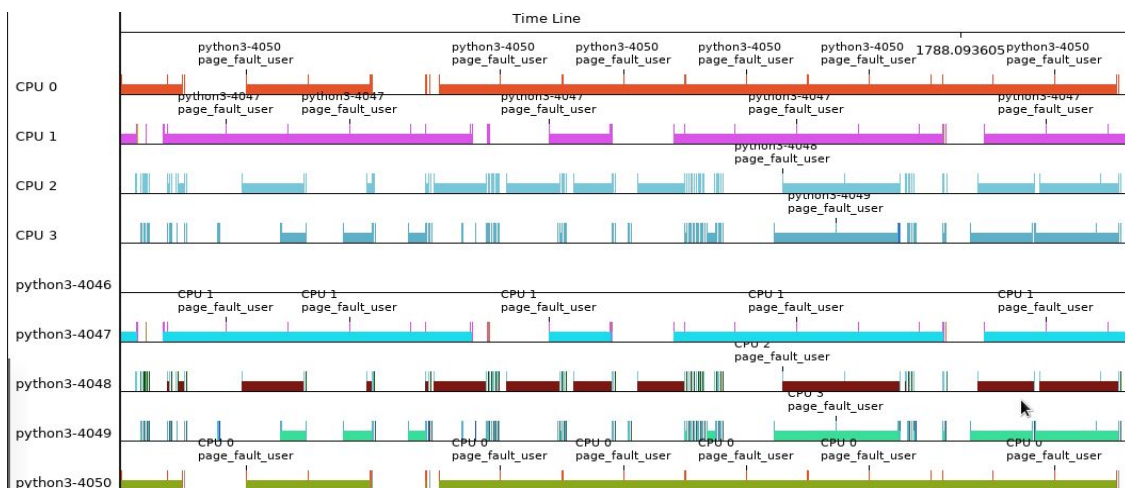
Processing:

```
> python3 process_queue_3.py 10000000
Tiempo total 5.152329444885254 segundos
Suma del resultado final: 50007380
```

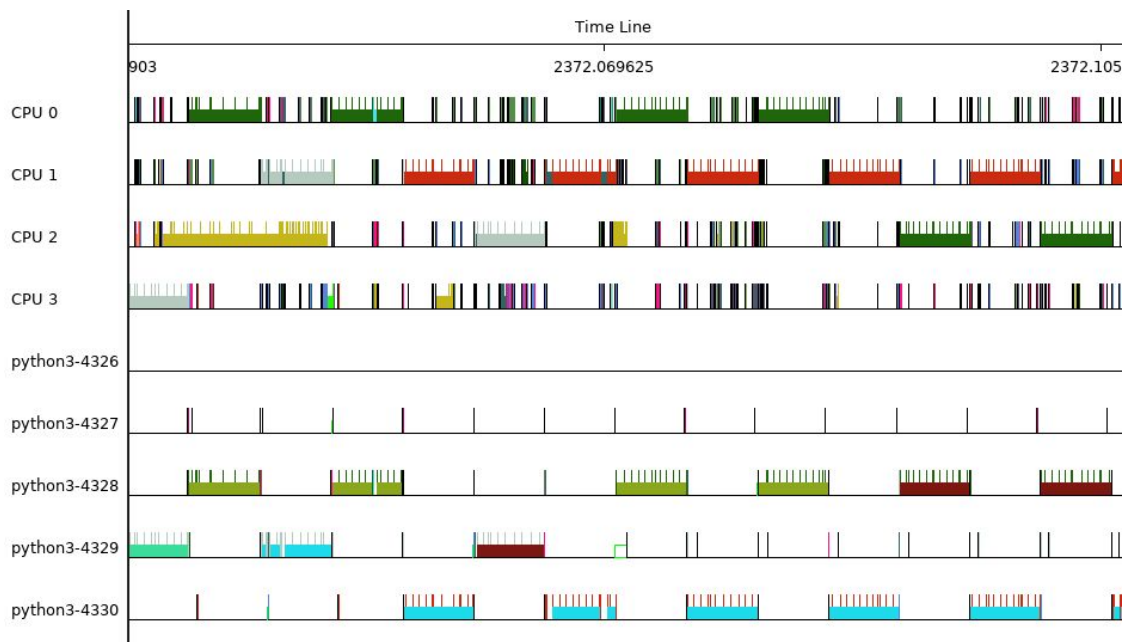
Threading

```
> python3 thread_queue_3.py 10000000
Tiempo total 9.926498889923096 segundos
Suma del resultado final: 49996160
```

Análisis usando trace-cmd con Processing:



Análisis usando trace-cmd con Threading:



Los análisis realizados se condicen con los supuestos del funcionamiento de las librerías threading y multiprocessing. Threading ejecuta los hilos de forma concurrente, un hilo a la vez condicionado por el mutex propio de la librería (GIL). La librería multiprocessing provee un mecanismo de ejecución de procesos en paralelo, se observa en la imagen de grabación del planificador del procesador momentos en que se están utilizando todos los núcleos al mismo tiempo.

Resultado de ejecución en una Raspberry PI 3 model B con un procesador Quad Core 1.2GHz Broadcom BCM2837 64bit, memoria 1GB RAM:

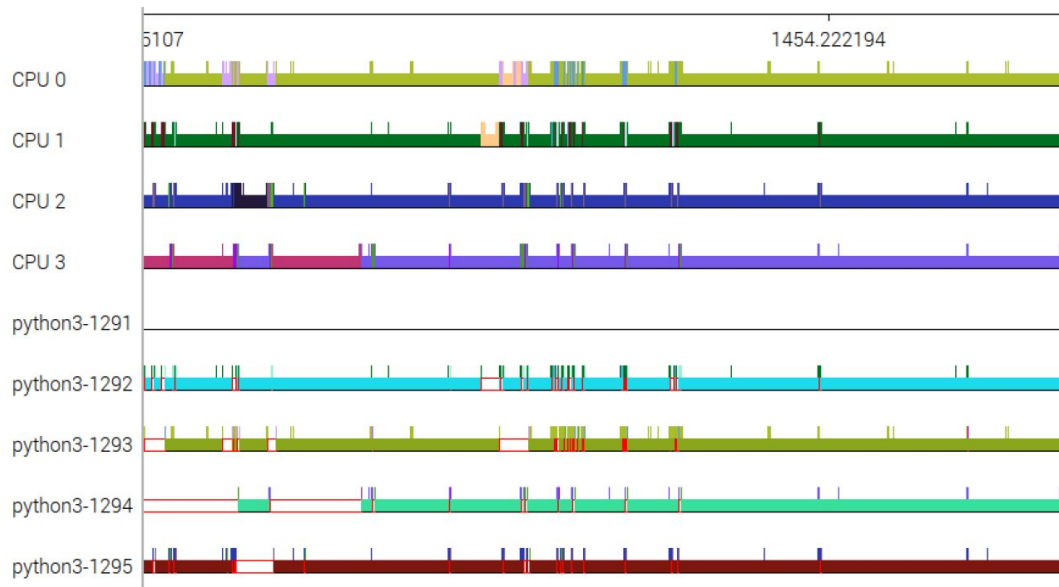
Processing:

```
> python3 process_queue_3.py 1000000
Tiempo total 4.256089448928833 segundos
Suma del resultado final: 4999597
```

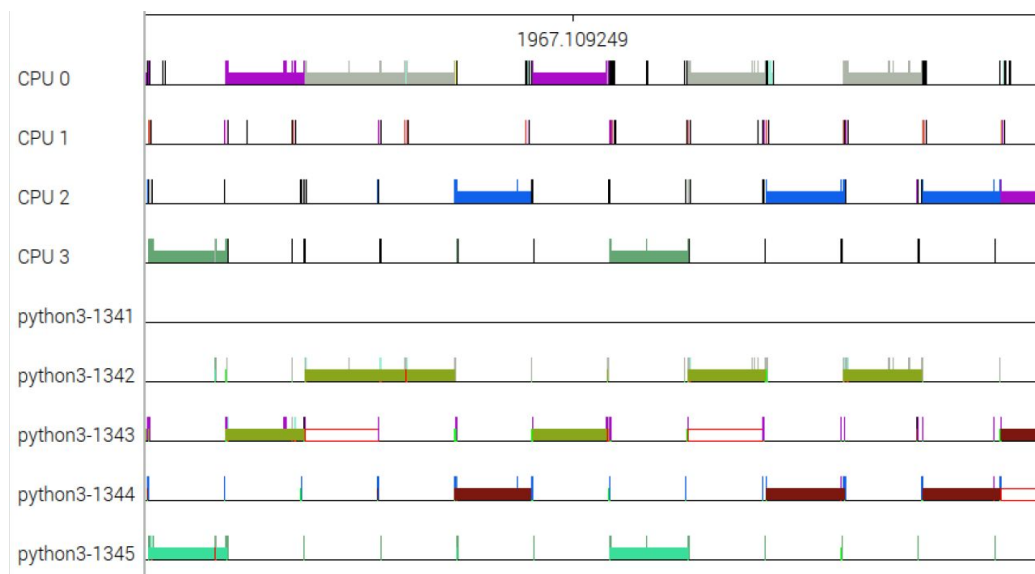
Threading:

```
> python3 thread_queue_3.py 1000000
Tiempo total 18.069430828094482 segundos
Suma del resultado final: 4996900
```


Análisis usando trace-cmd con Processing:



Análisis usando trace-cmd con Threading:



Rendimiento de CPU con Productor - Consumidor utilizando OpenCV - Multiprocessing vs Multithreading

El objetivo de este análisis consiste en comparar el rendimiento entre las librerías de Multiprocessing y Multithreading en base a la librería OpenCV empleando la función Canny para obtener los bordes de una imagen.

El programa consiste en un productor que carga las imágenes en una estructura de datos compartida (cola), diversos consumidores obtienen una imagen de la cola y la procesan hasta obtener la imagen con los bordes en base a la función Canny de la librería OpenCV, cuando el productor finaliza vuelve a consultar la cola de imagenes. El programa usa semáforos para sincronizar la cantidad de elementos en la cola con los permisos para producir.

Código implementado con Multiprocessing y Multithreading se puede ver en el repositorio de la Práctica Supervisada `opencv_process_multi_pro_multi_cons.py` y `opencv_process_threading_multi_pro_multi_cons.py` [\[2\]](#).

Resultado de ejecución en python3 en una computadora portátil con un procesador Intel Core i5-7200U con 2 nucleos - 4 subprocesos, memoria RAM 8 GB DDR4 2133MHz:

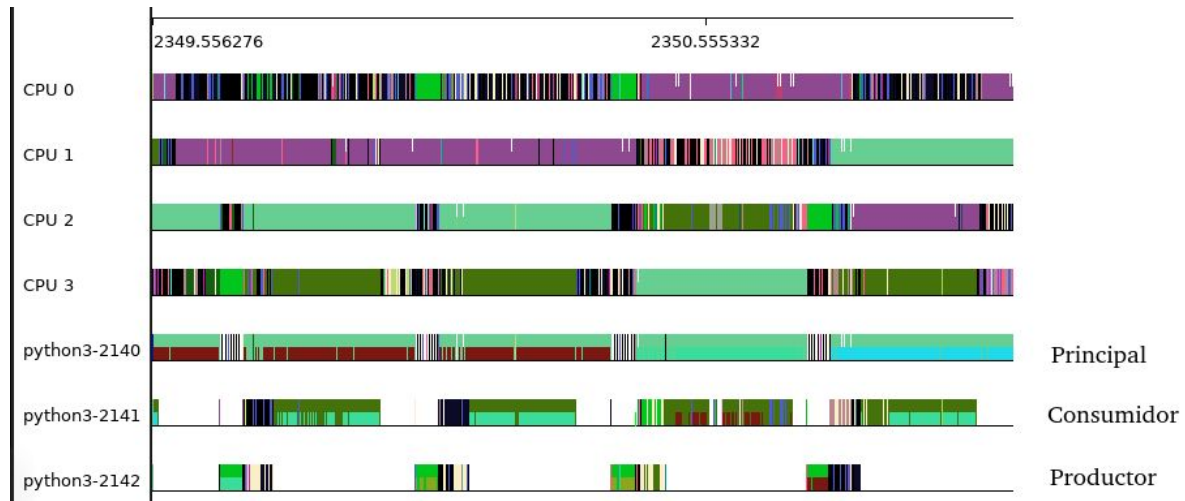
Processing:

```
> python3 opencv_process_multi_pro_multi_cons.py 1 8 15
Producer-1: i produced an images
Consumer-2: i consumed an images
Producer-1: i produced an images
Producer-1: i produced an images
Consumer-3: i consumed an images
Producer-1: i produced an images
Consumer-4: i consumed an images
Producer-1: i produced an images
Consumer-5: i consumed an images
Producer-1: i produced an images
Consumer-6: i consumed an images
Producer-1: i produced an images
Consumer-7: i consumed an images
Producer-1: i produced an images
```

Threading:

```
> python3 opencv_threading_multi_pro_multi_cons.py 1 4 15
produzco
Thread-1: i produced an images
Thread-2: i consumed an images
Thread-1: i produced an images
Thread-3: i consumed an images
Thread-1: i produced an images
Thread-4: i consumed an images
Thread-1: i produced an images
Thread-5: i consumed an images
Thread-1: i produced an images
Thread-2: i consumed an images
Thread-1: i produced an images
Thread-3: i consumed an images
```

Análisis usando trace-cmd con Processing (anulando la función Canny) con 1 productor y un consumidor:



Con la librería Canny deshabilitada el programa funciona como un productor que lee y guarda imágenes en una cola y un consumidor que saca elementos de la cola cada vez que están disponibles y los guarda en una carpeta.

A simple vista identificamos tres procesos corriendo, el proceso principal (padre), el proceso productor y el proceso consumidor.

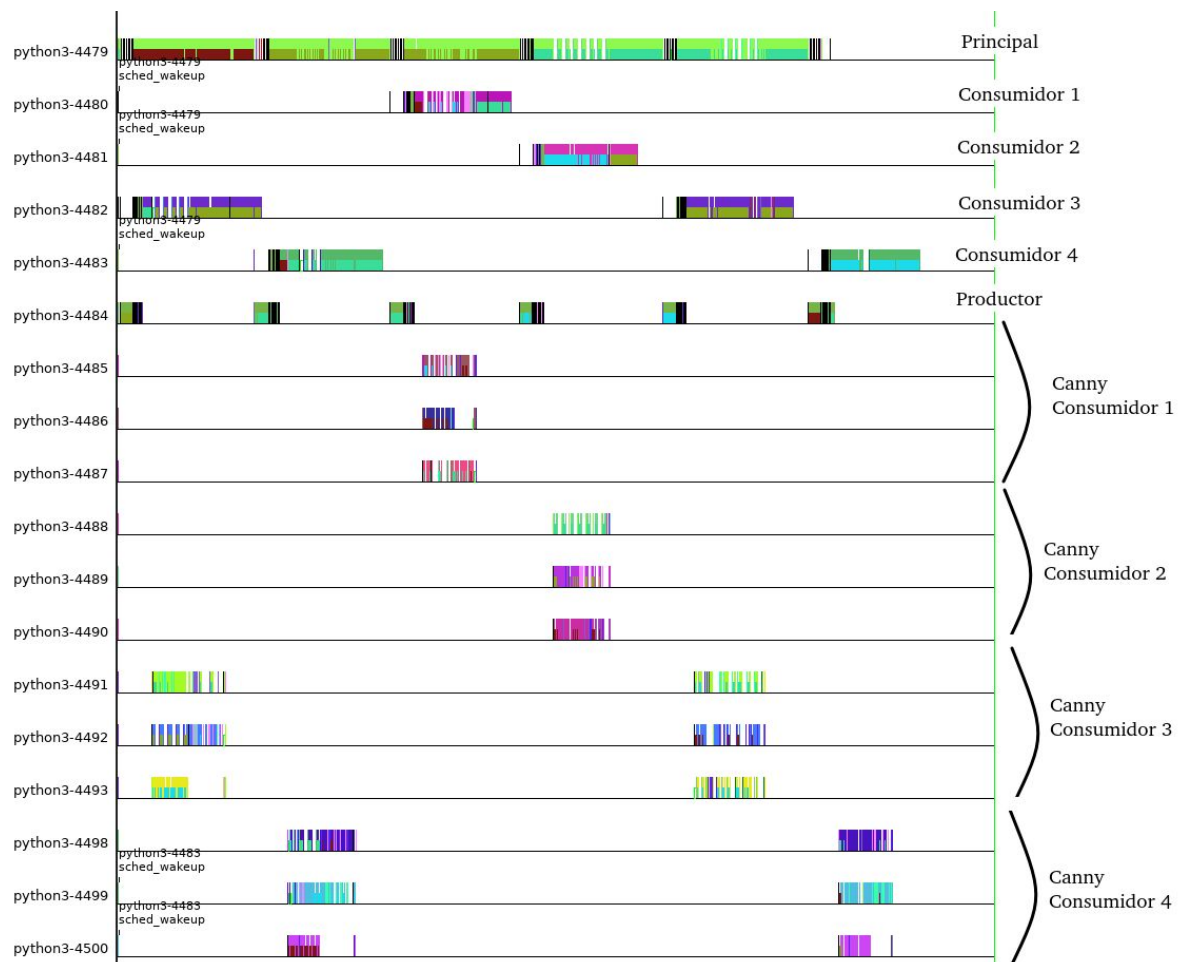
Análisis usando trace-cmd con Processing con un productor un consumidor y la función Canny implementada:



Con la librería Canny habilitada el programa funciona como un productor que lee y guarda imágenes en una cola y un consumidor que saca elementos de la cola cada vez que están disponibles los procesa con la función Canny y los guarda en una carpeta.

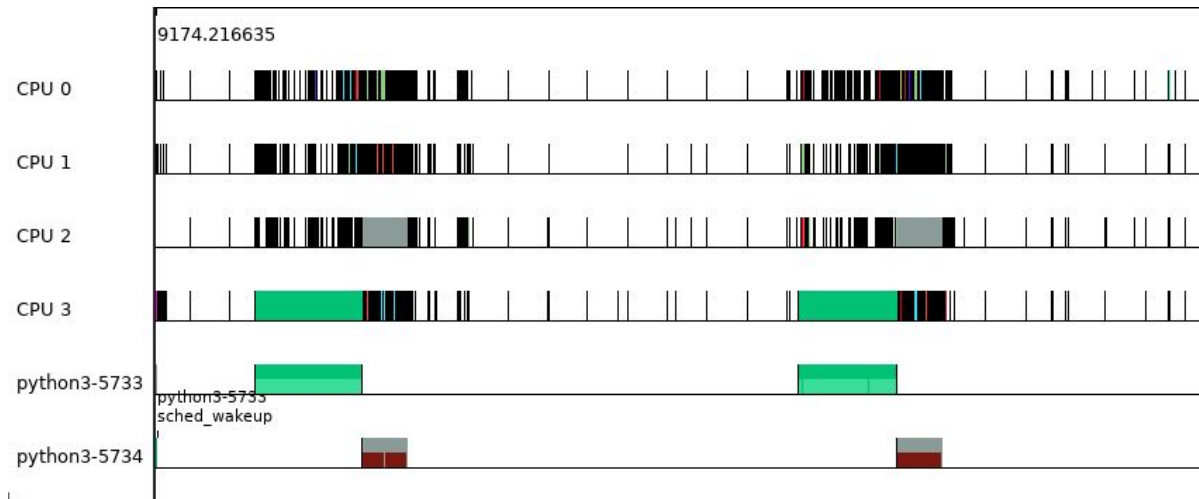
En este caso donde el productor carga la imagen y inmediatamente el consumidor comienza el procesamiento, la función Canny utiliza tres procesos paralelos.

Análisis usando trace-cmd con Processing con un productor, cuatro consumidores y la función Canny implementada:



En el gráfico observamos que para cada proceso consumidor Canny crea tres procesos propios que re-utiliza cada vez que el proceso consumidor llame a la función canny. Esto se puede observar en el proceso consumidor 3 y 4.

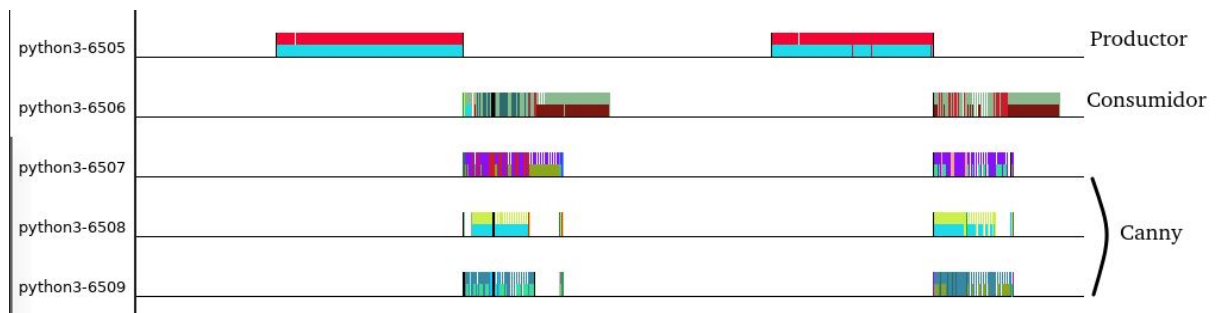
Análisis usando trace-cmd con Threading (anulando la función Canny) con 1 productor y un consumidor:



Con la librería Canny deshabilitada el programa funciona como un productor que lee y guarda imágenes en una cola y un consumidor que saca elementos de la cola cada vez que están disponibles y los guarda en una carpeta.

A simple vista identificamos dos threads corriendo productor y consumidor.

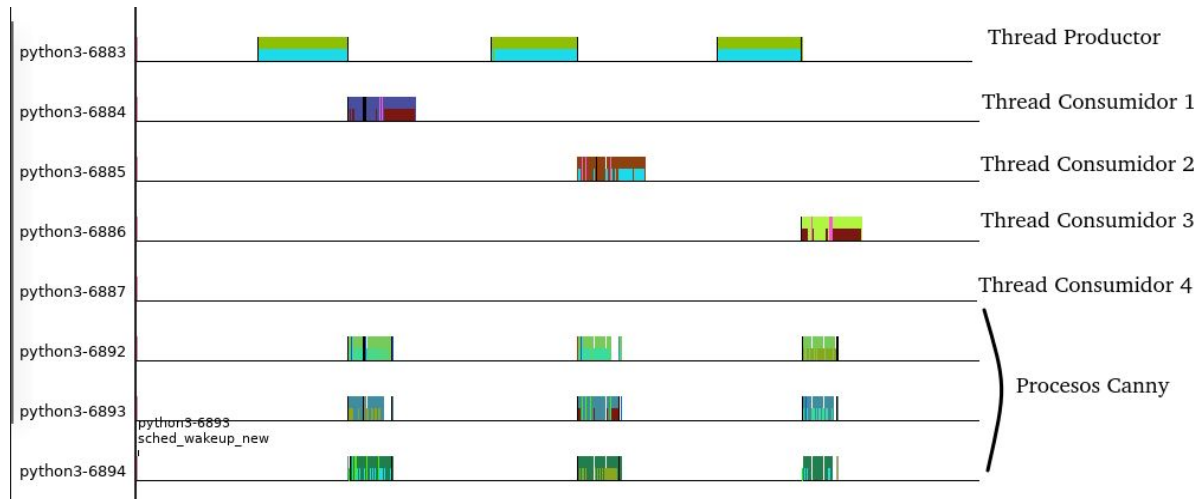
Análisis usando trace-cmd con Threading con un productor un consumidor y la función Canny implementada:



Con la librería Canny habilitada el programa funciona como un productor que lee y guarda imágenes en una cola y un consumidor que saca elementos de la cola cada vez que están disponibles los procesa con la función Canny y los guarda en una carpeta.

En este caso donde el productor carga la imagen y inmediatamente el consumidor comienza el procesamiento, la función Canny utiliza tres procesos paralelos.

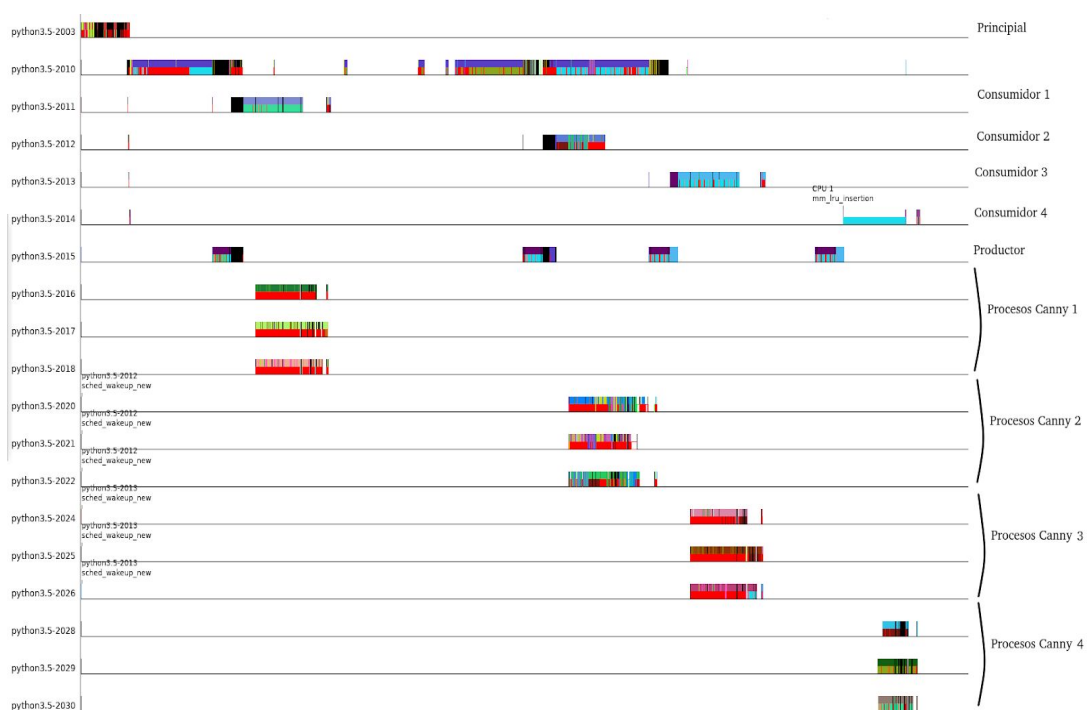
Análisis usando trace-cmd con Threading con un productor, cuatro consumidores y la función Canny implementada:



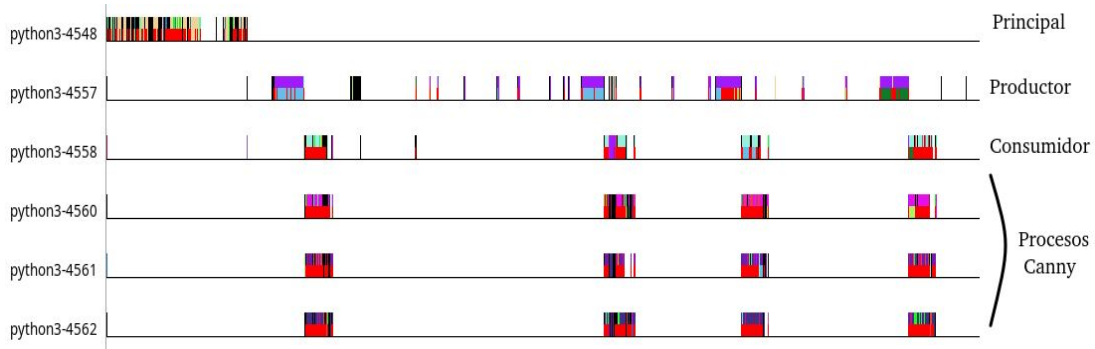
En el gráfico observamos que para cada Threading consumidor Canny se ejecuta de forma alternada, pero se los procesos Canny que crea son comunes a todos los threads (a diferencia de los procesos donde cada uno utilizaba sus propios procesos canny).

Resultado de ejecución en una Raspberry PI 3 model B con un procesador Quad Core 1.2GHz Broadcom BCM2837 64bit, memoria 1GB RAM:

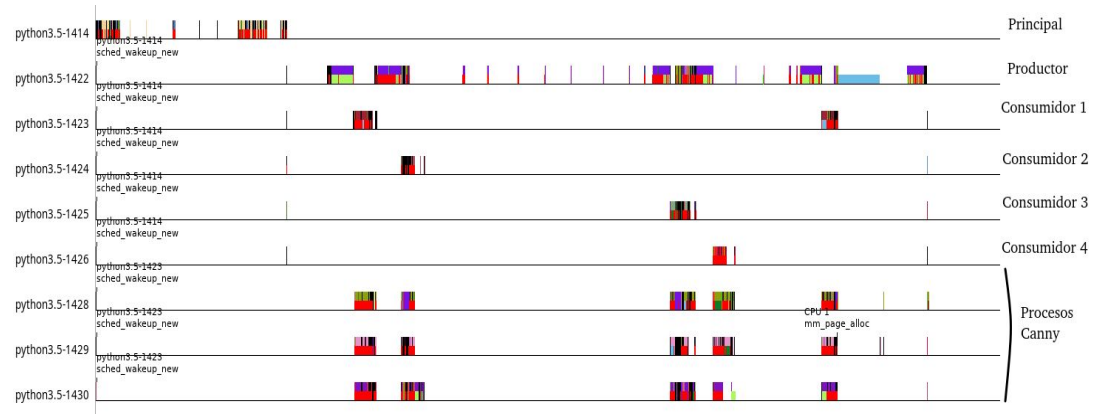
Análisis usando trace-cmd con Processing:



Análisis usando trace-cmd con Threading (un solo thread):



Análisis usando trace-cmd con Threading (4 threads):

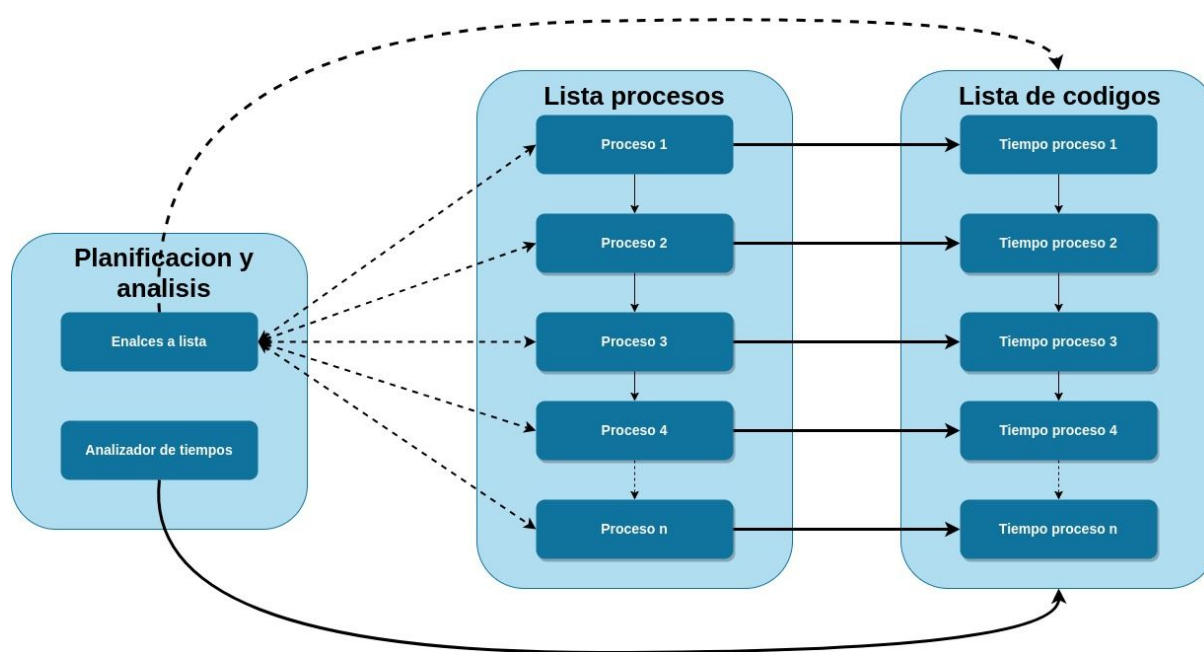


Desarrollo

En este apartado se diseña un mecanismo para medir la efectividad y velocidad de respuesta de la Raspberry Pi llevándola a límites extremos de procesamiento. Es decir, se sobrecarga la Raspberry mediante el lanzamiento de numerosos procesos que ejecutan funciones de cálculos matriciales y mergesort con la finalidad de medir el tiempo de respuesta.

En este punto se abandona el desarrolló con la librería threading y se enfoca únicamente en el uso de multiprocessing.

El marco utilizado para esta prueba se muestra en el siguiente esquema:



Tenemos un proceso padre (consumidor) que es el encargado de crear una lista de procesos y ejecutarlos al tiempo que crea una lista que es un espacio de memoria compartido en la que cada proceso (productor) al terminar de ejecutar una tarea encomendada escribe el tiempo actual en la posición de la lista que le corresponde.

Mientras los procesos productores están el pleno funcionamiento el proceso padre realiza un round and robin de la lista determinando cuando un proceso escribe un nuevo valor de tiempo actual, cuando esto ocurre el consumidor quita el tiempo de la lista hace la resta del tiempo actual y almacena ese valor en una lista como tiempo de respuesta. Toda la ejecución se da durante un tiempo determinado en el cual el consumidor (padre) finaliza todos lo hijos, para luego realizar un promedio de los tiempos de respuesta almacenados.

Cada proceso productor ejecuta un operaciones matriciales o mergesort, en el análisis se pretende variar la cantidad de procesos productores iniciando desde 1 hasta un máximo de 15 procesos y también la cantidad de elementos es decir tamaño de matriz o tamaño de lista del mergesort.

Códigos en python de análisis de matrices (se encuentran en el repositorio [2]):

- planificador_matrices.py
- sensor_matrices.py
- process_queue_3.py

Códigos en python de análisis usando merge short (se encuentran en el repositorio [2]):

- planificador_merge_lineal.py
- Sensor_merge_lineal.py
- MergeSort.py

Ejecución de pruebas finales

Para correr pruebas finales se utilizaron **tiempos_elementos.sh** y **tiempos_elementos.sh** ambos ejecutan el script **loadcpu.sh** para medir la carga del procesador durante la prueba (todos los scripts están detallados en el anexo).

Pruebas Matriciales

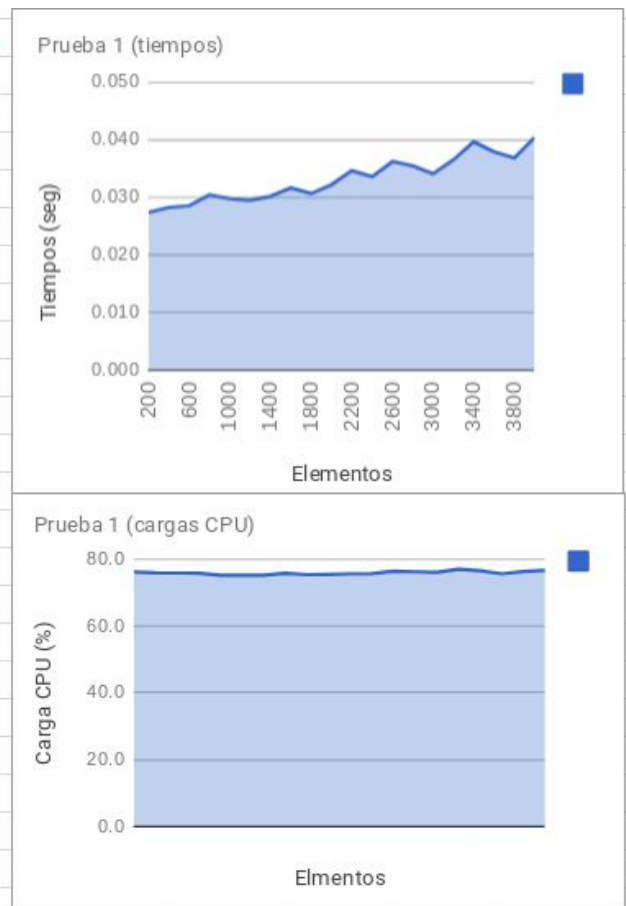
Prueba 1- 2 -3 (Matricial-Elementos):

Ejecución	Se ejecuta el tiempos_elementos.sh
Entorno	Raspberry Pi V1.2 con ARMv7 rev 4 de 4 núcleos y 1GB de memoria RAM
Descripción	El script tiempos_elemento.sh realiza un bucle partiendo de un número fijo de procesos (15) y modificando el número de elementos ejecutando varias instancias de planificador_matrices.py (consumidor) que se encarga de lanzar los procesos sensor_matrices.py (productores) y enlazarlos a un lista común para el análisis de tiempos de respuesta, los cuales se promedian dentro del mismo programa y se almacena la salida del mismo dentro del script. En total para cada valor de elementos (que se le pasa como parámetro a planificador_matrices.py al igual que el número fijo de procesos 15) se realizan 5 muestras y se promedian dentro del script. De esta forma el script nos devuelve un tiempo de respuesta promedio para cada valor distinto de elemento almacenando en un archivo de texto.
Observaciones	Cada proceso productor lanza el programa process_queue_3.py que a su vez crea 4 procesos adicionales.

Prueba 1(Matricial-Elementos):

Nombre	Prueba 1 matricial elementos
Parámetros	planificador_matrices.py recibe como parametros el número fijo de procesos durante la prueba (15) y se varían la cantidad de elementos de la matriz de 100 a 2000 con incrementos de 100 elementos. Realizando 5 muestras por cada valor de elementos .

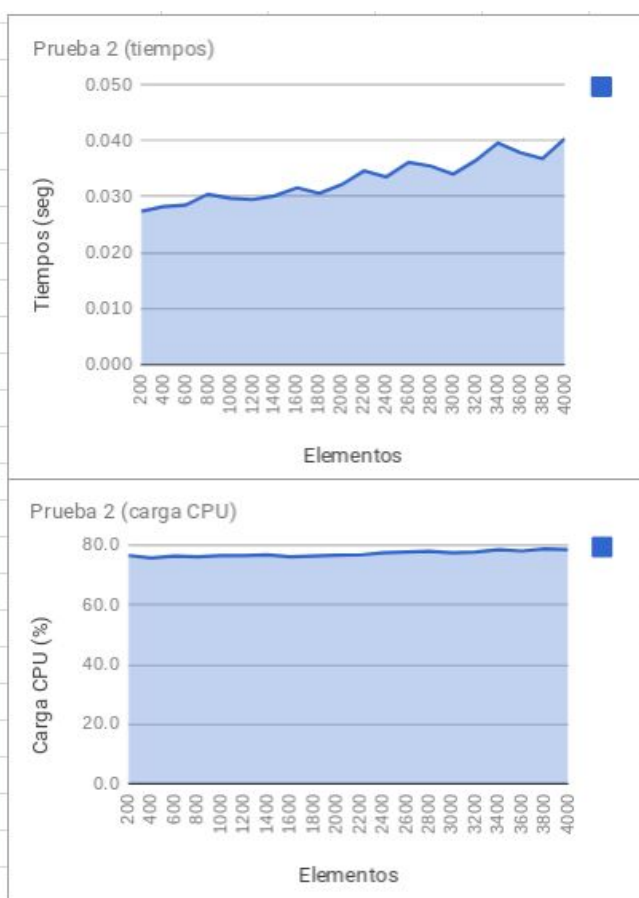
Elementos	Tiempos (seg)	Carga CPU (%)
100	0.027	76.2
200	0.026	75.9
300	0.027	75.9
400	0.027	75.8
500	0.027	75.2
600	0.027	75.2
700	0.028	75.2
800	0.030	75.8
900	0.031	75.4
1000	0.030	75.5
1100	0.030	75.7
1200	0.031	75.7
1300	0.031	76.4
1400	0.033	76.2
1500	0.034	76.1
1600	0.035	77.0
1700	0.036	76.5
1800	0.036	75.7
1900	0.036	76.3
2000	0.037	76.7



Prueba 2 (Matricial-Elementos):

Nombre	Prueba 2 matricial elementos
Parámetros	planificador_matrices.py recibe como parametros el número fijo de procesos durante la prueba (15) y se varían la cantidad de elementos de la matriz de 200 a 4000 con incrementos de 200 elementos. Realizando 5 muestras por cada valor de elementos .

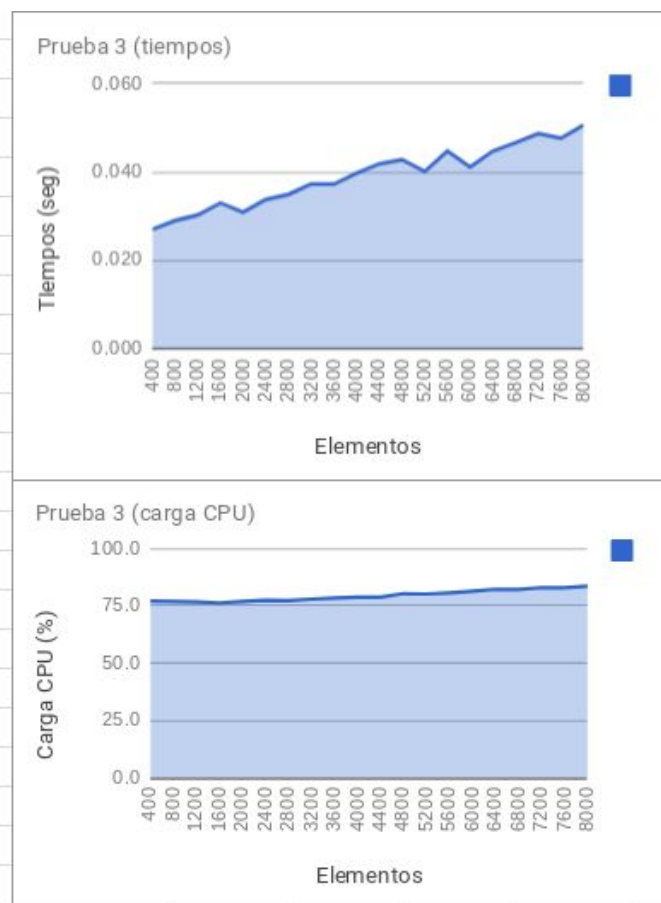
Elementos	Tiempos (seg)	Carga CPU (%)
200	0.027	76.6
400	0.028	75.8
600	0.029	76.4
800	0.030	76.1
1000	0.030	76.6
1200	0.029	76.5
1400	0.030	76.8
1600	0.032	76.2
1800	0.031	76.4
2000	0.032	76.6
2200	0.035	76.8
2400	0.034	77.4
2600	0.036	77.8
2800	0.035	78.0
3000	0.034	77.5
3200	0.036	77.8
3400	0.040	78.5
3600	0.038	78.1
3800	0.037	78.8
4000	0.040	78.6



Prueba 3 (Matricial-Elementos):

Nombre	Prueba 3 matricial elementos
Parámetros	planificador_matrices.py recibe como parametros el número fijo de procesos durante la prueba (15) y se varían la cantidad de elementos de la matriz de 400 a 8000 con incrementos de 400 elementos. Realizando 5 muestras por cada valor de elementos .

Elementos	Tiempos (seg)	Carga CPU (%)
400	0.027	77.1
800	0.029	76.9
1200	0.030	76.7
1600	0.033	76.3
2000	0.031	77.0
2400	0.034	77.5
2800	0.035	77.3
3200	0.037	77.9
3600	0.037	78.4
4000	0.040	78.8
4400	0.042	78.7
4800	0.043	80.3
5200	0.040	80.2
5600	0.045	80.7
6000	0.041	81.4
6400	0.045	82.2
6800	0.047	82.2
7200	0.049	83.0
7600	0.048	82.9
8000	0.051	83.7



Prueba 4 - 5 - 6 (Matricial-Procesos):

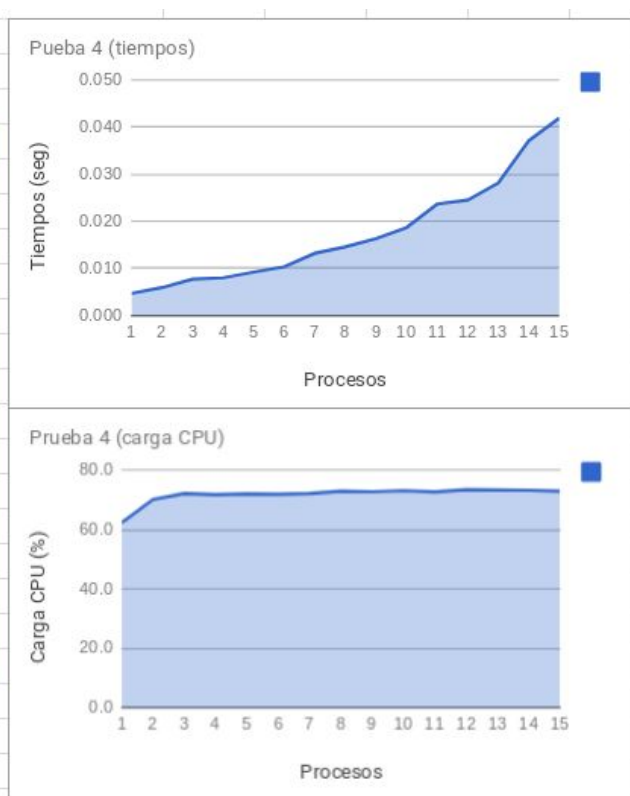
Ejecución	Se ejecuta el tiempos_elementos.sh
Entorno	Raspberry Pi V1.2 con ARMv7 rev 4 de 4 núcleos y 1GB de memoria RAM
Descripción	El script tiempos_elemento.sh realiza un bucle partiendo de un número fijo de elementos y modificando el número de procesos ejecutando varias instancias de planificador_matrices.py (consumidor) que se encarga de lanzar los procesos sensor_matrices.py (productores) y enlazarlos a un lista común para el análisis de tiempos de respuesta, los cuales se promedian dentro del mismo programa y se almacena la salida del mismo dentro del script. En total para cada valor de procesos (que se le pasa como parámetro a planificador_matrices.py al igual que el número fijo de elementos) se realizan 5 muestras y se promedian dentro del script. De esta forma el script nos devuelve un tiempo de respuesta promedio para cada valor distinto de procesos almacenando en un archivo de texto.
Parámetros	planificador_matrices.py recibe como parametros el número fijo de elementos durante la prueba (1000) y se varían la cantidad de procesos de

	1 a 15 con incrementos de a 1. Realizando 5 muestras por cada valor de elementos .
Observaciones	Cada proceso productor lanza el programa process_queue_3.py que a su vez crea 4 procesos adicionales.

Prueba 4 (Matricial-Procesos):

Nombre	Prueba 4 matricial procesos
Parámetros	planificador_matrices.py recibe como parametros el número fijo de elementos durante la prueba (1000) y se varían la cantidad de procesos de 1 a 15 con incrementos de a 1. Realizando 5 muestras por cada valor de elementos .

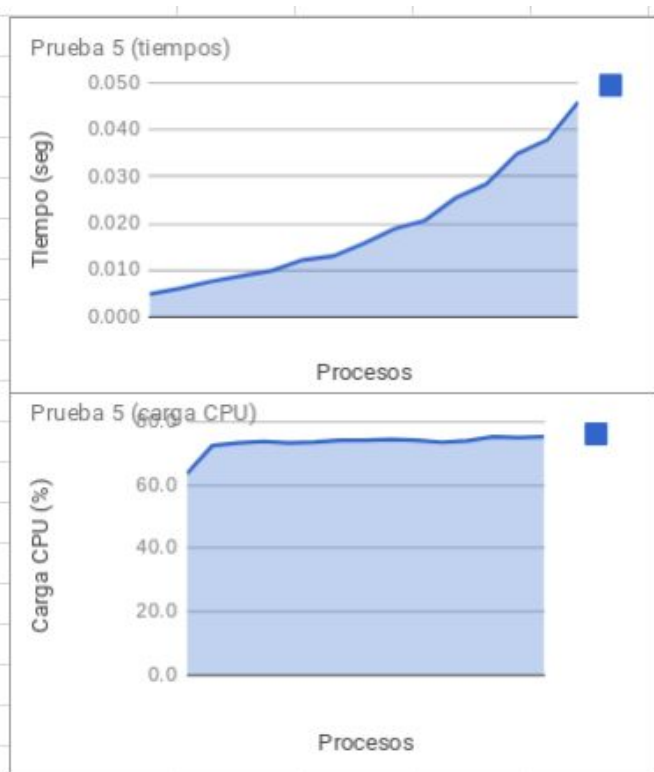
Procesos	Tiempos (seg)	Carga CPU (%)
1	0.005	62.4
2	0.006	70.2
3	0.008	72.1
4	0.008	71.8
5	0.009	72.0
6	0.010	71.9
7	0.013	72.2
8	0.015	72.9
9	0.016	72.8
10	0.019	73.1
11	0.024	72.7
12	0.025	73.5
13	0.028	73.4
14	0.037	73.3
15	0.042	72.9



Prueba 5 (Matricial-Procesos):

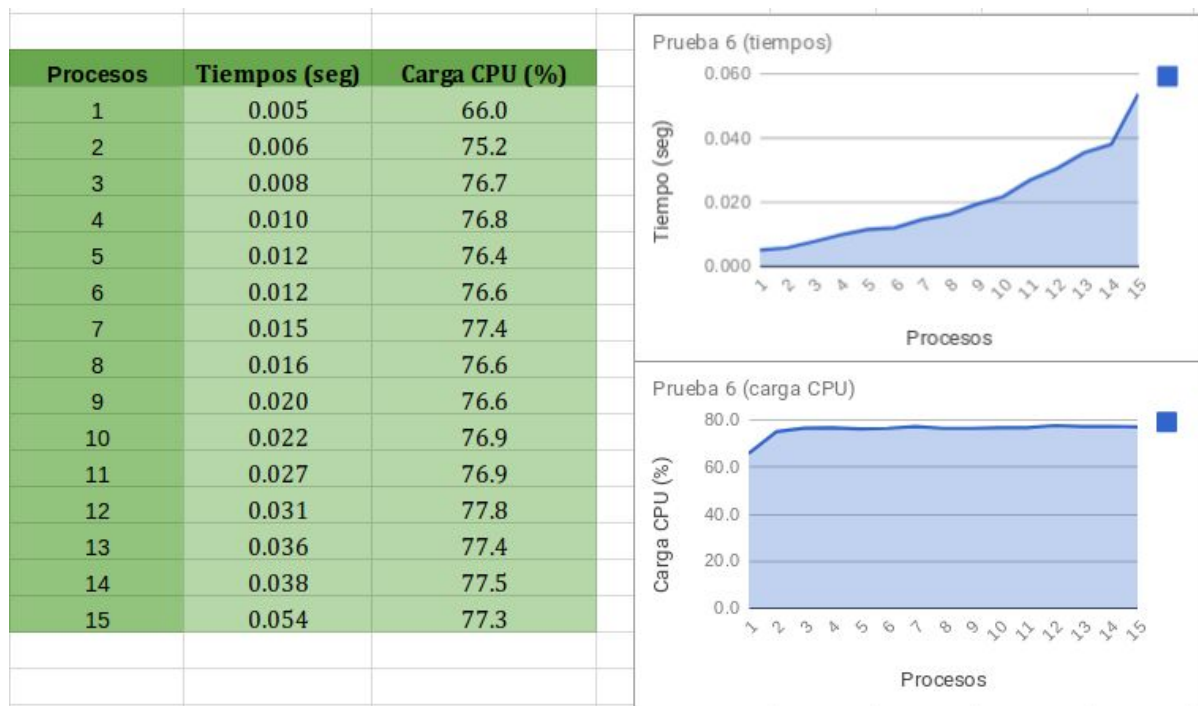
Nombre	Prueba 5 matricial procesos
Parámetros	planificador_matrices.py recibe como parametros el número fijo de elementos durante la prueba (2000) y se varían la cantidad de procesos de 1 a 15 con incrementos de a 1. Realizando 5 muestras por cada valor de elementos .

Procesos	Tiempos (seg)	Carga CPU (%)
1	0.005	63.7
2	0.006	72.6
3	0.008	73.4
4	0.009	73.9
5	0.010	73.5
6	0.012	73.7
7	0.013	74.3
8	0.016	74.3
9	0.019	74.6
10	0.021	74.3
11	0.026	73.7
12	0.028	74.1
13	0.035	75.4
14	0.038	75.2
15	0.046	75.4



Prueba 6 (Matricial-Procesos):

Nombre	Prueba 6 matricial procesos
Parámetros	planificador_matrices.py recibe como parametros el número fijo de elementos durante la prueba (4000) y se varían la cantidad de procesos de 1 a 15 con incrementos de a 1. Realizando 5 muestras por cada valor de elementos .



Pruebas Merge Sort - Multiprocessing

Entorno	Raspberry Pi V1.2 con ARMv7 rev 4 de 4 núcleos y 1GB de memoria RAM
Descripción	El script <code>tiempos_elemento.sh</code> realiza un bucle partiendo de un número fijo de elementos y modificando el número de procesos ejecutando varias instancias de <code>planificador_merge.py</code> (consumidor) que se encarga de lanzar los procesos <code>sensor_merge.py</code> (productores) y enlazarlos a un lista común para el análisis de tiempos de respuesta, los cuales se promedian dentro del mismo programa y se almacena la salida del mismo dentro del script. En total para cada valor de procesos (que se le pasa como parámetro a <code>planificador_merge.py</code> al igual que el número fijo de elementos) se realizan 5 muestras y se promedian dentro del script. De esta forma el script nos devuelve un tiempo de respuesta promedio para cada valor distinto de procesos almacenando en un archivo de texto.
Observaciones	Cada proceso productor lanza el programa <code>MergeSort.py</code> ejecutando solo Merge Sort lineal

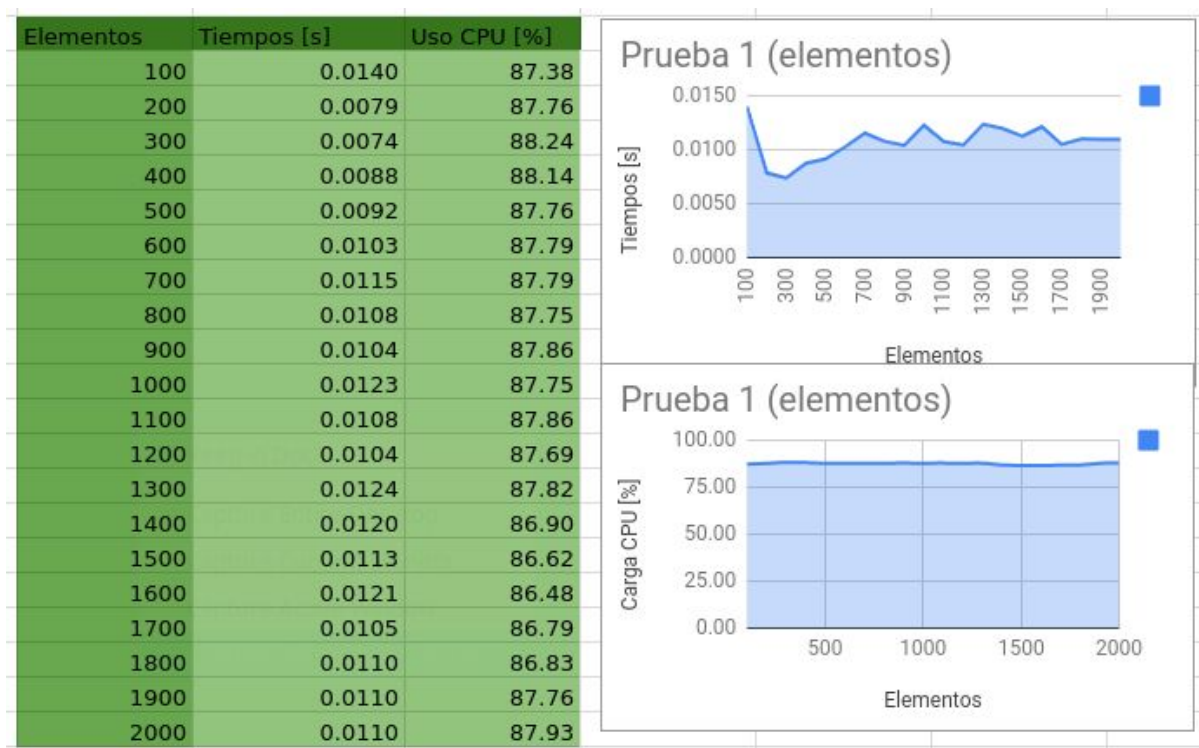
Pruebas 1 - 2 - 3 (Merge Sort - Elementos)

Ejecución	Se ejecuta tiempos_elementos.sh
-----------	--

Prueba 1 (Merge Sort -Elementos)

Nombre	Prueba 1 Merge elementos
Parámetros	planificador_merge.py recibe como parametros el número fijo de procesos igual a 15 y se varían la cantidad de elementos de 100 a 2000 con incrementos de 100. Realizando 5 muestras por cada valor de elementos .

Resultado de la prueba:

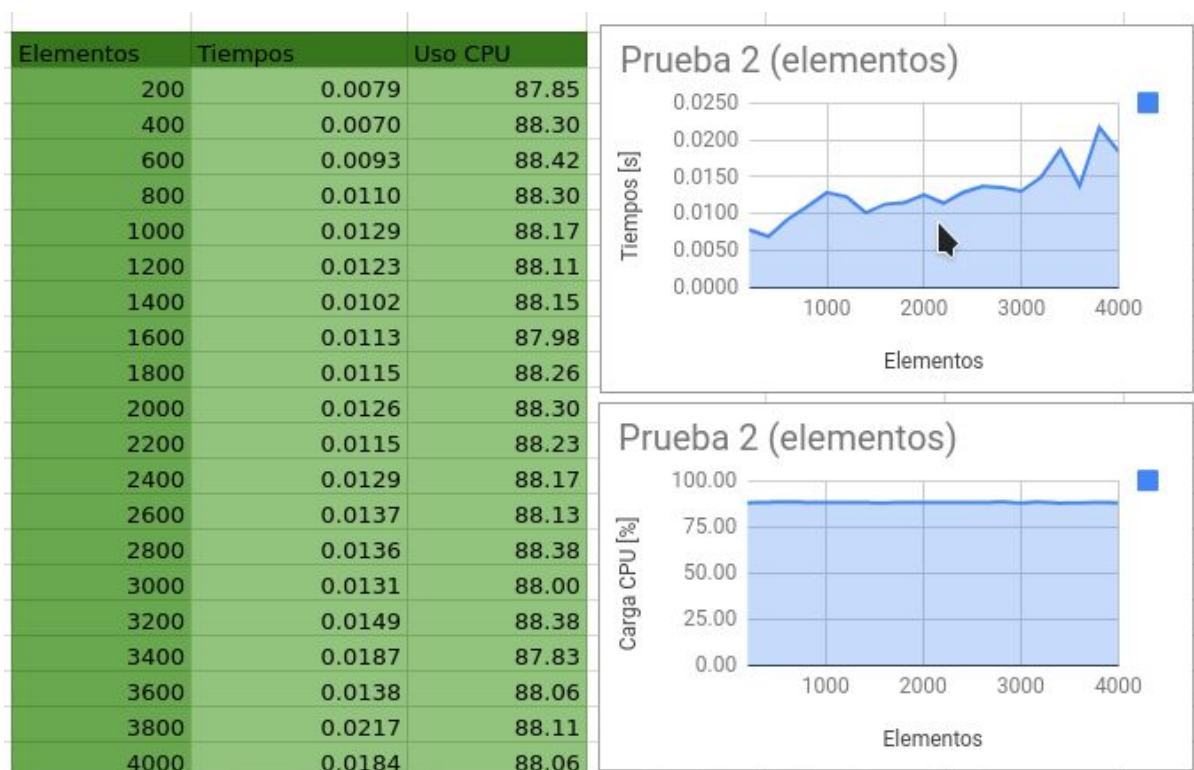


Prueba 2 (Merge Sort - Elementos)

Nombre	Prueba 2 Merge elementos
--------	--------------------------

Parámetros	planificador_merge.py recibe como parametros el número fijo de procesos igual a 15 y se varían la cantidad de elementos de 200 a 4000 con incrementos de 200. Realizando 5 muestras por cada valor de elementos .
------------	--

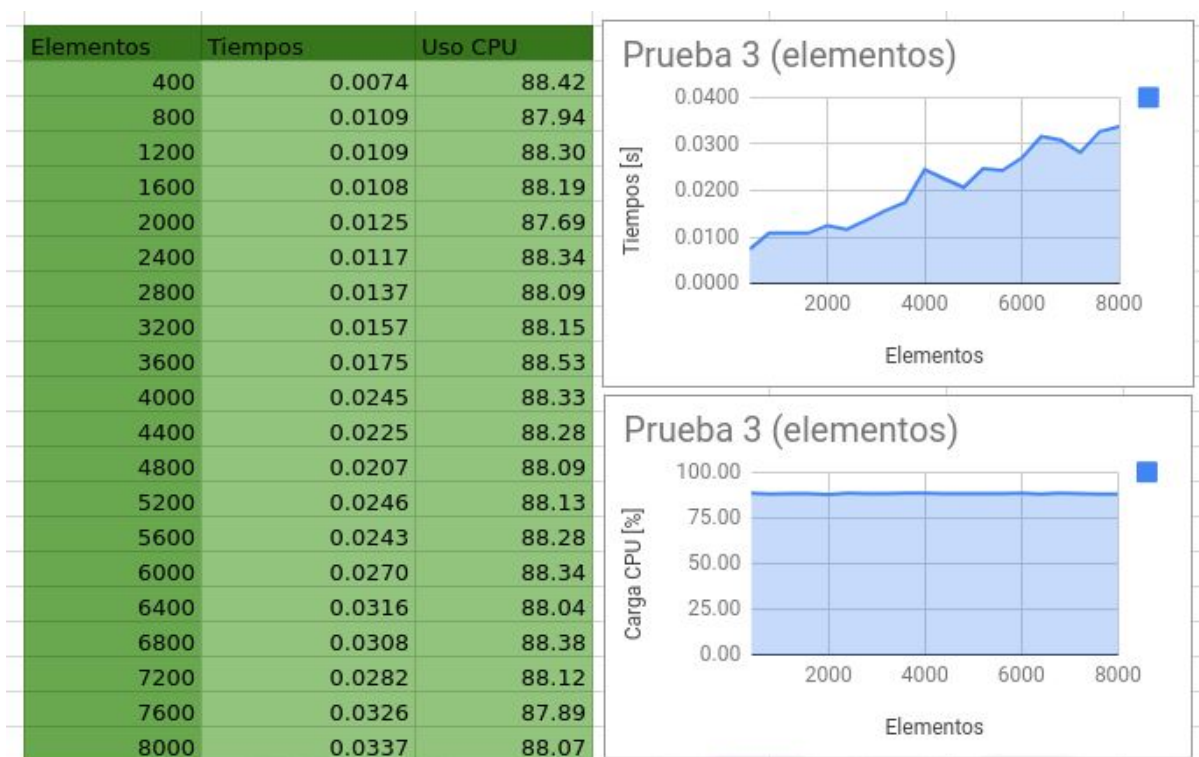
Resultado de la prueba:



Prueba 3 (Merge Sort - Elementos)

Nombre	Prueba 3 Merge elementos
Parámetros	planificador_merge.py recibe como parametros el número fijo de procesos igual a 15 y se varían la cantidad de elementos de 400 a 8000 con incrementos de 400. Realizando 5 muestras por cada valor de elementos .

Resultado de la prueba:



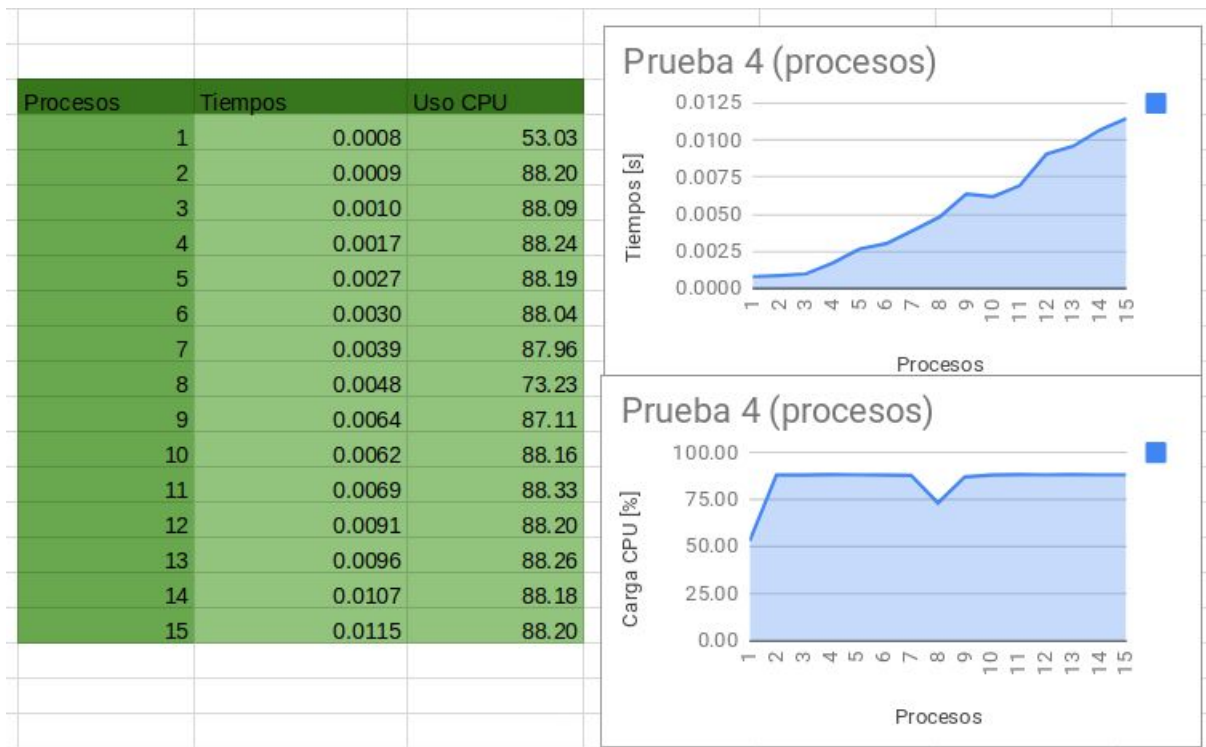
Pruebas 4 - 5 - 6 (Merge Sort - Procesos)

Ejecución	Se ejecuta tiempos_procesos.sh
-----------	---------------------------------------

Prueba 4 (Merge Sort - Procesos):

Nombre	Prueba 4 Merge procesos
Parámetros	planificador_merge.py recibe como parametros el número fijo de elementos igual a 1000 y se varían la cantidad de procesos de 1 a 15 con incrementos de 1. Realizando 5 muestras por cada valor de elementos .

Resultado de la prueba:

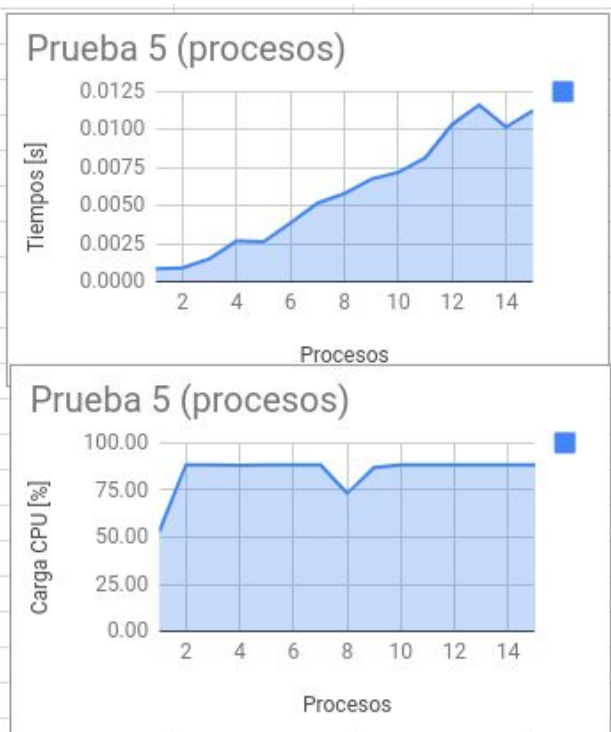


Prueba 5 (Merge Sort - Procesos):

Nombre	Prueba 5 Merge procesos
Parámetros	planificador_merge.py recibe como parametros el número fijo de elementos igual a 2000 y se varían la cantidad de procesos de 1 a 15 con incrementos de 1. Realizando 5 muestras por cada valor de elementos .

Resultado de la prueba:

Procesos	Tiempos	Uso CPU
1	0.0009	53.00
2	0.0009	88.43
3	0.0015	88.37
4	0.0027	88.09
5	0.0026	88.21
6	0.0039	88.43
7	0.0052	88.36
8	0.0058	73.37
9	0.0068	86.98
10	0.0072	88.36
11	0.0082	88.35
12	0.0104	88.35
13	0.0116	88.33
14	0.0102	88.28
15	0.0113	88.30

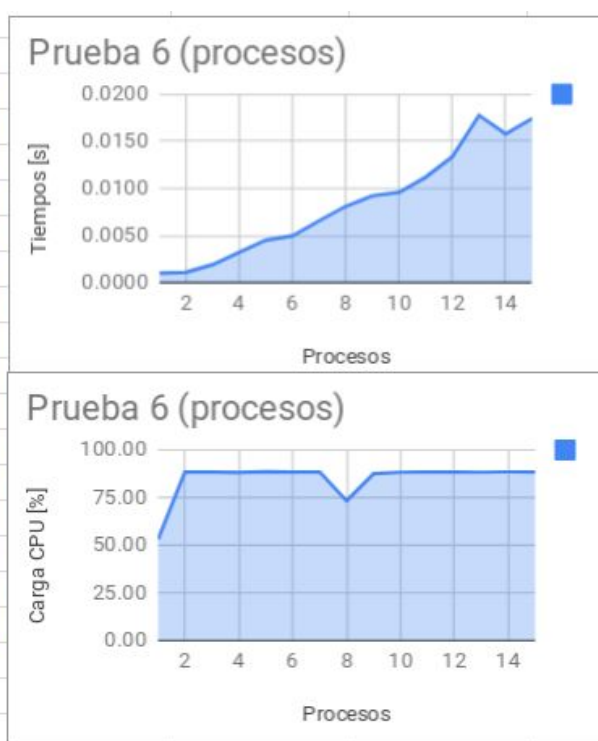


Prueba 6 (Merge Sort - Procesos):

Nombre	Prueba 6 Merge procesos
Parámetros	planificador_merge.py recibe como parametros el número fijo de elementos igual a 4000 y se varían la cantidad de procesos de 1 a 15 con incrementos de 1. Realizando 5 muestras por cada valor de elementos .

Resultado de la prueba:

Procesos	Tiempos	Uso CPU
1	0.0010	53.09
2	0.0011	88.37
3	0.0019	88.33
4	0.0032	88.17
5	0.0045	88.54
6	0.0050	88.43
7	0.0066	88.40
8	0.0081	73.33
9	0.0092	87.56
10	0.0096	88.29
11	0.0112	88.36
12	0.0134	88.33
13	0.0178	88.31
14	0.0158	88.48
15	0.0174	88.46



Pruebas Merge Sort - Threading

Entorno	Raspberry Pi V1.2 con ARMv7 rev 4 de 4 núcleos y 1GB de memoria RAM
Descripción	El script <code>tiempos_elemento.sh</code> realiza un bucle partiendo de un número fijo de elementos y modificando el número de procesos ejecutando varias instancias de <code>planificador_merge.py</code> (consumidor) que se encarga de lanzar los procesos <code>sensor_merge.py</code> (productores) y enlazarlos a un lista común para el análisis de tiempos de respuesta, los cuales se promedian dentro del mismo programa y se almacena la salida del mismo dentro del script. En total para cada valor de procesos (que se le pasa como parámetro a <code>planificador_merge.py</code> al igual que el número fijo de elementos) se realizan 5 muestras y se promedian dentro del script. De esta forma el script nos devuelve un tiempo de respuesta promedio para cada valor distinto de procesos almacenando en un archivo de texto.
Observaciones	Cada proceso productor lanza el programa <code>MergeSort.py</code> ejecutando solo Merge Sort lineal

Pruebas 1 - 2 - 3 (Merge Sort - Elementos)

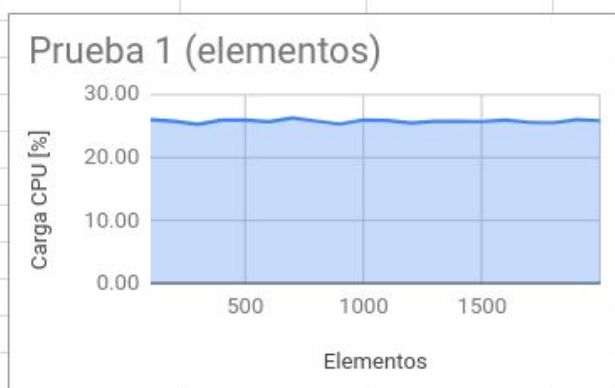
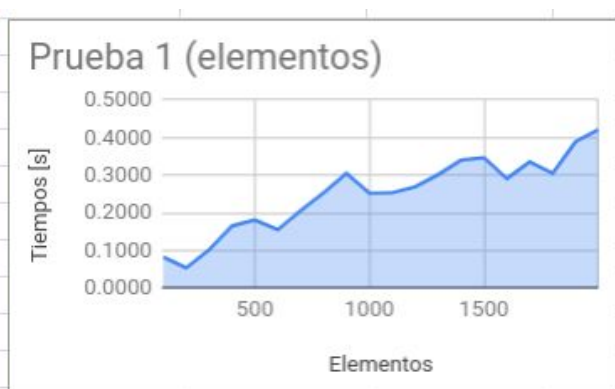
Ejecución	Se ejecuta <code>tiempos_elementos.sh</code>
-----------	---

Prueba 1

Nombre	Prueba 1 Merge elementos
Parámetros	planificador_merge.py recibe como parametros el número fijo de procesos igual a 15 y se varían la cantidad de elementos de 100 a 2000 con incrementos de 100. Realizando 5 muestras por cada valor de elementos .

Resultados de la prueba

Elementos	Tiempos [s]	Uso CPU [%]
100	0.0833	25.93
200	0.0539	25.68
300	0.1016	25.23
400	0.1651	25.88
500	0.1812	25.88
600	0.1554	25.62
700	0.2058	26.19
800	0.2534	25.69
900	0.3048	25.24
1000	0.2523	25.89
1100	0.2535	25.79
1200	0.2687	25.39
1300	0.3013	25.68
1400	0.3406	25.68
1500	0.3464	25.63
1600	0.2912	25.87
1700	0.3355	25.50
1800	0.3050	25.44
1900	0.3895	25.94
2000	0.4213	25.77



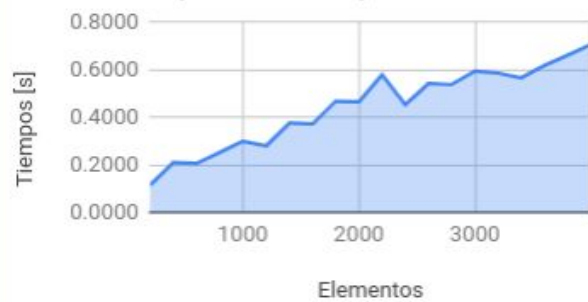
Prueba 2

Nombre	Prueba 2 Merge elementos
Parámetros	planificador_merge.py recibe como parametros el número fijo de procesos igual a 15 y se varían la cantidad de elementos de 200 a 4000 con incrementos de 200. Realizando 5 muestras por cada valor de elementos .

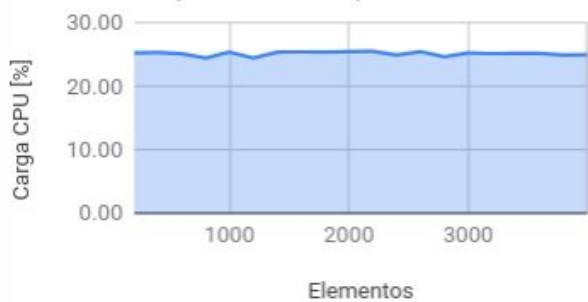
Resultados de la prueba

Elementos	Tiempos [s]	Uso CPU [%]
200	0.1174	25.35
400	0.2121	25.40
600	0.2081	25.20
800	0.2536	24.54
1000	0.3014	25.47
1200	0.2817	24.55
1400	0.3771	25.46
1600	0.3739	25.51
1800	0.4692	25.50
2000	0.4667	25.53
2200	0.5804	25.59
2400	0.4540	25.03
2600	0.5437	25.56
2800	0.5393	24.76
3000	0.5958	25.34
3200	0.5875	25.24
3400	0.5680	25.26
3600	0.6188	25.28
3800	0.6626	24.98
4000	0.7076	25.08

Prueba 2 (elementos)



Prueba 2 (elementos)



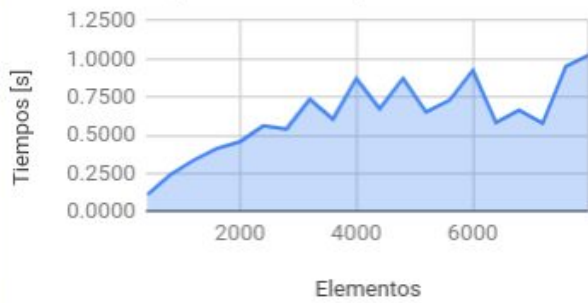
Prueba 3

Nombre	Prueba 3 Merge elementos
Parámetros	planificador_merge.py recibe como parametros el número fijo de procesos igual a 15 y se varían la cantidad de elementos de 400 a 8000 con incrementos de 400. Realizando 5 muestras por cada valor de elementos .

Resultados de la prueba

Elementos	Tiempos [s]	Uso CPU [%]
400	0.1099	25.67
800	0.2425	25.60
1200	0.3361	25.67
1600	0.4138	25.70
2000	0.4583	25.63
2400	0.5628	25.65
2800	0.5420	25.38
3200	0.7359	25.47
3600	0.6051	25.48
4000	0.8708	25.36
4400	0.6737	25.63
4800	0.8720	25.64
5200	0.6528	25.74
5600	0.7295	25.39
6000	0.9238	25.70
6400	0.5824	25.70
6800	0.6631	25.70
7200	0.5784	25.79
7600	0.9500	25.52
8000	1.0257	24.85

Prueba 3 (elementos)



Prueba 3 (elementos)



Pruebas 4 - 5 - 6 (Merge Sort - Hilos)

Ejecución	Se ejecuta tiempos_procesos.sh
-----------	---------------------------------------

Prueba 4

Nombre	Prueba 5 Merge hilos
Parámetros	planificador_merge.py recibe como parametros el número fijo de elementos igual a 1000 y se varían la cantidad de procesos de 1 a 15 con incrementos de 1. Realizando 5 muestras por cada valor de elementos .

Resultado de la prueba

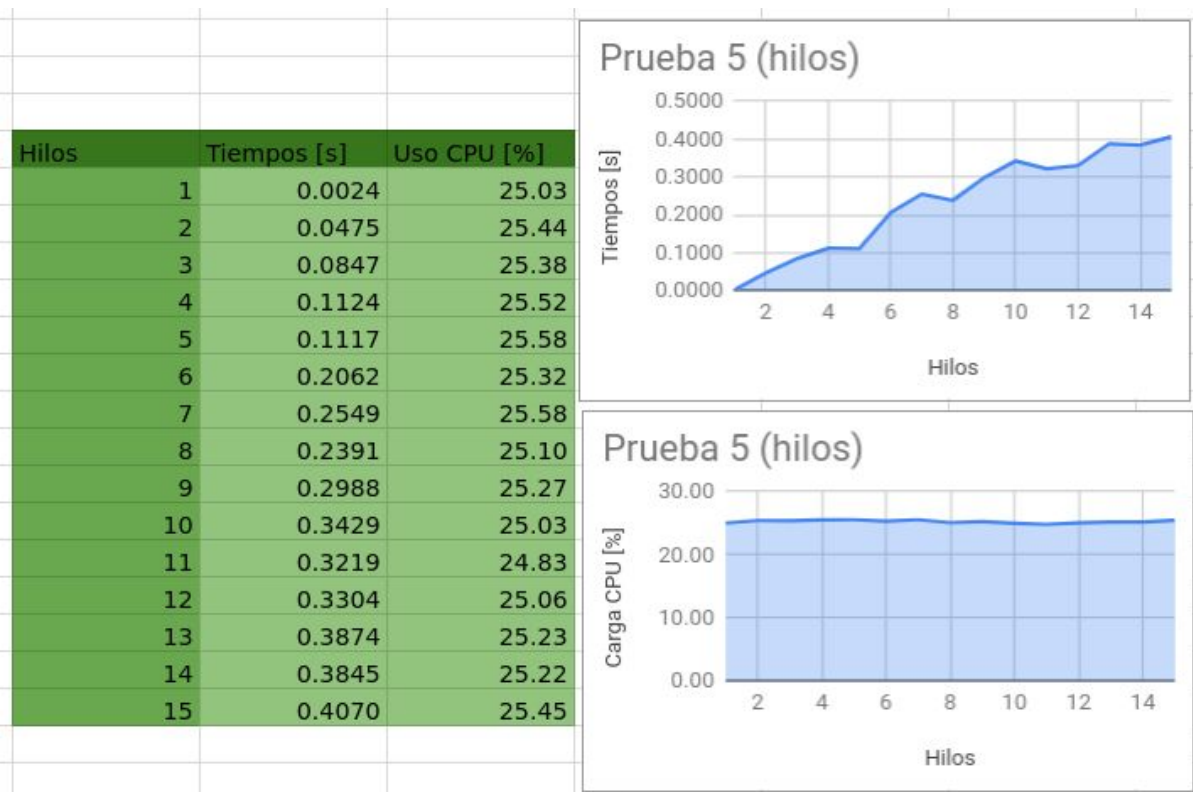
Hilos	Tiempos [s]	Uso CPU [%]
1	0.0026	25.20
2	0.0233	25.81
3	0.0385	25.84
4	0.0853	25.54
5	0.1033	25.62
6	0.1094	25.45
7	0.1516	25.83
8	0.1771	25.59
9	0.1833	25.67
10	0.2273	25.69
11	0.1934	24.94
12	0.2814	25.47
13	0.2244	25.08
14	0.2260	25.60
15	0.3031	25.86



Prueba 5

Nombre	Prueba 5 Merge hilos
Parámetros	planificador_merge.py recibe como parametros el número fijo de elementos igual a 2000 y se varían la cantidad de procesos de 1 a 15 con incrementos de 1. Realizando 5 muestras por cada valor de elementos .

Resultado de la prueba

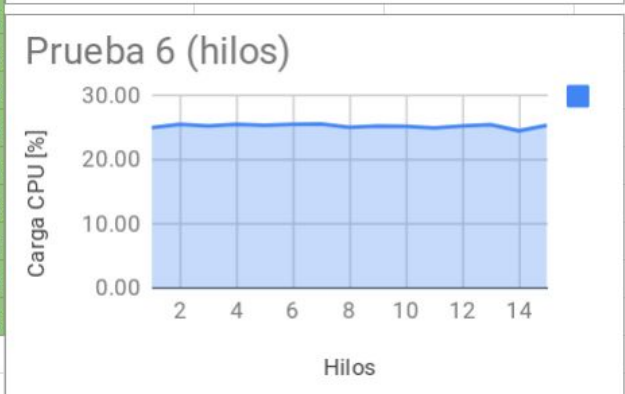


Prueba 6

Nombre	Prueba 6 Merge hilos
Parámetros	planificador_merge.py recibe como parametros el número fijo de elementos igual a 4000 y se varían la cantidad de procesos de 1 a 15 con incrementos de 1. Realizando 5 muestras por cada valor de elementos .

Resultado de la prueba

Hilos	Tiempos [s]	Uso CPU [%]
1	0.0028	25.10
2	0.0705	25.61
3	0.1326	25.35
4	0.1781	25.63
5	0.1551	25.47
6	0.2457	25.61
7	0.2483	25.69
8	0.3269	25.16
9	0.3196	25.34
10	0.4358	25.30
11	0.3980	25.03
12	0.4510	25.36
13	0.6291	25.55
14	0.6557	24.60
15	0.7732	25.49



Conclusión

Con la realización de esta práctica se adquirió práctica en la programación con Python y se utilizaron diferentes librerías destinadas a la ejecución paralela y concurrente, aprendiendo de sus ventajas, desventajas y rendimiento aplicado a una Raspberry Pi 3.

Se investigó sobre herramientas de profiling que permitían de manera gráfica observar la planificación de procesos e hilos del procesador de los diferentes programas que hacían uso de los módulos antes mencionados.

Se tomó conciencia de la importancia de adaptar el software al hardware en donde corren aplicaciones paralelas y concurrentes, pudiendo observar cómo se produce una mejora o degradación de tiempos de conclusión en la performance la Raspberry PI 3.

El análisis de rendimiento en la Raspberry Pi 3 utilizando la librería **multiprocessing** con operaciones matriciales se consiguió una sobrecarga promedio del 80% utilizando como máximo un total de 60 procesos en simultáneo dio un tiempo máximo de conclusión de 0.054 segundos en el peor caso.

Con un análisis similar utilizando la librería **multiprocessing**, con el algoritmo de ordenamiento Merge Sort se consiguió una sobrecarga promedio del 88% utilizando como máximo un total de 15 procesos en simultáneo, dando un tiempo máximo de conclusión de 0.034 segundos en el peor caso.

Utilizando ahora la librería **threading** el algoritmo de ordenamiento Merge Sort dió como resultado una sobrecarga promedio del 25% utilizando como máximo un total de 15 procesos en simultáneo dio un tiempo máximo de conclusión de 1 segundo en el peor caso.

Anexos

Trace-cmd

Esta herramienta no necesita ser instalada, pues ya viene en la imagen de cualquier distribución GNU/Linux, incluso para Raspbian que fue el sistema operativo utilizado para Raspberry PI.

La forma de uso es la siguiente:

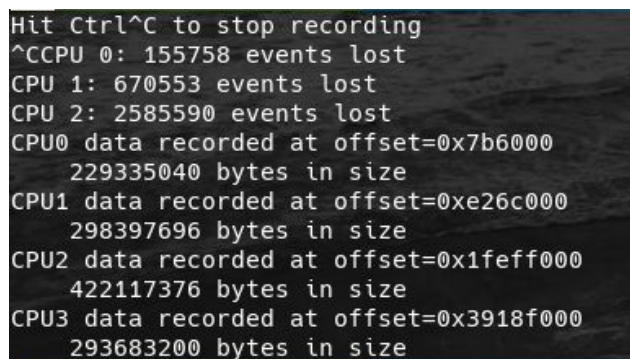
Primero debemos comenzar la grabación de los eventos mediante el siguiente comando: `#sudo trace-cmd record -o trace.dat -e all`

Donde `trace-cmd` es el programa, `record` es la opción que nos permite comenzar la grabación de los eventos, y en este caso mediante `"-e all"` se seleccionan todos los eventos de kernel y finalmente mediante `"-o"` se especifica el nombre del archivo de salida.

Una vez que comienza la grabación podemos empezar la ejecución de algún programa del que deseamos conocer características de su ejecución.

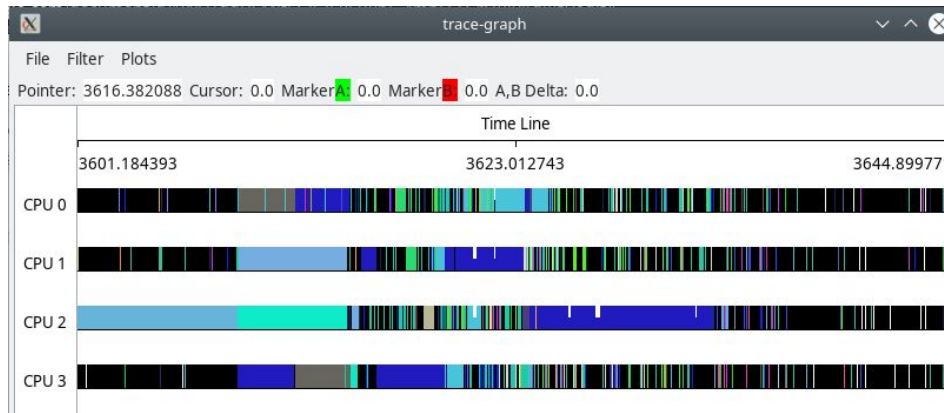
El programa nos dice que para detener la ejecución debemos presionar `ctrl+c` el cual es el equivalente a `"kill -9"` para la finalización de un proceso.

La salida muestra que luego de ejecutar `ctrl+c` un mensaje donde indica que se han grabado eventos para la cantidad de núcleos que posee el hardware.

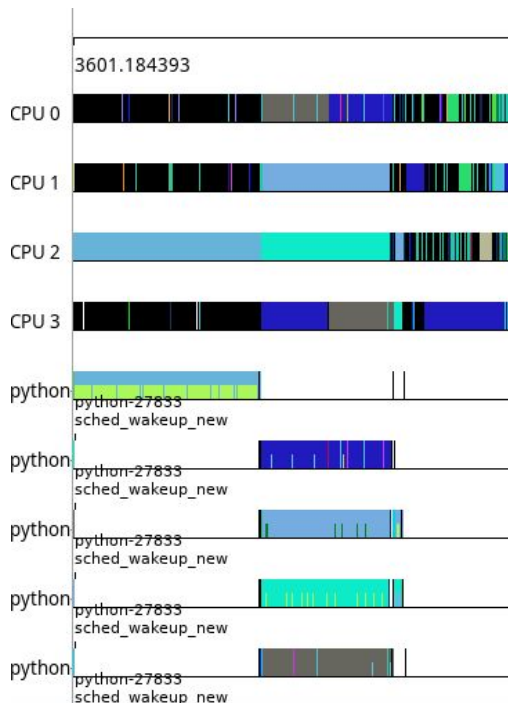


```
Hit Ctrl^C to stop recording
^CCPU 0: 155758 events lost
CPU 1: 670553 events lost
CPU 2: 2585590 events lost
CPU0 data recorded at offset=0x7b6000
      229335040 bytes in size
CPU1 data recorded at offset=0xe26c000
      298397696 bytes in size
CPU2 data recorded at offset=0x1feff000
      422117376 bytes in size
CPU3 data recorded at offset=0x3918f000
      293683200 bytes in size
```

Luego de esto, usamos el siguiente comando para visualizar lo guardado en el archivo `trace.dat` o cual sea el nombre se haya escogido: `trace-graph -i trace.dat`



Luego entrando en Plots>Tasks podemos seleccionar la tarea que deseamos filtrar, en nuestro caso buscaremos una de nombre “python” y aceptamos.



El proceso para filtrar es muy manual y tedioso y no se encontró otra manera de realizarlo con este programa. Se puede ver en esta última captura claramente una ejecución lineal y luego una ejecución en simultáneo con tantos procesos como núcleos posea el equipo.

Lttnng y Tracecompass

Primero se deben instalar los siguientes paquetes.

- lttnng-tools

- lttng-modules

Por ejemplo para instalarlo en la Raspberry se procedió a ejecutar los siguientes comandos.

```
#sudo apt-get install lttng-modules-dkms  
#sudo apt-get install liblttng-ust-dev  
#sudo apt-get install lttng-tools
```

Los siguientes comandos son usados para comenzar a grabar los eventos del kernel.

```
#sudo lttng-sessiond -b
```

Que inicializa el demonio de sesión

```
#lttng create
```

Crea un directorio donde se guardará lo que se graba. Al ejecutar el comando se visualiza un mensaje como el siguiente

```
Session auto-20181025-100949 created.  
Traces will be written in /home/miguel/lttng-traces/auto-20181025-100949
```

Donde indica la ruta:

```
#lttng enable-event -k -a
```

Que habilita los eventos del kernel, en este caso todos debido a la bandera -a

```
All Kernel events are enabled in channel channel0
```

```
#lttng start
```

Para comenzar a grabar los eventos, en este momento podemos correr un programa y cuando se quiera terminar ejecutamos:

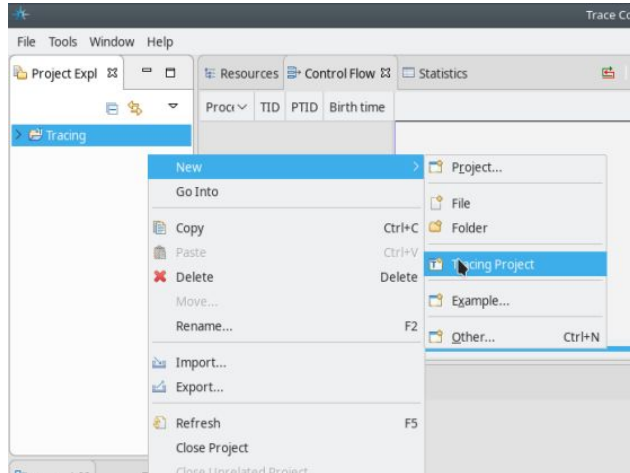
```
#lttng stop
```

Finalmente con

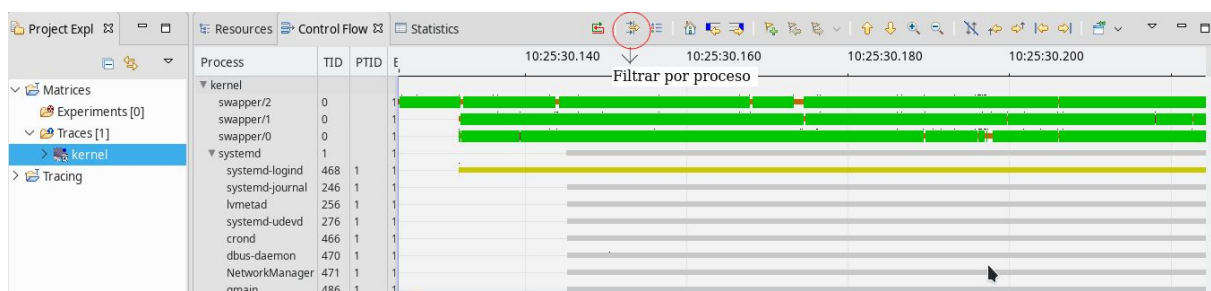
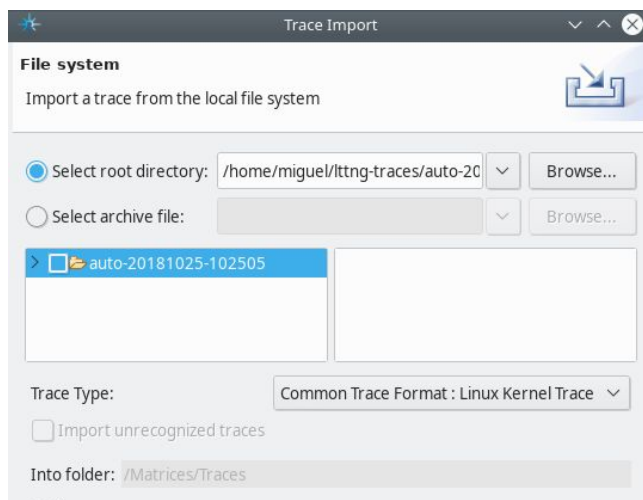
```
#lttng destroy
```


Se termina la sesión sin eliminar los datos grabados.

Para visualizar los datos ejecutamos trace compass y creamos un proyecto



Una vez creado el proyecto expandimos la carpeta e importamos en "traces" los datos grabados del lttng "select root directory" y colocando en trace type "Common Trace Format: Linux Kernel Trace"



Haciendo doble click en “kernel” se despliega un gráfico donde se visualiza la actividad, la cual se puede filtrar por proceso. En nuestro caso filtramos por “Python”



Modificando el zoom in y el zoom out se puede encontrar una imagen óptima del funcionamiento del programa. En este caso queda muy a simple vista como actúan los hilos y se ve claramente que no existe solapamiento.

Referencias

- [1] <https://gist.github.com/stephenmcd/39ded69946155930c347>
- [2] <https://MohamadSleiman@bitbucket.org/MiguelazoDS/pps.git>
- [3] <https://gist.github.com/philippstroehle/7864727>

Bibliografía

Procesos y Threads

Sistemas Operativos William Stallings 5ta Edición

http://mceron.cs.buap.mx/cap2_dis.pdf

Multithreading - Multiprocessing

<https://timber.io/blog/multiprocessing-vs-multithreading-in-python-what-you-need-to-know/>

<https://medium.com/@nbosco/multithreading-vs-multiprocessing-in-python-c7dc88b50b5b>

<https://www.ploggingdev.com/2017/01/multiprocessing-and-multithreading-in-python-3/>

https://sebastianraschka.com/Articles/2014_multiprocessing.html

Multiprocessing: Process vs Pool

<https://www.ellicium.com/python-multiprocessing-pool-process/>

GIL (Global Interpreter Locking)

<http://www.dabeaz.com/python/UnderstandingGIL.pdf>

Merge Sort

<https://gist.github.com/stephenmcd/39ded69946155930c347>

https://en.wikipedia.org/wiki/Merge_sort

Productor-Consumidor

<https://gist.github.com/cristipufu/2d8724a7b526ef57e73a4f1709fa5690>

<https://stackoverflow.com/questions/12474182/asynchronously-read-and-process-an-image-in-python>

OpenCV

https://docs.opencv.org/3.1.0/da/d22/tutorial_py_canny.html