Multi-hilado en Python

- Sistemas Paralelos UNLP -

ÍNDICE

Trabajo realizado por Jimena Villanueva

Introducción

Se busca dar una investigación sobre el soporte existente para multi-hilado en el lenguaje Python y cómo actúa el intérprete a la hora de hacer un programa en paralelo. Se explican los métodos básicos como los métodos para la creación y destrucción de los hilos, y los métodos necesarios para ejecutarlos. También se desarrollan los elementos que brinda el lenguaje para la sincronización de los hilos (semáforos, locks, cola de condiciones, etc) y comunicación entre ellos (manejo de la memoria local y global de los hilos).
Luego se comentan las ventajas y desventajas, y las limitaciones del modelo de multi hilado en Python para paralelismo y cómo es que Numba es una buena alternativa para este problema. Se mencionan y explican las primitivas principales y el funcionamiento de la librería.
Finalmente se implementan y se explica el desarrollo de dos algoritmos de multihilos en Python que computan una multiplicación de matrices, uno de los algoritmos usando el lenguaje puro y el otro utilizando Numba. Se concluye con la comparación de los speedup y eficiencia entre los distintos algoritmos y una breve conclusión general.

Implementación de la librería threading

Python soporta multi hilado a través de la librería ***Threading***. Este módulo construye interfaces de subprocesamiento de nivel superior sobre el módulo _thread de nivel inferior. Por otro lado, las versiones de Python utilizadas comúnmente son una implementación de **cpython** y por lo que utilizan **Global interpreter Lock (GIL).** Esto impide que varios threads estén trabajando simultáneamente, incluso en máquinas que poseen varios procesadores, por lo que nunca se podrá lograr un verdadero paralelismo.
Esto trae como desventaja la limitación de la condición de carrera, pero garantiza la seguridad del hilo.
El GIL se inventó para impedir que múltiples hilos decrementen la referencia de algún objeto mientras otros están haciendo uso de ella, es decir que previene que espacios en memoria sean liberados cuando aún están siendo utilizados. Además facilitó la implementación de Python e incrementó la velocidad de ejecución del mismo cuando se utiliza un thread. En caso que se quiera eliminar, se debe entender la manera en que las sentencias se vuelven atómicas con el uso de este y así proveer un mecanismo para que lo simule. Además se debe contemplar que varias funciones de librerías ya provistas hacen el uso del mismo, por lo que se deberían refactorizar. Podemos decir que eliminar el GIL traería más problemas que ventajas.

Objetos Thread y tipos

En Python un objeto Thread representa una determinada operación que se ejecuta como un subproceso independiente, es decir, representa a un hilo. Los subprocesos de Python son subprocesos reales del sistema de hilos POSIX (pthreads) o hilos de Windows. Toda la programación o cambio de hilo es totalmente administrado por el sistema operativo del host. En el detrás de escena de la creación de un hilo, python almacena lugar para una estructura de datos que contiene el estado del intérprete.
Hay tres tipos de hilos:

- *Hilos demonios*: el significado de esta marca es que la totalidad del programa Python finaliza cuando solo quedan subprocesos del demonio. El valor inicial es heredado del hilo creador o del argumento del constructor del demonio. Estos se cierran abruptamente y puede que sus recursos no se liberen correctamente y se corrompan.
- *Hilo dummy o extranjero*: son hilos de control iniciados fuera del módulo de subprocesos, como por ejemplo directamente de código C. Estos poseen una funcionalidad limitada, siempre se consideran vivos y demoníacos, y no se pueden esperar a través de la función join(). Además nunca son eliminados, ya que es imposible detectar la terminación de los mismos.
- *Hilo principal*: corresponde al hilo del control inicial del programa de Python, que por defecto no es un hilo demonio.

Creación de hilos en Python

Hay dos maneras a la hora de crear un hilo en Python: pasar un objeto invocable al constructor, es decir enviar una función al constructor la cual se ejecutará después de que se inicie el hilo, o sobreescribir el método run() en una subclase de Thread.
Como ya mencionamos anteriormente, cuando trabajamos con hilos en Python, lo haremos a través de la librería **threading.py.** Esta librería contiene la clase *Thread*, la cual nos permite crearlos. El constructor de dicha clase es el siguiente:

```
def __init__(group=None, target=None, name=None, args=(), kwargs={}, *, daemon=None):
```

Este constructor siempre debe ser llamado con argumentos de palabra clave. Los mismos son:

- *group*: debe ser None, y está reservado para futuras versiones.
- *target*: es la función o proceso que será invocado por la función run(). Por defecto es None.
- *name*: corresponde al nombre del hilo.
- *args:* consiste en una tupla de argumentos que se le puede pasar al thread. Por defecto es ().

- *kwargs*: es un diccionario de argumentos de palabras clave para invocar el constructor de la clase base. Por defecto es {}.
- *daemon*: establece explícitamente si el hilo es demoníaco o no. El valor por defecto es None, por lo que la propiedad demoníaca es heredada del hilo actual.

Si la subclase sobrescribe el constructor, debe asegurarse de invocar al constructor de la clase base (`Thread.__init__()`) antes de hacer cualquier otra cosa al hilo.

Podemos crear un objeto de la clase Thread de la siguiente forma:

```
thread = threading.Thread(target= function, args={})
```

Una vez que un objeto *thread* es creado, su actividad debe ser iniciada llamando al método ***start()*** del hilo. Con esta función se le asigna automáticamente un nombre y un identificador. Podemos obtener dichos valores con la función ***getName()*** para saber el nombre y la función ***ident()*** para conocer el id. A su vez, al iniciar el hilo se invoca el método ***run()*** en un hilo de control separado. Una vez que la actividad del hilo ha sido iniciada, el hilo se considera “vivo” y deja de estar vivo cuando su método run() termina. El método ***is_alive()*** verifica si el hilo está vivo.

Otros hilos pueden llamar al método ***join()*** de un hilo. Esto bloquea al hilo llamador hasta que el hilo cuyo método join() ha sido llamado termine. Esta función permite esperar a que todos los hilos finalicen y poder continuar ejecutando las sentencias restantes del programa. Esta función puede ser aplicada muchas veces sobre un mismo hilo.

Más allá de las funciones claves que nombramos para la ejecución de nuestro ejercicio, existen muchas más que a continuación nombraremos y explicaremos su uso, tanto de la clase Thread como de la librería Threading.

Métodos más importantes de la librería *Threading*

- ***threading.active_count()***: devuelve el número de Threads activos.
- ***threading.current_thread()***: devuelve el Thread actual, correspondiente al subproceso de control del que llama.
- ***threading.excepthook()***: maneja las excepciones no captadas lanzadas por el método run().
- ***threading.get_native_id()***: devuelve el ID de subproceso integral nativo del subproceso actual asignado por el kernel. Este es un número entero no negativo. Su

valor se puede utilizar para identificar de forma única este hilo en particular en todo el sistema .

- ***threading.enumerate()*:** devuelve una lista de todos los objetos Thread actualmente activos. La lista excluye los subprocesos terminados y los subprocesos que aún no se han iniciado.

- ***threading.main_thread()*:** devuelve el objeto Thread principal. Por lo general el hilo principal es el hilo desde el que se inició el intérprete de Python.

- ***threading.stack_size([size])*:** devuelve el tamaño de la pila de subprocesos utilizado al crear nuevos subprocesos. El argumento opcional *size* especifica el tamaño de la pila que se utilizará para los subprocesos creados posteriormente.

- ***threading.TIMEOUT_MAX*:** es el valor máximo permitido para el parámetro de tiempo de espera de las funciones de bloqueo. Especificar un tiempo de espera mayor que este valor generará un OverflowError.

- ***threading.local():*** administra datos locales de los subprocesos. Esta función crea una instancia local (o subclase) y almacena los atributos en ella. Los valores de instancia serán diferentes para hilos distintos.

Funciones de la clase Thread:

- ***getName()* y *setName()*:** obtienen y modifican el nombre que se le asigna a cada thread.
- ***is_alive()*:** es un método que corresponde a una función booleana, la cual responde true justo antes que se inicie el método run, hasta exactamente después de que el run finalice.
- ***isDaemon()* y *setDaemon()*:** la primera función devuelve verdadero si es un hilo demonio. La segunda función es para setear este argumento.
- ***ident()*:** devuelve el identificador del hilo o None en caso de que el hilo no se haya iniciado. Es un número entero distinto a 0. Los identificadores pueden ser reciclados cuando finaliza un hilo y otro es creado.

Métodos de sincronización

Existen ciertas **primitivas de sincronización** para asegurar que dos o más hilos no ejecuten simultáneamente una sección crítica, brindadas por el módulo de *threading*. Estas ayudan a evitar condiciones de carrera y aseguran una ejecución ordenada de las secciones críticas del código.
A continuación se explican algunas de ellas:

Locks

Esta primitiva permite a los hilos adquirir y liberar de forma segura un lock para garantizar que solo un hilo pueda acceder a un recurso compartido a la vez. En Python es el método de sincronización de más bajo nivel disponible. Se implementa directamente por el módulo *_thread*. Posee dos estados posibles: abierto (“locked”) o cerrado (“unlocked”), y se crea por defecto abierto. Para modificar estos estados posee dos métodos básicos *acquire()* (adquirir) y *release()* (liberar). Cuando el estado del lock está desbloqueado, la función *acquire()* cambia el estado a bloqueado y lo devuelve inmediatamente. Mientras que la función *release()* cambia del estado bloqueado a desbloqueado y lo devuelve inmediatamente. Este método solo debe ser llamado cuando el lock está bloqueado, sino se lanzará un error en ejecución.Es decir que cuando el estado es bloqueado, *acquire()* bloquea hasta que una llamada a *release()* en otro hilo lo cambie a abierto, luego la llamada a *acquire()* lo restablece a cerrado y retorna.
Cuando más de un hilo está bloqueado en *acquire()* esperando que el estado sea desbloqueado, sólo un hilo procederá cuando una llamada a *release()* restablezca el estado a desbloqueado. No está definido cuál de los hilos en espera procederá, y puede variar a través de las implementaciones. Se debe aclarar que todos los métodos se ejecutan de manera atómica.
La clase que implementa los objetos de esta primitiva es class *threading.lock*.

Funciones:

acquire(blocking= True)

Con esta función se adquiere el lock. Cuando se invoca con el argumento *blocking* establecido en True, el hilo se bloqueará hasta que adquiera el bloqueo. Si se establece en False, el hilo intentará adquirir el bloqueo, pero si no es posible, continuará su ejecución sin esperar. La función devuelve True si el bloqueo se adquirió con éxito.

release()

Libera un lock y puede ser llamada desde cualquier hilo, no solo del que ha adquirido el bloqueo. Una vez que se libera el lock, otros hilos pueden adquirirlo y acceder a la sección crítica. Si hay otros subprocesos bloqueados esperando a que se desbloquee, permite que continúe solo uno de ellos.
Si se invoca sobre un lock desbloqueado falla y lanza un error en tiempo de ejecución.

locked()

Devuelve verdadero si el lock está adquirido actualmente por algún hilo, y falso si está disponible para su adquisición.

Rlock

Es una variante de Lock, que permite a un único hilo adquirir el bloqueo múltiples veces. Además del estado de bloqueado y desbloqueado utilizados por las primitivas locks, utiliza el concepto de un hilo dueño y nivel de recursividad. Realiza un seguimiento de cuantas veces fue adquirido por un hilo en particular y debe ser liberado la misma cantidad de veces antes de que otros hilos puedan adquirirlo.
La clase que implementa los objetos de esta primitiva es class *threading.RLock*.

Funciones:

acquire(blocking= True)
: Con esta función se adquiere el lock. Si ya está adquirido por otro hilo, el hilo actual se bloqueará hasta que el lock esté disponible, a menos que se establezca blocking en false.

release()
: Se utiliza para liberar el bloqueo. En este caso, si el lock fue adquirido múltiples veces por el mismo hilo, se debe llamar a esta función la misma cantidad de veces para liberar completamente el lock. Si se llama más veces de las que se adquirió, lanzará un error en tiempo de ejecución.

locked()
: Devuelve verdadero si el lock está adquirido actualmente por algún hilo, y falso si está disponible para su adquisición.

Objeto condición

Esta clase implementa objetos de variable de condición. Una variable de condición permite que uno o más subprocesos esperen hasta que se cumpla una determinada condicion antes de continuar su ejecución.
La clase que implementa los objetos de esta primitiva es class *threading.Condition(lock=None)*.
Implementa las funciones de acquire() y release() ya que necesita adquirir y liberar el lock asociado a la condición.

Funciones:

wait(timeout=None)
: Esta función hace que el hilo actual espere hasta que otro hilo notifique en la misma condición. El hilo se bloquea y libera el lock asociado a la condición. Puede indicarse un tiempo de espera opcional (*timeout*) para limitar el tiempo que espera antes de continuar con su ejecución.

wait_for(predicate, timeout=None)
: Se espera hasta que una condición se evalúe como verdadera. El predicado debe ser un invocable cuyo resultado se interpretará como un valor booleano.
Este método puede llamar al método wait() repetidamente hasta que se satisfaga el predicado o hasta que se agote el tiempo de espera.

notify(n=1)

Esta función notifica a un hilo en espera que la condición cambio. Despierta al menos un hilo en espera, y si se proporciona el argumento opcional *n*, se despiertan hasta n hilos.

notify_all()

Funciona igual que el notify pero despierta a todos los hilos del objeto condición y los pone en estado listos para ejecutar.

Objeto Semáforo

Un semáforo administra un contador atómico que representa el número de llamadas a release() menos el número de llamadas acquires(), más un valor inicial.
Este objeto es útil para controlar el acceso a recursos compartidos en escenarios donde solo se permite un número limitado de hilos a la vez.
La clase que implementa los objetos de esta primitiva es class *threading.Semaphore(value=1).*
El argumento *value* especifica el valor inicial del semáforo, que determina la cantidad inicial de recursos disponibles.

Funciones:

acquire(blocking=True)

Se adquiere un recurso del semáforo. Si el recurso está disponible, se adquiere inmediatamente y el hilo continua con su ejecución. Si no está disponible, el hilo se bloquea hasta que uno esté disponible. El argumento *blocking* controla si el hilo se bloquea o no. Si está en True, el hilo se bloqueara hasta que se adquiera un recurso. Si está en False, el hilo intentará adquirir el recurso, pero si no está disponible continuará su ejecución sin esperar.

release(n=1)

Libera un recurso del semáforo, incrementando el contador interno en n. Si hay hilos bloqueados esperando recursos, uno de ellos se desbloqueara y adquirirá el recurso liberado. Si no hay hilos bloqueados, el recurso se incrementa en uno.

Objeto evento

Un objeto evento permite a los hilos esperar hasta que se establezca un estado determinado (bandera). La bandera inicialmente es falsa. Los hilos pueden esperar hasta que el evento se active, y una vez activado, todos los hilos que están esperando pueden continuar su ejecución. El evento se puede volver a restablecer para que los hilos vuelvan a esperar su activación.
La clase que implementa los objetos de esta primitiva es class *threading.Event.*

Funciones:

set()

Establece el estado del evento en activado, haciendo que todos los hilos que están esperando continúen su ejecución.

clear()

Restablece el estado del evento en desactivado, haciendo que los hilos que esperen se bloqueen hasta que se active nuevamente.

wait(timeout=None)

Hace que el hilo actual espere hasta que el evento esté activado. Si ya está activado, continua su ejecución inmediatamente. Se puede especificar un tiempo de espera *timeout* opcional para limitar el tiempo que espera el hilo antes de continuar.

Objeto temporizador

Un temporizador permite programar una función para que se ejecute después de un cierto tiempo. Se crea un temporizador con una duración específica, y una vez transcurrido el tiempo, se ejecuta la función asociada. Se puede iniciar y cancelar el temporizador.
La clase que implementa los objetos de esta primitiva es class *threading.Timer.*

start()

Inicia el temporizador y comienza a contar el tiempo.

cancel()

Cancela el temporizador, deteniendo la ejecución futura de la función asociada. Si el temporizador ya fue iniciado, no tendrá efecto en la ejecución actual

Objeto barrera

Una barrera permite a un grupo de hilos esperar hasta que todos hayan alcanzado un punto de sincronización antes de continuar su ejecución.
Esta clase proporciona una primitiva de sincronización simple para ser utilizada por un número fijo de subprocesos que necesitan esperar el uno al otro. Cada uno de los subprocesos intenta pasar la barrera llamando al método wait() y se bloqueará hasta que todos los subprocesos hayan realizado sus llamadas a wait(). En este punto, los hilos se liberan simultáneamente.
La barrera se puede reutilizar tantas veces como desee para el mismo número de subprocesos.
La clase que implementa los objetos de esta primitiva es class *threading.Barrier(parties, action=None, timeout=None).*
Con el argumento *parties* se indica el número de hilos que deben alcanzar la barrera antes de que se liberen todos. El argumento opcional *action* es una función que se ejecutará una vez que se libere la barrera.

Funciones:

wait (timeout = None)

Esta función hace que el hilo actual espere en la barrera hasta que todos los hilos hayan llegado. Cuando todos los hilos que forman parte de la barrera han llamado a esta función y se liberan simultáneamente para continuar con su ejecución.
Si se especifica un tiempo de espera *timeout* (opcional), el hilo esperará hasta ese tiempo antes de continuar.

reset()

Restablece la barrera a su estado inicial. Todos los hilos que estaban esperando en ella se liberan y se puede usar nuevamente.

Numba como alternativa de solución

Como solución al problema de falta de soporte para la ejecución paralela por el bloqueo del GIL, podemos utilizar la librería Numba. Esta es una librería de Python que se utiliza para acelerar el rendimiento de código al compilarlo en tiempo de ejecución. Ofrece capacidades de paralelismo mediante la utilización de threads y aceleración de hardware como GPU. Tiene la capacidad de generar código altamente optimizado y ejecutarlo eficientemente. A diferencia de Python, Numba utiliza el compilador Just-In-Time (JIT) para traducir el código Python en código de máquina de bajo nivel. Esto permite obtener un rendimiento cercano al de lenguajes compilados como C.
Numba ofrece la función ***@jit*** para decorar funciones y métodos en el código Python. Esto le indica a Numba que compile y optimice el código para aprovechar el paralelismo. Para asegurarse de que Numba genere código completamente compilado, se puede especificar '***nopython=true'*** y así aprovechar al máximo el paralelismo.
Otra característica de Numba útil para la ejecución en paralelo es ***'parallel=true'*** esto indica que se realice la paralelización automática en los bucles. El parámetro 'parallel' se utiliza en conjunto con la función ***'prange'*** que es la versión paralela del bucle range de Python. Esta función divide automáticamente el bucle en iteraciones independientes y las distribuye entre múltiples hilos para ejecutarlos simultáneamente. Por lo tanto, si utiliza prange junto a parallel=true le estamos indicando a Numba que realice la paralelización automática en los bucles, siempre y cuando sea posible. Numba analiza el bucle,y si detecta que las iteraciones son independientes entre sí, distribuye automáticamente las iteraciones entre los hilos. No todos los bucles pueden beneficiarse del paralelismo automático. Numba evalúa el bucle y decide si es posible y beneficioso paralelizarlo. Es importante evaluar si la paralelización es apropiada y si realmente se obtiene un beneficio significativo en términos de rendimiento.
En Numba la cantidad de hilos utilizada para la paralelización se determina automáticamente en función del número de núcleos de CPU disponibles en el sistema. Ajusta dinámicamente la cantidad de hilos utilizados para aprovechar al máximo los recursos disponibles del hardware.

Ejercicio práctico y comparación con Numba.

Como ejercicio práctico, se implementaron dos algoritmos multi hilos en Python que computan una multiplicación de matrices, uno utilizando el lenguaje puro y otro usando la alternativa Numba.

Desarrollo del algoritmo con Python:

Para el desarrollo con Python importamos la librería *threading.* Esta librería nos permite crear y utilizar hilos.
Comenzamos con la declaración de las matrices como arreglos de tamaño de N*N, las dos que se multiplicaran y una tercera para guardar el resultado.
Luego definimos la función que ejecutaran los procesos, *mult()*, donde se realiza la multiplicación de las matrices. Para llevar a cabo la multiplicación, se define un rango (de principio a fin) que procesara cada hilo, es decir que se divide la matriz en pequeñas partes para que cada proceso recorra simultáneamente junto a los demás. Luego, dentro de este rango, se suman los valores de la multiplicación de las celdas mientras se recorren todas las filas y columnas. Definimos variables auxiliares para almacenar el valor de las filas y columnas para optimizar la solución.
Una vez definida la función, creamos los threads y le enviamos como parámetros la función que ejecutaran (*mult*) y los valores de principio y fin para definir el rango de cada hilo. La creación la llevamos a cabo dentro de una iteración con la cantidad de hilos que queremos y los vamos agregando en un arreglo.
Cuando todos los threads fueron creados y agregados al arreglo, podemos iniciarlos recorriendo dicho arreglo e invocando a la función *start()* por cada hilo.
Finalmente, en una nueva iteración esperamos a que los hilos terminen llamando a la función *join()*.

Desarrollo del algoritmo con Numba:

Para el desarrollo de este algoritmo, no fue necesario modificar mucho código ya que Numba nos brinda el decorador @jit como mencionamos previamente. Sin embargo, detallaremos los cambios a continuación.
Comenzamos con la importación de Numba y Numpy.
Luego seteamos la variable de entorno *'NUMBA_THREADING_LAYER'* con el valor 'threadpool' para asegurarnos que Numba utilice un modelo de subprocesamiento basado en hilos para aprovechar los múltiples núcleos del procesador.
Continuamos seteando la cantidad de hilos que utilizara Numba para ejecutar paralelamente y declarando las matrices como vectores nuevamente. Para esto último, utilizamos la librería Numpy ya importada previamente, e inicializamos las dos matrices que se multiplicaran con valores random entre 0 y 1, y la matriz de resultados con valores 0.
Para la definición de la función *mult()* en este caso no es necesario definir un rango de valores de inicio a fin que recorrera cada hilo, ya que al utilizar *prange* de Numba, divide las iteraciones del bucle automáticamente entre los hilos disponibles. Además decoramos la función con el decorator *@jit* indicando los parámetros *nopython* y *parallel* en *True.* Por lo

tanto, al utilizar *prange* junto a *@jit(parallel=True)*, Numba realiza automáticamente la paralelización del bucle en función de los recursos disponibles en el sistema.

Ambos fueron compilados y ejecutados tanto en una máquina local como en una máquina virtual remota con el fin de comparar los tiempos de ejecución y generar una mejor conclusión.

Características de la máquina local:

AMD Ryzen™ 5 2500U

General Specifications

Plataforma: Computadora portátil
Familia de productos: AMD Ryzen™ Processors
Línea de productos: AMD Ryzen™ 5 Mobile Processors with Radeon™ Vega Graphics
N.º de núcleos de CPU: 4
N.º de subprocesos: 8
Reloj de aumento máx.: Hasta 3.6GHz
Reloj base: 2.0GHz
Caché L1 total: 384KB
Caché L2 total: 2MB
Caché L3 total: 4MB
TDP/TDP predeterminado: 15W
AMD Configurable TDP (cTDP): 12-25W
Processor Technology for CPU Cores: 14nm
Desbloqueados: No
Paquete: FP5
Temp. máx.: 95°C
Fecha de lanzamiento: 10/26/2017
***Compatible con SO:**
Windows 10 edición de 64-bits
RHEL x86 edición de 64-bits
Ubuntu x86 edición de 64-bits
*El soporte del sistema operativo (SO) variará según el fabricante.

Características de la VM (instancia de máquina virtual creada con Google Cloud):

Configuración de la máquina

Tipo de máquina	e2-highcpu-8
Plataforma de CPU	Unknown CPU Platform
Arquitectura	x86-64
CPU virtuales para proporción de núcleos	—
Núcleos visibles personalizados	—
Dispositivo de visualización	Inhabilitado Habilita esta opción para usar las herramientas de grabación y captura de pantalla
GPU	Ninguna

Métricas del algoritmo con *Python puro*:

Tiempo de ejecución del *algoritmo secuencial*

N =	256	512	1024
Local	3.6869386695	32.297171906	346.1014082431793
VM	4.3006650465	34.3844650811	285.1076575015

Tiempo de ejecución

Tiempo	N=	256	512	1024
máquina local	2	4.3814277647	37.97312382470	311.26555759430
	4	4.592253767	38.9246153614	332.6655012318
	8	4.62046076430	39.3635556059	295.72850956600
máquina virtual	2	2.43385174540	20.2909904373	167.18599465230
	4	2.42877527040	20.52892757180	170.8364684001
	8	2.44267835410	20.65427318140	169.053598235

SpeedUp

Speed Up	N =	256	512	1024
máquina local	2	0,802860394	0,082973644	1,04038864
	4	0,797959091	0,082048411	1,170334943
	8	0,841492515	0,008505271	1,111916818
máquina virtual	2	1,767020138	1,694568098	1,705332185
	4	1,770713453	1,674927488	1,668892246
	8	1,760635017	1,664762772	1,686492689

Eficiencia

Eficiencia	N =	256	512	1024
máquina local	2	0.401430197	0.041486822	0.52019432
	4	0.1994897728	0.02051210275	0.2925837358
	8	0.1051865644	0.001063158875	0.1389896023
máquina virtual	2	0.883510069	0.847284049	0.8526660925
	4	0.4426783633	0.418731872	0.4172230615
	8	0.2200793771	0.2080953465	0.2108115861

Métricas del algoritmo con *Numba*:

Tiempo de ejecución del *algoritmo secuencial*

N =	256	512	1024	2048	4098
Local	1.06204262860	1.2177070157	3.85459229310	17.0356421883	90.1883492689
VM	1.1292059950	1.1684684621	2.73676814960	15.4234519691	114.4592757383

Tiempo de ejecución

Tiempo	N =	256	512	1024	2048	4096
máquina local	**2**	1.06591557560	1.1658612531	2.9459061558	9.270289753	59.6341821113
	4	1.10242721880	1.1375412429	2.26645781210	4.6515792837	38.4626626270
	8	1.08736048620	1.1216634407	2.2995290775	5.8441946665	37.0054960043
máquina virtual	**2**	1.1171429004	1.064290978	1.90366732370	8.31502875390	57.4215338522
	4	1.102204826	1.0254827754	1.51609733130	4.72951315590	29.3122076877
	8	1.0925462955	1.0037076697	1.35615427380	3.3195439087	18.1488098538

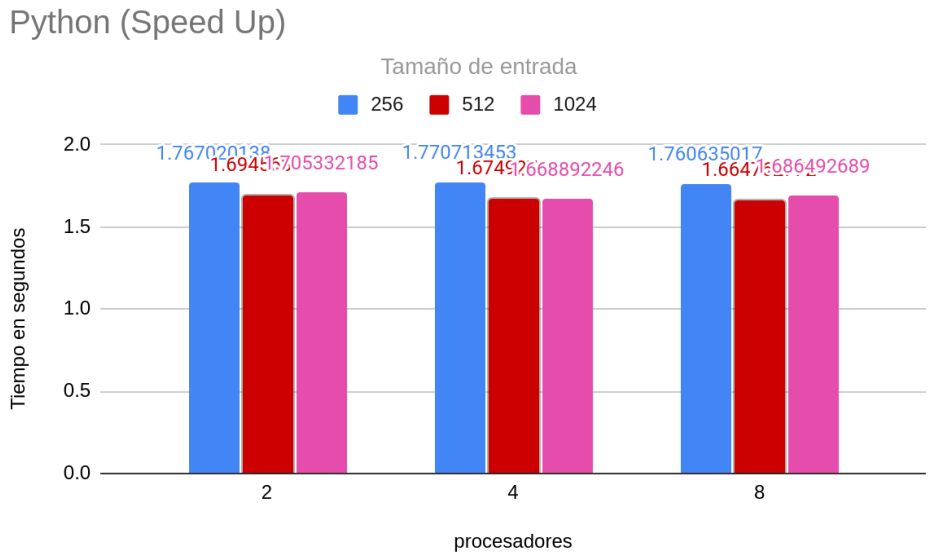
Speedup

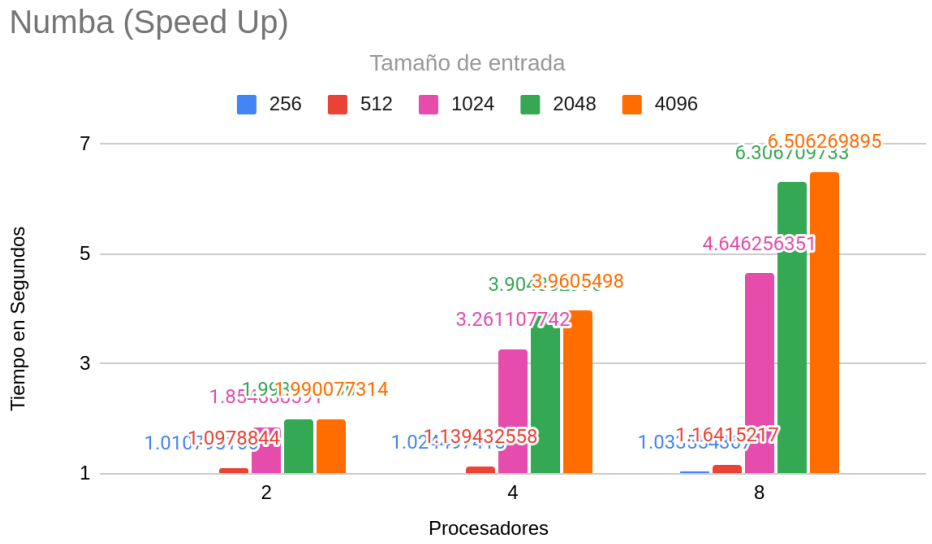
Speed Up	N =	256	512	1024	2048	4096
máquina local	**2**	3,458940608	1.1660118003	1,308457259	1,837660164	1,51235996
	4	3,344382837	1.0544090052	1,70071213	3,662335123	2,344828546
	8	3,390723423	1.11611986480	1,676252904	2,914968299	2,437160936
máquina virtual	**2**	1,010798166	1,0978844	1,854888591	1,993316236	1,990077314
	4	1,024497415	1,139432558	3,261107742	3,904832995	3,9605498
	8	1,033554367	1,16415217	4,646256351	6,306709733	6,506269895

Eficiencia

Eficiencia	N =	256	512	1024	2048	4096
máquina local	**2**	1.729470304	0.5830059002	0.6542286295	0.918830082	0.75617998
	4	0.8360957093	0.2636022513	0.4251780325	0.9155837808	0.5862071365
	8	0.4238404279	0.1395149831	0.209531613	0.3643710374	0.304645117

máquina virtual	**2**	0.505399083	0.5489422	0.9274442955	0.996658118	0.995038657
	4	0.2561243538	0.2848581395	0.8152769355	0.9762082488	0.99013745
	8	0.1291942959	0.1455190213	0.5807820439	0.7883387166	0.8132837369





Conclusión

Viendo y comparando las métricas de los distintos algoritmos podemos decir y asegurar que el algoritmo desarrollado con Numba es el que presenta mejores resultados, ya que hace un adecuado uso de los procesadores y cumple con el paralelismo en sí mismo.
Vale aclarar que, en ambos algoritmos obtuvimos mejores resultados corriéndolos sobre la VM, por lo que tomaremos estos tiempos como referencia.
En cuanto al algoritmo paralelo con Python puro podemos notar que no se explota al máximo la concurrencia ya que el lenguaje en sí presenta limitaciones cuando ejecuta algoritmos paralelamente. Este bloqueo del intérprete hace que, aunque el lenguaje admite hilos, no se aprovechen eficientemente. Esto se ve reflejado en los tiempos obtenidos, donde vemos que el tiempo no varía a medida que aumentamos la cantidad de procesos. Como consecuencia de esto, obtendremos valores similares en el cálculo de los speed ups y en la eficiencia, destacándose la obtenida con dos procesos sobre las demás.
En cambio con Numba, no tenemos esta similitud de los valores a medida que aumentamos la cantidad de procesos. Por lo que en este caso, con más procesos, menos tiempo tardará el algoritmo en ejecutarse. Así obtenemos speed ups más cercanos al óptimo en cada caso y a su vez, eficiencias cercanas a 1. Esto es así cuando tenemos un tamaño de entrada grande, ya que en los casos donde N es relativamente chico para el problema no se alcanzan buenos resultados (casos donde N es 256 y 512). Esto es debido a que prange tendrá un rendimiento óptimo cuando tenga una cantidad suficiente de trabajo independiente en cada iteración para justificar el costo de la paralelización. Por lo que a medida que crece N, obtendremos mejores resultados.

Bibliografía

https://www.geeksforgeeks.org/multithreading-python-set-1/
https://docs.python.org/es/3/library/threading.html
https://wiki.python.org/moin/GlobalInterpreterLock
https://numba.pydata.org/numba-doc/latest/user/parallel.html