UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

TÍTULO: Un Estudio Comparativo entre Traductores de Python para Aplicaciones Paralelas de Memoria Compartida

AUTORES: Andrés Milla

DIRECTOR: Enzo Rucci

CODIRECTOR:

ASESOR PROFESIONAL:

CARRERA: Licenciatura en Informática

Resumen

En la actualidad, Python es uno de los lenguajes más utilizados en diversas áreas de aplicación. Sin embargo, presenta limitaciones a la hora de poder optimizar y paralelizar aplicaciones debido a las limitaciones de su intérprete oficial, especialmente para aplicaciones CPU-bound. Para solucionar esta problemática han surgido traductores alternativos, aunque cada uno con un enfoque diferente y con su propia relación de costo-rendimiento. Ante la ausencia de estudios comparativos, se realizó una evaluación del rendimiento y esfuerzo de programación de dichos traductores, utilizando como caso de estudio N-Body, un problema popular y con alta demanda computacional.

Palabras Clave

Python, Numba, Cython, PyPy, N-body, HPC, CPU-bound, Multi-hilado, Rendimiento, Esfuerzo de programación, Optimización

Conclusiones

A partir de los resultados obtenidos, se concluye que PyPy y CPython produjeron un bajo rendimiento debido a sus limitaciones a la hora de paralelizar algoritmos; mientras que en sentido opuesto, Numba y Cython presentaron rendimientos notablemente superiores a los últimos y cercanos a los de una implementación de C+OpenMP, demostrando ser opciones viables para acelerar algoritmos numéricos. En cuanto al esfuerzo de programación, Cython requirió conocimiento adicional de C+OpenMP junto con un proceso más laborioso de ejecución. Por su parte, Numba tomó un mayor número de líneas de código aunque resultó más simple gracias a su enfoque de decoradores y a la posibilidad de reutilizar el código CPython de base.

Trabajos Realizados

- Implementaciones escritas y optimizadas en diferentes traductores de Python (CPython, PyPy, Numba, Cython) que computen N-Body sobre arquitecturas multicore.
- Un estudio comparativo de las soluciones para N-Body en arquitecturas multicore considerando rendimiento y esfuerzo de programación.

Trabajos Futuros

- Explorar otras capacidades y limitaciones de los traductores de Python no contempladas en este trabajo, como la utilización de GPUs.
- Replicar el estudio realizado considerando: (1) otros casos de estudio que sean computacionalmente intensivos pero cuyas características sean diferentes a las de N-Body; (2) otras arquitecturas multicore distintas a la usada en este trabajo. Ambas extensiones contribuirían a robustecer los resultados encontrados.
- Dado que existen otras tecnologías que permitan implementar paralelismo a nivel de procesos en Python, realizar una comparación entre ellas considerando no sólo el rendimiento sino también el costo de programación.

Fecha de la presentación: DICIEMBRE 2021

Universidad Nacional de La Plata

Facultad de Informática

Tesina de Grado

Un Estudio Comparativo entre Traductores de Python para Aplicaciones Paralelas de Memoria Compartida

Autor: Andrés Milla

Director: Dr. Enzo Rucci

Diciembre de 2021

Agradecimientos

A mi mamá y mi papá, por su apoyo incondicional y el esfuerzo que realizaron en todos estos años.

A mis amigos, por su colaboración y por el ánimo que me dieron todos los días.

A Enzo, por su excepcional trabajo como director y por acompañarme en cada paso dado.

A la Facultad de Informática, por brindarme grandes enseñanzas durante estos 4 años.

Resumen

En la actualidad, Python es uno de los lenguajes más utilizados en diversas áreas de aplicación, especialmente por su alto grado de legibilidad y su bajo esfuerzo de programación. Sin embargo, Python presenta limitaciones a la hora de poder optimizar y paralelizar aplicaciones debido a las limitaciones de su intérprete oficial (CPython), especialmente para aplicaciones *CPU-bound*. Para solucionar esta problemática han surgido traductores alternativos, aunque cada uno con un enfoque diferente y con su propia relación de costo-rendimiento. En primer lugar, se encuentra PyPy, un compilador JIT enfocado a optimizar algoritmos numéricos; en segundo lugar, aparece Numba, un compilador JIT que traduce Python en código de máquina optimizado a través de LLVM; y por último, se presenta Cython, un *superset* de Python que permite utilizar librerías de C. Las dos últimas herramientas cuentan con primitivas para paralelizar algoritmos, auto-vectorización de instrucciones, entre otras características habituales del procesamiento paralelo.

Ante la ausencia de estudios comparativos en el estado del arte, en esta tesina se realizó una evaluación de las prestaciones (rendimiento y esfuerzo de programación) de dichos traductores, utilizando como caso de estudio *N-Body*, un problema popular en simulación y con alta demanda computacional. Para cada uno de estos traductores se desarrollaron diferentes algoritmos, partiendo desde una versión base e introduciendo optimizaciones de forma incremental hasta llegar a la versión final. Estos fueron ejecutados en una arquitectura multicore Intel de 56 núcleos, en donde se pudo notar que PyPy y CPython produjeron un bajo rendimiento debido a sus limitaciones a la hora de paralelizar algoritmos; mientras que en sentido opuesto, Numba y Cython presentaron rendimientos notablemente superiores a los últimos y cercanos a los de una implementación de C+OpenMP, demostrando ser opciones viables para acelerar algoritmos numéricos. En cuanto al esfuerzo de programación, Cython requirió conocimiento adicional de C+OpenMP junto con un proceso más laborioso de ejecución. Por su parte, Numba tomó un mayor número de líneas de código aunque resultó más simple gracias a su enfoque de decoradores y a la posibilidad de reutilizar el código CPython de base.

En base a lo anterior, se puede afirmar que en contextos similares a los de este estudio tanto Numba como Cython pueden ser potentes herramientas para acelerar aplicaciones *CPU-bound* desarrolladas en Python. La elección entre uno y otro estará mayormente determinada por el enfoque que el equipo de desarrollo encuentre más conveniente, considerando las características propias de cada uno.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

En primer lugar, se presenta la motivación de este trabajo (Sección 1.1); posteriormente, se describen los objetivos y la metodología empleada (Sección 1.2); luego, se mencionan las contribuciones del trabajo (Sección 1.3); y por último, se mencionan las publicaciones que derivan de esta tesina (Sección 1.4) y se describe la organización del documento (Sección 1.5).

1.1. Motivación

Desde su surgimiento a comienzos de la década del 90, Python se ha convertido en uno de los lenguajes más populares en la actualidad. De acuerdo con la edición de Noviembre de 2021 del índice TIOBE, Python se ubica en la primera posición de los lenguajes más populares en el último año [1].

Python es un lenguaje de programación de alto nivel, interpretado, interactivo, dinámico y multi-paradigma [2]. Su notable poder de programación se debe a su sintaxis limpia y clara, la cual provoca que el esfuerzo de programación sea menor comparado con otros lenguajes [3, 4]. Entre las áreas de aplicación, se pueden mencionar desarrollo web, educación, aplicaciones de escritorio, desarrollo de videojuegos, inteligencia artificial, *web scraping*, procesamiento de imágenes, entre otras [5, 6].

En simultáneo al crecimiento de Python, el cómputo de altas prestaciones (HPC, por sus siglas en inglés) ha tomado mayor relevancia debido a la constante necesidad de disminuir el tiempo de respuesta y al crecimiento vertiginoso de los datos a procesar [7, 8, 9]. Por su parte, Python no ha sido ajeno a esta cuestión, existiendo diferentes alternativas que permiten explotar capacidades de concurrencia y paralelismo.

El intérprete oficial de Python es conocido como CPython [10], el cual presenta limitaciones al momento de implementar soluciones multi-hiladas. En

particular, el principal problema es la utilización de un componente llamado *Global Interpreter Lock* (GIL), el cual permite que solo un hilo se ejecute a la vez. Esta característica afecta fuertemente a las aplicaciones *CPU-bound* [1], ya que llevan a que su ejecución sea de forma secuencial. Para solucionar esta limitación, se suele utilizar procesos en vez de hilos, pero hay que tener en cuenta que el consumo de recursos es mayor y que aumenta el costo de programación por tener un espacio de direcciones distribuido [11].

Aunque existen intérpretes alternativos a CPython, algunos de estos también presentan el mismo problema. Tal es el caso de PyPy, un compilador *Just-In-Time* (JIT) [2] que está enfocado en el rendimiento del sistema [12, 13]. En sentido opuesto, existen intérpretes que optan por no utilizar el GIL en sus implementaciones, como por ejemplo Jython, una implementación de Python que se ejecuta sobre la máquina virtual de Java [14, 15]. Gracias a esto, Jython otorga la posibilidad de desarrollar programas multi-hilados sin la limitación del GIL. Lamentablemente, Jython emplea la versión 2 de Python, la cual se encuentra actualmente discontinuada [16, 17]. Esto último, limita el soporte a futuro para sus programas y la posibilidad de aprovechar las características que proveen las versiones posteriores del lenguaje.

Por otro lado, otros traductores optan por ofrecerle al programador desactivar este componente, tal como es el caso de Numba, un compilador JIT que traduce Python en código de máquina optimizado [18]. Numba utiliza una característica de Python conocida como decoradores [19], para intervenir lo menos posible en el código del programador.

Por último, se puede mencionar a Cython, un compilador estático que permite transpilar [3] Python a C, y luego compilarlo a código objeto [20]. También permite desactivar el GIL y utilizar librerías de C como OpenMP [21], lo cual resulta de suma utilidad para desarrollar programas multi-hilados.

Al momento de implementar una aplicación en Python, se debe seleccionar qué traductor utilizar. Esta elección es fundamental ya que no sólo impactará en el rendimiento del programa sino también en el tiempo requerido para desarrollo como también en el costo de mantenerlo a futuro. Para no tomar una decisión "a ciegas", resulta fundamental revisar la evidencia al respecto. Lamentablemente, la literatura disponible en la temática no es exhaustiva.

Si bien existen estudios que contemplan comparaciones de traductores, lo llevan a cabo usando versiones secuenciales [22, 23], lo que no permite evaluar sus capacidades de procesamiento paralelo. En sentido contrario, en el caso de sí

[1] Programas que realizan una gran cantidad de cálculos utilizando la CPU de manera exhaustiva.

[2] Compilación en tiempo de ejecución.

[3] Proceso que realiza una clase especial de compilador en la cual se genera código fuente en un lenguaje a partir del correspondiente a otro lenguaje.

usar paralelismo, lo hacen entre lenguajes y no entre traductores de Python [24, 25, 26, 27]. Por otra parte, en la mayoría de ellos no se hace un estudio sobre la productividad y el esfuerzo de programación que contrae desarrollar cada solución, una cuestión que día a día se torna más importante [28, 29].

En base a lo anterior, resulta fundamental conocer las ventajas y desventajas de diferentes traductores del lenguaje Python tanto en un paradigma secuencial como multi-hilado, así como las primitivas y funciones que permiten optimizar código. Por lo tanto, esta tesina se enfoca en su comparación considerando no sólo el rendimiento, sino también su productividad y esfuerzo de programación asociados. Como caso de estudio, se selecciona la simulación de N cuerpos computacionales (*N-Body*), un problema *CPU-bound* y que resulta popular en la comunidad HPC. Mediante este estudio comparativo se espera contribuir a programadores de Python para que conozcan fortalezas y debilidades al momento de implementar paralelismo en memoria compartida.

1.2. Objetivo y metodología

El objetivo de esta tesina consiste en comparar las prestaciones (rendimiento y esfuerzo de programación) de traductores del lenguaje Python en una arquitectura multicore, utilizando como caso de estudio el problema de N cuerpos computacionales con atracción gravitacional.

Para ello se llevaron a cabo las siguientes actividades:

- Se estudió el problema N-Body y sus requisitos computacionales.
- Se relevó la bibliografía existente en la temática a partir de la búsqueda en bases de datos especializadas.
- Se diseñaron y desarrollaron diferentes soluciones paralelas usando distintos traductores de Python al problema estudiado, considerando diversas optimizaciones aplicables.
- Se midió el rendimiento y el esfuerzo de programación entre las implementaciones mencionadas y se realizó un análisis comparativo.
- Se analizaron los resultados para determinar fortalezas y debilidades de los traductores de Python para aplicaciones que requieran cómputo intensivo.

1.3. Contribuciones

Entre las contribuciones de este trabajo, se pueden mencionar:

- Implementaciones optimizadas y escritas en diferentes traductores de Python que computen N-Body sobre arquitecturas multicore, las cuales se encontrarán en un repositorio web público para beneficio de la comunidad científica[4].
- Una comparación rigurosa de las soluciones para N-Body en arquitecturas multicore considerando rendimiento y esfuerzo de programación. Estos resultados servirán para que un programador de Python pueda identificar las fortalezas y debilidades a la hora de utilizarlos, y a su vez contribuirá en la evaluación del potencial de Python como un lenguaje de base alternativo en el ámbito de HPC.

1.4. Publicaciones

Este trabajo ha servido como base para las siguientes publicaciones y presentaciones orales:

- Andrés Milla y Enzo Rucci. «Acelerando Código Científico en Python usando Numba». En: *XXVII Congreso Argentino de Ciencias de la Computación (CACIC 2021)* (oct. de 2021), pág. 12. URL: `http://sedici.unlp.edu.ar/handle/10915/126012` [30]
- Andrés Milla y Enzo Rucci. *Acelerando aplicaciones paralelas en Python: Numba vs. Cython*. Conferencia. PyCon 2021, oct. de 2021. URL: `https://eventos.python.org.ar/events/pyconar2021/activity/448/` [31]

1.5. Organización del documento

El resto del documento se organiza de la siguiente manera. En primer lugar, se presenta el marco teórico sobre el que se basa el trabajo (Capítulo 2). Posteriormente se detallan las optimizaciones realizadas sobre *N-Body* utilizando los traductores CPython y Pypy, Numba y Cython (Capítulos 3, 4 y 5, respectivamente). Luego, se realiza una comparación sobre las prestaciones de los traductores mencionados (Capítulo 6) y, por último, se presentan las conclusiones y los posibles trabajos futuros (Capítulo 7).

[4] `https://github.com/Pastorsin/python-hpc-study/`

Capítulo 2

Marco teórico

En este capítulo, inicialmente se presenta a Python como lenguaje de programación junto con sus alternativas para optimizar algoritmos numéricos (Sección 2.1); posteriormente se detalla el problema *N-Body* (Sección 2.2); luego, se describe el estado del arte (Sección 2.3); y por último, se expone un resumen sobre el presente capítulo (Sección 2.4).

2.1. Python

Python es un lenguaje de programación creado en 1991 por Guido van Rossum [2]. Entre sus características más importantes se pueden destacar las siguientes:

- Es un lenguaje de alto nivel, interpretado, multiparadigma y con tipado dinámico.
- Se caracteriza porque posee una sintaxis clara y limpia, lo que lo convierte en un lenguaje con alto grado de legibilidad.
- Es de libre uso y distribución, incluso para fines comerciales [32]. Esto se debe a que el intérprete oficial (denominado CPython) está bajo una licencia de código abierto aprobada por la *Open Source Initiative*[1].
- Puede ejecutarse en diversas plataformas, entre las cuales se pueden mencionar Windows, macOS y Linux.

En la Figura 2.1 se puede apreciar la implementación y la ejecución del clásico programa "Hola Mundo" en Python. En el archivo `run.py` se encuentra

[1]Organización dedicada a la promoción del código abierto.

la implementación del programa, en el mismo se utiliza la función `print` para escribir el *string* "Hello World!" en la salida estándar del sistema. Luego, el programa es ejecutado a través de la línea de comandos utilizando el binario `python3`, el cual muestra en pantalla el *string* mencionado previamente.

Archivo `run.py`

```
print("Hello World!")
```

Ejecución

```
$ python3 run.py
Hello World!
```

Figura 2.1: Implementación y ejecución del clásico programa "Hola Mundo" en Python.

Tal como se mencionó en la Sección 1.1, Python es utilizado en diversas áreas debido a su alto poder de programación. Sin embargo, presenta fuertes limitaciones al momento de optimizar las aplicaciones por sus características de lenguaje interpretado. En las siguientes secciones se describen diversas alternativas para enmendar esta desventaja, aunque cada una lo realiza con un enfoque diferente y con su propia relación de costo-rendimiento.

2.1.1. NumPy

NumPy es un paquete dedicado a la computación científica y numérica [18]. Entre sus características se pueden destacar las siguientes:

- Ofrece una gran variedad de utilidades para el cómputo científico, de las cuales se pueden destacar diversas funciones matemáticas, generadores de números aleatorios, rutinas para álgebra lineal, entre otras.
- Mantiene la filosofía de sintaxis clara y limpia de Python, lo que hace que programadores con diferentes nivel de conocimiento puedan comprender las operaciones utilizadas.
- Es compatible con un gran número de plataformas basadas tanto en CPUs como GPUs.
- Su núcleo está escrito y optimizado en C, por lo que sus prestaciones se acercan a las que proveen implementaciones en este lenguaje.
- Permite realizar operaciones multidimensionales entre arreglos, denominadas *broadcasting* (ver Fig. 2.2). Tanto es así que los conceptos de vectorización, indexación y transmisión de NumPy son los estándares de facto de la computación de arreglos en la actualidad [18].

- Forma parte de un gran ecosistema de librerías y herramientas para diferentes fines, como por ejemplo *scikit-learn*[2], *biopython*[3], entre otras.

```
1 import numpy as np
2
3 N = 2
4
5 A = np.zeros((N, N))
6 B = np.ones((N, N))
7
8 AB = A - B
```

Figura 2.2: Implementación de una resta bidimensional utilizando NumPy.

A su vez, NumPy permite declarar el tipo de dato de los arreglos mediante el parámetro `dtype` y posibilita indicar la organización de memoria a través del parámetro `order`. Se puede optar por la organización de C (*Row major*) indicando `order='C'`, mientras que por otro lado se puede optar por la organización de Fortran (*Column major*) especificando `order='F'` (ver Fig. 2.3).

```
1 import numpy as np
2
3 N = 2
4
5 A = np.zeros((N, N), order="C", dtype=np.float64)
6 B = np.ones((N, N), order="C", dtype=np.float64)
7
8 AB = A - B
```

Figura 2.3: Implementación de una resta bidimensional utilizando NumPy indicando la organización de memoria junto con el tipo de dato.

Debido a las características mencionadas previamente, se considera a NumPy como una posible alternativa para acelerar código Python a través de sus funciones, las cuales se encuentran pre-compiladas en el lenguaje C.

2.1.2. PyPy

Como se mencionó previamente, CPython es *open-source*, por lo tanto existen una gran variedad de intérpretes alternativos [33] y algunos de ellos están

[2] `https://scikit-learn.org/stable/`
[3] `https://biopython.org/`

orientados a acelerar el cómputo numérico. Tal es el caso de PyPy, un compilador JIT que asegura incrementar la velocidad de ejecución y disminuir la memoria utilizada de los programas escritos en Python [12]. Para ello, se requieren dos condiciones:

- El programa no debe ser de ejecución corta, es decir, debe ejecutarse al menos por unos cuantos segundos. Si tarda menos, PyPy no podrá optimizar lo suficiente como para que la ganancia de tiempo se vea reflejada en la ejecución final [34].
- Debe utilizar la menor cantidad posible de paquetes escritos en C [18], ya que el compilador JIT no podrá optimizar el código fuera de Python. Por lo tanto, es interesante destacar que debido a que NumPy está escrito en C, no se podrá utilizar junto a PyPy para acelerar programas numéricos [34].

2.1.3. Threading

Python provee soporte para multi-hilado con el fin de acelerar la ejecución de las programas. Estas funcionalidades se pueden encontrar en el módulo `threading` [35], el cual se encuentra disponible de forma nativa en el lenguaje.

Creación de hilos El módulo ofrece dos opciones para crear hilos. A continuación se describen ambas.

1. *Basada en funciones*: Se debe instanciar la clase `Thread` indicando en el parámetro `target` la función que ejecutará el hilo. Además, se pueden especificar los argumentos que recibe la función mediante el parámetro nombrado `args` (ver Fig. 2.4).

```
1 from threading import Thread
2
3 def example(thread_id):
4     print(f"Thread id: {thread_id}!")
5
6 Thread(target=example, args=(0,))
```

Figura 2.4: Ejemplo de la creación de hilos basada en funciones utilizando el módulo `threading`.

2. *Basada en clases*: Se debe crear una clase que herede de la clase `Thread` y luego implementar el método `run`. En este caso, los argumentos se deben pasar a través del constructor de la clase (ver Fig. 2.5).

```python
from threading import Thread

class MyThread(Thread):
    def __init__(self, thread_id):
        super(MyThread, self).__init__()
        self.thread_id = thread_id

    def run(self):
        print(f"Thread id: {self.thread_id}")

MyThread(thread_id=0)
```

Figura 2.5: Ejemplo de la creación de hilos basada en clases utilizando el módulo `threading`

Ejecución de los hilos Para ejecutar los hilos se debe invocar al método de instancia `start`. En caso de querer bloquear al hilo llamador se debe utilizar el método de instancia `join` para esperar a que los hilos invocados terminen (ver Fig. 2.6).

```python
from threading import Thread

T = 4

def example(thread_id):
    print(f"Thread id: {thread_id}")

# Creación de T threads:
threads = [
    Thread(target=example, args=(thread_id,))
    for thread_id in range(T)
]

# Ejecución de los T threads:
for thread in threads:
    thread.start()

# Esperar por la finalización de los T threads:
for thread in threads:
    thread.join()
```

Figura 2.6: Ejemplo de la ejecución de hilos utilizando el módulo `threading`.

Destrucción de los hilos El módulo no ofrece una forma de destruir los hilos a partir de una sentencia, lo que recae en la responsabilidad al

programador. Por lo tanto, para destruir un hilo desde el hilo llamador se deberá realizar mediante excepciones, variables globales o alguna otra lógica impuesta por el programador.

Alcance de datos Al modelarse los hilos mediante funciones o clases, el alcance de datos es igual al del lenguaje, `threading` no ofrece reglas que modifiquen esto al nivel de librerías como OpenMP [21], es decir, si se desea trabajar con un dato global, se deberá definir un nivel por encima de las funciones o clases.

Sincronización `threading` ofrece mecanismos para sincronizar los hilos mediante exclusión mutua, como *locks* y semáforos a través de las clases `Lock` y `Semaphore`, respectivamente. Adicionalmente, el módulo permite sincronizar por condición mediante la clase `Condition` y establecer barreras a través de la clase `Barrier`.

En base a las características descritas previamente, hay que tener en cuenta que al utilizar `threading` como alternativa para acelerar código Python implica que el programador tenga que especificar el paralelismo explícitamente.

Por último, cabe destacar que Python provee otros dos módulos para emplear multi-hilado de forma nativa. El primero es conocido como `asyncio` pero su uso está destinado a programas de tipo *IO-bound* [4] [36], por lo tanto queda fuera del objetivo de este trabajo. Mientras que por otro lado, se encuentra la clase `ThreadPoolExecutor` correspondiente al módulo `concurrent.futures`, pero simplemente se trata de una abstracción al módulo `threading` descrito previamente [37].

2.1.3.1. Limitaciones

El GIL (*Global Interpreter Lock*) es un componente de CPython, el cual asegura que sólo un hilo se ejecute a la vez. Tal como dice su nombre, se trata de un *lock* a nivel global que asegura que el uso de los objetos de Python esté implícitamente a salvo de las consecuencias indeseables del acceso concurrente, como condiciones de carrera, *deadlocks*, entre otras [11].

Tal como se puede apreciar en la Fig. 2.7 esta característica afecta fuertemente a las aplicaciones *CPU-bound*, ya que llevan a que su ejecución sea prácticamente de forma secuencial, sumado al costo de adquirir y liberar el *lock* constantemente. Aun así, el módulo `threading` resulta adecuado para aplicaciones *IO-bound* dado que permite llevar a cabo operaciones de entrada/salida en forma concurrente y por ende reducir el tiempo de ejecución de este tipo de aplicaciones.

[4] Programas que realizan una gran cantidad de operaciones de Entrada/ Salida.

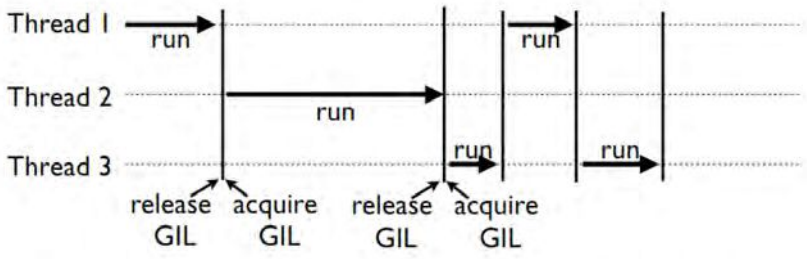

Figura 2.7: Comportamiento del GIL en una ejecución multi-hilada.

2.1.4. Multiprocessing

Multiprocessing es un paquete para Python que provee funcionalidades con el fin de paralelizar aplicaciones de forma similar al módulo `threading` (ver Sección 2.1.3) [38].

Gracias a la utilización de subprocesos en lugar de hilos, Multiprocessing no cuenta con la limitación del GIL descrita en la Sección 2.1.3.1. Sin embargo, hay que tener en cuenta que esto provoca un mayor consumo de recursos, lo que ocasiona *overheads* adicionales en la ejecución, y además, al no compartir memoria, se requiere un mayor esfuerzo de programación a la hora de comunicar datos. No obstante, este paquete queda fuera del alcance del trabajo por el objetivo propuesto.

2.1.5. Numba

Numba es un compilador JIT que permite traducir código Python a código de máquina optimizado a través de LLVM [5]. De acuerdo con su documentación, es capaz de alcanzar aceleraciones similares a las de lenguajes compilados como C, C++ y Fortran [18], sin necesidad de re-escribir su código gracias a un enfoque de anotaciones llamados decoradores [19].

Compilación JIT La librería ofrece dos modos de compilación: (1) modo objeto, el cual permite compilar código que haga uso de objetos; (2) modo *nopython*, que le permite a Numba generar código evitando a la API de CPython. Para indicar dichos modos, se utilizan los decoradores `@jit` y `@njit` (ver Fig. 2.8), respectivamente [18].

Por defecto, cada función será compilada al momento de ser invocada y se mantendrá en la caché para futuras llamadas. Sin embargo, la inclusión del parámetro `signature` provocará que la función sea compilada

[5]The LLVM Compiler Infrastructure, `https://llvm.org/`

```python
from numba import njit

# Equivalente a indicar
# @jit(nopython=True)
@njit
def f(x, y):
    return x + y
```

Figura 2.8: Compilación en modo *nopython*.

al momento de la declaración. En este último se indicará el tipo de dato del valor retornado y de los argumentos recibidos por la función.

Cabe destacar que Numba también ofrece una compilación *ahead-of-time* [39], lo que permite compilar el programa previo a su ejecución.

Tipos de datos Si bien Numba infiere los tipos de datos de las variables, también le da posibilidad al programador de indicarlos y controlar la organización de los datos en memoria de ellos (ver Fig. 2.9).

```python
from numba import njit, double

@njit(double(double[::1, :],
             double[:, ::1]))
def f(x, y):
    """
    x:  Vector 2D de tipo Double
        organizado por columnas.
    y:  Vector 2D de tipo Double
        organizado por filas.
    Retorna la suma del producto
    entre los vectores x e y.
    """
    return (x * y).sum()
```

Figura 2.9: Compilación en modo *nopython* con el parámetro `signature`

Multi-hilado Como se mencionó anteriormente, el decorador `@njit` le permite a Numba evitar la API de CPython, lo que le brinda la posibilidad al programador de desactivar el GIL mediante la sentencia `@njit(nogil=True)`, lo cual resulta de sumo interés para desarrollar programas paralelos [40].

Adicionalmente, Numba permite activar un sistema de paralelización automática estableciendo el parámetro `parallel=True`, como también

indicar una paralelización explícita mediante la función `prange` (ver Fig. 2.10), la cual distribuye las iteraciones entre los hilos de manera similar a la directiva `parallel for` de OpenMP.

Cabe destacar que Numba se encarga de traducir las zonas paralelas a diferentes API de hilos mediante librerías internas, las cuales son denominadas *threading layers* [41]. Las opciones disponibles son las siguientes:

- *default*: Es la *threading layer* por defecto y se encarga de seleccionar la mejor API de hilos de acuerdo con la configuración del sistema.
- *omp*: Esta *threading layer* utiliza OpenMP como API de hilos.
- *tbb*: Esta *threading layer* utiliza TBB[6] como API de hilos.
- *workqueue*: Es una implementación propia de Numba que evita la ausencia de *threading layer* en caso de que el sistema no soporte OpenMP ni TBB.

Como última opción, es importante mencionar que se puede utilizar el módulo `threading` y desactivar el GIL a través de Numba, lo que permitirá que los hilos se ejecuten de forma paralela. Sin embargo, hay que tener en cuenta que las *threading layers* mencionadas previamente no se activarán, y el programador deberá encargarse explícitamente de la distribución de trabajo entre los hilos y de su adecuada sincronización.

Alcance de datos Numba se encarga de declarar las variables como privadas a cada hilo si son declaradas dentro del alcance de la zona paralela, y además permite realizar reducciones sobre ellas.

Sincronización Numba aún no soporta nativamente primitivas que permitan controlar la sincronización de los hilos, como pueden ser semáforos o *locks* [18]. Sin embargo, se pueden utilizar las sentencias de sincronización del módulo `threading` descrito anteriormente (ver sección 2.1.3). Hay que tener en cuenta que en este último caso se estarían utilizando objetos de Python, y por lo tanto la ejecución va a sufrir el *overhead* provocado por los mismos.

Vectorización Numba delega en LLVM la autovectorización del código y la generación de instrucciones SIMD, pero le permite al programador controlar ciertos parámetros que podrían influir en esta tarea, como la precisión numérica mediante el argumento `fastmath=True`. También ofrece

[6] `https://www.intel.com/content/www/us/en/developer/tools/oneapi/onetbb.html`

```python
from numba import njit, double, prange

@njit(double(double[::1]), parallel=True)
def f(x):
    """
    x: Vector 1D.
    Retorna la suma del vector x mediante
    una reducción.
    """
    N = x.shape[0]
    z = 0

    for i in prange(N):
        z += x[i]

    return z
```

Figura 2.10: Compilación en modo *nopython* con el parámetro `parallel`

la posibilidad de utilizar *Intel SVML* en caso de estar disponible en el sistema [18].

Integración con NumPy Cabe destacar que Numba soporta un gran número de funciones de NumPy, lo cual le permite al programador controlar la organización de memoria de los arreglos y realizar operaciones entre ellos [18].

Soporte para GPUs Además de CPUs, Numba es capaz de aprovechar las capacidades de las GPUs, tanto de NVIDIA como de AMD.

2.1.6. Cython

Cython es un compilador estático para Python creado con el objetivo de escribir código en C aprovechando la sintaxis simple y clara de Python [20]. En otras palabras, Cython es un *superset* de Python que permite interactuar con funciones, tipos y librerías de C.

Flujo de programación Tal como se puede apreciar en la Fig. 2.11 el flujo de programación de Cython es muy diferente al que el programador de Python está habituado.

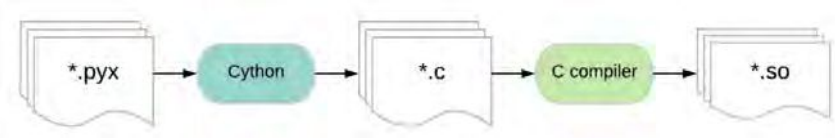

Figura 2.11: Flujo de programación en Cython.

La principal diferencia es que el archivo que contendrá el código fuente tendrá extensión `.pyx` a diferencia de Python, cuya extensión es `.py`.

Luego, este archivo se deberá compilar a través de un archivo `setup.py`, en donde se indican los flags de compilación para dar como salida los siguientes archivos:

1. Un archivo con extensión `.c`, el cual corresponde al código transpilado de Cython a C.
2. Un archivo binario con extensión `.so`, el cual corresponde a la compilación del archivo de C descrito previamente. Este último archivo, nos permitirá importar el módulo compilado en cualquier script de Python.

Tipos de datos Cython permite declarar variables utilizando los tipos de datos de C a partir de la sentencia `cdef` (ver Fig. 2.12). Si bien esto es opcional, la documentación lo recomienda para optimizar la ejecución del programa, ya que se evita la inferencia de tipos de CPython en tiempo de ejecución [42].

```
1  cdef int x, y, z
2  cdef float a, b[100], *c
3
4  cdef struct Point:
5      double x
6      double y
```

Figura 2.12: Variables declaradas con tipos de datos de C en Cython.

Por último, cabe destacar que Cython permite indicar la organización de memoria de los arreglos al igual que Numba (ver sección 2.1.5).

Funciones Existen tres modos para definir una función en Cython:

1. *Funciones de Python*: La función se especifica de la misma manera que se lo hace en Python, utilizando la palabra clave `def`. Las mismas reciben y retornan objetos de Python.

2. *Funciones de C*: Se especifican con la palabra clave `cdef`. Las mismas pueden recibir y retornar tanto objetos de Python como valores de C. Cabe destacar que este tipo de funciones no pueden ser invocadas por fuera del módulo, es decir, desde un *script* de Python.
3. *Función híbrida*: Se declara con la palabra clave `cpdef`. Es similar a una función de C, con la salvedad de que puede ser invocada fuera del módulo. Internamente es una función de Python que llama a una función de C, por lo que llamadas reiterativas pueden provocar *overhead* en el tiempo de ejecución del algoritmo [42].

Multi-hilado Cython provee soporte para utilizar OpenMP a través del módulo `cython.parallel`. El mismo permite utilizar la función `prange`, la cual posibilita paralelizar bucles mediante el constructor `parallel for` de OpenMP. A continuación se detallan sus argumentos:

- Los argumentos `start, stop, step` indican el comienzo, fin e intervalo del bucle, respectivamente.
- El argumento `nogil` permite desactivar el GIL de Python especificando el valor `True`.
- Los argumentos `schedule` y `chunksize` permiten indicar el *scheduling* del constructor `parallel for` de OpenMP. Los valores disponibles son: `static`, `dynamic`, `guided` y `runtime`.
- Se puede indicar la cantidad de hilos que van a intervenir en la sección paralela a través del argumento `num_threads`.

Alcance de datos Si se produce una asignación en un bloque `prange`, la variable es transpilada como `lastprivate` en el constructor `parallel for`. Esto significa que la variable contendrá el valor de la última iteración.

Por otro lado, las reducciones son identificadas si se utiliza un operador "*in-situ*". Por ejemplo, la operación estándar de la suma (`x = x + y`) no será identificada como reducción, pero la operación "*in-situ*" de la suma (`x += y`) sí (ver Fig. 2.13). Por último, cabe destacar que el índice utilizado en el bloque `prange` siempre será `lastprivate`.

```
1 from cython.parallel import prange
2
3 cdef int i
4 cdef int N = 30
5 cdef int total = 0
6
7 for i in prange(N, nogil=True):
8     total += i
```

Figura 2.13: Reducción utilizando el bloque `prange` de Cython.

Sincronización Se pueden utilizar todas las funciones de sincronización que provee OpenMP a través del módulo `openmp` de Cython.

Vectorización Cython delega la vectorización en el compilador de C que se esté utilizando. Si bien existen soluciones alternativas para forzar la vectorización, nativamente no es soportado por Cython.

Integración con NumPy Lamentablemente, las operaciones entre vectores de NumPy no son soportadas por Cython. Sin embargo, como se mencionó anteriormente se puede utilizar NumPy para controlar la organización de memoria de los arreglos.

Soporte para GPUs Nativamente Cython no provee ninguna funcionalidad para utilizar GPUs. De todas formas, al ser un *superset* de Python se puede combinar con librerías desarrolladas para este fin.

2.2. N-Body

El problema consiste en simular la evolución de un sistema compuesto por N cuerpos durante una cantidad de tiempo determinada. Dados la masa y el estado inicial (velocidad y posición) de cada cuerpo, se simula el movimiento del sistema a través de instantes discretos de tiempo. En cada uno de ellos, todo cuerpo experimenta una aceleración que surge de la atracción gravitacional del resto, lo que afecta a su estado y el de los demás cuerpos.

N-Body es sumamente utilizado para solucionar diversos problemas pertenecientes a diferentes áreas de la comunidad científica. Aunque principalmente se lo utiliza en la astrofísica para simular la fuerza de atracción entre galaxias y estrellas [43], también se puede mencionar uso en áreas muy diferentes como la biología computacional [44] o la computación gráfica [45].

2.2.1. Fundamentos

La física subyacente es fundamentalmente la mecánica de Newton [46] aplicada en un espacio tridimensional.

Cuando N es igual a 2, la atracción gravitacional entre dos cuerpos C_1 y C_2 se calcula mediante la ley de gravitación universal de Newton (ver Ecuación 2.1), en donde F corresponde a la magnitud de la fuerza gravitacional entre los cuerpos; G corresponde a la constante de gravitación universal [7]; m_1 y m_2 corresponden a las masas de los cuerpos C_1 y C_2, respectivamente. Finalmente, r corresponde a la distancia Euclídea [8] entre los cuerpos C_1 y C_2.

$$F = \frac{G \times m_1 \times m_2}{r^2} \tag{2.1}$$

Cuando N es mayor a 2, es decir, cuando hay más de dos cuerpos en el espacio. La fuerza de gravitación sobre un cuerpo se obtiene con la sumatoria de todas las fuerzas de gravitación ejercidas por los $N - 1$ cuerpos restantes.

La fuerza de atracción se traduce en una aceleración del cuerpo mediante la aplicación de la segunda ley de Newton, la cual está dada por la Ecuación 2.2, en donde F es el vector fuerza, que es calculado utilizando la magnitud obtenida con la ecuación de gravitación y la dirección con el sentido del vector que va desde el cuerpo afectado hacia el cuerpo que ejerce la atracción.

$$F = m \times a \tag{2.2}$$

La aceleración de un cuerpo se puede calcular a partir de la ecuación anterior, dividiendo la fuerza total por su masa (ver ecuación 2.3).

$$a = \frac{F}{m} \tag{2.3}$$

Mediante la aceleración se puede obtener la diferencia de posición de los cuerpos utilizando las ecuaciones del movimiento de la mecánica Newtoniana (ver Ecuaciones 2.4 y 2.5).

$$\frac{dx_i}{dt} = v_i \tag{2.4}$$

$$\frac{dv_i}{dt} = a_i \tag{2.5}$$

[7] Equivalente a 6.674×10^{11}

[8] Se calcula utilizando la fórmula $\sqrt{((x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2)}$, siendo (x_1, y_1, z_1) las coordenadas de $C1$ y (x_2, y_2, z_2) las coordenadas de C_2.

Por lo tanto, de las fórmulas anteriores podemos concluir que durante un pequeño intervalo de tiempo dt, la aceleración a_i del cuerpo C_i es aproximadamente constante, por lo que el cambio en velocidad está dado aproximadamente por la Ecuación 2.6.

$$dv_i = a_i dt \tag{2.6}$$

Luego, el cambio en la posición de un cuerpo es la integral de su velocidad y aceleración sobre el intervalo de tiempo dt, el cual se aproxima a la Ecuación 2.7. Esta fórmula emplea el método de integración *velocity verlet*, en el cual una mitad del cambio de posición emplea la velocidad "vieja" mientras que la otra considera la velocidad "nueva". A su vez, este método es la aplicación del esquema de integración *Leapfrog* [47].

$$dp_i = v_i dt + \frac{a_i}{2} dt^2 = (v_i + \frac{dv_i}{2}) dt \tag{2.7}$$

2.2.2. Algoritmo

En el Algoritmo 1 se puede apreciar el pseudocódigo de la solución a *N-Body* utilizando los fundamentos explicados en la Sección 2.2.1.

Algoritmo 1 N-Body

```
1: for i = 1, ..., N do
2:     for j = 1, ..., N do
3:         Calcular la fuerza de atracción que ejerce C_j sobre C_i
4:         Sumar las fuerzas que afectan a C_i
5:     end for
6:     Calcular el desplazamiento de C_i
7:     Actualizar las posiciones del cuerpo C_i
8: end for
```

Se puede notar que por cada cuerpo del espacio (línea 1) se calcula la fuerza gravitacional de Newton sobre los cuerpos restantes (líneas 2-5), y luego se aplica el método de integración *velocity verlet* para mover al cuerpo y actualizar las posiciones (líneas 6-7).

La complejidad de este algoritmo es de $\mathcal{O}(n^2)$, ya que se requieren n^2 iteraciones para computar las fuerzas que afectan a cada cuerpo del espacio (líneas 1-4).

A la hora de paralelizar el algoritmo hay que tener en cuenta las dependencias de datos. Es decir, el cálculo de las fuerzas depende de las iteraciones

anteriores y para mover al cuerpo se necesita previamente calcular las fuerzas que lo afectan.

Por último, cabe destacar que esta implementación es conocida en la bibliografía como el método directo (*all pairs*) [43]. Sin embargo, existen métodos alternativos que pueden resolver *N-Body* con una complejidad de $\mathcal{O}(n \log n)$ [48]. No obstante, estos últimos quedan por fuera del objetivo de este trabajo.

2.3. Estado del arte

En la literatura disponible, se pueden encontrar algunos trabajos que evalúan aspectos prestacionales de Python.

En [24] se realizó un estudio sobre lenguajes de alto nivel para resolver algoritmos de complejidad NP-Hard. Para ello, se realizaron métricas sobre escalabilidad, rendimiento y productividad. En particular, se eligió a Numba como traductor de Python, ya que requirió menos modificaciones en el código. El mismo, se colocó en segunda posición respecto a productividad, pero con un rendimiento muy por debajo al de C a medida que se aumenta la escalabilidad del problema.

En [25] se estudiaron potenciales lenguajes novedosos para la comunidad de HPC. En particular, Python fue considerado como un lenguaje candidato por sus características de simplicidad, pero se lo descartó por ser un lenguaje interpretado y con manejo automático de memoria.

En [22] se muestra a Cython como alternativa para cálculo numérico en soluciones mono-hiladas. El foco del estudio es optimizar la vectorización, al nivel de lenguajes como C y Fortran, es por esto que también se plantean herramientas como F2PY [49], la cual permite transpilar código Python a código Fortran. Como resultado, la utilización de Cython llegó a aportar rendimientos casi tan buenos como la utilización de Fortran y C, pero de manera más fácil para el programador.

En [26] se evalúa el rendimiento de Python frente a aplicaciones científicas. Para ello, se lo compara con lenguajes como Fortran y C, y se analiza qué tan eficiente resulta como lenguaje para implementar paralelismo. Como conclusión, se tuvo que Python es lo suficientemente eficiente para resolver aplicaciones científicas de forma paralela, pero esto requiere utilizar librerías para manejo de vectorización, como lo es NumPy, y transpiladores que permitan traducir los bucles de Python a código nativo de C o Fortran.

En [23] se hace un análisis de rendimiento entre las estructuras de datos que provee Python, utilizando CPython, Cython, Jython y Pypy. En particular, se elige como caso de estudio una aplicación empresarial llamada "*triReduce*", la cual realiza cálculos financieros de forma secuencial. Como resultado, se

encontró que ningún traductor fue superior a otro en términos de rendimiento para todos los casos.

En [27] se muestra que el pobre rendimiento de Python comparado a lenguajes como C, se puede minimizar si se utilizan compiladores JIT. Un ejemplo es Numba que aplica tanto a soluciones secuenciales como paralelas.

En resumen, se puede apreciar que los estudios existentes contemplan comparaciones de traductores pero usando versiones secuenciales, lo que no permite evaluar sus capacidades de procesamiento paralelo. En sentido contrario, en el caso de usar paralelismo, lo hacen entre lenguajes y no entre traductores de Python. Por otra parte, en la mayoría de ellos no se hace un estudio sobre la productividad y el esfuerzo de programación que contrae desarrollar cada solución, una cuestión que día a día se torna más importante [28, 29].

2.4. Resumen

En este capítulo se presentó a Python como un lenguaje de programación de alto nivel con un gran poder de programación gracias a sus características notables, de las cuales principalmente se destaca su alto grado de legibilidad. Sin embargo, por su naturaleza de lenguaje interpretado, Python presenta fuertes limitaciones al momento de optimizar las aplicaciones; por ende, surgen diferentes alternativas para enmendar este inconveniente.

Una alternativa de bajo costo consiste en utilizar las funciones precompiladas que provee NumPy, una librería para Python que está destinada a la computación científica. De forma similar, se puede optar por PyPy, un intérprete alternativo a CPython orientado a la optimización del cómputo numérico. En ambos enfoques la optimización se logra implícitamente por el uso de cada herramienta.

Por otra parte, las aplicaciones pueden optimizarse mediante paralelismo explícito usando multi-hilado; para ello Python ofrece el módulo `threading`. Resulta importante aclarar que sólo tiene sentido para aplicaciones *IO-Bound* debido al componente de CPython denominado GIL, el cual permite que solo un hilo se ejecute a la vez. En consecuencia, para aplicaciones *CPU-Bound*, se suelen utilizar procesos en vez de hilos a través del módulo `multiprocessing`. Aún cuando esta alternativa permite ejecución simultánea de procesos, hay que tener en cuenta que el consumo de recursos es mayor y que el costo de programación aumenta por tener un espacio de direcciones distribuido. Debido a esto, han surgido alternativas que buscan lograr paralelismo en ejecuciones multi-hiladas. Entre ellas, se encuentran Numba, un compilador JIT que permite paralelizar aplicaciones interviniendo lo menos posible en el código del programador; y Cython, un *superset* de Python que permite utilizar librerías

de C como OpenMP para paralelizar las aplicaciones.

Luego, en este capítulo se describió el caso de estudio elegido, *N-Body*, un problema *CPU-bound* con complejidad $\mathcal{O}(n^2)$ que es sumamente utilizado en áreas científicas como la astrofísica. El problema consiste en simular la evolución de un sistema compuesto por N cuerpos durante una cantidad de tiempo determinada.

Por último, se describieron trabajos existentes que evalúan las prestaciones de Python, tanto en un ambiente secuencial como paralelo, mostrando la ausencia de un estudio como el que se propone en el presente trabajo.

Capítulo 3

Optimización de N-Body usando CPython y PyPy

En este capítulo, se presentan las diferentes implementaciones propuestas (Sección 3.1); luego, se muestran los resultados experimentales realizados (Sección 3.2); y por último, se expone un resumen del presente capítulo (Sección 3.3).

3.1. Implementaciones

En esta sección se describen las diferentes implementaciones propuestas utilizando CPython y PyPy como intérpretes. Para ello, se siguió un enfoque incremental comenzando con una versión secuencial (denominada *naive*) y luego se exploraron diferentes optimizaciones que fueron incorporadas de acuerdo al resultado encontrado.

3.1.1. Implementación Naive

Inicialmente se desarrolló una implementación Python "pura" (denominada *naive*) siguiendo el pseudocódigo del Algoritmo 1, la cual puede verse en la Figura 3.1.

```python
def nbody(
    N, D,
    positions_x, positions_y, positions_z,
    masses,
    velocities_x, velocities_y, velocities_z,
    dp_x, dp_y, dp_z,
):
    # For each discrete instant of time
    for _ in range(D):
        # For every body that experiences a force
        for i in range(N):
            # Initialize the force of the body i
            forces_x = forces_y = forces_z = 0.0

            # Calculate gravitational force to the rest of the bodies
            # Newton's Law of Universal Attraction:
            # Fi = (G * mi * mj * (qj - qi)) / (|qj - qi| ^ 3)
            for j in range(N):
                # Calculate the distance to the body i:
                # (qj - qi)
                dpos_x = positions_x[j] - positions_x[i]
                dpos_y = positions_y[j] - positions_y[i]
                dpos_z = positions_z[j] - positions_z[i]
                # Calculate the distance magnitude:
                # |qj - qi|
                dsquared = (
                    dpos_x ** 2.0 + dpos_y ** 2.0 + dpos_z ** 2.0 + SOFT
                )
                # Calculate the mass factor:
                # (G * mi * mj)
                gm = GRAVITY * masses[j] * masses[i]
                # Calculate the Fi denominator:
                # 1 / |qj - qi| ^ 3
                d32 = dsquared ** -1.5
                # Calculate the force:
                # (G * mi * mj * (qj - qi)) / |qj - qi| ^ 3
                forces_x += gm * d32 * dpos_x
                forces_y += gm * d32 * dpos_y
                forces_z += gm * d32 * dpos_z

            # Calculate the acceleration vector of body i:
            # |M = F * A|  |A = F / M|
            aceleration_x = forces_x / masses[i]
            aceleration_y = forces_y / masses[i]
            aceleration_z = forces_z / masses[i]
            # Velocity Verlet integrator
            # V = V' + ((A * h) / 2) ; h = 1
            velocities_x[i] += aceleration_x * DT / 2.0
            velocities_y[i] += aceleration_y * DT / 2.0
            velocities_z[i] += aceleration_z * DT / 2.0
            # Save the differential position of body i
            dp_x[i] = velocities_x[i] * DT
            dp_y[i] = velocities_y[i] * DT
            dp_z[i] = velocities_z[i] * DT

        for i in range(N):
            # Update positions of the bodies
            positions_x[i] += dp_x[i]
            positions_y[i] += dp_y[i]
            positions_z[i] += dp_z[i]
```

Figura 3.1: Implementación CPython *naive*.

Tal como se puede observar en la Fig. 3.1, el bucle de la línea 9 itera por cada instante discreto de tiempo de la simulación. Luego, por cada cuerpo (línea 11) se calcula la fuerza gravitacional ejercidas por los demás cuerpos (líneas 18-39) y una vez hecho esto, se calcula el desplazamiento del cuerpo (líneas 41-54) utilizando el método de integración *velocity verlet*. Como último paso de la simulación, se mueven los cuerpos actualizando su posición (líneas 56-60).

Para finalizar, cabe destacar que la función es invocada utilizando listas de Python como estructura de datos.

3.1.2. Integración de NumPy

Como se mencionó en la Sección 2.1.1, la utilización de arreglos de NumPy permite acelerar el tiempo de cómputo, ya que su núcleo se encuentra implementado y optimizado en el lenguaje C. Por lo tanto, como segunda optimización se opta por utilizar dichos arreglos como estructura de datos. El código que computa la simulación es el mismo que el de la versión *naive* (ver Fig. 3.1).

3.1.3. Broadcasting

Se reemplazaron las operaciones tradicionales entre arreglos por las operaciones entre vectores que provee NumPy (*broadcasting*). A continuación se explica de forma detallada cada operación vectorial utilizada (ver Fig. 3.2).

- Para cada cuerpo (línea 5) se calcula la distancia con los cuerpos restantes (línea 10). Para ello se restan los vectores de posición de cada cuerpo (dimensión $(N, 3)$ [1]) con el vector de posición del cuerpo actual (dimensión (3)). Esto da como resultado un vector de dimensión $(N, 3)$, denominado *dpos*.

- En la línea 13 se calculan las magnitudes de las distancias, elevando al cuadrado el vector *dpos* (dimensión $(N, 3)$) y sumando cada vector de dimensión 3 a través del eje 1. Esto produce un vector de dimensión (N) denominado *dsquared*, al que luego se le suma la constante de suavizado[2].

- En la línea 16 se calculan los factores de masa, multiplicando el vector de la masas de cada cuerpo (dimensión (N)) por el vector del cuerpo

[1] Según la notación de ejes de NumPy [50], cada componente de la tupla indica la cantidad de elementos de la dimensión correspondiente. Por ejemplo, la tupla $(N, 3)$ indica dos dimensiones, la primera con N elementos y la segunda con 3.

[2] La suma de esta constante evita resultados numéricos inconsistentes.

actual y por la constante de gravitación universal de Newton. Esto da como resultado un vector de dimensión (N), denominado gm.

- En la línea 19 se eleva a $-\frac{3}{2}$ el vector $dsquared$ (dimensión (N)), el cual produce un vector de dimensión (N), denominado $d32$.
- En la línea 21 se calcula parcialmente el vector de fuerzas. Para ello se multiplica el vector gm (dimensión (N)) por el vector $d32$ (dimensión (N)), y luego se transforma la dimensión a $(N, 1)$. Es decir, como salida se obtendrá un vector de N elementos, y cada uno de ellos tendrá un elemento. Esto es necesario debido a las reglas de *broadcasting* de NumPy [51] para la operación siguiente.
- Finalmente, en la línea 24 se totalizan las fuerzas que afectan al cuerpo actual. Para esto, el vector obtenido previamente (dimensión $(N, 1)$) se multiplica por el vector $dpos$ (dimensión $(N, 3)$), dando como resultado un vector parcial de dimensión $(N, 3)$. Luego, se suman los elementos a través del eje 0 y se obtiene el vector de fuerza de dimensión (3).
- En la línea 28 se calcula la aceleración dividiendo el vector de fuerzas (dimensión (3)) por la masa del cuerpo actual.
- En la línea 31 se actualiza el vector de velocidad del cuerpo actual (dimensión (3)) sumando el vector de aceleración multiplicado por la constante $\frac{DT}{2}$.
- En la línea 33 se calcula el vector de diferencial de posición (dimensión (3)), y luego en la línea 37 se lo utiliza para actualizar la posición de todos los cuerpos.

```
1  def nbody(N, D, positions, masses, velocities, dp):
2      # For each discrete instant of time
3      for _ in range(D):
4          # For every body that experiences a force
5          for i in range(N):
6              # Newton's Law of Universal Attraction:
7              # Fi = (G * mi * mj * (qj - qi)) / (|qj - qi| ^ 3)
8              # Calculate the distances to the body i:
9              # (qj - qi)
10             dpos = np.subtract(positions, positions[i])
11             # Calculate the distance magnitudes:
12             # |qj - qi|
13             dsquared = (dpos ** 2.0).sum(axis=1) + SOFT
14             # Calculate the mass factors:
15             # (G * mi * mj)
16             gm = masses * (masses[i] * GRAVITY)
17             # Calculate the Fi denominator:
18             # 1 / |qj - qi| ^ 3
19             d32 = dsquared ** -1.5
20             # (G * mi * mj) / |qj - qi| ^ 3
21             gm_d32 = (gm * d32).reshape(N, 1)
22             # Calculate the forces:
23             # (G * mi * mj * (qj - qi)) / |qj - qi| ^ 3
24             forces = np.multiply(gm_d32, dpos).sum(axis=0)
25
26             # Calculate the acceleration vector of body i:
27             # |M = F * A|  |A = F / M|
28             acceleration = forces / masses[i]
29             # Velocity Verlet integrator
30             # V = V' + ((A * h) / 2) ; h = 1
31             velocities[i] += acceleration * DT / 2.0
32             # Save the differential position of body i
33             dp[i] = velocities[i] * DT
34
35         # Update positions of the bodies
36         for i in range(N):
37             positions[i] += dp[i]
```

Figura 3.2: Implementación CPython con *broadcasting*.

3.1.4. Localidad de datos

Con el fin de mejorar la localidad de los datos[3], se implementó una versión que itera los cuerpos de a bloques, en forma similar a [52]. En la misma se busca minimizar el tráfico a la memoria principal utilizando a la memoria caché de manera más adecuada.

Como se puede notar en la Figura 3.3 el bucle de la línea 4 itera sobre bloques de cuerpos, y en el bucle más interno de la línea 13, se calculará la fuerza de atracción gravitacional de Newton y luego la integración de Verlet. Por último, en el bucle de la línea 46 se actualizan las posiciones iterando de

[3]Localidad de datos refiere al fenómeno que se produce en un programa cuando este tiende a acceder a las mismas ubicaciones de memoria (o a ubicaciones cercanas) en un periodo de tiempo particular.

a bloques.

```
def nbody(N, D, positions, masses, velocities, dp):
    # For each discrete instant of time
    for _ in range(D):
        for first in range(0, N, BLOCKSIZE):
            last = first + BLOCKSIZE

            # Init the force
            forces = np.zeros((BLOCKSIZE, 3), dtype=DATATYPE)

            # Calculate gravitational force to the rest of the bodies
            # Newton's Law of Universal Attraction:
            # Fi = (G * mi * mj * (qj - qi)) / (|qj - qi| ^ 3)
            for j in range(N):
                # Newton's Law of Universal Attraction:
                # Fi = (G * mi * mj * (qj - qi)) / (|qj - qi| ^ 3)
                # Calculate the distances to the body i:
                # (qj - qi)
                dpos = np.subtract(positions[j], positions[first:last])
                # Calculate the distance magnitudes:
                # |qj - qi|
                dsquared = (dpos ** 2.0).sum(axis=1) + SOFT
                # Calculate the mass factors:
                # (G * mi * mj)
                gm = masses[first:last] * (masses[j] * GRAVITY)
                # Calculate the Fi denominator:
                # 1 / |qj - qi| ^ 3
                d32 = dsquared ** -1.5
                # (G * mi * mj) / |qj - qi| ^ 3
                gm_d32 = (gm * d32).reshape(BLOCKSIZE, 1)
                # Calculate the forces:
                # (G * mi * mj * (qj - qi)) / |qj - qi| ^ 3
                forces += np.multiply(gm_d32, dpos)

            # Calculate the acceleration vector of body i:
            # |M = F * A|  |A = F / M|
            # f_i = i - first
            aceleration = forces / masses[first:last].reshape(BLOCKSIZE, 1)
            # Velocity Verlet integrator
            # V = V' + ((A * h) / 2) ; h = 1
            velocities[first:last] += aceleration * DT / 2.0
            dp[first:last] = velocities[first:last] * DT

        # Update positions of the bodies
        for first in range(0, N, BLOCKSIZE):
            last = first + BLOCKSIZE
            positions[first:last] += dp[first:last]
```

Figura 3.3: Implementación CPython utilizando bloques.

3.1.5. Multi-hilado

Se utilizó el módulo `threading` de Python para obtener una versión multi-hilada (ver Fig. 3.4). Para ello, cada hilo se encargará de computar una porción de los cuerpos denominados *chunks* para repartir la carga de trabajo.

Dada las dependencias de datos descritas en la Sección 2.2.2, se necesita

explicitar dos barreras para sincronizar los hilos, una para esperar que todos los hilos calculen el diferencial de posición (línea 35) y otra para esperar que todos los hilos actualicen las posiciones (línea 41).

```
1  def nbody(N, D, positions, masses, velocities, dp, chunk, barrier):
2      # For each discrete instant of time
3      for _ in range(D):
4          # For every body that experiences a force
5          for i in chunk:
6              # Newton's Law of Universal Attraction:
7              # Fi = (G * mi * mj * (qj - qi)) / (|qj - qi| ^ 3)
8              # Calculate the distances to the body i:
9              # (qj - qi)
10             dpos = np.subtract(positions, positions[i])
11             # Calculate the distance magnitudes:
12             # |qj - qi|
13             dsquared = (dpos ** 2.0).sum(axis=1) + SOFT
14             # Calculate the mass factors:
15             # (G * mi * mj)
16             gm = masses * (masses[i] * GRAVITY)
17             # Calculate the Fi denominator:
18             # 1 / |qj - qi| ^ 3
19             d32 = dsquared ** -1.5
20             # (G * mi * mj) / |qj - qi| ^ 3
21             gm_d32 = (gm * d32).reshape(N, 1)
22             # Calculate the forces:
23             # (G * mi * mj * (qj - qi)) / |qj - qi| ^ 3
24             forces = np.multiply(gm_d32, dpos).sum(axis=0)
25
26             # Calculate the acceleration vector of body i:
27             # |M = F * A|  |A = F / M|
28             acceleration = forces / masses[i]
29             # Velocity Verlet integrator
30             # V = V' + ((A * h) / 2) ; h = 1
31             velocities[i] += acceleration * DT / 2.0
32             # Save the differential position of body i
33             dp[i] = velocities[i] * DT
34
35         barrier.wait()
36
37         # Update positions of the bodies
38         for i in chunk:
39             positions[i] += dp[i]
40
41         barrier.wait()
```

Figura 3.4: Implementación CPython multi-hilada utilizando `threading`.

3.2. Resultados experimentales

En esta sección se presentan y analizan los resultados experimentales obtenidos. Primeramente se detalla el diseño experimental empleado (Sección 3.2.1) y posteriormente se analizan los rendimientos alcanzados (Sección 3.2.2).

3.2.1. Diseño experimental

Todas las pruebas fueron realizadas en un servidor Dell Poweredge equipado con 2×Intel Xeon Platinum 8276 de 28 núcleos (2 hilos hw por núcleo) y 256 GB de memoria RAM. El sistema operativo fue Ubuntu 20.04.2 LTS mientras que los intérpretes utilizados fueron PyPy v7.3.1 y Python v3.8.10 junto con NumPy v1.20.1.

Para la evaluación de las implementaciones, se varió la carga de trabajo (N = {256, 512, 1024, 2048, 4096, 8192}) y el número de hilos (T = {2,4,8,16}) , mientras que el número de pasos de simulación se mantuvo fijo (I=100). Cada optimización propuesta, fue aplicada y evaluada incrementalmente a partir de la versión *naive* [4]. Para evaluar el rendimiento se emplea la métrica GFLOPS (mil millones de FLOPS), utilizando la fórmula $GFLOPS = \frac{20 \times N^2 \times I}{t \times 10^9}$, donde N es el número de cuerpos, I es el número de pasos, t es el tiempo de ejecución (en segundos) y el factor 20 representa la cantidad de operaciones en punto flotante requerida por cada interacción [5].

En la Tabla 3.1 se describe la versión correspondiente a cada optimización probada. El código fuente de cada una de ellas puede verse en el repositorio de este trabajo [6].

Tabla 3.1: Versionado de las implementaciones de CPython y PyPy.

Versión	Etiqueta	NumPy	*Broadcasting*	Multi-hilado	dtype[1]
python-0.0	Naive	No	No	No	float64
python-1.0	NumPy	Sí	No	No	float64
python-2.0	NumPy+*broadcasting*	Sí	Sí	No	float64
python-3.0	Bloques	Sí	Sí	No	float64
python-4.0	`threading`	Sí	Sí	Sí	float64

[1] Tipo de dato de NumPy.

3.2.2. Rendimiento

En la Figura 3.5 se pueden observar los rendimientos obtenidos de la versión *naive* junto con la incorporación de NumPy al variar N. Se puede notar que la incorporación de arreglos de NumPy sin utilizar *broadcasting* empeoró el rendimiento 2.9× en promedio, lo que se debe a que los valores se guardan directamente en arreglos de NumPy y al accederlos se deben convertir a

[4] Cada versión previa está etiquetada como *Referencia* en todos los gráficos.

[5] Una convención ampliamente aceptada en la literatura para este problema.

[6] `https://github.com/Pastorsin/python-hpc-study/tree/main/src/versions/tesina`

objetos de Python, por lo cual se producen conversiones innecesarias. En cambio, en la versión *naive* esto no sucede debido a que los valores ya se guardan directamente como objetos de Python.

El problema anterior se solucionó al incorporar *broadcasting*, es decir, al realizar operaciones vectoriales entre los arreglos de NumPy. Esto evita las conversiones innecesarias debido a que las operaciones son llevadas a cabo internamente en el núcleo de NumPy [51]. Se puede notar que el rendimiento mejoró 10× en promedio con respecto a la versión *naive*.

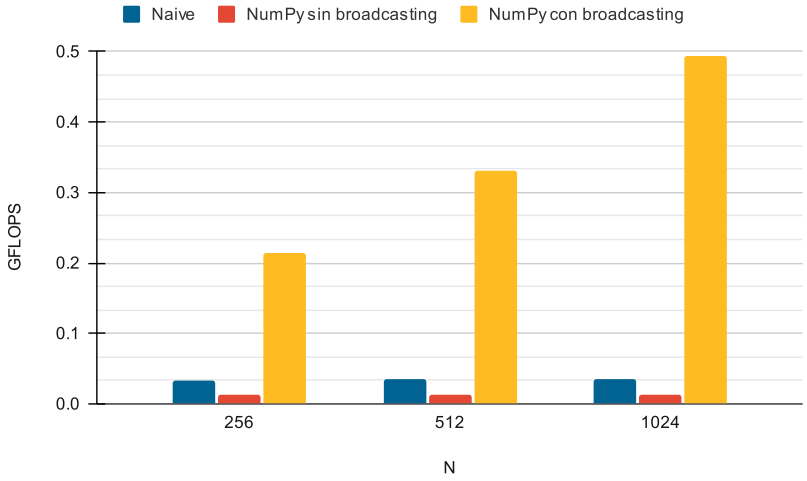

Figura 3.5: Rendimiento obtenido con la incorporación de NumPy al variar N.

En la Figura 3.6 se puede ver el rendimiento obtenido para diferentes tamaños de bloques al variar N. Sin embargo, el procesamiento por bloques no mejoró la solución, degradando el rendimiento 3.4× en promedio.

Tal como se explicó en la Sección 2.1.3.1, la utilización de `threading` para implementar la solución multi-hilada empeoró el rendimiento como se puede observar en la Figura 3.7. Este comportamiento se debe al GIL que posee CPython, el cual no permite que los hilos se ejecuten de forma paralela y a su vez provoca un *overhead* adicional, ya que los hilos adquieren y liberan el GIL constantemente. Esto último se puede apreciar en las Figuras 3.8 y 3.9, en donde se muestran capturas del comportamiento del GIL utilizando la herramienta *per4m* [53]. A su vez, se puede observar que cada hilo pasa aproximadamente el 50 % de su tiempo de ejecución bloqueado.

Por último, en la Figura 3.10 se presenta el rendimiento obtenido con CPython en comparación con PyPy. Se puede notar que el rendimiento de CPython tiende a crecer a medida que el tamaño aumenta, mientras que el rendimiento de PyPy se mantiene constante.

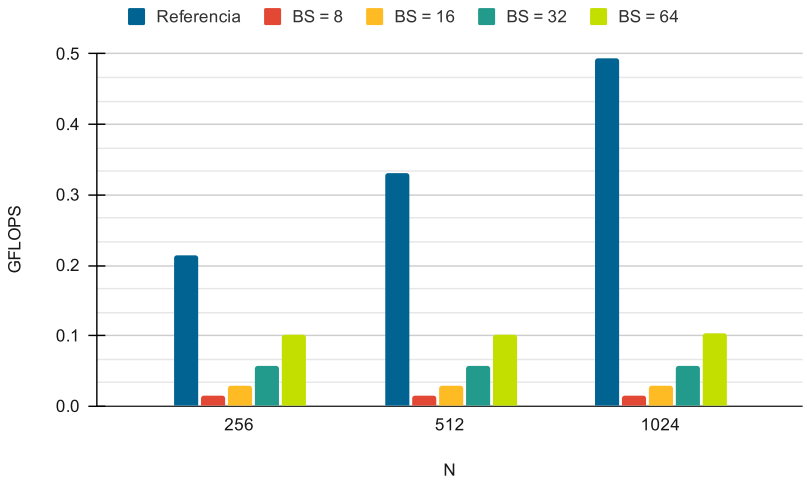

Figura 3.6: Rendimiento obtenido del procesamiento de a bloques al variar N y BS (tamaño del bloque).

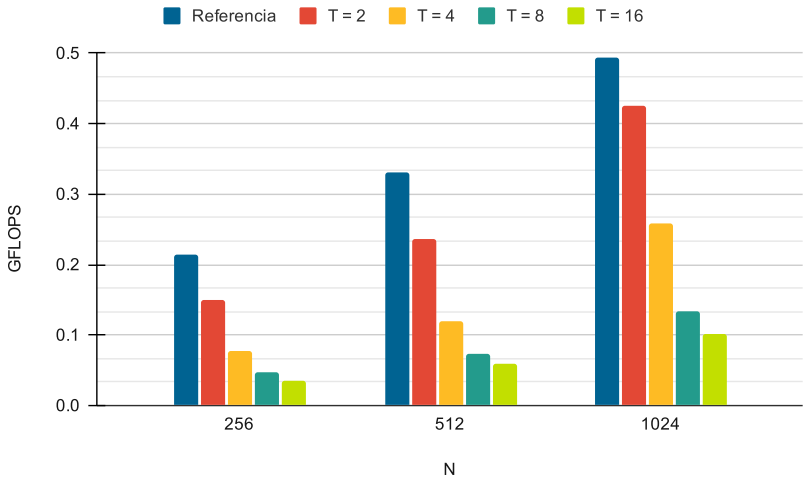

Figura 3.7: Rendimiento obtenido de la solución multi-hilada utilizando `threading` al variar N y T (número de hilos).

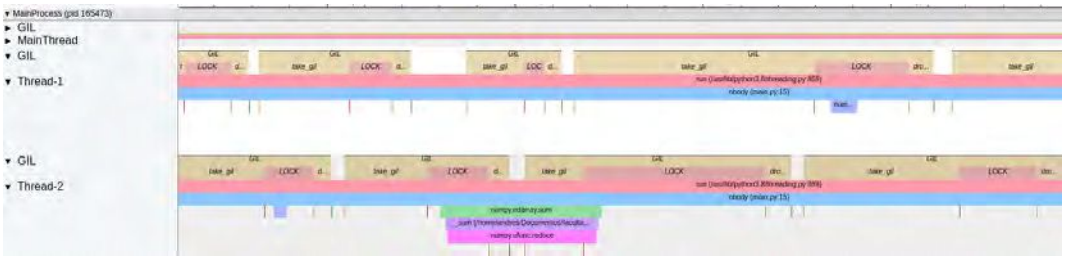

Figura 3.8: Comportamiento del GIL en la solución multi-hilada usando `threading`.

```
Summary of threads:

PID         total(us)     no gil%✅     has gil%❗     gil wait%❌
-------   -----------   ----------   -----------   ------------
165473*     234255.3         98.2          1.3           0.5
165485      181101.0          8.8         39.4          51.8
165486      179208.7          9.0         41.0          50.0

High 'no gil' is good (✅), we like low 'has gil' (❗),
 and we don't want 'gil wait' (❌). (* indicates main thread)
```

Figura 3.9: Resumen del comportamiento del GIL en la solución multi-hilada usando `threading`.

Cabe destacar que la implementación ejecutada con PyPy fue la versión *python-0.0*, ya que las demás implementaciones utilizan arreglos de NumPy y como se mencionó anteriormente PyPy es incapaz de optimizarlos (ver Sección 2.1.2). Por otro lado, tampoco se evaluó la versión multi-hilada, debido a que PyPy cuenta con el GIL al igual que CPython y los rendimientos serán similares al último [13].

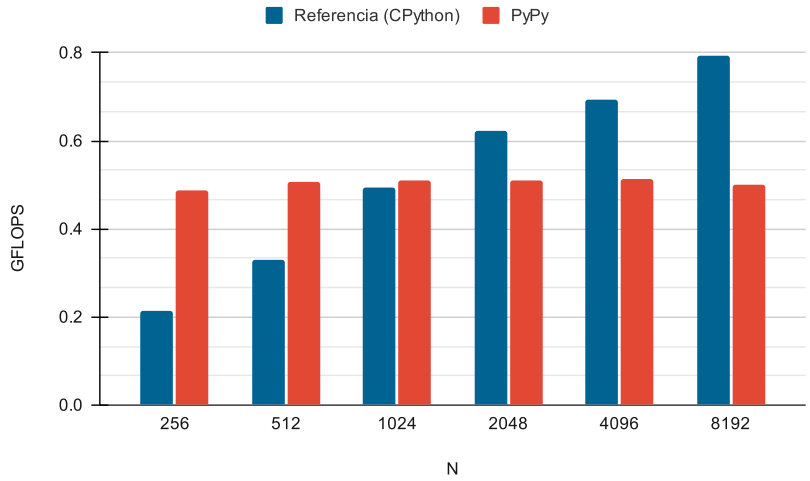

Figura 3.10: Rendimiento obtenido utilizando CPython y PyPy al variar N.

3.3. Resumen

La implementación *naive* de *N-Body* utilizando CPython, apenas alcanzó los 0.4 GFLOPS en promedio. Para mejorar la solución, se integró NumPy; sin embargo, el rendimiento empeoró 1.9× en promedio, ya que se producían conversiones innecesarias al utilizar sus arreglos. Para solucionarlo, se optó por utilizar el *broadcasting* provisto por el mismo, lo que efectivamente logró una mejora de 10× en promedio.

Posteriormente, se aplicó un procesamiento por bloques con el fin de mejorar la localidad de datos y por ende el rendimiento. Sin embargo, esto fue contraproducente y empeoró la solución 3.4× en promedio. A su vez, se probó multi-hilar la solución utilizando el módulo `threading`, pero también se degradó notablemente el rendimiento debido a que los hilos no se ejecutaron de forma paralela por las restricciones del GIL, pasando bloqueados el 50 % del tiempo total.

Por último, se probó ejecutar la solución *naive* utilizando PyPy como intérprete. En comparación a CPython, resultó superior en los tamaños más

chicos probados, pero a medida que se aumentó el tamaño del problema, PyPy se mantuvo constante y el rendimiento de CPython aumentó. Por lo tanto, la versión de CPython junto con el *broadcasting* de NumPy resultó superior en los escenarios con la mayor cantidad de cuerpos probados.

Capítulo 4

Optimización de N-Body usando Numba

En este capítulo, se presentan las diferentes implementaciones propuestas (Sección 4.1); luego, se muestran los resultados experimentales realizados (Sección 4.2); y por último, se expone un resumen del presente capítulo (Sección 4.3).

4.1. Implementaciones

En esta sección se describen las diferentes implementaciones propuestas utilizando Numba. Para ello, se siguió un enfoque incremental comenzando con una versión secuencial (denominada *naive*) y luego se exploraron diferentes optimizaciones que fueron incorporadas de acuerdo al resultado encontrado.

4.1.1. Implementación Naive

Como implementación inicial se optó por la descrita en la Sección 3.1.3, la cual utiliza las operaciones entre vectores que provee NumPy (*broadcasting*).

4.1.2. Integración de Numba

La primera versión de Numba se obtuvo al incluir el decorador `njit` en la implementación *naive* (ver líneas 1-8 de la Fig. 4.1). Se indicó que el código se compile con precisión relajada mediante el parámetro `fastmath` (línea 6), con el modelo de división de NumPy para evitar la verificación de división por cero (línea 7) [18] y con *Intel SVML*, el cuál es inferido por Numba por estar disponible en el sistema.

```
1  @njit(
2      void(
3          int64, int64,
4          double[:, ::1], double[::1], double[:, ::1], double[:, ::1],
5      ),
6      fastmath=True,
7      error_model="numpy",
8  )
9  def nbody(N, D, positions, masses, velocities, dp):
10     # For each discrete instant of time
11     for _ in range(D):
12         # For every body that experiences a force
13         for i in range(N):
14             # Newton's Law of Universal Attraction:
15             # Fi = (G * mi * mj * (qj - qi)) / (|qj - qi| ^ 3)
16             # Calculate the distances to the body i:
17             # (qj - qi)
18             dpos = np.subtract(positions, positions[i])
19             # Calculate the distance magnitudes:
20             # |qj - qi|
21             dsquared = (dpos ** 2.0).sum(axis=1) + SOFT
22             # Calculate the mass factors:
23             # (G * mi * mj)
24             gm = masses * (masses[i] * GRAVITY)
25             # Calculate the Fi denominator:
26             # 1 / |qj - qi| ^ 3
27             d32 = dsquared ** -1.5
28             # (G * mi * mj) / |qj - qi| ^ 3
29             gm_d32 = (gm * d32).reshape(N, 1)
30             # Calculate the forces:
31             # (G * mi * mj * (qj - qi)) / |qj - qi| ^ 3
32             forces = np.multiply(gm_d32, dpos).sum(axis=0)
33
34             # Calculate the acceleration vector of body i:
35             # |M = F * A|  |A = F / M|
36             acceleration = forces / masses[i]
37             # Velocity Verlet integrator
38             # V = V' + ((A * h) / 2) ; h = 1
39             velocities[i] += acceleration * DT / 2.0
40             # Save the new position
41             dp[i] = velocities[i] * DT
42
43         # Update positions of the body i
44         for i in range(N):
45             positions[i] += dp[i]
```

Figura 4.1: Implementación Numba *naive*

4.1.3. Multi-hilado

Para introducir paralelismo a nivel de hilos, se utilizó la sentencia `prange`. Para ello, fue necesario antes separar el bucle que itera sobre los cuerpos (línea 13 de la Fig. 4.1) junto con el bucle que actualiza las posiciones (línea 44 de la Fig. 4.1) en dos funciones. La primera función paralela se encarga de computar la ley de atracción gravitacional de Newton y la integración de Verlet (líneas 1-37 de la Fig. 4.2), mientras que la otra función actualiza la posición de los

cuerpos (líneas 40-49 de la Fig. 4.2).

```
1  @njit(
2      void(int64, double[:, ::1], double[::1], double[:, ::1], double[:, ::1]),
3      fastmath=True,
4      parallel=True,
5      error_model="numpy",
6  )
7  def calculate_positions(N, positions, masses, velocities, dp):
8      # For every body that experiences a force
9      for i in prange(N):
10         # Newton's Law of Universal Attraction:
11         # Fi = (G * mi * mj * (qj - qi)) / (|qj - qi| ^ 3)
12         # Calculate the distances to the body i:
13         # (qj - qi)
14         dpos = np.subtract(positions, positions[i])
15         # Calculate the distance magnitudes:
16         # |qj - qi|
17         dsquared = (dpos ** 2.0).sum(axis=1) + SOFT
18         # Calculate the mass factors:
19         # (G * mi * mj)
20         gm = masses * (masses[i] * GRAVITY)
21         # Calculate the Fi denominator:
22         # 1 / |qj - qi| ^ 3
23         d32 = dsquared ** -1.5
24         # (G * mi * mj) / |qj - qi| ^ 3
25         gm_d32 = (gm * d32).reshape(N, 1)
26         # Calculate the forces:
27         # (G * mi * mj * (qj - qi)) / |qj - qi| ^ 3
28         forces = np.multiply(gm_d32, dpos).sum(axis=0)
29
30         # Calculate the acceleration vector of body i:
31         # |M = F * A|  |A = F / M|
32         acceleration = forces / masses[i]
33         # Velocity Verlet integrator
34         # V = V' + ((A * h) / 2) ; h = 1
35         velocities[i] += acceleration * DT / 2.0
36         # Save the differential position of body i
37         dp[i] = velocities[i] * DT
38
39
40 @njit(
41     void(int64, double[:, ::1], double[:, ::1]),
42     fastmath=True,
43     parallel=True,
44     error_model="numpy",
45 )
46 def update_positions(N, positions, dp):
47     # Update positions of the bodies
48     for i in prange(N):
49         positions[i] += dp[i]
```

Figura 4.2: Implementación Numba paralela con *broadcasting*.

4.1.4. Arreglos con tipos de datos simples

Se reemplazaron las operaciones vectoriales de NumPy (*broadcasting*) por operaciones numéricas, y las estructuras bidimensionales fueron sustituidas por unidimensionales para ayudar a Numba a la hora de autovectorizar el código (ver Figuras 4.4 y 4.5).

4.1.5. Operaciones matemáticas

Se propone evaluar alternativas para el cálculo del denominador de la ley de atracción universal de Newton por:

1. Calcular la potencia positiva y luego dividir.
2. Multiplicar por el inverso multiplicativo, calculando la potencia positiva previamente.

Adicionalmente, se ponen a prueba las siguientes funciones para calcular las potencias:

1. Función `pow` del módulo `math` de Python
2. Función `power` que provee NumPy.

4.1.6. Vectorización

Como se indicó en la Sección 2.1.5, Numba delega la autovectorización en LLVM. Aun así, se indicaron los flags `avx512f, avx512dq, avx512cd, avx512bw, avx512vl` para favorecer el uso de esta clase particular de instrucciones considerando la plataforma objetivo.

Para esto último, fue necesario especificar el archivo `.numba_config.yaml` con el contenido de la Figura 4.3.

```
ENABLE_AVX: 1
CPU_FEATURES: +avx512f,+avx512dq,+avx512cd,+avx512bw,+avx512vl
```

Figura 4.3: Especificación de instrucciones AVX-512 con Numba.

```python
@njit(
    void(
        int64,
        double[::1], double[::1], double[::1],
        double[::1],
        double[::1], double[::1], double[::1],
        double[::1], double[::1], double[::1],
    ),
    fastmath=True, parallel=True, error_model="numpy",
)
def calculate_positions(
    N,
    positions_x, positions_y, positions_z,
    masses,
    velocities_x, velocities_y, velocities_z,
    dp_x, dp_y, dp_z,
):
    # For every body that experiences a force
    for i in prange(N):
        # Initialize the force of the body i
        forces_x = forces_y = forces_z = 0.0
        # Calculate gravitational force to the rest of the bodies
        # Newton's Law of Universal Attraction:
        # Fi = (G * mi * mj * (qj - qi)) / (|qj - qi| ^ 3)
        for j in range(N):
            # Calculate the distance to the body i:
            # (qj - qi)
            dpos_x = positions_x[j] - positions_x[i]
            dpos_y = positions_y[j] - positions_y[i]
            dpos_z = positions_z[j] - positions_z[i]
            # Calculate the distance magnitude:
            # |qj - qi|
            dsquared = (dpos_x ** 2.0) + (dpos_y ** 2.0) + (dpos_z ** 2.0) + SOFT
            # Calculate the mass factor:
            # (G * mi * mj)
            gm = GRAVITY * masses[j] * masses[i]
            # Calculate the Fi denominator:
            # 1 / |qj - qi| ^ 3
            d32 = dsquared ** -1.5
            # Calculate the force:
            # (G * mi * mj * (qj - qi)) / |qj - qi| ^ 3
            forces_x += gm * d32 * dpos_x
            forces_y += gm * d32 * dpos_y
            forces_z += gm * d32 * dpos_z

        # Calculate the acceleration vector of body i:
        # |M = F * A|  |A = F / M|
        aceleration_x = forces_x / masses[i]
        aceleration_y = forces_y / masses[i]
        aceleration_z = forces_z / masses[i]
        # Velocity Verlet integrator
        # V = V' + ((A * h) / 2) ; h = 1
        velocities_x[i] += aceleration_x * DT / 2.0
        velocities_y[i] += aceleration_y * DT / 2.0
        velocities_z[i] += aceleration_z * DT / 2.0
        # Save the differential position of body i
        dp_x[i] = velocities_x[i] * DT
        dp_y[i] = velocities_y[i] * DT
        dp_z[i] = velocities_z[i] * DT
```

Figura 4.4: Implementación Numba sin *broadcasting* de la función que calcula las posiciones de los cuerpos.

```
1  @njit(
2      void(
3          int64,
4          double[::1], double[::1], double[::1],
5          double[::1], double[::1], double[::1],
6      ),
7      fastmath=True,
8      parallel=True,
9      error_model="numpy",
10 )
11 def update_positions(
12     N,
13     positions_x, positions_y, positions_z,
14     dp_x, dp_y, dp_z,
15 ):
16     # Update positions of the bodies
17     for i in prange(N):
18         positions_x[i] += dp_x[i]
19         positions_y[i] += dp_y[i]
20         positions_z[i] += dp_z[i]
```

Figura 4.5: Implementación Numba sin *broadcasting* de la función que actualiza las posiciones de los cuerpos.

4.1.7. Localidad de datos

Se desarrolló una versión que emplea procesamiento por bloques para favorecer la localidad de datos, de manera similar a la descrita en la Sección 3.1.4. Para ello, el bucle de la línea 19 de la Figura 4.4 iterará sobre bloques de cuerpos, y en otros dos bucles más internos, se calculará la fuerza de atracción gravitacional de Newton (línea 20 de la Fig. 4.6) y la integración de Verlet (línea 43 de la Fig. 4.6) respectivamente.

4.1.8. Threading layer

Se varió la API de hilos a través de las *threading layers* que utiliza Numba para traducir las regiones paralelas. Para ello, se rotaron las opciones: *default*, *workqueue*, *omp* (OpenMP) y `threading`. Para las tres primeras no se tuvo que modificar el código fuente, ya que Numba se encarga de traducir el bloque `prange` a la API seleccionada. Sin embargo, no sucedió lo mismo para el caso de `threading`, ya que como se puede ver en las Figuras 4.7 y 4.8 se tuvo que codificar la distribución de hilos junto con la especificación del parámetro `nogil=True` para desactivar el GIL.

Cabe destacar que la opción *tbb* no fue utilizada debido a que no se encontraba disponible en el equipo de soporte.

```python
def calculate_positions(
    N,
    positions_x, positions_y, positions_z,
    masses,
    velocities_x, velocities_y, velocities_z,
    dp_x, dp_y, dp_z,
):
    # Init the force
    forces_x = np.zeros(N, dtype=DATATYPE)
    forces_y = np.zeros(N, dtype=DATATYPE)
    forces_z = np.zeros(N, dtype=DATATYPE)

    for b in prange(BLOCKS):
        first = b * BLOCKSIZE
        last = first + BLOCKSIZE
        # Calculate gravitational force to the rest of the bodies
        # Newton's Law of Universal Attraction:
        # Fi = (G * mi * mj * (qj - qi)) / (|qj - qi| ^ 3)
        for j in range(N):
            for i in range(first, last):
                # Calculate the distance to the body i:
                # (qj - qi)
                dpos_x = positions_x[j] - positions_x[i]
                dpos_y = positions_y[j] - positions_y[i]
                dpos_z = positions_z[j] - positions_z[i]
                # Calculate the distance magnitude:
                # |qj - qi|
                dsquared = (
                    (dpos_x ** 2.0) + (dpos_y ** 2.0) + (dpos_z ** 2.0) + SOFT
                )
                # Calculate the mass factor:
                # (G * mi * mj)
                gm = GRAVITY * masses[j] * masses[i]
                # Calculate the Fi denominator:
                # 1 / |qj - qi| ^ 3
                d32 = dsquared ** -1.5
                # Calculate the force:
                # (G * mi * mj * (qj - qi)) / |qj - qi| ^ 3
                forces_x[i] += gm * dpos_x * d32
                forces_y[i] += gm * dpos_y * d32
                forces_z[i] += gm * dpos_z * d32

        for i in range(first, last):
            # Calculate the acceleration vector of body i:
            # |M = F * A|  |A = F / M|
            # f_i = i - first
            aceleration_x = forces_x[i] / masses[i]
            aceleration_y = forces_y[i] / masses[i]
            aceleration_z = forces_z[i] / masses[i]
            # Velocity Verlet integrator
            # V = V' + ((A * h) / 2) ; h = 1
            velocities_x[i] += aceleration_x * DT / 2.0
            velocities_y[i] += aceleration_y * DT / 2.0
            velocities_z[i] += aceleration_z * DT / 2.0
            # Save the differential position of body i
            dp_x[i] = velocities_x[i] * DT
            dp_y[i] = velocities_y[i] * DT
            dp_z[i] = velocities_z[i] * DT
```

Figura 4.6: Implementación Numba utilizando bloques.

```python
@njit(
    void(
        int64, int64[::1],
        float32[::1], float32[::1], float32[::1],
        float32[::1],
        float32[::1], float32[::1], float32[::1],
        float32[::1], float32[::1], float32[::1],
    ),
    nogil=True, fastmath=True, error_model="numpy",
)
def calculate_positions(
    N, chunk,
    positions_x, positions_y, positions_z,
    masses,
    velocities_x, velocities_y, velocities_z,
    dp_x, dp_y, dp_z,
):
    # For every body that experiences a force
    for i in chunk:
        # Initialize the force of the body i
        forces_x = forces_y = forces_z = 0.0
        # Calculate gravitational force to the rest of the bodies
        for j in range(N):
            # Calculate the distance to the body i:
            dpos_x = positions_x[j] - positions_x[i]
            dpos_y = positions_y[j] - positions_y[i]
            dpos_z = positions_z[j] - positions_z[i]
            # Calculate the distance magnitude:
            dsquared = (dpos_x ** 2.0) + (dpos_y ** 2.0) + (dpos_z ** 2.0) + SOFT
            # Calculate the mass factor:
            gm = GRAVITY * masses[j] * masses[i]
            # Calculate the Fi denominator:
            d32 = dsquared ** -1.5
            # Calculate the force:
            forces_x += gm * d32 * dpos_x
            forces_y += gm * d32 * dpos_y
            forces_z += gm * d32 * dpos_z

        # Calculate the acceleration vector of body i:
        aceleration_x = forces_x / masses[i]
        aceleration_y = forces_y / masses[i]
        aceleration_z = forces_z / masses[i]
        # Velocity Verlet integrator
        velocities_x[i] += aceleration_x * DT / 2.0
        velocities_y[i] += aceleration_y * DT / 2.0
        velocities_z[i] += aceleration_z * DT / 2.0
        # Save the differential position of body i
        dp_x[i] = velocities_x[i] * DT
        dp_y[i] = velocities_y[i] * DT
        dp_z[i] = velocities_z[i] * DT
```

Figura 4.7: Implementación Numba de la función que calcula las posiciones de los cuerpos utilizando `threading`.

```python
@njit(
    void(
        int64[::1],
        float32[::1], float32[::1], float32[::1],
        float32[::1], float32[::1], float32[::1],
    ),
    nogil=True,
    fastmath=True,
)
def update_positions(
    chunk,
    positions_x, positions_y, positions_z,
    dp_x, dp_y, dp_z,
):
    # Update positions of the bodies
    for i in chunk:
        positions_x[i] += dp_x[i]
        positions_y[i] += dp_y[i]
        positions_z[i] += dp_z[i]
```

Figura 4.8: Implementación Numba de la función que actualiza las posiciones de los cuerpos utilizando `threading`.

```python
def nbody(
    N, D, chunk,
    positions_x, positions_y, positions_z,
    masses,
    velocities_x, velocities_y, velocities_z,
    dp_x, dp_y, dp_z,
    barrier,
):
    # For each discrete instant of time
    for _ in range(D):
        calculate_positions(
            N, chunk,
            positions_x, positions_y, positions_z,
            masses,
            velocities_x, velocities_y, velocities_z,
            dp_x, dp_y, dp_z,
        )

        barrier.wait()

        update_positions(
            chunk,
            positions_x, positions_y, positions_z,
            dp_x, dp_y, dp_z,
        )

        barrier.wait()
```

Figura 4.9: Implementación Numba utilizando `threading`.

4.2. Resultados experimentales

En esta sección se presentan y analizan los resultados experimentales obtenidos. Para ello, inicialmente se especifica el diseño experimental empleado (Sección 4.2.1) y luego se analizan los rendimientos obtenidos (Sección 4.2.2).

4.2.1. Diseño experimental

Todas las pruebas fueron realizadas utilizando el servidor descrito en la Sección 3.2.1 junto con Numba v0.52.0. Se mantuvo la configuración de I, mientras que se varió el número de hilos ($T = \{1, 28, 56, 112\}$) y la carga de trabajo ($N = \{4096, 8192, 16384, 32768, 65536, 131072, 262144, 524288\}$).

Para evaluar el rendimiento se utilizó la métrica GFLOPS definida en dicha sección.

En la Tabla 4.1 se describe la versión correspondiente a cada optimización probada, junto con las opciones de compilación y el tipo de dato utilizado. El código fuente de cada una de ellas puede verse en el repositorio de este trabajo [1].

Tabla 4.1: Versionado de las implementaciones de Numba.

Versión	Etiqueta	njit	fastmath	SVML	AVX-512	parallel	dtype[1]
python-2.0	Naive	No	No	No	No	No	float64
numba-0.0	Integración de Numba	Sí	Sí	Sí	No	No	float64
numba-1.0	Paralela	Sí	Sí	Sí	No	Sí	float64
numba-2.0	Sin *broadcasting*	Sí	Sí	Sí	No	Sí	float64
numba-3.0	1º variante matemática	Sí	Sí	Sí	No	Sí	float64
numba-3.1	2º variante matemática	Sí	Sí	Sí	No	Sí	float64
numba-3.2	`math.pow`	Sí	Sí	Sí	No	Sí	float64
numba-3.3	`numpy.power`	Sí	Sí	Sí	No	Sí	float64
numba-3.4	AVX-512	Sí	Sí	Sí	Sí	Sí	float64
numba-4.0	Bloques	Sí	Sí	Sí	Sí	Sí	float64
numba-5.0	float64 estricta	Sí	No	Sí	Sí	Sí	float64
numba-5.1	float32 relajada	Sí	Sí	Sí	Sí	Sí	float32
numba-5.2	float32 estricta	Sí	No	Sí	Sí	Sí	float32
numba-6.0	*workqueue*	Sí	Sí	Sí	Sí	Sí	float32
numba-6.1	*omp*	Sí	Sí	Sí	Sí	Sí	float32
numba-6.2	`threading`	Sí	Sí	Sí	Sí	Sí	float32

[1] Tipo de dato de NumPy.

[1] `https://github.com/Pastorsin/python-hpc-study/tree/main/src/versions/tesina`

4.2.2. Rendimiento

En la Fig. 4.10 se pueden observar los rendimientos al activar las opciones de compilación y aplicar multi-hilado al variar N. Aunque las opciones de compilación de Numba (`njit+fastmath+svml`) no tienen incidencia prácticamente en el rendimiento de esta versión, sí se aprecia una mejora importante al utilizar hilos para computar el problema. En particular, se puede notar una mejora en promedio de 33× y 38× para 56 y 112 hilos, respectivamente.

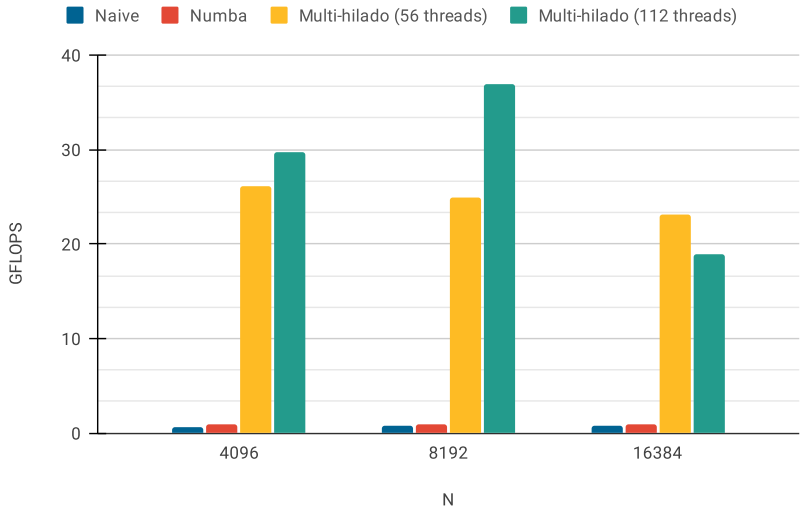

Figura 4.10: Rendimientos obtenidos para opciones de compilación y multi-hilado al variar N.

En la Fig. 4.11 se puede apreciar la mejora significativa que produce emplear arreglos con tipos de datos simples en lugar de compuestos (un promedio de 41× para el caso de 112 hilos). Si bien el segundo simplifica la codificación, también implica organizar los datos en forma de arreglo de estructuras (en inglés, *array of structures*), lo que impone limitaciones al aprovechamiento de las capacidades SIMD del procesador [54]. Adicionalmente, también se puede notar que el uso de *hyper-threading* reporta una mejora de aproximadamente 78 % en este caso.

De la Fig. 4.12 se puede observar que prácticamente no hay cambios en el rendimiento por el uso de los diferentes cálculos matemáticos y funciones de potencia que fueron descritos en la Sección 4.1.5. Esto se debe a que, independientemente de cuál opción se utilice, el código máquina resultante es siempre el mismo. Un caso similar se da al indicar explícitamente que utilice instrucciones AVX-512. Tal como se mencionó en la sección 2.1.5, Numba intenta

autovectorizar el código a través de LLVM. Al observar el código máquina, se notó que las instrucciones generadas ya hacían uso de estas extensiones.

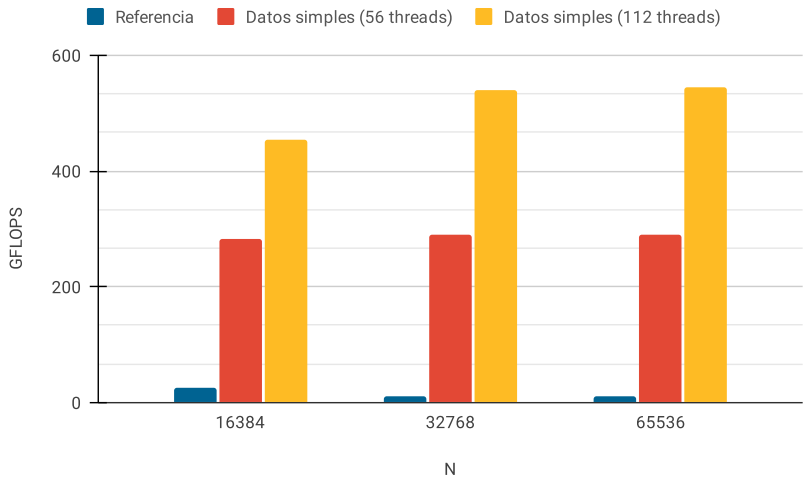

Figura 4.11: Rendimientos obtenidos de la optimización paralela al variar N.

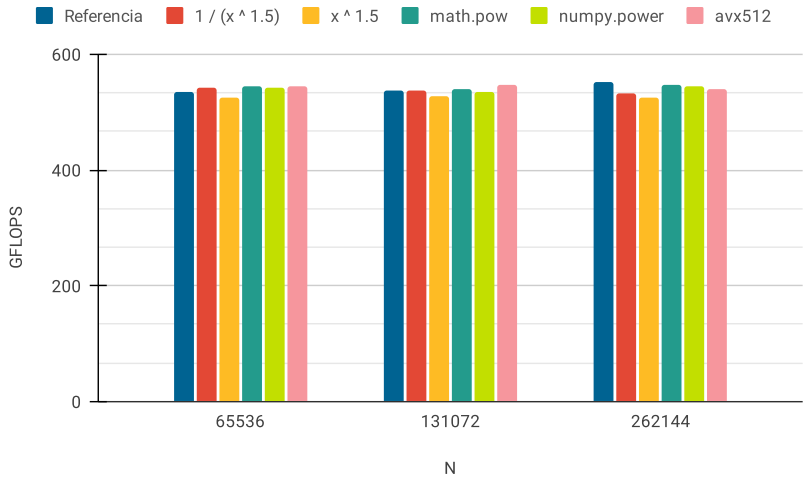

Figura 4.12: Rendimientos obtenidos utilizando el uso de diferentes cálculos matemáticos, funciones de potencia e instrucciones AVX512 al variar N.

El procesamiento por bloques descrito en la sección 4.1.7 no mejoró el rendimiento de la solución, tal como se puede observar en la Fig. 4.13. La pérdida de rendimiento se relaciona con que esta reorganización del cómputo

produce fallos en LLVM a la hora de autovectorizar. Lamentablemente, debido a que Numba no ofrece primitivas para indicar la utilización de instrucciones SIMD de forma explícita, no hay manera de enmendarlo.

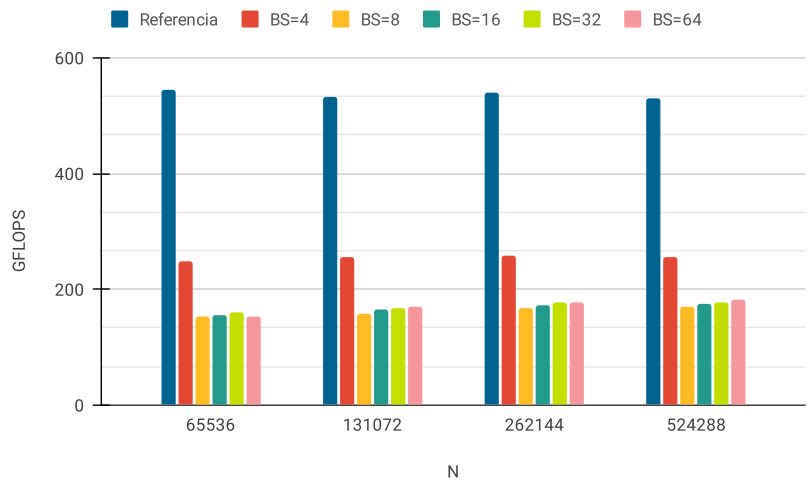

Figura 4.13: Rendimiento obtenido del procesamiento de a bloques al variar N.

En la Fig. 4.14 se muestran los rendimientos obtenidos para la relajación de precisión al variar el tipo de dato y la carga de trabajo (N). Se puede observar que el uso del tipo de datos `float32` (en lugar de `float64`) conlleva a una mejora de hasta 2.8× GFLOPS, a costo de una reducción en la precisión del resultado. En forma similar, se puede notar claramente la importante aceleración que produce relajar la precisión en ambos tipos de datos: `float32` (17.2× en promedio) y `float64` (11.4× en promedio). En particular el pico de rendimiento es de 1524/536 GFLOPS en simple/doble precisión. Resulta importante mencionar que esta versión logra una aceleración de 687× en comparación a la implementación *naive* (caso `float64`).

Por último, en la Figura 4.15 se puede observar que al utilizar 112 hilos, OpenMP y la *threading layer* predeterminada que provee Numba, superan por un promedio de 9.3 % a las demás. A su vez, se puede notar que el uso de `threading` está por debajo de todas las *threading layers* probadas. Esto ocurre debido al *overhead* generado por la utilización de objetos de Python para sincronizar los hilos, mientras que en las demás *threading layers* no sucede lo mismo ya que la sincronización se realiza sobre su propia API evitando el uso de los mismos.

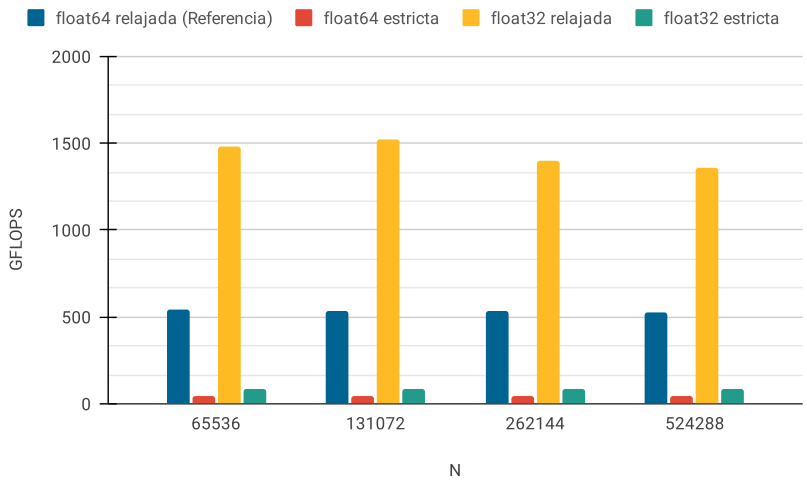

Figura 4.14: Rendimiento obtenido para la relajación de precisión al variar el tipo de dato y N.

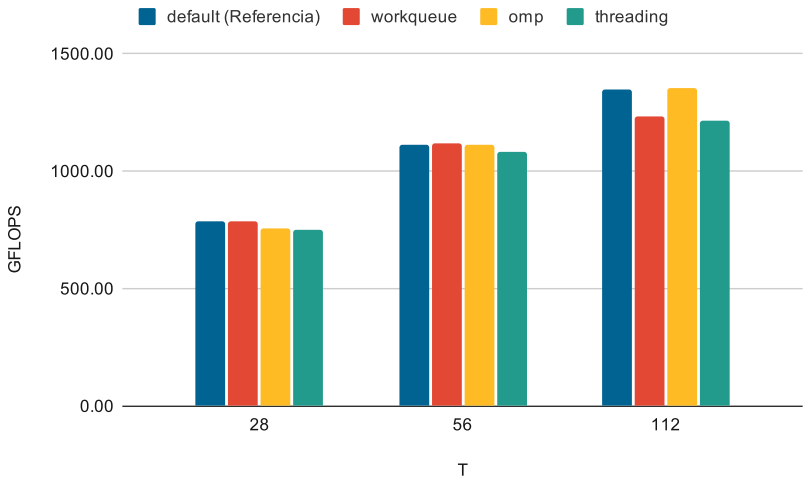

Figura 4.15: Rendimiento obtenido con las diferentes *threading layers* de Numba al variar T y fijar $N = 524288$.

4.3. Resumen

En este capítulo, se comenzó a optimizar *N-Body* partiendo de la implementación naive de CPython con *broadcasting*. A continuación se describen las optimizaciones introducidas y las principales conclusiones del análisis experi-

mental:

- La integración de Numba, utilizando el decorador `njit` junto con instrucciones *Intel SVML* y el parámetro de compilación `fastmath`; no tuvo incidencia en el rendimiento.
- La incorporación de paralelismo a nivel de hilos logró una mejora en promedio de 38× para 112 hilos, gracias a la ejecución paralela de los mismos.
- El reemplazo de cada operación vectorial por operaciones tradicionales entre arreglos, es decir, evitar el uso de *broadcasting*; logró una aceleración en promedio de 41× debido a un mejor aprovechamiento de las capacidades SIMD del procesador.
- La variación de diferentes cálculos matemáticos, instrucciones AVX-512, y funciones de potencia no reportaron mejoras de rendimiento significativas, ya que en todos los casos el código máquina resultante fue similar.
- La solución por bloques fue contraproducente y degradó el rendimiento con todos los tamaños de bloques probados, debido a fallos de LLVM a la hora de auto-vectorizar.
- La utilización del tipo de dato `float32` logró una mejora de hasta 2.8× GFLOPS, a costo de una menor representación en el resultado final.
- La relajación de precisión aceleró la solución 17.2× y 11.4× para los tipos de datos `float32` y `float64` respectivamente.
- La *threading layer* predeterminada fue superior al resto en términos de rendimiento. En particular, se notó que el rendimiento al utilizar `threading` fue degradado, esto se debe a la utilización de objetos de Python para realizar la sincronización entre los hilos.

Por último, cabe destacar que el pico de rendimiento fue de 1524 y 536 GFLOPS en simple y doble precisión respectivamente. Además, resulta importante mencionar que la versión final de Numba logró una aceleración de 687× en comparación a la implementación *naive* al utilizar `float64` como tipo de dato.

Capítulo 5

Optimización de N-Body usando Cython

En este capítulo, se presentan las diferentes implementaciones propuestas (Sección 5.1); luego, se muestran los resultados experimentales obtenidos (Sección 5.2); y por último, se expone un resumen del presente capítulo (Sección 5.3).

5.1. Implementaciones

En esta sección se describen las diferentes implementaciones propuestas utilizando Cython. Para ello, se siguió un enfoque incremental comenzando con una versión secuencial (denominada *naive*) y luego se exploraron diferentes optimizaciones que fueron incorporadas de acuerdo al resultado encontrado.

5.1.1. Implementación Naive

Como implementación inicial (*naive*) se optó por la descrita en la Sección 3.1.2. En esta se utilizan como estructura de datos a diferentes arreglos de NumPy, lo que permite un manejo más flexible de la memoria, en particular a lo que refiere a organización y tipos de datos.

5.1.2. Integración de Cython

Se mantuvo el mismo código de la implementación *naive* (ver Sección 5.1.1), pero se la compiló utilizando Cython. Para esto, simplemente se cambió la extensión del código `.py` por `.pyx`.

5.1.3. Tipado explícito

Tal como se puede ver en la Figura 5.1 se indicaron explícitamente los tipos de datos que provee Cython, con el fin de disminuir la interacción con la API de CPython.

Inicialmente, para disminuir verificaciones innecesarias en tiempo de ejecución, se indican las siguientes directivas de compilación (líneas 1-4) [55]:

- *boundcheck* (línea 1) evita verificaciones errores de índice sobre los arreglos.
- *wraparound* (línea 2) evita que los arreglos se pueden indexar en relación con el final. Por ejemplo, en Python si A es un arreglo, con la sentencia `A[-1]` podremos obtener su último elemento.
- *nonecheck* (línea 3) evita verificaciones por variables que pueden llegar a tomar el valor *None*.
- *cdivision* (línea 4) realiza la división a través de C evitando la API de CPython.

En la línea 5 se indica un tipo de función híbrida a través de la sentencia `cpdef`, lo que permite que la función sea importada desde otras aplicaciones desarrolladas en Python (ver Sección 2.1.6). Luego, en las líneas 6-10 se especifican los tipos de datos de Cython, que posteriormente serán transpilados a tipos de datos de C (ver Sección 2.1.6). En particular, los arreglos se especifican con el tipo de dato `double[::1]`, el cual asegura que los argumentos recibidos sean arreglos de NumPy contiguos en memoria [56]. Finalmente en las líneas 12-16 se declaran los tipos de datos pertenecientes a las variables locales a través de la sentencia `cdef`.

Para notar la diferencia entre esta implementación y la anterior (ver Sección 5.1.2), se utiliza la herramienta *annotate* provista por Cython para generar un reporte que muestra la interacción del código con la API de CPython [55]. Gracias a esto, se puede notar que en la implementación anterior (ver Fig. 5.2) la interacción es alta, ya que se emplean los tipos de datos de Python. En sentido opuesto, al incorporar los tipos de datos de Cython, se puede apreciar que la interacción es casi nula (ver Fig. 5.3), lo cual probablemente causará un menor *overhead* en la ejecución.

5.1.4. Multi-hilado

En esta versión se introduce paralelismo a nivel de hilos a través de la sentencia `prange` que provee Cython. Para ello, se reemplazaron las sentencias

```
@boundscheck(False)
@wraparound(False)
@nonecheck(False)
@cdivision(True)
cpdef void nbody(
    int N, int D,
    double[::1] positions_x, double[::1] positions_y, double[::1] positions_z,
    double[::1] masses,
    double[::1] velocities_x, double[::1] velocities_y, double[::1] velocities_z,
    double[::1] dp_x, double[::1] dp_y, double[::1] dp_z,
):
    cdef double forces_x, forces_y, forces_z
    cdef double aceleration_x, aceleration_y, aceleration_z
    cdef double dpos_x, dpos_y, dpos_z
    cdef double dsquared, gm, d32
    cdef int i, j

    for _ in range(D):
        # For every body that experiences a force
        for i in range(N):
            # Initialize the force of the body i
            forces_x = forces_y = forces_z = 0.0

            # Calculate gravitational force to the rest of the bodies
            # Newton's Law of Universal Attraction:
            # Fi = (G * mi * mj * (qj - qi)) / (|qj - qi| ^ 3)
            for j in range(N):
                # Calculate the distance to the body i:
                # (qj - qi)
                dpos_x = positions_x[j] - positions_x[i]
                dpos_y = positions_y[j] - positions_y[i]
                dpos_z = positions_z[j] - positions_z[i]
                # Calculate the distance magnitude:
                # |qj - qi|
                dsquared = (dpos_x ** 2.0) + (dpos_y ** 2.0) + (dpos_z ** 2.0) + SOFT
                # Calculate the mass factor:
                # (G * mi * mj)
                gm = GRAVITY * masses[j] * masses[i]
                # Calculate the Fi denominator:
                # 1 / |qj - qi| ^ 3
                d32 = dsquared ** -1.5
                # Calculate the force:
                # (G * mi * mj * (qj - qi)) / |qj - qi| ^ 3
                forces_x += gm * d32 * dpos_x
                forces_y += gm * d32 * dpos_y
                forces_z += gm * d32 * dpos_z

            # Calculate the acceleration vector of body i:
            # |M = F * A|  |A = F / M|
            aceleration_x = forces_x / masses[i]
            aceleration_y = forces_y / masses[i]
            aceleration_z = forces_z / masses[i]
            # Velocity Verlet integrator
            # V = V' + ((A * h) / 2) ; h = 1
            velocities_x[i] += aceleration_x * DT / 2.0
            velocities_y[i] += aceleration_y * DT / 2.0
            velocities_z[i] += aceleration_z * DT / 2.0
            # Save the differential position of body i
            dp_x[i] = velocities_x[i] * DT
            dp_y[i] = velocities_y[i] * DT
            dp_z[i] = velocities_z[i] * DT

        # Update positions of the bodies
        for i in range(N):
            positions_x[i] += dp_x[i]
            positions_y[i] += dp_y[i]
            positions_z[i] += dp_z[i]
```

Figura 5.1: Implementación Cython con tipado explícito.

```
+08: def nbody(
 09:     N, D,
 10:     positions_x, positions_y, positions_z,
 11:     masses,
 12:     velocities_x, velocities_y, velocities_z,
 13:     dp_x, dp_y, dp_z,
 14: ):
 15:
+16:     for _ in range(D):
 17:
 18:         # For every body that experiences a force
+19:         for i in range(N):
 20:             # Initialize the force of the body i
+21:             forces_x = 0.0
+22:             forces_y = 0.0
+23:             forces_z = 0.0
 24:
 25:             # Calculate gravitational force to the rest of the bodies
 26:             # Newton's Law of Universal Attraction:
 27:             # Fi = (G * mi * mj * (qj - qi)) / (|qj - qi| ^ 3)
+28:             for j in range(N):
 29:                 # Calculate the distance to the body i:
 30:                 # (qj - qi)
+31:                 dpos_x = positions_x[j] - positions_x[i]
+32:                 dpos_y = positions_y[j] - positions_y[i]
+33:                 dpos_z = positions_z[j] - positions_z[i]
 34:                 # Calculate the distance magnitude:
 35:                 # |qj - qi|
+36:                 dsquared = (dpos_x ** 2.0) + (dpos_y ** 2.0) + (dpos_z ** 2.0) + SOFT
 37:                 # Calculate the mass factor:
 38:                 # (G * mi * mj)
+39:                 gm = GRAVITY * masses[j] * masses[i]
 40:                 # Calculate the Fi denominator:
 41:                 # 1 / |qj - qi| ^ 3
+42:                 d32 = dsquared ** -1.5
 43:                 # Calculate the force:
 44:                 # (G * mi * mj * (qj - qi)) / |qj - qi| ^ 3
+45:                 forces_x += gm * d32 * dpos_x
+46:                 forces_y += gm * d32 * dpos_y
+47:                 forces_z += gm * d32 * dpos_z
 48:
 49:             # Calculate the acceleration vector of body i:
 50:             # |M = F * A|  |A = F / M|
+51:             aceleration_x = forces_x / masses[i]
+52:             aceleration_y = forces_y / masses[i]
+53:             aceleration_z = forces_z / masses[i]
 54:             # Velocity Verlet integrator
 55:             # V = V' + ((A * h) / 2) ; h = 1
+56:             velocities_x[i] += aceleration_x * DT / 2.0
+57:             velocities_y[i] += aceleration_y * DT / 2.0
+58:             velocities_z[i] += aceleration_z * DT / 2.0
 59:             # Save the differential position of body i
+60:             dp_x[i] = velocities_x[i] * DT
+61:             dp_y[i] = velocities_y[i] * DT
+62:             dp_z[i] = velocities_z[i] * DT
 63:
 64:         # Update positions of the bodies
+65:         for i in range(N):
+66:             positions_x[i] += dp_x[i]
+67:             positions_y[i] += dp_y[i]
+68:             positions_z[i] += dp_z[i]
```

Figura 5.2: Anotación de objetos de Python sobre la implementación en Cython sin explicitar tipos de datos.

Figura 5.3: Anotación de objetos de Python sobre la implementación en Cython explicitando los tipos de datos.

`range` de las líneas 20 y 64 (ver Fig. 5.1) por sentencias `prange`. Particularmente, se les indicó utilizar la política *static* como *schedule* para distribuir equitativamente la carga de trabajo entre los hilos considerando la regularidad del cómputo. Adicionalmente, se desactivó el GIL a través del argumento *nogil* para permitir que los mismos se ejecuten de forma paralela (ver Fig. 5.4).

Por último, cabe destacar que la sentencia `prange` es transpilada a una directiva `parallel for` de OpenMP [57]. Por lo tanto, la misma posee una barrera implícita que permite sincronizar a los hilos para cumplir con las dependencias de datos descritas en el Algoritmo 1.

5.1.5. Operaciones matemáticas

Se optó por evaluar las mismas alternativas para el cálculo del denominador de la ley de atracción universal de Newton que las descritas en la Sección 4.1.5. En particular no se variaron las funciones de potencia para evitar la interacción con la API de CPython.

5.1.6. Localidad de datos

Con la finalidad de aprovechar la memoria caché de una manera más adecuada, se aplicó una optimización que itera los cuerpos de a bloques, similar a la descrita en la Sección 4.1.7.

Tal como se puede apreciar en la Figura 5.5, el bucle de la línea 20 de la Fig. 5.4 se desdobla en otros dos bucles internos. El primero de ellos calculará la fuerza de atracción gravitacional de Newton, mientras que el segundo calculará el desplazamiento mediante el método de integración *velocity verlet*.

```
@boundscheck(False)
@wraparound(False)
@nonecheck(False)
@cdivision(True)
cpdef void nbody(
    int N, int D, int T,
    double[::1] positions_x, double[::1] positions_y, double[::1] positions_z,
    double[::1] masses,
    double[::1] velocities_x, double[::1] velocities_y, double[::1] velocities_z,
    double[::1] dp_x, double[::1] dp_y, double[::1] dp_z,
):
    cdef double forces_x, forces_y, forces_z
    cdef double aceleration_x, aceleration_y, aceleration_z
    cdef double dpos_x, dpos_y, dpos_z
    cdef double dsquared, gm, d32
    cdef int i, j

    for _ in range(D):
        # For every body that experiences a force
        for i in prange(N, nogil=True, schedule="static", num_threads=T):
            # Initialize the force of the body i
            forces_x = forces_y = forces_z = 0.0

            # Calculate gravitational force to the rest of the bodies
            # Newton's Law of Universal Attraction:
            # Fi = (G * mi * mj * (qj - qi)) / (|qj - qi| ^ 3)
            for j in range(N):
                # Calculate the distance to the body i:
                # (qj - qi)
                dpos_x = positions_x[j] - positions_x[i]
                dpos_y = positions_y[j] - positions_y[i]
                dpos_z = positions_z[j] - positions_z[i]
                # Calculate the distance magnitude:
                # |qj - qi|
                dsquared = (dpos_x ** 2.0) + (dpos_y ** 2.0) + (dpos_z ** 2.0) + SOFT
                # Calculate the mass factor:
                # (G * mi * mj)
                gm = GRAVITY * masses[j] * masses[i]
                # Calculate the Fi denominator:
                # 1 / |qj - qi| ^ 3
                d32 = dsquared ** -1.5
                # Calculate the force:
                # (G * mi * mj * (qj - qi)) / |qj - qi| ^ 3
                forces_x = forces_x + gm * d32 * dpos_x
                forces_y = forces_y + gm * d32 * dpos_y
                forces_z = forces_z + gm * d32 * dpos_z

            # Calculate the acceleration vector of body i:
            # |M = F * A|  |A = F / M|
            aceleration_x = forces_x / masses[i]
            aceleration_y = forces_y / masses[i]
            aceleration_z = forces_z / masses[i]
            # Velocity Verlet integrator
            # V = V' + ((A * h) / 2) ; h = 1
            velocities_x[i] += aceleration_x * DT / 2.0
            velocities_y[i] += aceleration_y * DT / 2.0
            velocities_z[i] += aceleration_z * DT / 2.0
            # Save the differential position of body i
            dp_x[i] = velocities_x[i] * DT
            dp_y[i] = velocities_y[i] * DT
            dp_z[i] = velocities_z[i] * DT

        # Update positions of the bodies
        for i in prange(N, nogil=True, schedule="static", num_threads=T):
            positions_x[i] += dp_x[i]
            positions_y[i] += dp_y[i]
            positions_z[i] += dp_z[i]
```

Figura 5.4: Implementación Cython paralela.

```
@boundscheck(False)
@wraparound(False)
@nonecheck(False)
@cdivision(True)
cpdef void nbody(
    int N, int D, int T,
    double[::1] positions_x, double[::1] positions_y, double[::1] positions_z,
    double[::1] masses,
    double[::1] velocities_x, double[::1] velocities_y, double[::1] velocities_z,
    double[::1] dp_x, double[::1] dp_y, double[::1] dp_z,
    double[::1] forces_x, double[::1] forces_y, double[::1] forces_z,
):
    cdef double aceleration_x, aceleration_y, aceleration_z
    cdef double dpos_x, dpos_y, dpos_z
    cdef double dsquared, gm, d32

    cdef int i, j, b_i

    for _ in range(D):

        # For every body that experiences a force
        for b_i in prange(0, N, BLOCKSIZE, nogil=True, schedule="static", num_threads=T):
            # Initialize forces
            for i in range(b_i, b_i + BLOCKSIZE):
                forces_x[i] = 0.0
                forces_y[i] = 0.0
                forces_z[i] = 0.0

            # Calculate gravitational force to the rest of the bodies
            # Newton's Law of Universal Attraction:
            # Fi = (G * mi * mj * (qj - qi)) / (|qj - qi| ^ 3)
            for j in range(N):
                for i in range(b_i, b_i + BLOCKSIZE):
                    # Calculate the distance to the body i:
                    # (qj - qi)
                    dpos_x = positions_x[j] - positions_x[i]
                    dpos_y = positions_y[j] - positions_y[i]
                    dpos_z = positions_z[j] - positions_z[i]
                    # Calculate the distance magnitude:
                    # |qj - qi|
                    dsquared = (dpos_x ** 2.0) + (dpos_y ** 2.0) + (dpos_z ** 2.0) + SOFT
                    # Calculate the mass factor:
                    # (G * mi * mj)
                    gm = GRAVITY * masses[j] * masses[i]
                    # Calculate the Fi denominator:
                    # 1 / |qj - qi| ^ 3
                    d32 = dsquared ** -1.5
                    # Calculate the force:
                    # (G * mi * mj * (qj - qi)) / |qj - qi| ^ 3
                    forces_x[i] += gm * d32 * dpos_x
                    forces_y[i] += gm * d32 * dpos_y
                    forces_z[i] += gm * d32 * dpos_z

            for i in range(b_i, b_i + BLOCKSIZE):
                # Calculate the acceleration vector of body i:
                # |M = F * A|  |A = F / M|
                aceleration_x = forces_x[i] / masses[i]
                aceleration_y = forces_y[i] / masses[i]
                aceleration_z = forces_z[i] / masses[i]
                # Velocity Verlet integrator
                # V = V' + ((A * h) / 2) ; h = 1
                velocities_x[i] += aceleration_x * DT / 2.0
                velocities_y[i] += aceleration_y * DT / 2.0
                velocities_z[i] += aceleration_z * DT / 2.0
                # Save the differential position of body i
                dp_x[i] = velocities_x[i] * DT
                dp_y[i] = velocities_y[i] * DT
                dp_z[i] = velocities_z[i] * DT

        # Update positions of the bodies
        for i in prange(N, nogil=True, schedule="static", num_threads=T):
            positions_x[i] += dp_x[i]
            positions_y[i] += dp_y[i]
            positions_z[i] += dp_z[i]
```

Figura 5.5: Implementación Cython utilizando bloques.

5.2. Resultados experimentales

En esta sección se describen y analizan los resultados experimentales obtenidos. Para ello, inicialmente se especifica el diseño experimental empleado (Sección 5.2.1) y luego se analizan los rendimientos obtenidos (Sección 5.2.2).

5.2.1. Diseño experimental

Todas las pruebas fueron realizadas utilizando el servidor descrito en la Sección 3.2.1, junto con Cython v0.29.22 y el compilador ICC v19.1.0.166. Se mantuvo la configuración de I, mientras que se varió el número de hilos (T = {1, 112}) y la carga de trabajo (N = {256, 512, 1024, 4096, 8192, 16384, 32768, 65536, 131072, 262144, 524288}). Para evaluar el rendimiento se utilizó la métrica GFLOPS definida en dicha sección.

En la Tabla 5.1 se describe la versión correspondiente a cada optimización probada, junto con las opciones de compilación y el tipo de dato utilizado. El código fuente de cada una de ellas puede verse en el repositorio de este trabajo [1].

Por último, cabe destacar que todas las versiones de Cython fueron compiladas utilizando el flag *-O3* junto con los flags adicionales indicados en la Tabla 5.1. A continuación se describen brevemente estos últimos:

- `-xCORE-AVX512`: Fuerza a utilizar las instrucciones AVX-512 sobre cualquier otro conjunto SIMD.
- `-qopt-zmm-usage=high`: Indica maximizar el uso de vectorización AVX-512.
- `-march=native`: Indica el uso de los flags más adecuados para el procesador subyacente.
- `-ffast-math` y `-fp-model fast=2`: Indican relajar la precisión de punto flotante.

[1] `https://github.com/Pastorsin/python-hpc-study/tree/main/src/versions/tesina`

Tabla 5.1: Versionado de las implementaciones de Cython.

Versión	Etiqueta	Flags de compilación	Multi-hilado	Tipado	dtype[1]
python-1.0	Naive	-	No	No	float64
cython-0.0	Integración de Cython	-	No	No	float64
cython-1.0	Tipado	-	No	Sí	float64
cython-1.1	Tipado+AVX	-xCORE-AVX512 -qopt-zmm-usage=high	No	Sí	float64
cython-1.2	Tipado+AVX+fp	-xCORE-AVX512 -qopt-zmm-usage=high -fp-model fast=2	No	Sí	float64
cython-1.3	Tipado+march+fmath	-march=native -ffast-math	No	Sí	float64
cython-2.0	Paralela	-march=native -ffast-math	Sí	Sí	float64
cython-3.0	1º variante matemática	-march=native -ffast-math	Sí	Sí	float64
cython-3.1	2º variante matemática	-march=native -ffast-math	Sí	Sí	float64
cython-4.0	Bloques	-march=native -ffast-math	Sí	Sí	float64
cython-5.0	float64 estricta	-march=native	Sí	Sí	float64
cython-5.1	float32 relajada	-march=native -ffast-math	Sí	Sí	float32
cython-5.2	float32 estricta	-march=native	Sí	Sí	float32

[1] Tipo de dato de NumPy.

5.2.2. Rendimiento

En la Figura 5.6 se puede observar que la integración de Cython (sin especificar los tipos de datos de las variables) no produjo una mejora significativa con respecto a la versión *naive*. Esto ocurre ya que el código resultante está utilizando la API de CPython para ejecutarse (ver Fig. 5.2). Por otra parte, al tipar las variables con los tipos de datos de Cython se reduce esta interacción (ver Fig. 5.3) y por ende el rendimiento mejora notablemente (547.7× en promedio).

Tal como se puede apreciar en la Figura 5.7, indicar la utilización de instrucciones AVX-512 logró una mejora 1.7× en promedio. En particular, el rendimiento obtenido con el flag `-march=native` estuvo levemente por encima (1.4 GFLOPS en promedio) con respecto al flag `-xCORE-AVX512 -qopt-zmm-usage=high`.

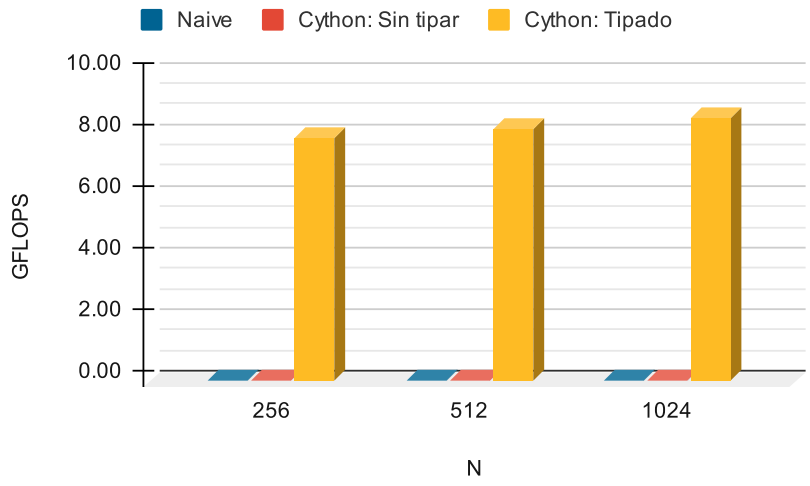

Figura 5.6: Rendimientos obtenidos de la integración de Cython con y sin tipado explícito al variar N.

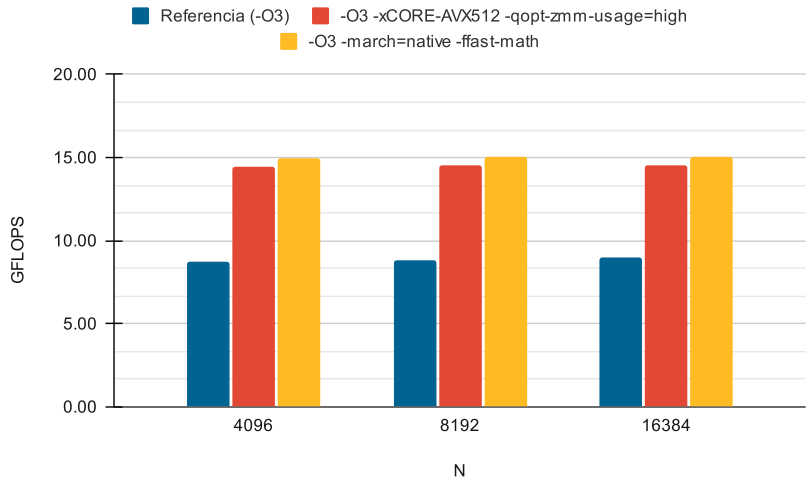

Figura 5.7: Rendimientos obtenidos para las opciones de compilación de Cython al variar N.

En la Figura 5.8 se puede ver que la solución multi-hilada con 56 y 112 threads logró una mejora notable de 21.1× y 34.6× en promedio respectivamente.

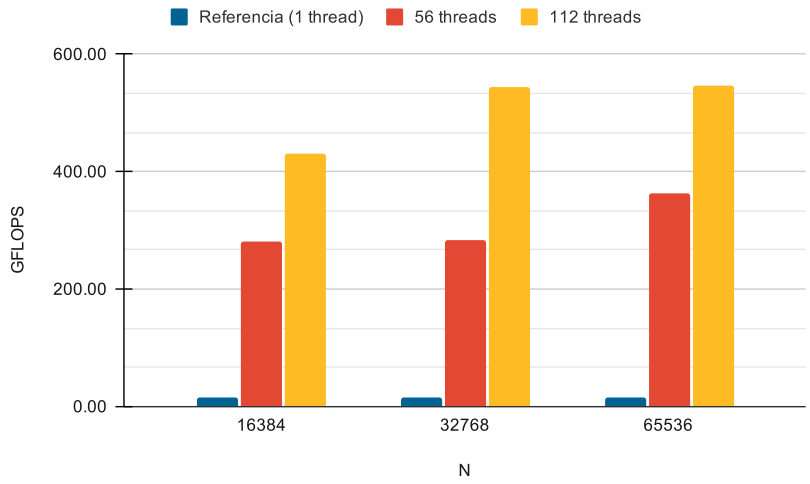

Figura 5.8: Rendimientos obtenidos de la solución multi-hilada con Cython al variar N.

En la Figura 5.9 se pueden observar los rendimientos obtenidos al aplicar las operaciones matemáticas descritas en la Sección 5.1.5. En la misma, se puede notar que emplear una división directa degradó el rendimiento un 41 %; mientras que calcular el inverso multiplicativo por potencia positiva no reportó mejoras significantes.

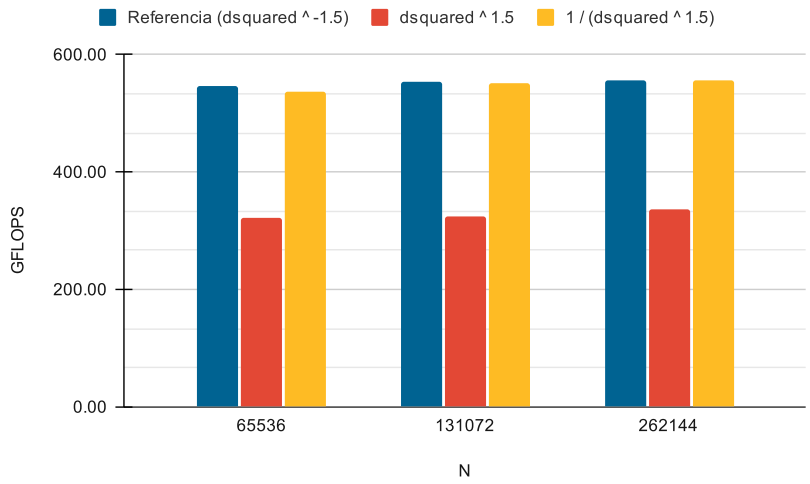

Figura 5.9: Rendimientos obtenidos para los cálculos matemáticos con Cython al variar N.

En la Figura 5.10 se puede notar que el procesamiento de bloques empeoró notablemente el rendimiento de la solución para todos los tamaños de bloques probados. Esto se debe a que el compilador identifica falsas dependencias en el código y no genera las instrucciones SIMD correspondientes. Lamentablemente esto no se puede enmendar, ya que Cython no ofrece una manera de indicarle al compilador que es seguro vectorizar las operaciones.

Por último, en la Figura 5.11 se puede ver que la relajación de precisión prácticamente no tuvo incidencias en el rendimiento obtenido. Sin embargo, utilizar *float* como tipo de dato mejoró el rendimiento notablemente (1362 GFLOPS en promedio) a costo de una menor representación en el resultado final.

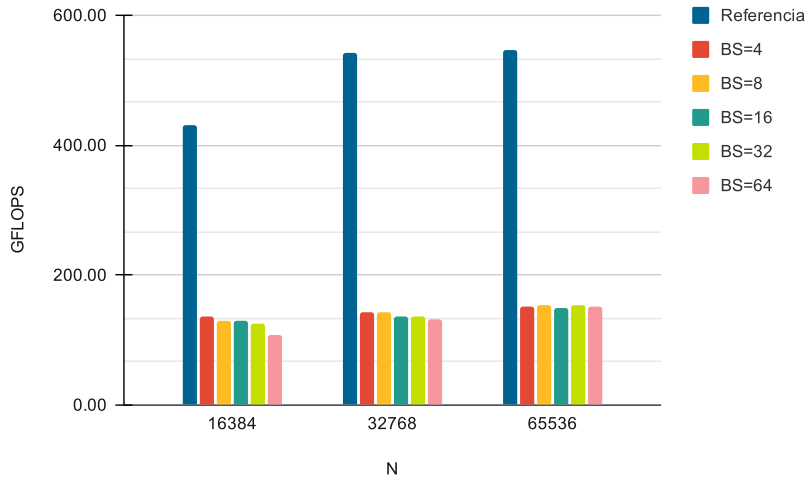

Figura 5.10: Rendimientos obtenidos de procesamiento por bloques utilizando Cython y variando N.

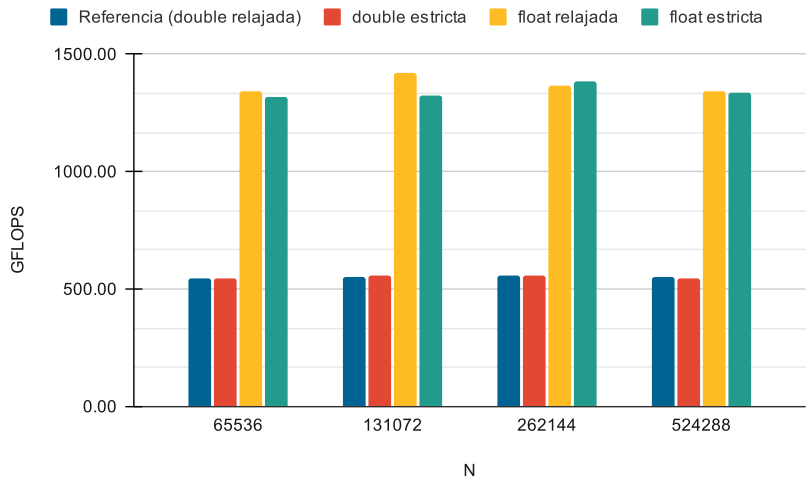

Figura 5.11: Rendimientos obtenidos para la relajación de precisión al variar el tipo de dato y N con Cython.

5.3. Resumen

En este capítulo, se comenzó a optimizar *N-Body* partiendo de la implementación naive de CPython con arreglos de NumPy (sin *broadcasting*). A continuación se describen las optimizaciones introducidas y las principales conclusiones del análisis experimental:

- La simple integración de Cython sin aplicar ninguna modificación sobre el código, no reportó mejoras significativas en el rendimiento obtenido.
- La aplicación de los tipos de datos de Cython mejoró notablemente el rendimiento (547.7× en promedio), ya que se redujo prácticamente toda interacción con la API de CPython.
- La indicación de flags de compilación para vectorizar y relajar la precisión (`-march=native` y `-ffast-math` respectivamente) aumentó el rendimiento 1.7× en promedio.
- La incorporación de paralelismo a nivel de hilos, aceleró la solución 34.6× en promedio gracias a la posibilidad de utilizar OpenMP y de desactivar el GIL de CPython.
- La variación de cálculos matemáticos no mostró una mejora significativa en el rendimiento obtenido.

- El procesamiento por bloques degradó el rendimiento de la solución para todos los tamaños de bloques probados, ya que el compilador detectó falsas dependencias durante la generación de instrucciones SIMD.
- La utilización del tipo de dato `float32` (simple precisión), mejoró notablemente el rendimiento (1362 GFLOPS en promedio) a costo de una menor representación en el resultado final.
- La relajación de precisión no reportó mejoras significativas tanto en simple como doble precisión.

Por último, cabe destacar que el pico de rendimiento fue de 1423.8 y 559.8 GFLOPS en simple y doble precisión respectivamente. Además, resulta importante mencionar que la versión final de Cython logró una aceleración de $46,647\times$ (doble precisión) en comparación a la implementación *naive* de este capítulo.

Capítulo 6

Comparación de prestaciones de traductores de Python

En este capítulo, se realiza una comparación sobre las prestaciones de los traductores utilizados para resolver *N-Body*. Inicialmente se presenta una comparación sobre los rendimientos obtenidos (Sección 6.1); luego, sobre el esfuerzo de programación empleado (Sección 6.2); y por último, se expone un resumen sobre el presente capítulo (Sección 6.3).

6.1. Rendimiento

En esta sección, se presenta el diseño experimental empleado (Sección 6.1.1); y posteriormente, se expone una comparación sobre el rendimiento de los traductores utilizados en el presente trabajo (Sección 6.1.2).

6.1.1. Diseño experimental

Para la realización de este capítulo, no se realizaron experimentos adicionales, si no que se utilizaron los correspondientes a las implementaciones de Python, Numba y Cython, que resultaron más rápidas en términos de rendimiento (ver Secciones 3.2.2, 4.2.2 y 5.2.2 respectivamente). Adicionalmente, se incluyó una comparación con una solución a *N-Body* optimizada en C+OpenMP para tener una referencia de la aceleración lograda con las soluciones desarrolladas. Para ello, se utilizó la implementación presentada en [58] utilizando el compilador ICC del servidor mencionado en la Sección 5.2.1.

Para evaluar el rendimiento se utilizó la métrica GFLOPS definida en la Sección 3.2.1, como también las mismas configuraciones de N, I y T.

A modo de resumen para el lector, en la Tabla 6.1 se puede ver una breve descripción de las versiones utilizadas para esta comparación.

Tabla 6.1: Versionado de las optimizaciones finales.

Versión	Etiqueta	Flags de compilación	Multi-hilado	Precisión
python-2.0	NumPy+*broadcasting*	-	No	Doble
numba-2.0	Sin *broadcasting*	-njit -fastmath -SVML -parallel	Sí	Doble
numba-5.1	float32 relajada	-njit -fastmath -SVML -parallel	Sí	Simple
cython-2.0	Paralela	-O3 -qopenmp -march=native -ffast-math	Sí	Doble
cython-5.1	float32 relajada	-O3 -qopenmp -march=native -ffast-math	Sí	Simple
c-11[1]	Desenrollado+fp-mode	-O3 -qopenmp -fp-model fast=2 -xCORE-AVX512 -qopt-zmm-usage=high	Sí	Simple
c-13[1]	Double+fp-mode	-O3 -qopenmp -fp-model fast=2 -xCORE-AVX512 -qopt-zmm-usage=high	Sí	Doble

[1] Versionado correspondiente según [58].

6.1.2. Comparación

En primer lugar, cabe aclarar que las versiones con CPython y PyPy no fueron incluidas en las figuras comparativas debido a su bajo rendimiento (0.5 GFLOPS en promedio).

En la Figura 6.2 se presenta una comparación entre las optimizaciones finales de Numba y Cython al variar la carga de trabajo y el tipo de dato. Se puede observar que al utilizar doble precisión, Cython resultó levemente más rápido que Numba por un promedio de 16.7 GFLOPS; mientras que en simple precisión, Numba resultó superior por un promedio de 73 GFLOPS. Estos valores

representan mejoras de 3 % y 5 %, respectivamente . A su vez, resulta importante mencionar que ambas versiones finales de Numba y Cython lograron una aceleración promedio de 1018× y 1050× respectivamente en comparación a la mejor implementación de CPython (caso `float64`).

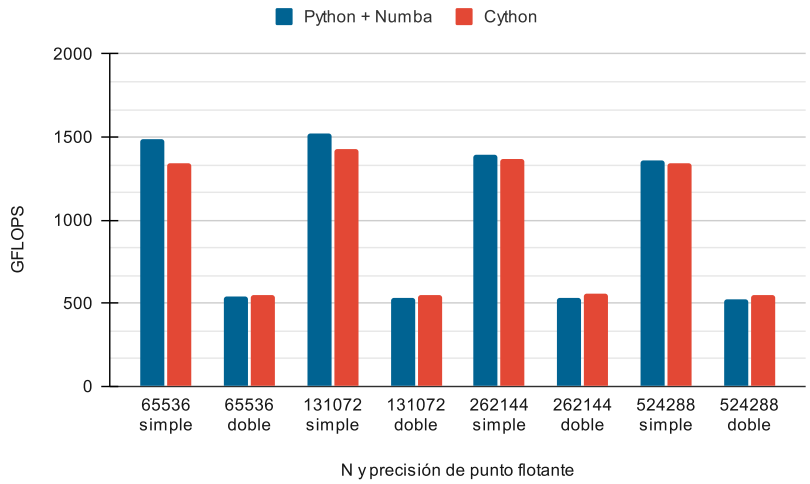

Figura 6.1: Comparación de rendimiento de las versiones finales entre Numba y Cython variando el tipo de dato y N.

Adicionalmente, en la Figura 6.2 se presenta una comparación con la solución optimizada en C+OpenMP. En la misma, se puede notar que no hay diferencias significativas de rendimiento al emplear doble precisión; mientras que en simple precisión, la versión de C+OpenMP superó ampliamente a las implementaciones de Numba y Cython por un promedio de 989.8 GFLOPS (1.7×) y 1062.9 GFLOPS (1.8×) respectivamente. Esto último se debe a que el procesamiento de a bloques en C+OpenMP permitió explotar la localidad de datos de una mejor manera, mientras que en las demás implementaciones, las versiones de a bloques presentaron limitaciones a la hora de vectorizar por falsas dependencias. Lamentablemente, ambos traductores no brindaron herramientas para enmendarlo (ver Secciones 4.2.2 y 5.2.2).

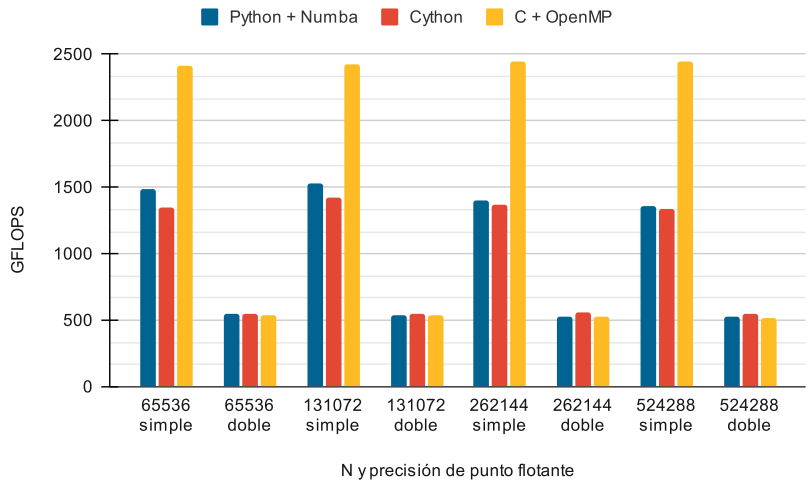

Figura 6.2: Comparación de rendimiento de las versiones finales entre Python+Numba, C+OpenMP y Cython variando el tipo de dato y N.

6.2. Esfuerzo de programación

A continuación se presenta el diseño experimental seguido (Sección 6.2.1) y la comparación de esfuerzo programación descrita anteriormente (Sección 6.2.2).

6.2.1. Diseño experimental

Las versiones utilizadas son las que resultaron más rápidas en términos de rendimiento, las mismas se encuentran resumidas en la Tabla 6.1. Sin embargo, cabe destacar que en la última tabla se encuentran diferentes subversiones de cada implementación debido a la variación del tipo de dato, pero el código de la simulación es el mismo.

Tal como se mencionó en los objetivos de esta tesina (ver Sección 1.2), es deseable analizar y comparar el esfuerzo de programación requerido por cada solución presentada. Para ello, inicialmente se plantea una comparación cuantitativa a través del indicador SLOC (*Source Lines Of Code*), el cual permite contabilizar las líneas de código [59]. Sin embargo, la subjetividad de este indicador dificulta medir de manera exacta el costo de programación empleado. Por lo tanto, adicionalmente se realiza una comparación cualitativa con el fin de complementar al primer análisis y permitirle al lector un mejor entendimiento del esfuerzo de programación requerido por cada solución.

Por último, cabe destacar que para el cálculo del indicador SLOC, se utilizó la herramienta *cloc*[1].

6.2.2. Comparación

En la Figura 6.3 se pueden observar la cantidad de líneas de código de las optimizaciones finales discriminando instrucciones, comentarios y líneas en blanco. En particular, la solución de Python requirió solo 14 líneas de código, lo que representa el 29.8 % y el 15.2 % de las líneas de código de Cython y Numba respectivamente. Esto se debe a la característica *broadcasting* provista por NumPy, la cual reduce el código notablemente gracias a la posibilidad de realizar operaciones entre arreglos.

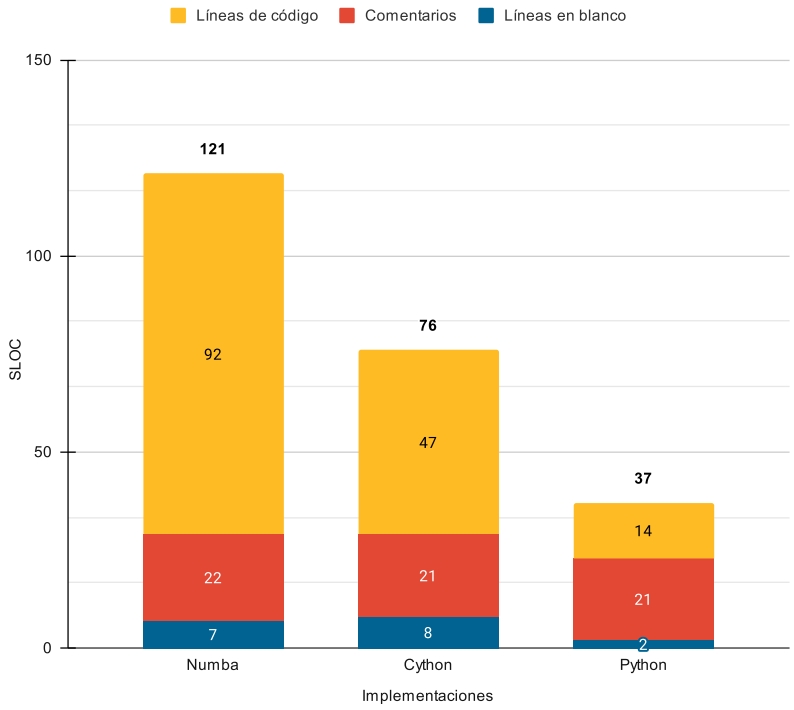

Figura 6.3: Cantidad de líneas de código de las optimizaciones finales.

Por otra parte, se puede notar que la solución Numba requirió 92 líneas de código, lo que representa 6.6× y 2× más que las implementaciones de Python y Cython respectivamente. Sin embargo, hay que tener en cuenta que la primera contiene las opciones de compilación declaradas en la propia implementación

[1] `https://github.com/AlDanial/cloc`

(representa 32.6 % del código total) y, además, contiene llamadas a 2 funciones adicionales que actúan como zonas paralelas, mientras que en las versiones de Python y Cython no se encuentra dicha modularización.

A continuación, se describen los aspectos cualitativos que fueron identificados al momento de desarrollar cada solución.

Con respecto a la versión de Python, si bien requirió menos líneas de código, el desarrollo de las operaciones vectoriales llevó un alto esfuerzo de programación que no se ve representado por la cantidad de líneas de código.

Se puede notar que Cython requirió menos líneas de código, pero a su vez, se necesitó de un conocimiento adicional de C y OpenMP a la hora de paralelizar el problema. Mientras que en Numba, simplemente se necesitó indicar la sentencia `prange`, y luego la librería se encargó de traducir las zonas paralelas a la *threading layer* correspondiente. Sin embargo, a favor de Cython, esto permite un mayor grado de control sobre la sincronización de los hilos, ya que para llevarla a cabo podemos interactuar con la API de OpenMP. En sentido opuesto, Numba no permite nativamente la sincronización explícita de los hilos y se deben utilizar librerías externas como `threading` para lograrlo.

Por otra parte, en la solución de Cython se tuvo que modificar el código de la simulación para declarar los tipos, mientras que en Numba se los declaró a través del parámetro `signature` en el decorador `njit` sin interferir en el código de la función ya desarrollada.

Para finalizar, se debe destacar que la implementación de Cython necesitó 3 archivos para llevar a cabo la solución: (1) El archivo de compilación *setup.py*. (2) El archivo fuente de la simulación *cynbody.pyx*. (3) El ejecutable *run.py*. Por lo tanto, al momento de ejecutar cada simulación, fue necesario compilar el código fuente, luego importarlo a través de un *script* de Python y finalmente ejecutarlo mediante la línea de comandos; mientras que por otra parte, en las versiones de Python y Numba simplemente se precisó un archivo para almacenar el código fuente y un comando para ejecutar la simulación.

6.3. Resumen

En este capítulo se evaluaron las prestaciones de los traductores de Python utilizados en los capítulos anteriores (CPython, PyPy, Numba y Cython).

En primer lugar, se realizó un análisis sobre el rendimiento de los traductores mencionados, y, adicionalmente se presentó una comparación con una implementación de C con el fin de tener una referencia de la aceleración lograda. Posteriormente, se realizó un análisis cuantitativo y cualitativo sobre el esfuerzo de programación que llevó cada solución. A continuación se destacan las principales conclusiones alcanzadas:

Rendimiento.

- Las versiones de CPython y PyPy obtuvieron un bajo rendimiento de 0.5 GFLOPS en promedio. Esto se debe a las limitaciones propias de los lenguajes interpretados y al componente GIL, el cual no permitió paralelizar la solución a nivel de hilos como los demás traductores.
- Las versiones de Numba y Cython no presentaron una diferencia de rendimiento significativa entre ellas. En simple precisión, Numba resultó superior a Cython mejorando solo el 3 % del rendimiento; mientras que en doble precisión, Cython superó levemente a Numba con una mejora de solo el 6 %.
- Se logró una aceleración de 1018×/1050× al utilizar Numba/Cython en comparación a la mejor implementación de CPython (en doble precisión).
- Tanto Numba como Cython lograron rendimientos prácticamente iguales a C+OpenMP al emplear doble precisión. Sin embargo, en simple precisión, C+OpenMP resultó 1.7× y 1.8× más rápido que Numba y Cython respectivamente.

Esfuerzo de programación.

- CPython requirió menos líneas de código que los demás traductores, solo necesitó 14 líneas de código para llevar a cabo la solución gracias al *broadcasting* de NumPy; sin embargo, se debe destacar que las operaciones entre arreglos llevaron un esfuerzo considerable de desarrollo que no se ve representado por el número de instrucciones utilizadas.
- Cython requirió menos líneas de código que Numba a costo de un conocimiento adicional de C y OpenMP, junto con la necesidad de intervenir en el código fuente ya desarrollado y de requerir una mayor cantidad de archivos fuentes. Esto último fue más sencillo en Numba, sobre todo gracias a su enfoque de decoradores, su abstracción a la capa de hilos subyacente y su compilación en tiempo de ejecución.

Por último, de los anteriores análisis se concluye que tanto las versiones de Numba y Cython alcanzaron prácticamente el mismo rendimiento; mientras que CPython requirió un menor esfuerzo de programación que los demás traductores a costo de un pobre rendimiento. En particular, Cython tomó menos líneas de código que Numba, pero requirió conocimiento adicional de C y OpenMP, junto con un proceso más arduo de ejecución.

Capítulo 7

Conclusiones y trabajos futuros

En la actualidad, el uso de Python está presente en diversas áreas de la computación debido a sus notables características, de las cuales principalmente se destaca su alto grado de legibilidad y bajo esfuerzo de programación. Sin embargo, Python presenta fuertes limitaciones a la hora de optimizar aplicaciones debido a su naturaleza de lenguaje interpretado y al componente GIL de CPython, el cual impide que más de un hilo se ejecute a la vez, anulando la posibilidad de implementar paralelismo a nivel de hilos.

Para enmendar las desventajas mencionadas previamente han surgido traductores alternativos, aunque cada uno con un enfoque diferente y con su propia relación de costo-rendimiento. Entre ellos, se encuentran: (1) PyPy, un compilador JIT que asegura incrementar la velocidad de ejecución y disminuir la memoria utilizada de los programas escritos en Python. (2) Numba, un compilador JIT que permite traducir código Python a código de máquina optimizado a través de LLVM. (3) Cython, un *superset* de Python que permite utilizar librerías de C como OpenMP para paralelizar las aplicaciones.

En este trabajo se realizó una comparación de las prestaciones (rendimiento y esfuerzo de programación) de los traductores mencionados anteriormente. En particular, se eligió como caso de estudio a *N-Body*, un problema paralelizable de alta demanda computacional y considerado *CPU-bound*. Para ello, se realizaron diferentes algoritmos para cada traductor, partiendo desde una versión base y aplicando optimizaciones incrementales hasta llegar a la versión final. En ese sentido, se exploraron los beneficios de usar multi-hilado, procesamiento de a bloques, *broadcasting*, diferentes cálculos matemáticos y funciones de potencias, vectorización, tipado explícito y distintas API de hilos.

Considerando los resultados obtenidos, se puede decir que no hubo diferencias significativas entre los rendimientos de Numba y Cython. Sin embargo, ambos traductores mejoraron de manera notable el rendimiento de CPython, presentando rendimientos cercanos a los de una implementación en C+OpenMP.

No sucedió lo mismo con PyPy, ya que no logró mejorar el rendimiento de CPython+NumPy debido a su incapacidad de paralelizar debido al GIL.

Por otra parte, CPython requirió un menor esfuerzo de programación que los demás traductores a costo de su bajo rendimiento. Luego, Cython resultó con menos líneas de código, pero requirió un conocimiento adicional de C+OpenMP junto con un proceso más laborioso de ejecución. En sentido opuesto, Numba tomó un mayor número de líneas de código debido a la especificación de opciones de compilación en el mismo código y la modularización empleada; sin embargo, Numba resultó más simple a la hora de desarrollar gracias a su enfoque de decoradores y a la posibilidad de mantener el mismo código que la implementación original.

En base a lo anterior, se puede afirmar que en contextos similares a los de este estudio tanto Numba como Cython pueden ser potentes herramientas para acelerar aplicaciones *CPU-bound* desarrolladas en Python. La elección entre uno y otro estará mayormente determinada por el enfoque que el equipo de desarrollo encuentre más conveniente, considerando las características propias de cada uno.

Como trabajos futuros, resulta de interés extender este estudio mediante los siguientes aspectos:

- Explorar otras capacidades y limitaciones de los traductores de Python no contempladas en este trabajo, como la utilización de GPUs.
- Replicar el estudio realizado considerando: (1) otros casos de estudio que sean computacionalmente intensivos pero cuyas características sean diferentes a las de *N-Body*; (2) otras arquitecturas multicore distintas a la usada en este trabajo. Ambas extensiones contribuirían a robustecer los resultados encontrados.
- Dado que existen otras tecnologías que permitan implementar paralelismo a nivel de procesos en Python, realizar una comparación entre ellas considerando no sólo el rendimiento sino también el costo de programación.

Bibliografía

[1] TIOBE Software BV. *TIOBE Index for November 2021*. Nov. de 2021. URL: https://www.tiobe.com/tiobe-index/.

[2] *General Python FAQ — Python 3.9.5 documentation*. URL: https://docs.python.org/3/faq/general.html#what-is-python (visitado 07-06-2021).

[3] *Python vs Java: What's The Difference? – BMC Software — Blogs*. URL: https://www.bmc.com/blogs/python-vs-java/ (visitado 07-06-2021).

[4] *Python vs C++ Comparison: Compare Python vs C++ Speed and More*. URL: https://www.bitdegree.org/tutorials/python-vs-c-plus-plus/ (visitado 07-06-2021).

[5] *Top 12 Fascinating Python Applications in Real-World [2021] — upGrad blog*. URL: https://www.upgrad.com/blog/python-applications-in-real-world/ (visitado 07-06-2021).

[6] *Applications for Python — Python.org*. URL: https://www.python.org/about/apps/ (visitado 07-06-2021).

[7] *Accelerating High Performance Computing (HPC) for Population-level Genomics*. URL: https://www.hpcwire.com/2019/09/30/accelerating-high-performance-computing-hpc-for-population-level-genomics/ (visitado 15-06-2021).

[8] *What Is High-Performance Computing (HPC)? How It Works — NetApp*. URL: https://www.netapp.com/data-storage/high-performance-computing/what-is-hpc/ (visitado 15-06-2021).

[9] *High performance computing: Do you need it? — Network World*. URL: https://www.networkworld.com/article/3444399/high-performance-computing-do-you-need-it.html (visitado 15-06-2021).

[10] *python/cpython*. Jun. de 2021. URL: https://github.com/python/cpython (visitado 07-06-2021).

[11] *What Is the Python Global Interpreter Lock (GIL)? – Real Python*. URL: https://realpython.com/python-gil/ (visitado 07-06-2021).

[12] *PyPy*. URL: https://www.pypy.org/ (visitado 07-06-2021).

[13] *Frequently Asked Questions — PyPy documentation*. URL: https://doc.pypy.org/en/latest/faq.html#does-pypy-have-a-gil-why (visitado 15-06-2021).

[14] *Home — Jython*. URL: https://www.jython.org/ (visitado 07-06-2021).

[15] *Chapter 19: Concurrency — Definitive Guide to Jython latest documentation*. URL: https://jython.readthedocs.io/en/latest/Concurrency/#no-global-interpreter-lock (visitado 15-06-2021).

[16] *Sunsetting Python 2 — Python.org*. URL: https://www.python.org/doc/sunset-python-2/ (visitado 16-06-2021).

[17] *Jython Developer's Guide — Jython Developer's Guide*. URL: https://jython-devguide.readthedocs.io/en/latest/#status-of-jython-branches (visitado 15-06-2021).

[18] *Numba documentation — Numba 0.53.1-py3.7-linux-x86_64.egg documentation*. URL: https://numba.readthedocs.io/en/stable/index.html (visitado 17-08-2021).

[19] *7. Decorators — Python Tips 0.1 documentation*. URL: https://book.pythontips.com/en/latest/decorators.html (visitado 07-06-2021).

[20] *Cython: C-Extensions for Python*. URL: https://cython.org/ (visitado 07-06-2021).

[21] *Home - OpenMP*. URL: https://www.openmp.org/ (visitado 07-06-2021).

[22] I Wilbers, Hans Petter Langtangen y Åsmund Ødegård. «Using Cython to Speed Up Numerical Python Programs». En: ene. de 2009, págs. 495-512. ISBN: 978-82-519-2421-4.

[23] Alexander Roghult. «Benchmarking Python Interpreters: Measuring Performance of CPython, Cython, Jython and PyPy». En: 2016.

[24] Jan Gmys y col. «A comparative study of high-productivity high-performance programming languages for parallel metaheuristics». en. En: *Swarm and Evolutionary Computation* 57 (sep. de 2020), pág. 100720. ISSN: 22106502. DOI: 10.1016/j.swevo.2020.100720. URL: https://linkinghub.elsevier.com/retrieve/pii/S2210650220303734 (visitado 07-06-2021).

[25] F. Wilkens. «Evaluation of performance and productivity metrics of potential programming languages in the HPC environment». En: 2015.

[26] Xing Cai, Hans Petter Langtangen y Halvard Moe. «On the Performance of the Python Programming Language for Serial and Parallel Scientific Computations». en. En: *Scientific Programming* 13.1 (2005), págs. 31-56. ISSN: 1058-9244, 1875-919X. DOI: 10.1155/2005/619804. URL: http://www.hindawi.com/journals/sp/2005/619804/abs/ (visitado 07-06-2021).

[27] M. Varsha y col. «A Review of Existing Approaches to Increase the Computational Speed of the Python Language». en. En: *International Journal of Research in Engineering, Science and Management* (2019). ISSN: 25815792.

[28] *Microsoft's new research lab studies developer productivity and well-being — VentureBeat*. URL: https://venturebeat.com/2021/05/25/microsofts-new-research-lab-studies-developer-productivity-and-well-being/ (visitado 03-11-2021).

[29] *Coiling Python Around Hybrid Quantum Systems*. URL: https://www.nextplatform.com/2021/05/19/coiling-python-around-hybrid-quantum-systems/ (visitado 03-11-2021).

[30] Andrés Milla y Enzo Rucci. «Acelerando Código Científico en Python usando Numba». En: *XXVII Congreso Argentino de Ciencias de la Computación (CACIC 2021)* (oct. de 2021), pág. 12. URL: http://sedici.unlp.edu.ar/handle/10915/126012.

[31] Andrés Milla y Enzo Rucci. *Acelerando aplicaciones paralelas en Python: Numba vs. Cython*. Conferencia. PyCon 2021, oct. de 2021. URL: https://eventos.python.org.ar/events/pyconar2021/activity/448/.

[32] *About Python™ — Python.org*. URL: https://www.python.org/about/ (visitado 03-11-2021).

[33] *Alternative Python Implementations — Python.org*. URL: https://www.python.org/download/alternatives/ (visitado 03-11-2021).

[34] *PyPy - Features — PyPy*. URL: https://www.pypy.org/features.html (visitado 04-11-2021).

[35] *threading — Paralelismo basado en hilos — documentación de Python - 3.8.12*. URL: https://docs.python.org/es/3.8/library/threading.html (visitado 02-11-2021).

[36] *Coroutines and Tasks — Python 3.10.0 documentation*. URL: https://docs.python.org/3/library/asyncio-task.html (visitado 02-11-2021).

[37] *concurrent.futures — Launching parallel tasks — Python 3.10.0 documentation*. URL: https://docs.python.org/3/library/concurrent.futures.html#concurrent.futures.ThreadPoolExecutor (visitado 02-11-2021).

[38] *multiprocessing — Process-based parallelism — Python 3.10.0 documentation*. URL: https://docs.python.org/3/library/multiprocessing.html (visitado 24-11-2021).

[39] *Compiling code ahead of time — Numba 0.52.0.dev0+274.g626b40e-py3.7-linux-x86_64.egg documentation*. URL: https://numba.pydata.org/numba-doc/dev/user/pycc.html#pycc (visitado 06-11-2021).

[40] *Compiling Python code with @jit — Numba 0.53.1-py3.7-linux-x86_64.egg documentation*. URL: https://numba.readthedocs.io/en/stable/user/jit.html (visitado 05-08-2021).

[41] *The Threading Layers — Numba 0.50.1 documentation*. URL: https://numba.pydata.org/numba-doc/latest/user/threading-layer.html (visitado 08-12-2021).

[42] *Language Basics — Cython 3.0.0a9 documentation*. URL: https://cython.readthedocs.io/en/latest/src/userguide/language_basics.html (visitado 06-11-2021).

[43] Enzo Rucci y col. «Simulación de N Cuerpos Computacionales sobre Intel Xeon Phi KNL». En: *Actas del XXV Congreso Argentino de Ciencias de la Computación (CACIC 2019)*. 2019, págs. 194-204. ISBN: 978-987-688-377-1.

[44] Peter L. Freddolino y col. «Challenges in protein-folding simulations». En: *Nature Physics* 6.10 (2010), págs. 751-758. ISSN: 1745-2481. DOI: 10.1038/nphys1713. URL: https://doi.org/10.1038/nphys1713.

[45] Rhushabh Goradia. *Global Illumination for Point Models*. Fourth Annual Progress Seminar, 2008. https://tinyurl.com/y5nqel39. 2008.

[46] Paul Tipler. *Physics for Scientists and Engineers: Mechanics, Oscillations and Waves, Thermodynamics*. Freeman y Co, 2004.

[47] Peter Young. *The leapfrog method and other "symplectic" algorithms for integrating Newton's laws of motion*. Inf. téc. https://young.physics.ucsc.edu/115/leapfrog.pdf. Physics Department, University of California, USA, abr. de 2014.

[48] Josh Barnes y Piet Hut. «A hierarchical O (N log N) force-calculation algorithm». En: *nature* 324.6096 (1986), págs. 446-449.

[49] Pearu Peterson. *pearu/f2py*. Abr. de 2021. URL: https://github.com/pearu/f2py (visitado 07-06-2021).

[50] *Glossary — NumPy v1.21 Manual*. URL: https://numpy.org/doc/stable/glossary.html#term-axis (visitado 08-12-2021).

[51] *NumPy*. URL: https://numpy.org/ (visitado 07-06-2021).

[52] Manuel Costanzo y col. «Performance vs Programming Effort between Rust and C on Multicore Architectures: Case Study in N-Body». En: *2021 XLVII Latin American Computer Conference (CLEI)*. 2021, In press.

[53] *maartenbreddels/per4m: Profiling and tracing information for Python using viztracer and perf, the GIL exposed*. URL: https://github.com/maartenbreddels/per4m (visitado 08-11-2021).

[54] Intel Corp. *How to Manipulate Data Structure to Optimize Memory Use on 32-Bit Intel® Architecture*. https://tinyurl.com/26h62f76. 2018.

[55] *Source Files and Compilation — Cython 3.0.0a9 documentation*. URL: https://cython.readthedocs.io/en/latest/src/userguide/source_files_and_compilation.html (visitado 04-11-2021).

[56] *Cython for NumPy users — Cython 3.0.0a9 documentation*. URL: https://cython.readthedocs.io/en/latest/src/userguide/numpy_tutorial.html#efficient-indexing-with-memoryviews (visitado 11-11-2021).

[57] *Using Parallelism — Cython 3.0.0a9 documentation*. URL: https://cython.readthedocs.io/en/latest/src/userguide/parallelism.html (visitado 12-11-2021).

[58] Enzo Rucci y col. «Optimization of the N-Body Simulation on Intel's Architectures Based on AVX-512 Instruction Set». En: *Computer Science – CACIC 2019*. Springer International Publishing, 2020, págs. 37-52. ISBN: 978-3-030-48325-8.

[59] Kaushal Bhatt y col. «Analysis of source lines of code (SLOC) metric». En: *International Journal of Emerging Technology and Advanced Engineering* 2.5 (2012), págs. 150-154.