Comparación de Rendimiento y Esfuerzo de Programación entre Numba y Cython para una Aplicación Multi-hilada de Alto Rendimiento

Andrés Milla[1] and Enzo Rucci[2][0000−0001−6736−7358] ✉

[1] Facultad de Informática, UNLP.
La Plata (1900), Bs As, Argentina
andressmilla@gmail.com
[2] III-LIDI, Facultad de Informática, UNLP – CIC.
La Plata (1900), Bs As, Argentina
erucci@lidi.info.unlp.edu.ar

Resumen En la actualidad, Python es uno de los lenguajes más utilizados en diversas áreas de aplicación. Sin embargo, éste presenta limitaciones a la hora de poder optimizar y paralelizar aplicaciones debido a 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. Este trabajo es una continuación de otros previos, donde se presenta una comparación de rendimiento más justa y actualizada de los traductores Numba y Cython para el caso de estudio *N-Body* (un problema popular con alta demanda computacional). Además, se realiza un análisis comparativo del esfuerzo de programación requerido por ambas soluciones, lo que brinda un segundo criterio a la hora de optar entre ellos.

Keywords: N-body · CPU-bound · Programación paralela · Costo de programación · Python

1. Introducción

En la actualidad, a pesar de que Python se ha convertido en uno de los lenguajes más populares [20], sigue siendo considerado "lento" en comparación a otros lenguajes compilados como C, C++ y Fortran; especialmente para que aquellas aplicaciones intensivas en CPU (*CPU-bound*) [3]. Entre las causas de su pobre rendimiento, se encuentran su naturaleza de lenguaje interpretado y sus limitaciones al momento de implementar soluciones multi-hiladas [15]. En particular, el principal problema es la utilización de un componente llamado *Global Interpreter Lock* (GIL) en el intérprete oficial CPython. Este último sólo admite que un único un hilo se ejecute a la vez, lo que lleva a que su ejecución sea

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

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, como es el caso de PyPy [10]. En sentido opuesto, existen intérpretes que optan por no utilizar el GIL en sus implementaciones, como por ejemplo Jython [5]. Lamentablemente, Jython emplea una versión discontinuada de Python, lo que 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 [8]. Numba utiliza una característica de Python conocida como decoradores [1], para intervenir lo menos posible en el código del programador. Por último, se puede mencionar a Cython, un compilador estático que permite transpilar [4] Python a C, y luego compilarlo a código objeto [3]. También permite desactivar el GIL y utilizar librerías de C como OpenMP [4], 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 [22,18], lo que no permite evaluar sus capacidades de procesamiento paralelo. En sentido contrario, en el caso de sí usar paralelismo, lo hacen entre lenguajes y no entre traductores de Python [14,23,13,21]. 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 [7,2].

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. Por lo tanto, este artículo se enfoca en su comparación considerando no sólo el rendimiento, sino también su productividad y esfuerzo de programación asociados. Este artículo es una continuación de trabajos previos [17,16], presentado las siguientes nuevas contribuciones:

- Una re-evaluación de los rendimientos de las implementaciones Numba y Cython considerando que Intel ha adoptado recientemente a LLVM [5] como *backend* de su compilador de C/C++ [6]. De esta manera, se ofrece una

[4] 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.

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

comparación más justa y actualizada (ambos operan con el mismo *backend* ahora).
- Un análisis comparativo del esfuerzo de programación requerido para las soluciones Numba y Cython desarrolladas, el cual permite considerar un segundo criterio a la hora de optar entre ellos.

Mediante este estudio comparativo se espera contribuir a programadores de Python para que conozcan fortalezas y debilidades al momento de implementar aplicaciones multi-hiladas de alto rendimiento. El resto del artículo se organiza de la siguiente forma. La Sección 2 introduce el marco teórico para esta investigación. Luego, la Sección 3 describe las implementaciones realizadas. A continuación, la Sección 4 analiza los resultados experimentales mientras que la Sección 5 resume las conclusiones junto al trabajo futuro.

2. Marco teórico

2.1. Numba

Numba es un compilador JIT que permite traducir código Python a código de máquina optimizado a través de LLVM [6]. De acuerdo con su documentación, es capaz de alcanzar aceleraciones similares a las de lenguajes compilados como C, C++ y Fortran [8], sin necesidad de re-escribir su código gracias a un enfoque de anotaciones llamados decoradores [1].

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. 1), respectivamente [8].

```
1  from numba import njit
2
3  # Equivalent to indicating
4  # @jit(nopython=True)
5  @njit
6  def f(x, y):
7      return x + y
```

Figura 1: Compilación en modo *nopython*.

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 al momento de la declaración. Además, también posibilitará indicar los tipos de datos que usará la función y controlar la organización de los datos [8] en memoria (ver Fig. 2).

Multi-hilado 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. 3), la cual distribuye las iteraciones entre los hilos de manera similar a la directiva `parallel for`

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

```
1   from numba import njit, double
2
3   @njit(double(double[::1, :],
4                double[:, ::1]))
5   def f(x, y):
6       """
7       x:  Vector 2D of the "Double" type
8           organized in columns.
9       y:  Vector 2D of the "Double" type
10          organized in rows.
11      Returns the sum of the product
12      of vectors x and y.
13      """
14      return (x * y).sum()
```

Figura 2: Compilación en modo *nopython* con el parámetro `signature`

```
1   from numba import njit, double, prange
2
3   @njit(double(double[:,:]), parallel=True)
4   def f(x):
5       """
6       x: 2D Vector.
7       Returns the sum of vector x
8       through a reduction.
9       """
10      N = x.shape[0]
11      z = 0
12
13      for i in prange(N):
14          z += x[i]
15
16      return z
```

Figura 3: Compilación en modo *nopython* con el parámetro `parallel`

de OpenMP. Además, también soporta reducciones y se encarga de declarar las variables como privadas a cada hilo si son declaradas dentro del alcance de la zona paralela. Lamentablemente, Numba aún no soporta primitivas que permitan controlar la sincronización de los hilos, como pueden ser semáforos o *locks* [8].

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 la posibilidad de utilizar *Intel SVML* en caso de estar disponible en el sistema [8].

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 [9,8].

2.2. 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 [3]. En otras palabras, Cython es un *superset* de Python que permite interactuar con funciones, tipos y librerías de C.

Compilación Tal como se puede apreciar en la Fig. 4 el flujo de programación de Cython es muy diferente al que el programador de Python está habituado. 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 podrá compilar a través de un archivo `setup.py`, en donde se indican los flags de compilación para dar como salida (1) un archivo con extensión `.c`, el cual corresponde al código transpilado de Cython a C y (2) un archivo binario con extensión `.so`, el cual corresponde a la compilación del archivo de C descrito previamente. Este último nos permitirá importar el módulo compilado en cualquier script de Python.

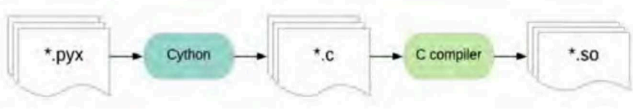

Figura 4: Flujo de programación en Cython.

Tipos de datos Cython permite declarar variables utilizando los tipos de datos de C a partir de la sentencia `cdef` (ver Fig. 5). 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. Además, Cython permite indicar la organización de memoria de los arreglos al igual que Numba [3].

Multi-hilado Cython provee soporte para utilizar OpenMP a través del módulo `cython.parallel`. El mismo contiene la función `prange`, la cual posibilita paralelizar bucles mediante el constructor parallel for de OpenMP. A su vez, esta función permite desactivar el GIL e indicar el *scheduling* de OpenMP a través de los argumentos `nogil` y `schedule` respectivamente.

Cabe destacar que todas las asignaciones declaradas dentro del bloques `prange` son transpiladas como `lastprivate`, mientras que las reducciones solo 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. 6).

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.

```
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 5: Variables declaradas con tipos de datos de C en Cython.

```
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 6: Reducción utilizando el bloque `prange` de Cython.

2.3. Caso de estudio: N-Body

En esta sección se presenta el caso de estudio elegido: la simulación de N cuerpos computacionales (*N-Body*), un problema *CPU-bound* de complejidad computacional $O(n^2)$ y que resulta popular en la comunidad HPC.

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.

```
1    for t in range(PASOS):
2        for i in range(N):
3            for j in range(N):
4                Calcular la fuerza ejercida por C_j sobre C_i
5                Totalizar las fuerzas ejercidas sobre C_i
6            Calcular el desplazamiento de C_i
7            Mover C_i
```

Figura 7: Pseudo-código del algoritmo N-Body

La simulación se realiza en 3 dimensiones y la atracción gravitacionales entre dos cuerpos C_i y C_j se computa de acuerdo con la mecánica Newtoniana. Más información se puede encontrar en [19].

El pseudo-código de la versión directa se muestra en la Fig. 7. Este problema presenta dos dependencias de datos que se pueden observar en la figura anterior. En primer lugar, ningún cuerpo puede moverse hasta tanto el resto haya computado sus interacciones. En segundo lugar, ningún cuerpo puede avanzar al siguiente paso hasta tanto el resto haya alcanzado el paso actual.

3. Implementaciones N-Body

En esta sección se describen las diferentes implementaciones propuestas.

3.1. Implementación con Numba

A continuación se describen las diferentes optimizaciones que fueron consideradas en la implementación con Numba (ver Fig. 8).

Arreglos de NumPy Como estructura de datos se optó por arreglos de NumPy, debido a que permiten controlar la organización de memoria de los datos.

Opciones de compilación Se indicó que el código fuera compilado con los siguientes parámetros:

- `signature` (línea 1): A través de este parámetro se indicó que los arreglos sean contiguos en memoria para ayudar a Numba a detectar un mayor número de instrucciones vectoriales.
- `parallel=True` (línea 9): Activa la paralelización.

- `fastmath=True` (línea 9): Activa la relajación de precisión.
- `error_model="numpy"` (línea 9): Permite utilizar el modelo de división de NumPy, el cual evita la verificación de división por cero, y por ende, menos comparaciones en tiempo de ejecución.

Multi-hilado Se introdujo paralelismo a nivel de hilos a través de la sentencia `prange`. Particularmente, se crearon dos zonas paralelas; la primera se encarga de calcular la ley de atracción gravitacional de Newton y la integración de Verlet (ver Fig. 8a), mientras que la segunda simplemente actualiza la posición de los cuerpos (ver Fig. 8b).

Operaciones con tipos de datos simples Aunque las operaciones vectoriales de NumPy (*broadcasting*) simplifican la codificación, impactan negativamente en términos de rendimiento [16].

Vectorización Si bien se le indicó a Numba la utilización de instrucciones AVX-512 como parámetro de compilación, se constató que ya hacía uso correctamente de este conjunto de extensiones.

Threading layer Se varió la API de hilos a través de las *threading layers* que utiliza Numba para traducir las regiones paralelas. En particular, se seleccionó *omp* (OpenMP) por presentar mejores rendimientos.

3.2. Implementación con Cython

A continuación se describen las diferentes optimizaciones que fueron consideradas en la implementación con Cython (ver Fig. 9).

Opciones de compilación Inicialmente, se indicaron los siguientes parámetros de compilación:

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

Tipado explícito Se especificaron los tipos de datos de Cython, y particularmente, se indicó que los arreglos sean contiguos en memoria. Esta optimización permite que las variables sean transpiladas utilizando tipos de datos de C, evitando la API de CPython y el *overhead* generado por el último.

```
@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 the forces exerted on body i
        # by every other body
        for j in range(N):
            # Newton's law of Universal Gravitation
            dpos_x = positions_x[j] - positions_x[i]
            dpos_y = positions_y[j] - positions_y[i]
            dpos_z = positions_z[j] - positions_z[i]
            dsquared = (
                (dpos_x ** 2.0) + (dpos_y ** 2.0) +
                (dpos_z ** 2.0) + SOFT
            )
            gm = GRAVITY * masses[j] * masses[i]
            d32 = dsquared ** -1.5
            # Sum the forces
            forces_x += gm * d32 * dpos_x
            forces_y += gm * d32 * dpos_y
            forces_z += gm * d32 * dpos_z

        # Calculate acceleration of body i
        aceleration_x = forces_x / masses[i]
        aceleration_y = forces_y / masses[i]
        aceleration_z = forces_z / masses[i]
        # Calculate new velocity of body i
        velocities_x[i] += aceleration_x * DT / 2.0
        velocities_y[i] += aceleration_y * DT / 2.0
        velocities_z[i] += aceleration_z * DT / 2.0
        # Calculate new 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
```

(a) Función que calcula las posiciones de los cuerpos.

```
@njit(
    void(
        int64,
        double[::1], double[::1], double[::1],
        double[::1], double[::1], double[::1],
    ),
    fastmath=True,
    parallel=True,
    error_model="numpy",
)
def update_positions(
    N,
    positions_x, positions_y, positions_z,
    dp_x, dp_y, dp_z,
):
    # For every body that experienced a force
    for i in prange(N):
        # Update position of body i
        positions_x[i] += dp_x[i]
        positions_y[i] += dp_y[i]
        positions_z[i] += dp_z[i]
```

(b) Función que actualiza las posiciones de los cuerpos.

Figura 8: Implementación con Numba

Multi-hilado Se introduce paralelismo a nivel de hilos a través de la sentencia `prange` que provee Cython. 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. Por último, se desactivó el GIL a través del argumento `nogil` para permitir que los mismos se ejecuten de forma paralela.

4. Resultados Experimentales

4.1. Diseño experimental

Todas las pruebas fueron realizadas en un sistema equipado con 2×Intel Xeon Platinum 8276 de 28 núcleos (2 hilos hw por núcleo) y 256 GB de memoria RAM.

```
@cython.cdivision(False)
@cython.profile(False)
@cython.boundscheck(False)
@cython.wraparound(True)
cpdef void nbody(
    int N, int D, int T,
    double[:,:] positions_x, double[:,:] positions_y, double[:,:] positions_z,
    double[:,:] masses,
    double[:,:] velocities_x, double[:,:] velocities_y, double[:,:] velocities_z,
    double[:,:] dp_x, double[:,:] dp_y, double[:,:] dp_z,
):
    cdef double forces_x, forces_y, forces_z
    cdef double acceleration_x, acceleration_y, acceleration_z
    cdef double dpos_x, dpos_y, dpos_z
    cdef double dsquared, gm, d32
    cdef int i, j

    # For each discrete instant of time
    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 the forces exerted on body i by every other body
            for j in range(N):
                # Newton's law of Universal Gravitation
                dpos_x = positions_x[j] - positions_x[i]
                dpos_y = positions_y[j] - positions_y[i]
                dpos_z = positions_z[j] - positions_z[i]
                dsquared = (dpos_x ** 2.0) + (dpos_y ** 2.0)
                    + (dpos_z ** 2.0) + SOFT
                gm = GRAVITY * masses[j] * masses[i]
                d32 = dsquared ** -1.5
                # Add the forces
                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 acceleration of body i
            acceleration_x = forces_x / masses[i]
            acceleration_y = forces_y / masses[i]
            acceleration_z = forces_z / masses[i]
            # Calculate new velocity of body i
            velocities_x[i] += acceleration_x * DT / 2.0
            velocities_y[i] += acceleration_y * DT / 2.0
            velocities_z[i] += acceleration_z * DT / 2.0
            # Calculate new 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 every body that experienced a force
        for i in prange(N, nogil=True, schedule="static", num_threads=T):
            # Update position of body i
            positions_x[i] += dp_x[i]
            positions_y[i] += dp_y[i]
            positions_z[i] += dp_z[i]
```

Figura 9: Implementación con Cython.

El sistema operativo fue Ubuntu 20.04.2 LTS y el intérprete utilizado fue Python v3.8.10 junto con Numba v0.52.0 y NumPy v1.20.1. Por el otro lado, se utilizó Cython v0.29.22 con el compilador Intel v2022.0.0.20211123.

Para la evaluación de las implementaciones, se varió la carga de trabajo al usar diferentes números de cuerpos: $N = \{65536, 131072, 262144, 524288\}$ mientras que tanto el número de hilos (T=112) como el número de pasos de simulación (I=100) se mantuvieron fijos.

Por último, cabe destacar que la versión de Cython fue compilada utilizando el *flag* `-O3` junto con los siguientes *flags* adicionales: `march=native` (indica el uso de los flags más adecuados para el procesador subyacente) y `fp-model fast=2` (indica relajar la precisión de punto flotante).

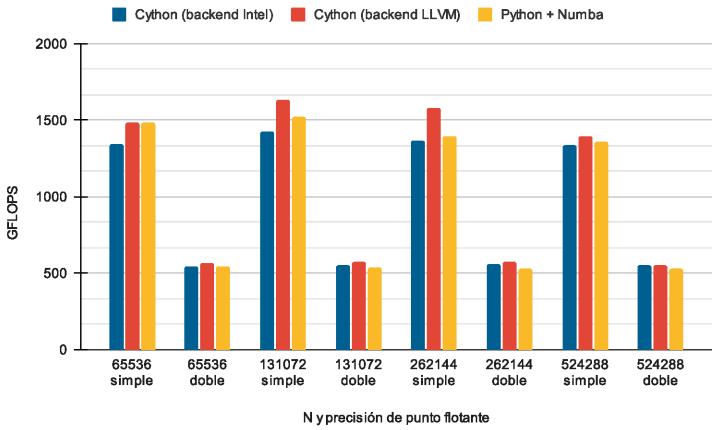

Figura 10: Comparación de rendimiento de las versiones finales entre Numba y Cython variando el tipo de dato y N.

4.2. Rendimiento

Al igual que en trabajos previos, se emplea la métrica GFLOPS para evaluar el rendimiento [16]. En la Fig. 10 se presenta una comparación entre las optimizaciones finales de Numba y Cython al variar la carga de trabajo y el tipo de dato. En el mismo, se puede observar que la incorporación de LLVM como *backend* mejoró las prestaciones de la versión anterior (*backend* propio de Intel); en particular, el rendimiento se incrementó (en promedio) 10 % y 3 % en simple y doble precisión, respectivamente. Por otro lado, resulta importante notar que la versión previa de Cython sólo lograba superar a Numba en precisión doble. Gracias a la mejora anterior, la nueva versión Cython resulta superior con ambos tipos de datos, logrando incrementos (en promedio) de 5 % y 6 % en simple y doble precisión, respectivamente.

4.3. Esfuerzo de Programación

En este trabajo el esfuerzo de programación se estimó en función del indicador SLOC (*Source Lines Of Code*), el cual permite contabilizar las líneas de código [12]. 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.

En la Fig. 11 se puede observar la cantidad de líneas de código de cada versión discriminando instrucciones, comentarios y líneas en blanco (medidas con la herramienta *cloc*[7]). La solución con Numba requirió 92 líneas de código, lo que

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

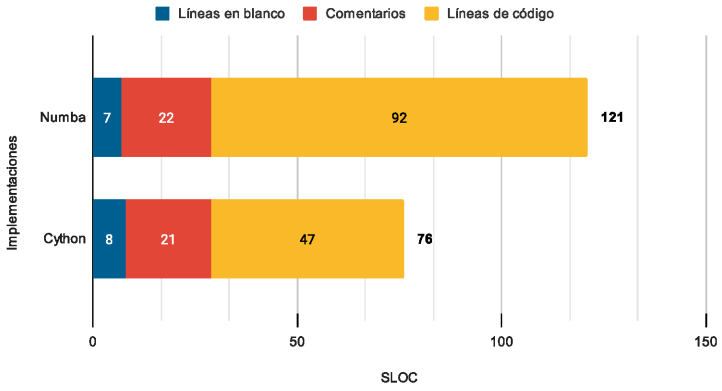

Figura 11: Cantidad de líneas de código de las implementaciones Numba y Cython para N-Body.

representa casi el doble que implementación de Cython. Sin embargo, hay que tener en cuenta que la primera contiene las opciones de compilación declaradas en la propia implementación (representa 32.6 % del código total) y, además, contiene llamadas a 2 funciones adicionales que actúan como zonas paralelas, mientras que en la implementación de Cython no se encuentra dicha modularización.

A continuación, se describen los aspectos cualitativos que fueron identificados al momento de desarrollar cada solución. Se puede mencionar 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 para lograrlo (por ej. `threading`).

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 Numba simplemente se precisó un archivo para almacenar el código fuente y un comando para ejecutar la simulación.

5. Conclusiones y Trabajo Futuro

En este trabajo se realizó una comparación más justa, actualizada y amplia de las prestaciones (rendimiento y esfuerzo de programación) de los traductores Numba y Cython para el caso de estudio *N-Body*.

Por un lado, los resultados muestran que la incorporación de LLVM como *backend* del compilador Intel para C repercute positivamente en el rendimiento de la versión Cython en comparación al *backend* clásico. A diferencia del estudio anterior, esta mejora lleva a que Cython resulte levemente superior a Numba tanto en doble como en simple precisión.

Por otro lado, en cuanto a esfuerzo de programación, 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:

- 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.

Referencias

1. 7. Decorators — Python Tips 0.1 documentation, `https://book.pythontips.com/en/latest/decorators.html`
2. Coiling Python Around Hybrid Quantum Systems, `https://www.nextplatform.com/2021/05/19/coiling-python-around-hybrid-quantum-systems/`
3. Cython: C-Extensions for Python, `https://cython.org/`
4. Home - OpenMP, `https://www.openmp.org/`
5. Home | Jython, `https://www.jython.org/`
6. Intel c/c++ compilers complete adoption of llvm, `https://www.intel.com/content/www/us/en/developer/articles/technical/adoption-of-llvm-complete-icx.html`

7. Microsoft's new research lab studies developer productivity and well-being | VentureBeat, https://venturebeat.com/2021/05/25/microsofts-new-research-lab-studies-developer-productivity-and-well-being/
8. Numba documentation — Numba 0.53.1-py3.7-linux-x86_64.egg documentation, https://numba.readthedocs.io/en/stable/index.html
9. NumPy, https://numpy.org/
10. PyPy, https://www.pypy.org/
11. What Is the Python Global Interpreter Lock (GIL)? – Real Python, https://realpython.com/python-gil/
12. Bhatt, K., Tarey, V., Patel, P., Mits, K.B., Ujjain, D.: Analysis of source lines of code (sloc) metric. International Journal of Emerging Technology and Advanced Engineering **2**(5), 150–154 (2012)
13. Cai, X., Langtangen, H.P., Moe, H.: On the Performance of the Python Programming Language for Serial and Parallel Scientific Computations. Scientific Programming **13**(1), 31–56 (2005). https://doi.org/10.1155/2005/619804
14. Gmys, J., Carneiro, T., Melab, N., Talbi, E.G., Tuyttens, D.: A comparative study of high-productivity high-performance programming languages for parallel metaheuristics. Swarm and Evolutionary Computation **57**, 100720 (Sep 2020). https://doi.org/10.1016/j.swevo.2020.100720
15. Marowka, A.: Python accelerators for high-performance computing. The Journal of Supercomputing **74**(4), 1449–1460 (Apr 2018). https://doi.org/10.1007/s11227-017-2213-5
16. Milla, A., Rucci, E.: Performance comparison of python translators for a multithreaded cpu-bound application. In: Pesado, P., Gil, G. (eds.) Computer Science – CACIC 2021. pp. 21–38. Springer International Publishing, Cham (2022)
17. Milla, A., Rucci, E.: Acelerando Código Científico en Python usando Numba. XXVII Congreso Argentino de Ciencias de la Computación (CACIC 2021) p. 12 (Oct 2021), http://sedici.unlp.edu.ar/handle/10915/126012
18. Roghult, A.: Benchmarking Python Interpreters: Measuring Performance of CPython, Cython, Jython and PyPy. Master's thesis, School of Computer Science and Communication, Royal Institute of Technology, Sweden (2016)
19. Rucci, E., Moreno, E., Pousa, A., Chichizola, F.: Optimization of the n-body simulation on intel's architectures based on avx-512 instruction set. In: Computer Science – CACIC 2019. pp. 37–52. Springer International Publishing (2020)
20. TIOBE Software BV: TIOBE Index for November 2021 (11 2021), https://www.tiobe.com/tiobe-index/
21. Varsha, M., Yashashree, S., Ramdas, D.K., Alex, S.A.: A Review of Existing Approaches to Increase the Computational Speed of the Python Language. International Journal of Research in Engineering, Science and Management (2019)
22. Wilbers, I., Langtangen, H.P., Odegard, A.: Using cython to speed up numerical python programs. In: Proceedings of MekIT. pp. 495–512 (2009)
23. Wilkens, F.: Evaluation of performance and productivity metrics of potential programming languages in the HPC environment. Bachelor's thesis, Faculty of Mathematics, Informatics und Natural Sciences, University of Hamburg, Germany (2015)