Análisis y Aplicación de la Criba de Eratóstenes

Rodrigo Vargas Sánchez

8 de junio de 2025

Resumen

Este informe explora la Criba de Eratóstenes, un algoritmo clásico para la identificación de números primos. Se estudia su fundamento teórico y se implementa tanto en bajo como en alto nivel, comparando sus tiempos de ejecución y eficiencia computacional. Finalmente, se analiza por qué se producen las diferencias de rendimiento.

Índice

1. Introducción

1.1. Definición y relevancia de los números primos

Los números primos son aquellos enteros mayores que 1 que solo tienen dos divisores positivos: el 1 y ellos mismos. Es decir, un número p es primo si no existe otro número entero positivo distinto de 1 y de p que lo divida exactamente. Ejemplos clásicos de números primos incluyen el 2, 3, 5, 7, 11 y 13.

La importancia de los números primos radica en que son los "bloques fundamentales"de los números naturales. Según el Teorema Fundamental de la Aritmética, todo número entero mayor que 1 puede expresarse de manera única como un producto de números primos, sin importar el orden de los factores. Esta propiedad convierte a los primos en piezas clave dentro de la teoría de números y en la base de muchas áreas de las matemáticas.

Además de su valor teórico, los números primos tienen aplicaciones prácticas muy importantes. En la actualidad, desempeñan un rol esencial en campos como la criptografía, especialmente en sistemas de cifrado como RSA, donde la dificultad de factorizar grandes números compuestos garantiza la seguridad de la información.

1.2. Breve historia de la Criba de Eratóstenes

La Criba de Eratóstenes es uno de los algoritmos más antiguos conocidos para encontrar números primos. Fue ideado por Eratóstenes de Cirene, un matemático, astrónomo y geógrafo griego que vivió entre los años 276 a.C. y 194 a.C. Su método consiste en eliminar iterativamente los múltiplos de cada número primo comenzando por el 2, lo que permite encontrar todos los primos menores a un número dado de manera eficiente.

A pesar de su simplicidad, la Criba de Eratóstenes destaca por su eficiencia en la generación de números primos cuando se aplica a conjuntos de datos moderadamente grandes. Su enfoque ha sido la base para muchos algoritmos más modernos de búsqueda de primos y continúa siendo utilizado tanto en la enseñanza como en aplicaciones computacionales.

1.3. Objetivo del informe

El presente informe tiene como objetivo principal analizar el algoritmo de la Criba de Eratóstenes desde una perspectiva tanto teórica como práctica. Para ello, se propone implementar el algoritmo en lenguajes de bajo y alto nivel, realizar pruebas de rendimiento, y comparar los tiempos de ejecución obtenidos. Se busca comprender cómo la elección del lenguaje y el nivel de abstracción influyen en la eficiencia del algoritmo y, a partir de los resultados, extraer conclusiones sobre su comportamiento computacional y su aplicabilidad.

2. Marco Teórico

2.1. Números primos

Definición formal. Un número primo es un número natural mayor que 1 que tiene exactamente dos divisores positivos distintos: el 1 y él mismo. Formalmente, se dice que un número $p \in \mathbb{N}$ es primo si:

$$\forall d \in \mathbb{N},\ d \mid p \Rightarrow d = 1 \vee d = p$$

Es decir, no existe ningún número natural d tal que $1 < d < p$ y $d \mid p$.

Propiedades clave.

- **Teorema Fundamental de la Aritmética:** Todo número natural mayor que 1 puede descomponerse de forma única (excepto por el orden de los factores) como producto de números primos.
- **Infinitud de los números primos:** Euclides demostró que existen infinitos números primos.
- **Todos los primos mayores que 2 son impares:** El número 2 es el único primo par, ya que cualquier otro número par es divisible por 2.
- **No siguen un patrón aritmético simple:** Aunque parecen distribuidos de forma caótica, su densidad disminuye de manera predecible según el Teorema de los Números Primos.

Aplicaciones relevantes.

- **Criptografía:** Algoritmos como RSA dependen de la factorización de grandes números compuestos, una tarea computacionalmente difícil.
- **Generación de números pseudoaleatorios:** Algunos algoritmos se apoyan en propiedades de los primos para generar secuencias más seguras.
- **Sistemas de codificación y compresión:** Los números primos ayudan a mejorar la eficiencia y robustez en la detección y corrección de errores.
- **Algoritmos distribuidos y hashing:** Se emplean primos para minimizar colisiones y distribuir datos de forma uniforme.

2.2. Criba de Eratóstenes

La Criba de Eratóstenes es un algoritmo eficiente para encontrar todos los números primos menores o iguales a un número n. El algoritmo funciona de la siguiente manera:

1. Se crea una lista de números enteros del 2 al n.
2. Comenzando desde el primer número (2), se marca como primo. Luego, se elimina de la lista todos sus múltiplos (excepto el mismo 2), ya que estos no son primos.
3. Se repite el proceso con el siguiente número no marcado en la lista. Este número será el siguiente primo. Se eliminan de la lista todos sus múltiplos.
4. El proceso continúa hasta llegar a un número cuya raíz cuadrada es mayor que n, ya que cualquier número compuesto mayor que n tendrá al menos un factor menor o igual a $\sqrt{n}$, que ya habrá sido marcado.
5. Los números restantes en la lista son todos primos.

Este proceso asegura que solo los números que no son múltiplos de ningún otro número más pequeño que n sean marcados como primos.

Ejemplo ilustrativo. Supongamos que queremos encontrar todos los números primos menores o iguales a 30 utilizando la Criba de Eratóstenes.

1. Comenzamos con la lista de números del 2 al 30:

   $$2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30$$

2. El primer número es 2. Marcamos los múltiplos de 2 (4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30).

3. El siguiente número no marcado es 3. Marcamos los múltiplos de 3 (9, 12, 15, 18, 21, 24, 27, 30).

4. El siguiente número no marcado es 5. Marcamos los múltiplos de 5 (10, 15, 20, 25, 30).

5. Continuamos con 7, marcando los múltiplos (14, 21, 28).

6. Al llegar a 11, ya no es necesario continuar, ya que $11^2 = 121$ es mayor que 30.

7. Los números no marcados en la lista son los primos:

   $$2, 3, 5, 7, 11, 13, 17, 19, 23, 29$$

Complejidad computacional: $O(n \log \log n)$. La Criba de Eratóstenes es un algoritmo muy eficiente para encontrar números primos en un rango dado. Su complejidad computacional es $O(n \log \log n)$, donde n es el número hasta el cual se buscan los primos. Esto se debe a que:

- En cada iteración del algoritmo, se marcan múltiplos de cada número primo. El número de múltiplos marcados para cada primo p es aproximadamente $\frac{n}{p}$.
- Al sumar las contribuciones de todos los primos, se obtiene una serie logarítmica que lleva a la complejidad de $O(n \log \log n)$, lo que hace que el algoritmo sea mucho más eficiente que otros métodos de prueba de primalidad.

Este rendimiento es significativamente mejor que el de otros enfoques ingenuos de búsqueda de primos, como la comprobación de cada número indivisibilidad por todos los números menores que él.

2.3. Criba Segmentada de Eratóstenes

La Criba Segmentada de Eratóstenes es una optimización del algoritmo clásico de la Criba de Eratóstenes, diseñada para reducir el uso de memoria cuando se busca generar números primos en un rango grande. En lugar de trabajar sobre una única lista de tamaño n, la versión segmentada divide el rango en bloques más pequeños (segmentos), que se procesan secuencialmente.

Motivación. Cuando n es muy grande, la lista de tamaño n necesaria para la criba clásica puede superar la capacidad de memoria del sistema. La criba segmentada permite procesar el rango por partes, manteniendo en memoria solo un segmento del tamaño de $\sqrt{n}$ a la vez, lo que hace posible ejecutar la criba en máquinas con recursos limitados.

Funcionamiento. El algoritmo segmentado funciona de la siguiente manera:

1. Se calculan primero todos los primos menores o iguales a $\sqrt{n}$ utilizando la criba clásica. Estos primos se usan como base para marcar múltiplos en los segmentos.
2. El rango $[2, n]$ se divide en segmentos de tamaño conveniente (por ejemplo, $\Delta = 10^6$).
3. Para cada segmento $[l, r]$, se crea una lista local de booleanos que representa si un número está marcado como primo.
4. Por cada primo $p \leq \sqrt{n}$, se marcan los múltiplos de p dentro del segmento actual. Esto se realiza comenzando desde el múltiplo más pequeño de p que se encuentra dentro del segmento.
5. Al finalizar el marcado en el segmento, los números no marcados son primos dentro de ese bloque.
6. Se repite el proceso para el siguiente segmento hasta cubrir todo el rango hasta n.

Ventajas. La principal ventaja de la criba segmentada es su bajo consumo de memoria, ya que solo requiere espacio proporcional al tamaño de un segmento, en lugar de mantener todo el rango en memoria. Esto permite encontrar números primos en rangos muy grandes sin comprometer los recursos del sistema.

Complejidad. La complejidad temporal de la criba segmentada sigue siendo $O(n \log \log n)$, al igual que la criba clásica, pero con una eficiencia espacial mucho mejor. Esta optimización es especialmente útil en implementaciones donde se necesita generar primos en intervalos grandes, como en criptografía o teoría de números computacional.

3. Implementación

3.1. Selección de algoritmo

Para la generación de números primos se implementó la Criba Segmentada de Eratóstenes en lugar de la versión clásica. Esta decisión se debe a las limitaciones de memoria presentes al ejecutar el algoritmo en un entorno tipo notebook, donde los recursos disponibles no permiten trabajar con listas de gran tamaño. La implementación clásica requiere mantener en memoria una lista de tamaño n, lo cual resulta inviable para valores grandes de n, ya que puede exceder la capacidad del sistema y provocar errores de ejecución. En cambio, la versión segmentada permite dividir el rango en bloques más pequeños y procesarlos de manera secuencial, lo que reduce considerablemente el uso de memoria sin sacrificar eficiencia en el tiempo de ejecución.

3.2. Especificaciones del entorno de ejecución

Característica	**Detalle**
Sistema Operativo	Ubuntu 22.04.1 LTS (Linux 6.8.0-60-generic)
Arquitectura	x86_64
Modelo del Procesador	Intel Core i7-1165G7 (11ª Gen) @ 2.80GHz
Núcleos (Físicos)	4 núcleos (1 socket)
Hilos por Núcleo	2 hilos por núcleo
Frecuencia Máxima del CPU	4.7 GHz
Frecuencia Mínima del CPU	400 MHz
Total de CPU(s)	8 (4 núcleos * 2 hilos por núcleo)
Caché L1 (Datos)	192 KiB (4 instancias)
Caché L1 (Instrucciones)	128 KiB (4 instancias)
Caché L2	5 MiB (4 instancias)
Caché L3	12 MiB (1 instancia)
Tarjeta Gráfica	Intel Iris Xe Graphics (integrada en el i7-1165G7)

3.3. Pseudocodigo de Criba Segmentada de Erastótenes

Algorithm 1: Criba Segmentada

```
1  Function SimpleSieve(limit):
2      Crear un arreglo booleano primos[0..limit] y asignar true a todos
3      primos[0] ← false, primos[1] ← false
4      for i ← 2 to ⌊√limit⌋ do
5          if primos[i] es true then
6              for j ← i × i to limit i do
7                  primos[j] ← false
8      return lista de índices i donde primos[i] es true
9  Function SegmentedSieve(n, tam_segmento):
10     limite ← ⌊√n⌋
11     primosBase ← SimpleSieve(limite)
12     primosFinal ← lista vacía
13     for low ← 0 to n tam_segmento do
14         high ← mín(low + tam_segmento − 1, n)
15         Crear arreglo booleano segmento[0..high-low] y asignar true a todos
16         foreach p en primosBase do
17             start ← máx(p × p, ⌈low/p⌉ × p)
18             for m ← start to high p do
19                 segmento[m - low] ← false
20         for i ← máx(2, low) to high do
21             if segmento[i - low] es true then
22                 Agregar i a primosFinal
23     return primosFinal
```

3.4. Implementación en Python

Descripción del enfoque:

La implementación en Python utiliza el algoritmo de Criba Segmentada para calcular los números primos hasta un límite n. Primero se genera una lista de primos base mediante la criba de Eratóstenes hasta $\sqrt{n}$. Luego, el intervalo $[0, n]$ se divide en segmentos de tamaño fijo, y en cada segmento se marcan los múltiplos de los primos base como no primos. De esta manera, se reduce el consumo de memoria y se mejora el rendimiento al trabajar por bloques.

Fragmento representativo del código:

Listing 1: Segmented Sieve en Python

```
def segmented_sieve(n, segment_size=1000):
    limit = int(math.sqrt(n))
    base_primes = simple_sieve(limit)

    primes = []
```

```
    for low in range(0, n + 1, segment_size):
        high = min(low + segment_size - 1, n)
        segment = [True] * (high - low + 1)

        for p in base_primes:
            start = max(p * p, (low + p - 1) // p * p)
            for multiple in range(start, high + 1, p):
                segment[multiple - low] = False

        for i in range(max(low, 2), high + 1):
            if segment[i - low]:
                primes.append(i)

    return primes
```

Medición de rendimiento:
Se utilizaron los módulos `time` y `tracemalloc` para medir el tiempo de ejecución y el consumo de memoria, respectivamente. Estas herramientas permitieron obtener datos comparables con las demás implementaciones en otros lenguajes.

3.5. Perfilado de rendimiento del código Python

Para evaluar el desempeño de la implementación en Python del algoritmo de criba segmentada, se realizó un perfilado detallado utilizando *line profiling* con un valor de $n = 10{,}000{,}000$ y un tamaño de segmento de aproximadamente $3,162$.

Resultados generales:

- Número de primos encontrados: 664,579
- Tiempo total de ejecución: 54,441.72 ms (aprox. 54.4 segundos)
- Memoria pico utilizada: 23.46 MB

Resumen del análisis línea a línea:

- La función `simple_sieve` para generar los primos base consume alrededor de 2 ms, un tiempo muy bajo comparado con el total.
- La inicialización y creación del segmento en cada iteración (líneas 23) representa aproximadamente un 0.2 % del tiempo, pero ocurre 3,163 veces.
- Los bucles internos son los más costosos:
  - La línea que calcula el inicio para marcar múltiplos (línea 26) consume un 8.2 % del tiempo.
  - El bucle que marca los múltiplos como no primos (líneas 27 y 28) consume conjuntamente más del 63 % del tiempo total. Esto indica que la operación de marcado es el cuello de botella principal.

- La iteración para agregar primos al resultado (líneas 30-32) representa un 26.8 % del tiempo.

El perfilado revela que el paso más costoso es la marcación de múltiplos dentro de cada segmento, lo que sugiere realizar una optimización en esa parte.

Datos detallados del perfilado:

```
Total time: 32.77 s
Primos encontrados: 664,579
Tiempo: 54,441.72 ms
Memoria pico: 23.46 MB

Línea clave        % Tiempo     Descripción
-----------------------------------------------
26                 8.2%       Cálculo del inicio para marcado
27, 28             63.1%      Marcado de múltiplos como no primos
30-32              26.8%      Iteración para agregar primos al resultado
```

3.6. Implementación en Python con NumPy

La versión con NumPy del algoritmo de criba segmentada utiliza arreglos booleanos optimizados para mejorar la manipulación y marcado de números primos. Esto permite aprovechar operaciones vectorizadas que aceleran considerablemente la ejecución en comparación con listas nativas de Python.

Diferencias principales respecto a la versión pura en Python:

- Uso de `numpy.ones` para crear arrays booleanos que representan la criba, lo que permite aplicar operaciones sobre segmentos completos sin necesidad de iterar elemento por elemento.
- Marcado de múltiplos usando slicing y funciones vectorizadas, como `sieve[i*i:limit+1:i] = False` y `segment[segment_indices] = False`, en lugar de bucles anidados explícitos.
- La función `numpy.nonzero` se utiliza para extraer los índices que representan números primos, evitando bucles y comprobaciones individuales.

Fragmento representativo del código:

```
def simple_sieve(limit):
    sieve = np.ones(limit + 1, dtype=bool)
    sieve[:2] = False
    for i in range(2, int(math.sqrt(limit)) + 1):
        if sieve[i]:
            sieve[i*i:limit+1:i] = False
    return np.nonzero(sieve)[0]

def segmented_sieve(n, segment_size=100000):
    limit = int(math.sqrt(n))
```

```
    base_primes = simple_sieve(limit)

    primes = []

    for low in range(0, n + 1, segment_size):
        high = min(low + segment_size - 1, n)
        segment = np.ones(high - low + 1, dtype=bool)

        for p in base_primes:
            start = max(p * p, (low + p - 1) // p * p)
            segment_indices = np.arange(start, high + 1, p) - low
            segment[segment_indices] = False

        segment_primes = np.nonzero(segment)[0] + low
        segment_primes = segment_primes[segment_primes >= 2]
        primes.extend(segment_primes.tolist())

    return primes
```

3.7. Implementación en C++

La implementación en C++ del algoritmo de criba segmentada sigue los mismos principios algorítmicos que sus equivalentes en Python, pero aprovechando características del lenguaje como tipado estático, estructuras de datos eficientes y control explícito de memoria. Esto permite obtener mejoras importantes en rendimiento y eficiencia en tiempo de ejecución.

Diferencias clave respecto a las versiones en Python:

- Utiliza `std::vector<bool>` para representar la criba de números primos, lo que ofrece un uso de memoria más compacto que las listas o arrays booleanos estándar.
- El ciclo principal se ejecuta de forma similar, pero con acceso más directo a memoria y sin la sobrecarga de tipado dinámico de Python.

Fragmento representativo del código:

```
std::vector<int> segmented_sieve(int n, int segment_size) {
    int limit = static_cast<int>(std::sqrt(n));
    std::vector<int> base_primes = simple_sieve(limit);
    std::vector<int> primes;

    for (int low = 0; low <= n; low += segment_size) {
        int high = std::min(low + segment_size - 1, n);
        std::vector<bool> segment(high - low + 1, true);

        for (int p : base_primes) {
            int start = std::max(p * p, ((low + p - 1) / p) * p);
            for (int multiple = start; multiple <= high; multiple += p) {
                segment[multiple - low] = false;
```

```
            }
        }

        for (int i = std::max(low, 2); i <= high; i++) {
            if (segment[i - low]) {
                primes.push_back(i);
            }
        }
    }

    return primes;
}
```

Consideracion adicional:

- La función `simple_sieve` también se implementó de forma similar a la versión de Python, pero usando bucles tradicionales y `std::vector`.

3.8. Implementación en C++ con paralelización usando OpenMP

Esta versión de la criba segmentada en C++ introduce paralelismo a través de la biblioteca OpenMP, lo cual permite aprovechar múltiples núcleos del procesador para acelerar el proceso de búsqueda de números primos. Se mantiene la lógica del algoritmo secuencial, pero se paraleliza el procesamiento de cada segmento del rango a cribar.

Diferencias clave respecto a la versión secuencial:

- Se utiliza la directiva `#pragma omp parallel for` para paralelizar el ciclo principal que recorre los segmentos.
- Cada hilo opera sobre su propio vector de primos `primes_per_thread[segment]`, evitando condiciones de carrera y garantizando la independencia entre hilos.
- Al final del proceso, los resultados parciales de cada hilo se combinan en un único vector global de primos.

Fragmento representativo del código:

```
std::vector<int> segmented_sieve(int n, int segment_size) {
    int limit = static_cast<int>(std::sqrt(n));
    std::vector<int> base_primes = simple_sieve(limit);
    std::vector<int> primes;

    int num_segments = (n + segment_size - 1) / segment_size;
    std::vector<std::vector<int>> primes_per_thread(num_segments);

    #pragma omp parallel for schedule(dynamic)
    for (int segment = 0; segment < num_segments; segment++) {
        int low = segment * segment_size;
```

```
        int high = std::min(low + segment_size - 1, n);
        std::vector<bool> segment_vec(high - low + 1, true);

        for (int p : base_primes) {
            int start = std::max(p * p, ((low + p - 1) / p) * p);
            for (int multiple = start; multiple <= high; multiple += p) {
                segment_vec[multiple - low] = false;
            }
        }

        for (int i = std::max(low, 2); i <= high; i++) {
            if (segment_vec[i - low]) {
                primes_per_thread[segment].push_back(i);
            }
        }
    }

    for (const auto& segment_primes : primes_per_thread) {
        primes.insert(primes.end(), segment_primes.begin(), segment_primes.end());
    }

    return primes;
}
```

Consideraciones adicionales:

- La paralelización se adapta mejor a valores grandes de `n` y segmentos de tamaño adecuado, donde el trabajo de cada hilo es suficientemente costoso para justificar la sobrecarga del paralelismo.

4. Resultados

4.1. Método de evaluación de rendimiento

Para evaluar el rendimiento de las distintas implementaciones del algoritmo de Criba de Eratóstenes segmentada, se utilizó un procedimiento uniforme en todos los lenguajes y versiones implementadas. Cada código fue ejecutado midiendo el tiempo de ejecución.

El tamaño de entrada n se varió de forma creciente, comenzando desde 10^4 hasta 10^8, aumentando en potencias de 10. En cada caso, se eligió un `segment_size` igual a $\sqrt{n}$, lo que permite balancear el uso de memoria con el rendimiento computacional.

El siguiente pseudocódigo resume el procedimiento empleado para las mediciones:

```
función run_tests():
    ns ← [10_000 × 10^i para i en 0..6]  # n desde 10^4 hasta 10^8
    para cada n en ns:
        segment_size ← raíz cuadrada de n
```

```
iniciar trazado de memoria
tiempo_inicio ← tiempo actual

primos ← criba_segmentada(n, segment_size)

tiempo_fin ← tiempo actual
```

Este mismo procedimiento fue adaptado a cada lenguaje respetando sus respectivas bibliotecas para medir tiempo. En Python, se utilizó `time` y en C++ se usó `std::chrono`.

4.2. Gráficos de resultados

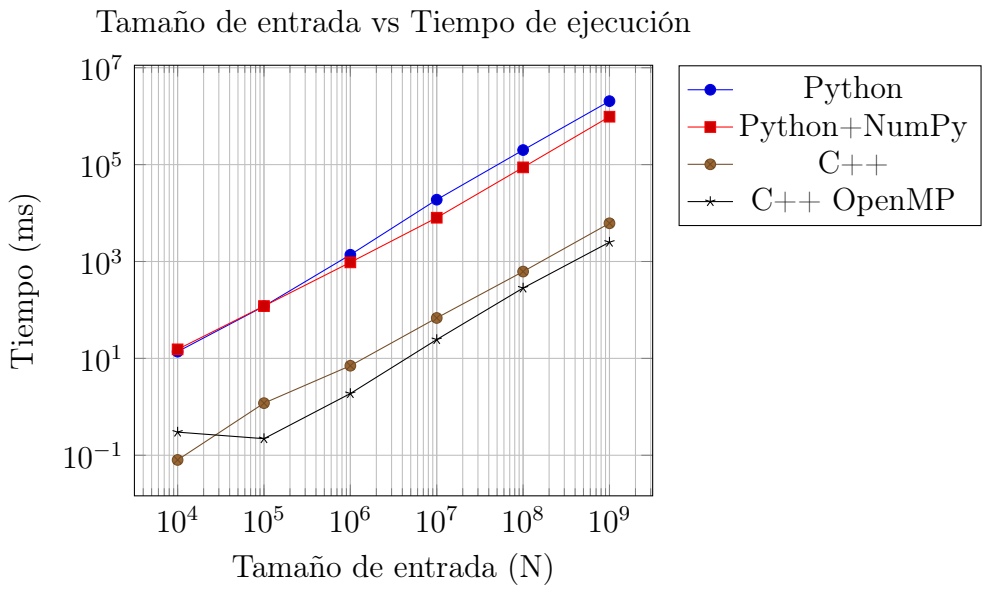

Figura 1: Tiempo de ejecución según tamaño de entrada. Escala log-log.

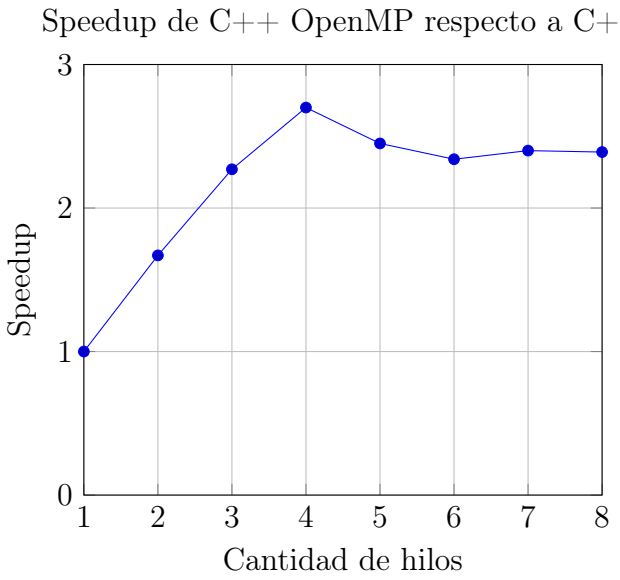

Figura 2: Speedup de OpenMP respecto a versión secuencial en C++ en función de cantidad de hilos

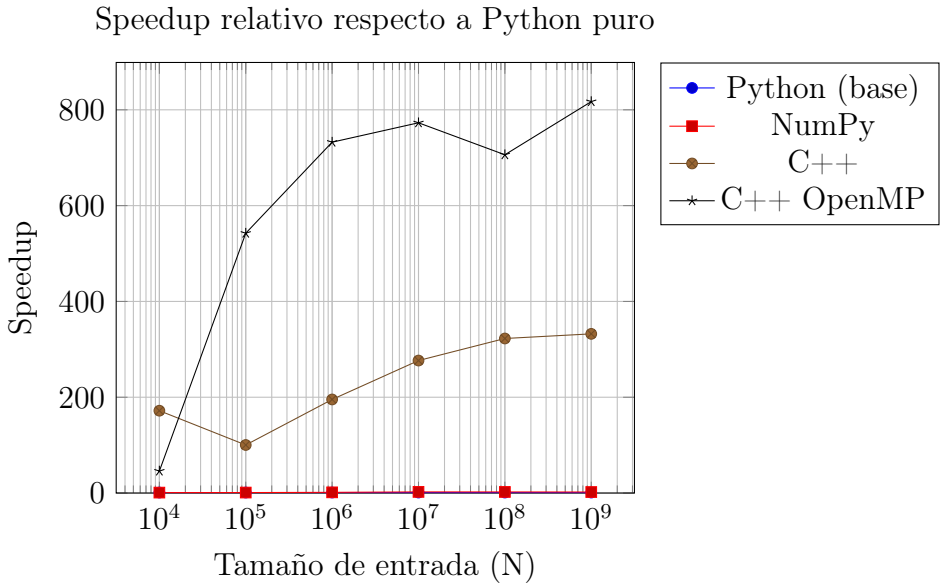

Figura 3: Speedup relativo respecto a Python puro

5. Análisis y Discusión

Los resultados obtenidos a partir de las mediciones de tiempo permiten analizar el impacto del lenguaje de programación y las técnicas de optimización utilizadas.

En primer lugar, se observa que la implementación en C++ presenta el mejor rendimiento en todos los tamaños de entrada evaluados, superando ampliamente a las versiones en Python. Esta diferencia se debe en gran medida a la naturaleza compilada de C++, que permite una ejecución más cercana al hardware, mientras que Python es un lenguaje interpretado con una sobrecarga considerable en operaciones intensivas.

La versión paralela en C++ utilizando OpenMP logra una mejora significativa en el rendimiento respecto a la versión secuencial, con un speedup máximo cercano a 2.7 al utilizar 4 hilos. Sin embargo, se evidencia que el rendimiento no escala linealmente con el número de hilos. A partir de 5 hilos, el *speedup* comienza a disminuir levemente, lo que sugiere la presencia de una saturación del rendimiento debido a la sobrecarga de sincronización y al uso compartido de recursos como la memoria.

En cuanto a las implementaciones en Python, se aprecia que el uso de NumPy introduce ciertas mejoras en los tiempos de ejecución para entradas grandes. Esto se debe a que NumPy permite operaciones vectorizadas y está implementado en C, lo que reduce la sobrecarga del intérprete de Python. No obstante, incluso con esta optimización, Python+NumPy es notablemente más lento que C++ en órdenes de magnitud.

El gráfico de *speedup* relativo respecto a Python evidencia esta diferencia de rendimiento: C++ supera a Python en órdenes de magnitud para tamaños grandes, mientras que la versión con NumPy muestra una mejora progresiva, aunque más moderada. La versión paralela con OpenMP presenta el mayor *speedup* relativo en todos los tamaños de entrada evaluados, lo cual confirma la efectividad de la paralelización para este tipo de algoritmos altamente computacionales.

En resumen, los resultados demuestran que, si bien Python es útil para prototipos y pruebas rápidas, su rendimiento para tareas intensivas como el cálculo de números primos mediante la Criba de Eratóstenes segmentada es considerablemente inferior. Por otro lado, las implementaciones en C++ especialmente cuando se aprovecha la paralelización con OpenMP ofrecen una solución mucho más eficiente y escalable.

6. Conclusión

En este estudio se evaluaron cuatro implementaciones del algoritmo de Criba de Eratóstenes segmentada: Python puro, Python con NumPy, C++ secuencial y C++ paralelo utilizando OpenMP. Los resultados muestran diferencias significativas en el rendimiento, evidenciando el impacto del lenguaje y las técnicas de optimización empleadas.

La implementación en C++ secuencial fue considerablemente más rápida que las versiones en Python, debido a la eficiencia propia de un lenguaje compilado. La paralelización mediante OpenMP mejoró aún más los tiempos de ejecución, logrando un *speedup* relevante hasta un cierto número de hilos, aunque con limitaciones por la sobrecarga y recursos compartidos.

Por su parte, el uso de NumPy en Python aportó mejoras respecto a Python puro, especialmente en entradas de mayor tamaño, pero no logró acercarse al rendimiento de C++.

En conclusión, para aplicaciones que requieren procesamiento intensivo y escalabilidad, C++ con paralelización es la opción más adecuada. Python sigue siendo útil para desarrollo rápido y prototipado, pero presenta limitaciones significativas en eficiencia para este tipo de algoritmos.

Este análisis resalta la importancia de seleccionar herramientas y técnicas apropiadas según las demandas computacionales del problema.