UNIVERSIDAD TÉCNICA DEL NORTE

Facultad de Ingeniería en Ciencias Aplicadas

Carrera de Ingeniería en Sistemas Computacionales

IMPLEMENTACIÓN DE ALGORITMOS SOBRE ARQUITECTURA MULTINÚCLEO PARA OPTIMIZAR EL ALTO COSTE COMPUTACIONAL AL PROCESAR GRANDES VOLÚMENES DE DATOS

Trabajo de grado previo a la obtención del título de Ingeniero en Sistemas Computacionales

Autor:

Neider Fabricio Requene Estacio

Tutor:

MSc. Cosme Ortega

Ibarra – Ecuador

2022

UNIVERSIDAD TÉCNICA DEL NORTE

BIBLIOTECA UNIVERSITARIA

AUTORIZACIÓN DE USO Y PUBLICACIÓN A FAVOR DE LA UNIVERSIDAD TÉCNICA DEL NORTE

1. IDENTIFICACIÓN DE LA OBRA

En cumplimiento del Art. 144 de la Ley de Educación Superior, hago la entrega del presente trabajo a la Universidad Técnica del Norte para que sea publicado en el Repositorio Digital Institucional, para lo cual pongo a disposición la siguiente información:

DATOS DE CONTACTO			
CÉDULA DE IDENTIDAD:	0803936020		
APELLIDOS Y NOMBRES:	Requene Estacio Neider Fabricio		
DIRECCIÓN:	Sánchez y Cifuentes 14-65		
EMAIL:	nfrequenee@utn.edu.ec		
TELÉFONO FIJO:		**TELÉFONO MÓVIL:**	0984896158

DATOS DE LA OBRA	
TÍTULO:	Implementación de algoritmos sobre arquitectura multinúcleo para optimizar el alto coste computacional al procesar grandes volúmenes de datos
AUTOR (ES):	Requene Estacio Neider Fabricio
FECHA: DD/MM/AAAA	11-10-2022
SOLO PARA TRABAJOS DE GRADO	
PROGRAMA:	[X] **PREGRADO** ☐ **POSGRADO**
TITULO POR EL QUE OPTA:	Implementación de algoritmos sobre arquitectura multinúcleo para optimizar el alto coste computacional al procesar grandes volúmenes de datos
ASESOR /DIRECTOR:	Msc. Ortega Bustamante Cosme MacArthur

2. CONSTANCIAS

El autor (es) manifiesta (n) que la obra objeto de la presente autorización es original y se la desarrolló, sin violar derechos de autor de terceros, por lo tanto, la obra es original y que es (son) el (los) titular (es) de los derechos patrimoniales, por lo que asume (n) la responsabilidad sobre el contenido de la misma y saldrá (n) en defensa de la Universidad en caso de reclamación por parte de terceros.

Ibarra, a los 11 días del mes de octubre de 2022

EL AUTOR:

(Firma).....

Nombre: Requene Estacio Neider Fabricio

CERTIFICADO DEL DIRECTOR DE TRABAJO DE GRADO

UNIVERSIDAD TÉCNICA DEL NORTE

FACULTAD DE INGENIERÍA EN CIENCIAS APLICADAS

CERTIFICACIÓN DEL DIRECTOR

Por medio del presente yo Ing. Cosme Ortega, MSc, certifico que el Sr. Requene Estacio Neider Fabricio, portador de la cédula de ciudadanía Nro. 0803936020. Ha trabajado en el desarrollo del proyecto de tesis "Implementación de algoritmos sobre arquitectura multinúcleo para optimizar el alto coste computacional al procesar grandes volúmenes de datos", previo a la obtención del título de Ingeniería en Sistemas Computacionales, lo cual ha realizado en su total responsabilidad. Es todo cuanto puedo certificar en honor a la verdad.

Atentamente:

1001580396 COSME MACARTHUR ORTEGA BUSTAMANTE

Firmado digitalmente por 1001580396 COSME MACARTHUR ORTEGA BUSTAMANTE
Fecha: 2022.10.11 16:28:03 -05'00'

Ing. Cosme Ortega, MSc

DIRECTOR DE TESIS

Dedicatoria

A Dios, que me ha guiado en cada uno de los pasos que he dado en mi vida.

A mis padres: **Nielsen Requene** y **Cristina Estacio**, quienes han ayudado en mi formación desde niño, inculcando valores y principios. Ya se acabó la espera, sé que ha pasado bastante tiempo y por eso se los dedico a ustedes.

A mis tíos: **Ruth Ramírez** y **Andrés Valencia,** quienes me acompañaron durante el inicio de la carrera y en gran medida me ayudaron a final este proceso.

A **toda mi familia**, quiénes de una u otra manera se hicieron presentes con palabras de aliento, para seguir adelante y llegar a la meta.

Al mi **director de tesis**, quién fue una pieza fundamental para seguir adelante en la meta propuesta gracias por su apoyo y dedicación para culminar.

Neider Fabricio Requene Estacio

Agradecimiento

Agradezco primeramente a Dios por la familia, amigos y docentes que formaron parte de este arduo proceso.

A la Facultad de Ingeniería en Ciencias Aplicadas (FICA), de la Universidad Técnica del Norte y a cada uno de los señores docentes y personal administrativo.

De manera especial al Msc. Cosme Ortega, tutor del trabajo de grado quien con su apoyo incondicional me guío para la elaboración y culminación del presente trabajo.

A los excelentes docentes que con su empeño y de dedicación sembraron su granito de arena para lograr ser un buen profesional.

A mis compañeros quienes a lo largo de la carrera me brindaron la ayuda necesaria cada día en cada paso que se dio.

A la gloriosa Universidad Técnica del Norte por brindarme sus conocimientos y la dicha de poder conocer a tan buenos docentes y amigos.

Tabla de contenidos

Índices de figuras

Índice de tablas

Resumen

El proyecto plateado tiene la finalidad de presentar algunas de las técnicas de programación para optimizar algoritmos de alto costo computacional.

Se realizó un marco teórico sobre la complejidad algorítmica, los procesadores multinúcleo y los lenguajes de programación que permiten realizar procesamiento en paralelo. Este se realizó con una búsqueda mixta entre artículos científicos y bases de datos de confianza de desarrolladores.

Partiendo del marco teórico se seleccionó los tres lenguajes de programación: Python, C y Java. De estos se realizó un análisis de su rendimiento de en forma secuencial y se realizó las optimizaciones con los lenguajes Python y C.

En Python se realizó la implementación del algoritmo de la Matriz Distancia Euclidiana con cinco variaciones, haciendo uso de los tipos de datos nativos del lenguaje, además de librerías optimizadas para el procesamiento matricial como lo son Numpy y Numba.

Para el lenguaje C, las optimizaciones se las realizó mediante directivas de compilador. Además, se hizo uso de las directrices de OpenMP para aplicar el multiprocesamiento.

El resultado de las optimizaciones se evaluó en base al tiempo de procesamiento y la aceleración, donde la implementación con la librería Numba resultó ser la óptima para un volumen de datos grande. Mientras, que la optimización con el lenguaje C presentó la óptima para un volumen de datos pequeño.

INTRODUCCIÓN

Antecedentes

Debido a la alta carga computacional por el incremento de la cantidad de datos para ser procesados, existe la necesidad de optimizar los tiempos en la ejecución de algoritmos, para esto es necesario utilizar todo el potencial que ofrece una computadora. Una de las soluciones que se plantea es el diseño e implementación de algoritmos en paralelo (Castillos Reyes, Acevedo Martínez, & Luzua Farias, 2016).

Con el paso de los años la evolución de los procesadores ha ido mejorando su rendimiento debido a su integración con arquitecturas multinúcleo, brindando la capacidad de multiprocesamiento para contribuir con nuevas técnicas de programación para aquellos que se dedican al desarrollo de software (Jiménez Cancino, 2009). Los procesadores multinúcleo combinan dos a más microprocesadores que habitualmente se les encuentra en circuitos integrados, exhibiendo una cierta forma de paralelismo a nivel de hilos, con múltiples procesos ejecutándose al mismo tiempo unidos por un canal de alta velocidad y equilibrando la carga de trabajo entre ellos (Soto, 2017).

Situación actual

Existen programas de computación complejos como por ejemplo la predicción del clima donde se maneja grandes cantidades de datos que necesitan ser evaluados e interpretados en el menor tiempo posible, teniendo un alto consumo de recursos, el cual crea una brecha en los tiempos de ejecución, por lo cual se establecen algoritmos eficientes para reducir tanto, tiempos como recursos (Díaz, 2010).

En la actualidad las empresas que brindan servicios tecnológicos se han encontrado con un gran problema respecto al crecimiento y manejo de enormes cantidades de datos, los cuales necesitan de un alto presupuesto económico para su ejecución, para algunas empresas esto no es viable aún, peor para los clientes (investigadores, comunidad) que hacen uso de los servicios, una de las formas de minimizar el excesivo costo y permitir el avance de las empresas es la aplicación de procesamiento en paralelo (Trujillo Rasúa, 2009).

Prospectiva

La propuesta de la investigación permitirá optimizar la ejecución de códigos complejos, como por ejemplo algoritmos de matemática computacional, debido a que con el pasar del tiempo los programas para computadoras personales se vuelven cada vez más

robustos, por lo cual se necesita un alto consumo en tiempo y recursos creando una necesidad de generar algoritmos que permitan ahorrar excesivos costes de computación.

Con el fin de apoyar a la comunidad de investigación científica sobre temas de optimización y uso de poder computacional se crea un proyecto de investigación sobre la aplicación de algoritmos eficientes en procesadores multinúcleo.

Planteamiento del problema

Existe la necesidad de reducir costos computacionales como son tiempo de ejecución y recursos de procesamiento, por lo que se hace necesario usar tecnologías de procesamiento multinúcleo en equipos personales. Debido a que existe una brecha de conocimiento para el diseño e implementación de algoritmos complejos para el procesamiento de grandes volúmenes de información. Por lo que es necesario realizar una investigación de optimización de algoritmos en procesadores multinúcleo.

Para poder definir el diagrama de Planteamiento de Problema se utilizó el instrumento de investigación de identificación y clasificación de problemas (Matríz Vester). ver **¡Error! No se encuentra el origen de la referencia.**.

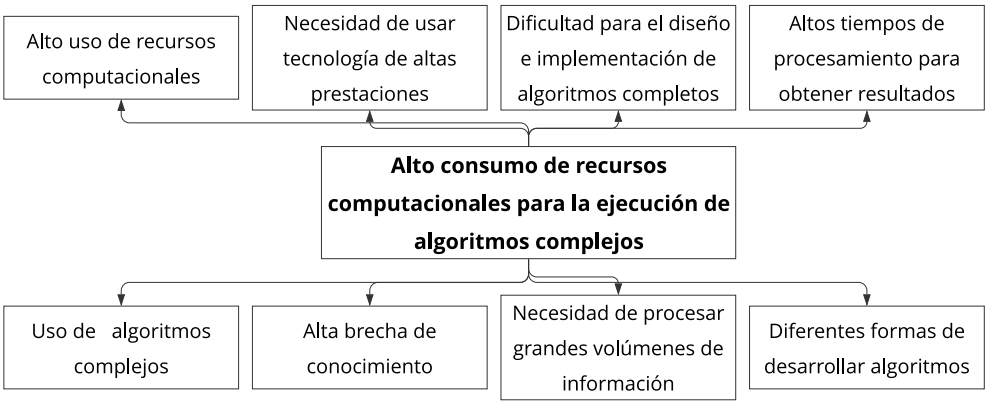

Fig. 1. Diagrama de planteamiento del problema.
Fuente: Elaboración propia

Objetivos

Objetivo general

Implementar algoritmos sobre arquitectura multinúcleo para optimizar el alto coste computacional al procesar grandes volúmenes de datos.

Objetivos específicos

- Elaborar un marco teórico de la arquitectura multinúcleo y el uso de herramientas para la ejecución de algoritmos de alto costo computacional.
- Estudiar técnicas de programación de algoritmos con lenguajes que soportan arquitecturas multinúcleo.
- Evaluar los algoritmos de alto costo computacional con las métricas definidas en la investigación.
- Validar los resultados utilizando métricas y factores de evaluación determinados en la investigación.

Alcance

El proyecto planteado tiene una finalidad de medir los costes de computación en procesadores con arquitecturas multinúcleo en computadores personales, con el sistema operativo Windows.

Implementando algoritmos de procesamiento matricial de alto costo computacional que permitan medir el rendimiento de las arquitecturas multinúcleo. Se definirán métricas de evaluación para medir la optimización de los algoritmos en base a las recomendaciones sugeridas en la investigación de programación sobre arquitectura multinúcleo.

El conjunto de datos sintéticos que se utilizarán durante el proceso de experimentación de los algoritmos, serán obtenidos de los repositorios de Machine Learning de la Universidad de California (UCI - University of California, Irvine).

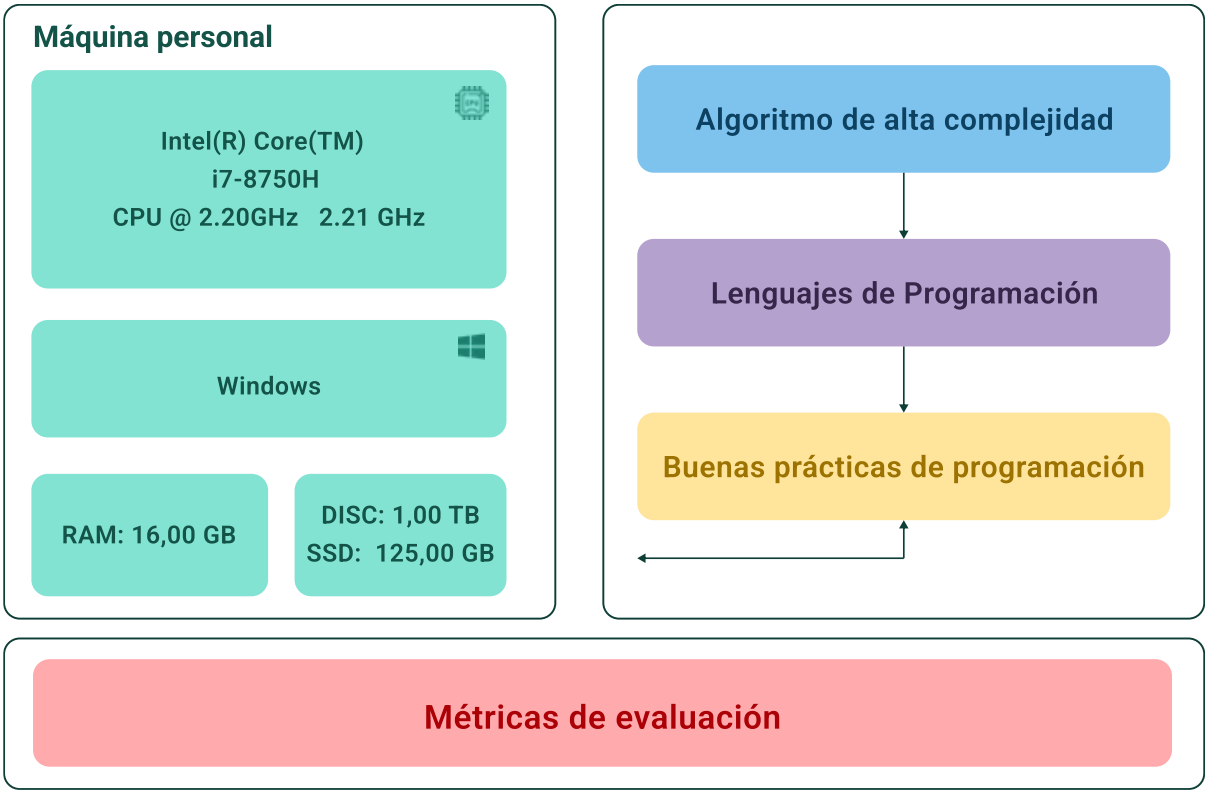

Fig. 2. Diagrama de proceso.
Fuente: Elaboración propia

Este proyecto de acuerdo con las características y el propósito de la investigación se enmarca en el paradigma cuantitativo, aplicado y experimental, bajo el enfoque empírico analítico y de diseño experimental, puesto que se desarrollará algoritmos matemáticos, se implementará prototipos de software, y se realizará pruebas de rendimiento de algoritmos y de resultados.

Justificación

Político

El proyecto se enfoca en el objetivo 9: Industria, innovación e Infraestructura de los Objetivos de Desarrollo Sostenible:

9.4 De aquí a 2030, modernizar la infraestructura y reconvertir las industrias para que sean sostenibles, utilizando los recursos con mayor eficacia y promoviendo la adopción de tecnologías y procesos industriales limpios y ambientalmente racionales, y logrando que todos los países tomen medidas de acuerdo con sus capacidades respectivas.

9.5 Aumentar la investigación científica y mejorar la capacidad tecnológica de los sectores industriales de todos los países, en particular los países en desarrollo, entre otras cosas fomentando la innovación y aumentando considerablemente, de aquí a 2030, el número de personas que trabajan en investigación y desarrollo por millón de habitantes y los

gastos de los sectores público y privado en investigación y desarrollo (Naciones Unidas, 2018).

Tecnológica

Los procesadores multinúcleo combinan de dos a más microprocesadores que habitualmente se les encuentra en circuitos integrados, puede existir una manera de paralelismo con múltiples procesos ejecutándose al mismo tiempo unidos por un canal de alta velocidad y equilibrando la carga de trabajo entre ellos (Soto, 2017).

En el ámbito del paralelismo además de la medición de tiempo de ejecución se pueden obtener otras métricas como la aceleración (Speedup), eficiencia(efficiency), etc., para observar el funcionamiento de las computadoras con tareas asignadas (Müller, 2011).

Teórica

La investigación sobre la implementación de algoritmos de alto costo computacional y el uso de procesadores multinúcleo apoyará a la motivación del uso de estas tecnologías para abrir oportunidades diferentes de programación a personas que están encargadas en el ámbito del desarrollo.

Método de investigación

Se utilizará el método científico que servirá de guía para la resolución de la pregunta de investigación planteada y la creación de las hipótesis porque usa un conjunto de técnicas y procedimientos para la obtención de resultados fiables dando soluciones a problemas de investigación.

Contexto

La TABLA 1 muestra de forma estructurada los diferentes estudios que se realizaron acerca del procesamiento multinúcleo.

TABLA 1 Repositorios externos

Trabajo	Enlace	Diferencia
2007: A nivel nacional en la ciudad de Cuenca se encuentra desarrollado un proyecto llamado Análisis de rendimiento de un clúster HPC y, arquitecturas manycore y multicore.	http://dspace.ucuenca.edu.ec/bitstream/123456789/28554/1/Trabajo%20de%20Titulaci%C3%B3n.pdf	Se realizará el análisis del rendimiento sobre un computador personal.

2009: Análisis del impacto de la arquitectura multinúcleo en cómputo paralelo.	https://tesis.ipn.mx/bitstream/handle/123456789/5918/1411.pdf?sequence=1&isAllowed=y	Se realizará el análisis del rendimiento sobre el sistema operativo Windows, con un conjunto de datos sintéticos.
2012: Artículos de investigación sobre tecnologías y lenguajes de programación concurrentes y/o paralelos.	http://www.unipamplona.edu.co/unipamplona/portalIG/home_77/recursos/paralela/21112016/art7_pmultinucleo.pdf	Se realizará un sobre tres lenguajes de programación.
2016: Programación paralela sobre arquitecturas heterogéneas.	http://bdigital.unal.edu.co/54267/1/1053771802.2016.pdf	Se utilizará algoritmos de procesamiento matricial para el análisis del rendimiento.
2015: Desarrollo de aplicaciones con técnicas de programación paralela para el análisis del procesamiento 3D de imágenes de microscopía.	https://biblioteca.utb.edu.co/notas/tesis/0068243.pdf	Se aplicará técnicas de programación sobre arquitectura multinúcleo.

CAPÍTULO 1

Marco teórico

A continuación, se muestra de una manera estructurada el contenido del presente capítulo el cual conceptualiza los puntos esenciales que involucra la implementación de algoritmos sobre una arquitectura multinúcleo para optimizar el alto coste computacional al procesar grandes volúmenes de datos, ver TABLA 2.

TABLA 2 Esquematización del marco teórico

Tema principal	Tema secundario	Sumario
Algoritmos de alto costo computacional.	Complejidad algorítmica Notación Big-O Complejidad algorítmica en las estructuras de datos. Complejidad algorítmica en los algoritmos de ordenamiento	
Estudio de procesadores multinúcleo	Procesadores de un solo núcleo Procesadores multinúcleos Hyper-Threading Ventajas y desventajas del procesamiento multinúcleo	
Herramientas para programación en procesadores multinúcleo.	Lenguaje C Java Python Go	
Buenas prácticas de programación.		

1.1. Algoritmos de alto costo computacional

Para comprender qué es un algoritmo de alto costo computacional, se debe partir por entender el concepto de la complejidad algorítmica y los tipos de recursos requeridos durante el cómputo para resolver un problema. Además, será necesario realizar un breve análisis de la notación utilizada para efectuar el cálculo de la complejidad de un algoritmo.

1.1.1. Complejidad algorítmica

La complejidad algorítmica es una medida de cuánto tardará un algoritmo en completarse dada una entrada de tamaño *n* (Padmanabhan, 2020). Si bien la complejidad suele expresarse en términos de tiempo, a veces la complejidad también se analiza en términos de espacio, esto significa la cantidad de memoria requerida por el algoritmo.

Según (Britannica T, 2011), la complejidad computacional es el costo inherente de resolver un problema en computación científica a gran escala, medido por el número de operaciones requeridas, así como la cantidad de memoria utilizada y el orden en que se utiliza. El resultado de un análisis de complejidad es una estimación de la rapidez con que aumenta el tiempo de solución a medida que aumenta el tamaño del problema, que se puede utilizar para analizar problemas y ayudar en el diseño de algoritmos para su solución.

Generalmente el análisis de la complejidad de un algoritmo es útil cuando se comparan algoritmos o se buscan mejoras. La complejidad algorítmica cae dentro de una rama de la informática teórica llamada teoría de la complejidad computacional (Padmanabhan, 2020).

La complejidad computacional es un continuo, ya que algunos algoritmos requieren tiempo lineal (es decir, el tiempo requerido aumenta directamente con el número de elementos o nodos en la lista, gráfico o red que se procesa), mientras que otros requieren tiempo cuadrático o incluso exponencial para completarlo (es decir, el tiempo requerido aumenta con el número de elementos al cuadrado o con el exponencial de ese número). En el apartado de este continuo se encuentran los problemas intratables, aquellos cuyas soluciones no pueden implementarse de manera eficiente. Para estos problemas, los científicos informáticos buscan encontrar algoritmos heurísticos que casi puedan resolver el problema y ejecutarse en un tiempo razonable (Britannica T, 2011).

Para ilustrar lo mencionado en el párrafo anterior, suponga que está buscando un artículo específico en una lista larga sin clasificar, probablemente lo comparará con cada artículo. El tiempo de búsqueda es proporcional al tamaño de la lista. Aquí se dice que la complejidad es lineal (Padmanabhan, 2020).

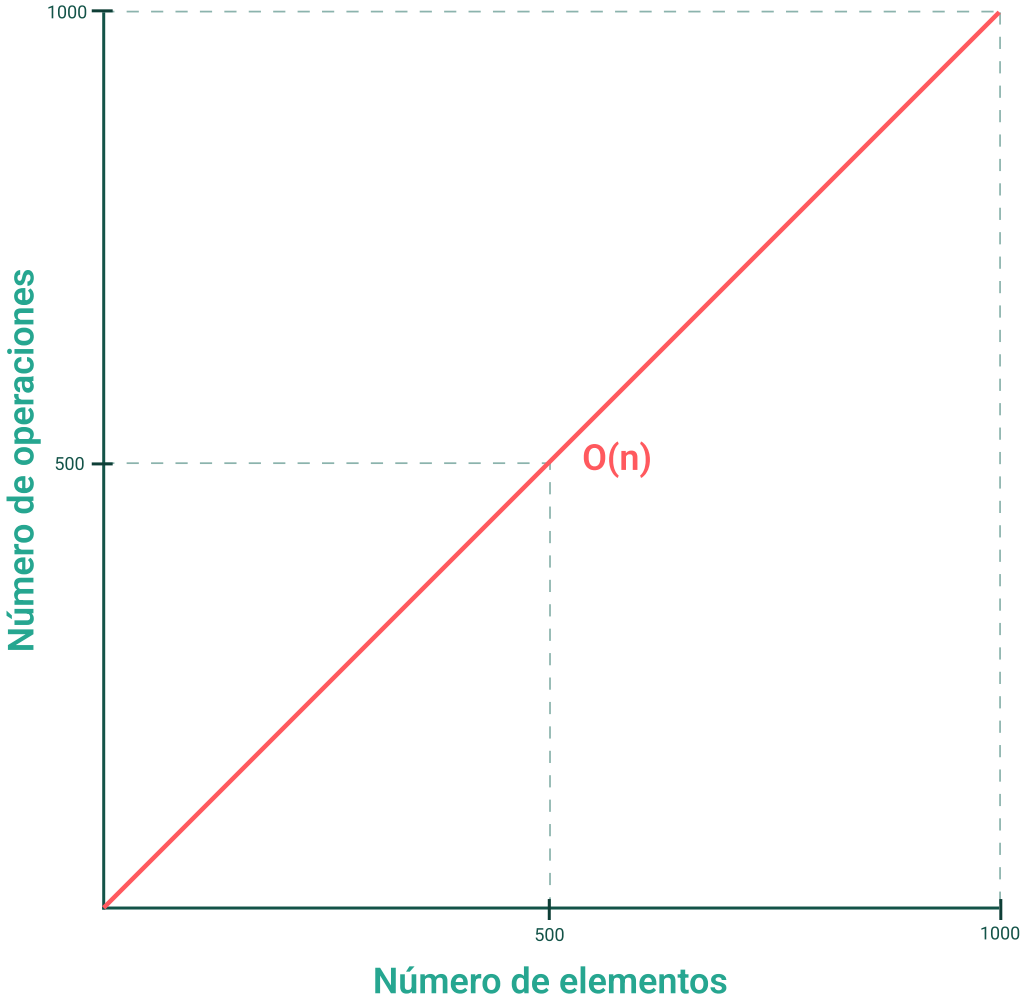

Fig. 3. Complejidad lineal

Fuente: (Escobar, 2019)

Por otro lado, si busca una palabra en un diccionario, la búsqueda será más rápida porque las palabras están ordenadas, conoce el orden y puede decidir rápidamente si necesita pasar a páginas anteriores o posteriores. Este es un ejemplo de complejidad logarítmica (Padmanabhan, 2020).

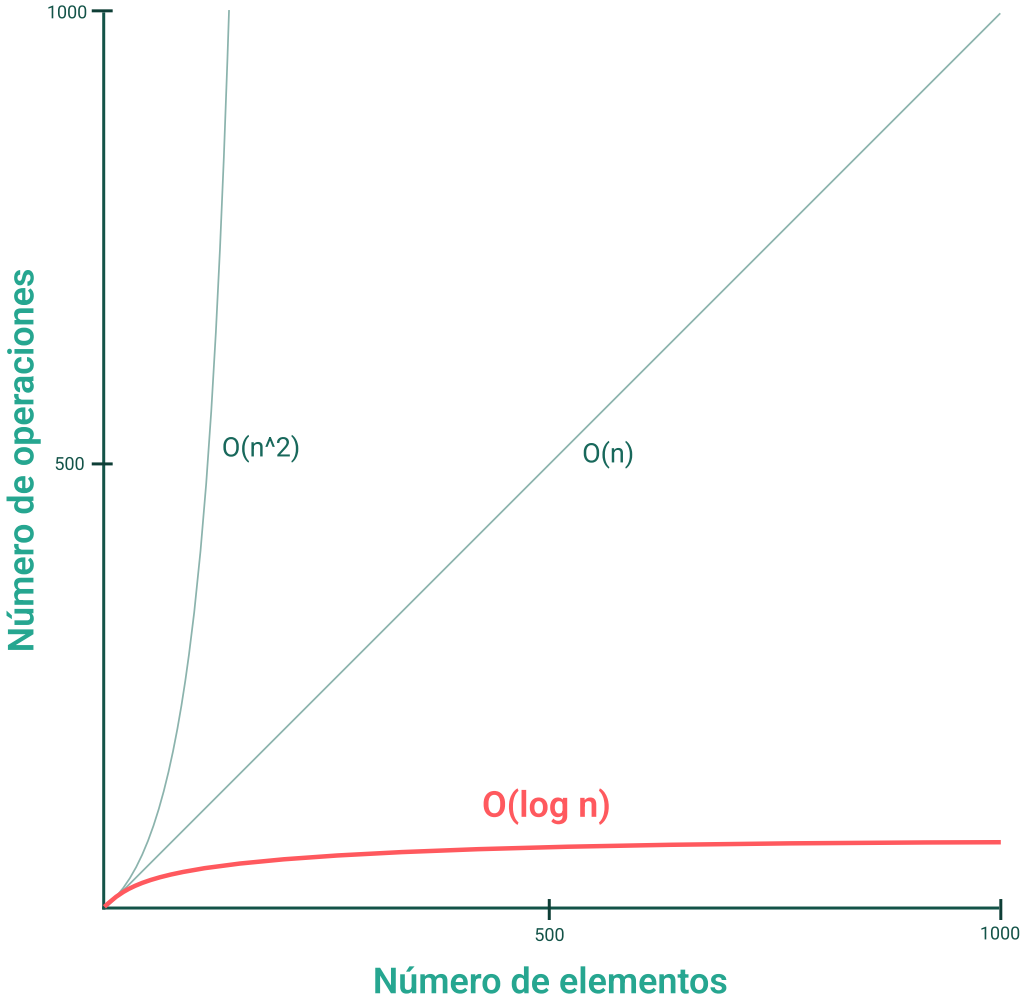

Fig. 4. Complejidad logarítmica

Fuente: (Escobar, 2019)

Si se le pide que elija la primera palabra de un diccionario, esta operación tiene una complejidad de tiempo constante, independientemente del número de palabras del diccionario. Así mismo, unirse al final de una cola en un banco es de complejidad constante independientemente de la longitud de la cola (Padmanabhan, 2020).

Supongamos que se le da una lista sin clasificar y se le pide que busque todos los duplicados, entonces la complejidad se vuelve cuadrática. La búsqueda de duplicados para un elemento es de complejidad lineal. Si se realiza esto para todos los elementos, la complejidad se vuelve cuadrática. De manera similar, si a todas las personas en una habitación se les pide que se den la mano con todas las demás, la complejidad es cuadrática (Padmanabhan, 2020).

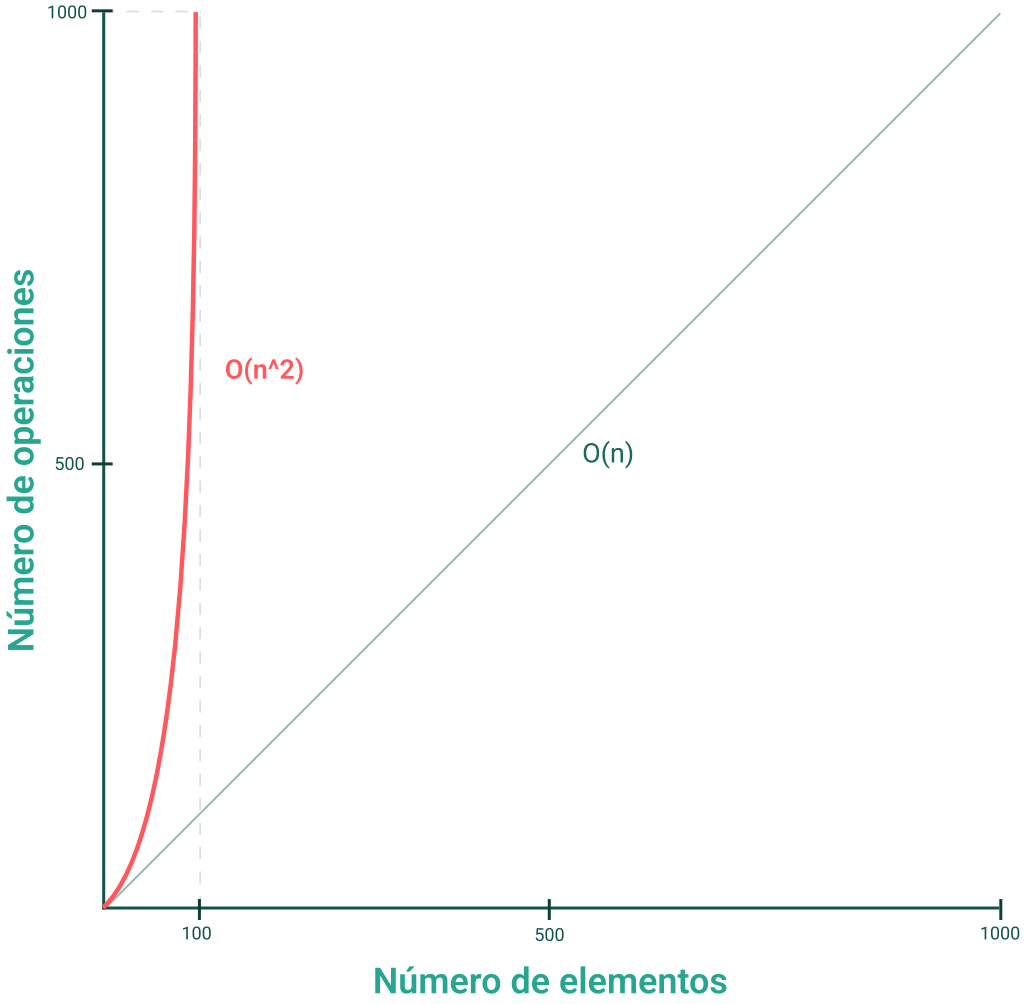

Fig. 5. Complejidad exponencial

Fuente: (Escobar, 2019)

1.1.2. Notación Big-O

La complejidad algorítmica permite entender cómo se va a comportar un algoritmo cuando se incrementa la cantidad de los datos de entrada. Teniendo en cuenta que la complejidad del algoritmo es independiente de la velocidad del computador en el que se ejecuta y se define utilizando la notación Big-O. La idea es conocer cuántas operaciones se necesitan a medida que aumentan los datos de entrada (Escobar, 2019).

La notación Big-O es la notación predominante para representar la complejidad algorítmica. Debido a que proporciona un límite superior de complejidad y, por lo tanto, significa el peor rendimiento del algoritmo. Con tal notación, es fácil comparar diferentes algoritmos porque la notación dice claramente cómo escala el algoritmo cuando aumenta el tamaño de entrada. Esto a menudo se denomina orden de crecimiento (Padmanabhan, 2020).

En la notación Big-O el tiempo de ejecución constante se los representa con ***O(1)***; el crecimiento lineal está representado por ***O(n)***; el crecimiento logarítmico por ***O(log n)***; el crecimiento logarítmico-lineal por ***O(n log n)***; el crecimiento cuadrático por ***O(n^2)***; el crecimiento exponencial por ***O(2^n)***; y el crecimiento factorial que está representado por ***O(n!)*** (Nasar, 2016).

El orden de crecimiento de la complejidad algorítmica se puede comparar de mejor a peor:

$$\boldsymbol{O(1) < O(\log n) < O(\sqrt{n}) < O(n) < O(n \log n) < O(n^2) < O(2^n) < O(n!)}$$

En el análisis de complejidad, solo se retiene el término dominante. Por ejemplo, si un algoritmo requiere $\boldsymbol{2n^3 + \log n + 4}$ operaciones, se dice que su orden (complejidad) es $\boldsymbol{O(n^3)}$ debido a que $\boldsymbol{2n^3}$ es el término dominante.

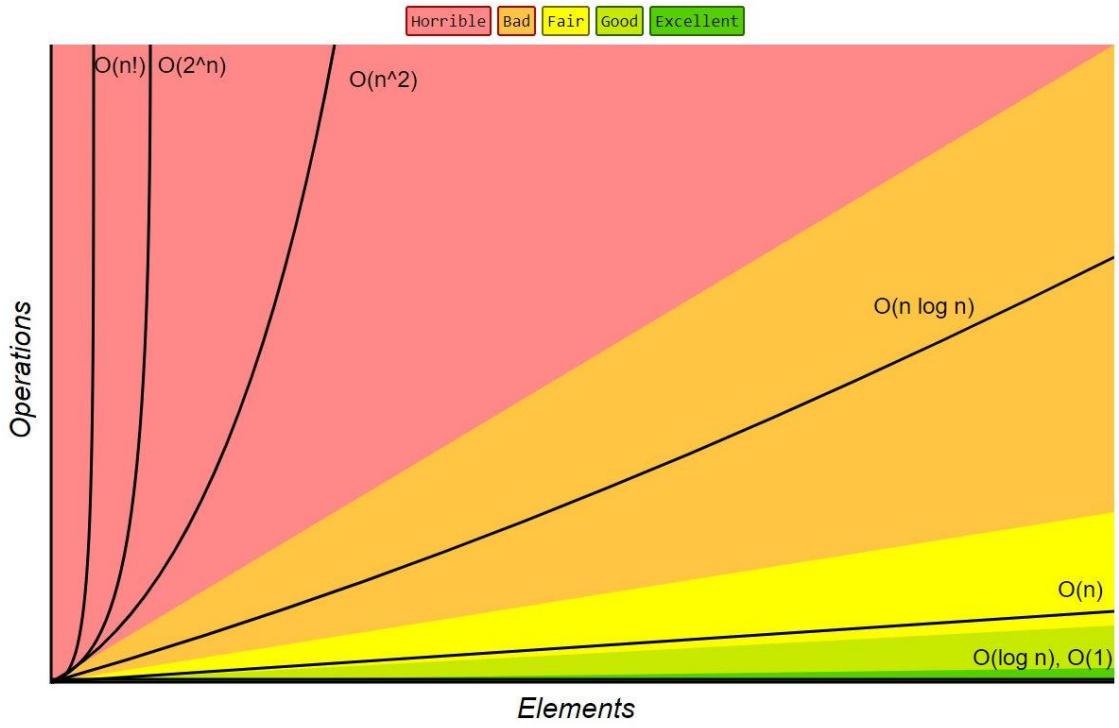

Fig. 6. Orden de crecimiento de los algoritmos especificados en notación Big-O.
Fuente: (Padmanabhan, 2020)

1.1.3. Complejidad algorítmica en las estructuras de datos

Las estructuras de datos sólo almacenan datos, pero se debe tomar en cuenta la complejidad algorítmica cuando se realiza operaciones con ellas. Por este motivo es necesario analizar operaciones como la inserción, eliminación, la búsqueda e indexación. Con

el objetivo de elegir la estructura de datos adecuada para reducir la complejidad (Padmanabhan, 2020).

En la Fig. 7 se muestra un conjunto de estructuras de datos y su complejidad al efectuar algún tipo de operación:

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Fig. 7. Complejidad de operaciones sobre estructuras de datos.
Fuente: (Chart, 2013)

1.1.4. Complejidad algorítmica en los algoritmos de ordenamiento

El algoritmo de ordenamiento más simple es probablemente Bubblesort (burbuja), pero es cuadrático en el caso promedio y, por lo tanto, no es eficiente. Las mejores alternativas son aquellas con complejidad logarítmica lineal: Quicksort, Mergesort, Heapsort, etc. Si la lista ya está ordenada, la complejidad del mejor caso ocurre con Bubblesort, Timsort, Insertionsort y Cubesort, todos completados en tiempo lineal (Padmanabhan, 2020).

En la Fig. 8 se presenta una lista de algoritmos de ordenamiento de datos y la complejidad que conlleva cada uno:

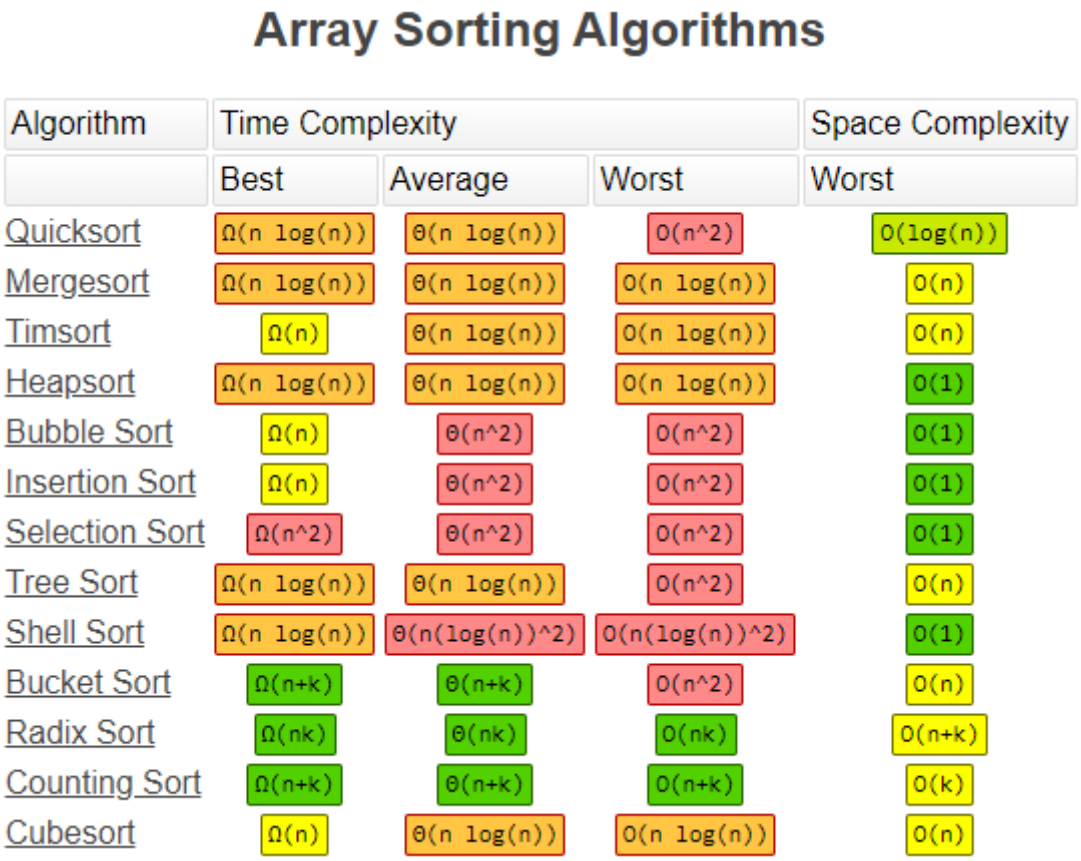

Fig. 8. Complejidad de los algoritmos de ordenamiento

Fuente: (Chart, 2013)

1.2. Estudio de procesadores multinúcleo

El avance de la tecnología ha permitido que el desarrollo de procesadores eficientes y veloces sea un tema de gran interés para las empresas fabricantes de hardware hoy en día. Y una gran palabra de moda en el hardware es la de *multinúcleo* (multi-core), debido a que empresas como AMD e Intel lanzan procesadores con más núcleos que nunca (Millar, 2020).

Un procesador multinúcleo es un circuito integrado único (también conocido como chip multiprocessor o CMP) que contiene unidades de procesamiento de núcleos múltiples, comúnmente conocidas como núcleos (Firesmith, 2017).

En un procesador de un solo núcleo, el rendimiento de la CPU está limitado por el tiempo necesario para comunicarse con la memoria caché y la RAM. Aproximadamente el 75% del tiempo de la CPU se usa esperando los resultados del acceso a la memoria. Para mejorar el rendimiento de sus procesadores, los fabricantes han lanzado más máquinas multinúcleo. Una CPU que ofrece varios núcleos puede funcionar significativamente mejor que una CPU de un solo núcleo de la misma velocidad (Millar, 2020).

Múltiples núcleos permiten que las PC ejecuten múltiples procesos al mismo tiempo con mayor facilidad, aumentando su rendimiento cuando realiza múltiples tareas o bajo las demandas de aplicaciones y programas potentes (Millar, 2020).

1.2.1. Procesadores de un solo núcleo

Originalmente, las CPU tenían un solo núcleo. Eso significaba que la CPU física tenía una sola unidad central de procesamiento (Hoffman, 2018). La Fig. 8 presenta los componentes principales de una arquitectura de un solo núcleo.

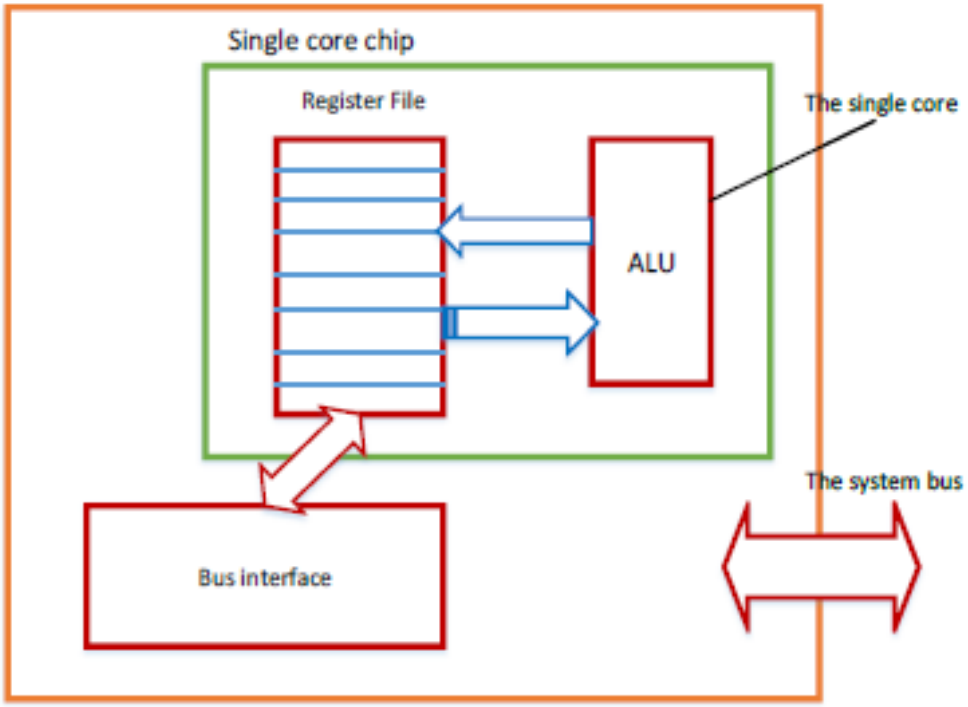

Fig. 9. Arquitectura de un sólo núcleo
Fuente: (Surakhi et al., 2018)

1.2.2. Procesadores multinúcleos

Se refiere a otra versión avanzada de la arquitectura de procesadores. Debido a que contienen más de un núcleo en el mismo chip para aumentar el rendimiento del procesamiento. Cada uno de estos núcleos tiene su propia canalización y recursos para ejecutar diferentes instrucciones sin causar ningún problema (Surakhi et al., 2018).

Para aumentar el rendimiento, los fabricantes agregan "núcleos" adicionales o unidades de procesamiento central. Una CPU de doble núcleo tiene dos unidades centrales de procesamiento, por lo que el sistema operativo la ve como dos CPU. Una CPU con dos

núcleos, por ejemplo, podría ejecutar dos procesos diferentes al mismo tiempo (Hoffman, 2018).

La Fig.10 muestra teóricamente la arquitectura de un sistema en el que un único sistema operativo host asigna 14 aplicaciones de software a los núcleos de un procesador de cuatro núcleos homogéneos. En esta arquitectura, hay tres niveles de caché, que son progresivamente más grandes, pero más lentos: L1 (que consta de una caché de instrucciones y una caché de datos), L2 y L3. Es necesario tener en cuenta que las cachés L1 y L2 son locales para un solo núcleo, mientras que L3 se comparte entre los cuatro núcleos.

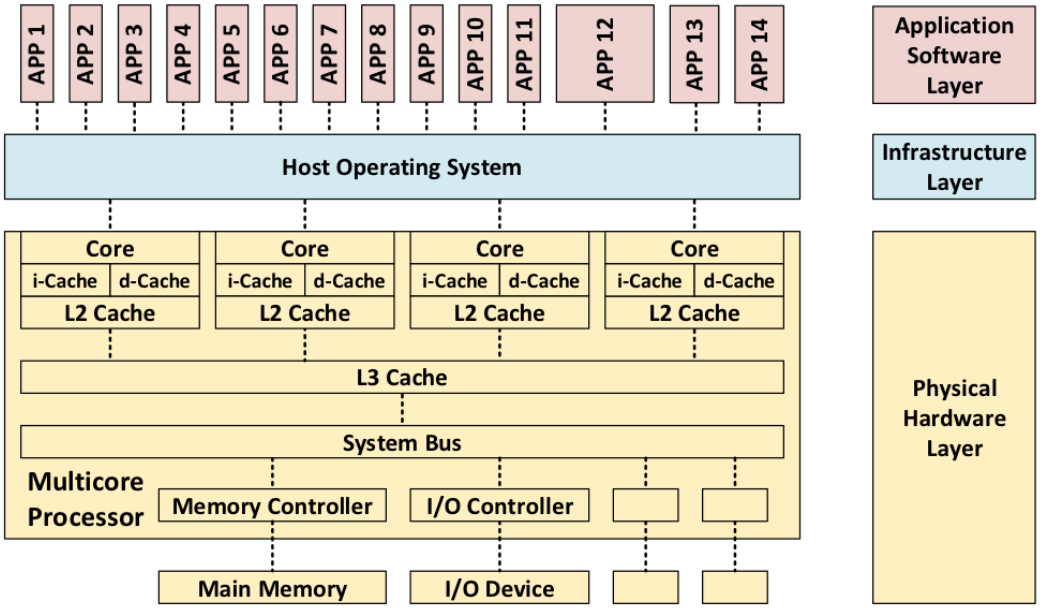

Fig. 10. Arquitectura multinúcleo
Fuente: (Firesmith, 2017)

Basado en la idea de un procesador multinúcleo que contiene varios procesadores 'núcleos' y paquetes dentro de un solo procesador físico, aumenta el rendimiento general de la máquina al permitir más de una instrucción para ser ejecutada en la misma unidad de tiempo. Como resultado, los procesadores multinúcleo simples tienen un mejor rendimiento que procesadores complejos de un solo núcleo (Surakhi et al., 2018).

1.2.3. Hyper-Threading

Antes de continuar con las siguientes secciones, es necesario abarcar este tema; debido a que permitirá explicar cómo vez el sistema operativo los núcleos del procesador, y

por qué en ciertas ocasiones la cantidad de núcleos físicos aparece duplicada en el sistema operativo.

Según (Hoffman, 2018) la técnica del *hyper-threading* fue el primer intento que realizó Intel por llevar la computación paralela a los ordenadores personales. Debutó en las CPU de escritorio con el Pentium 4 HT en 2002. Los Pentium 4 de la época tenían un solo núcleo de CPU, por lo que en realidad solo podía realizar una tarea a la vez, incluso si podía cambiar entre tareas con la suficiente rapidez, que parecía una multitarea. Y *hyper-threading* intentó compensar eso.

Un único núcleo de CPU físico con *Hyper-Threading* aparece como dos CPU lógicas en un sistema operativo. La CPU sigue siendo una sola CPU, por lo que es una situación engañosa. Si bien el sistema operativo ve dos CPU para cada núcleo, el hardware de la CPU real solo tiene un único conjunto de recursos de ejecución para cada núcleo. La CPU finge tener más núcleos que los que tiene y usa su propia lógica para acelerar la ejecución del programa. En otras palabras, se engaña al sistema operativo para que vea dos CPU por cada núcleo de CPU real (Hoffman, 2018).

Hyper-threading permite que los dos núcleos de CPU lógicos compartan recursos de ejecución física. Esto puede acelerar un poco las cosas: si una CPU virtual está detenida y esperando, la otra CPU virtual puede tomar prestados sus recursos de ejecución. Hyper-threading puede ayudar a acelerar el sistema, pero no es tan bueno como tener núcleos adicionales reales (Hoffman, 2018).

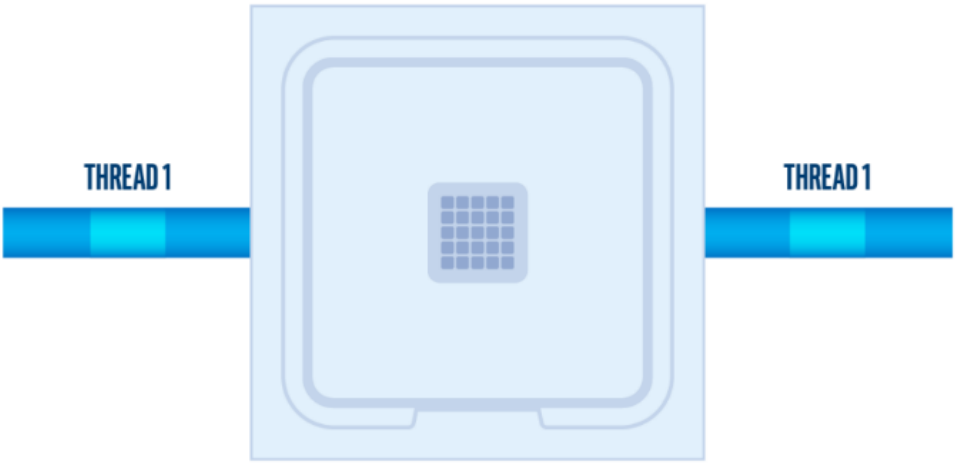

Fig. 11. Hyper-Threading
Fuente: (Intel Corporation, 2020)

Cuando la tecnología Hyper-Threading Intel está activa, la CPU expone dos contextos de ejecución por núcleo físico. Esto significa que un núcleo físico ahora funciona como dos "núcleos lógicos" que manejan distintos subprocesos de software. Por ejemplo, el procesador

Intel® Core™ i9-10900K de diez núcleos tiene 20 subprocesos cuando tiene habilitado Hyper-Threading (Intel Corporation, 2020).

Dos núcleos lógicos pueden ejecutar las tareas con más eficiencia que un núcleo de un único subproceso tradicional. La tecnología Hyper-Threading, al aprovechar el tiempo inactivo cuando el núcleo antiguamente esperaba que finalizaran otras tareas, mejora el rendimiento de la CPU (hasta un 30 % en las aplicaciones del servidor) (Intel Corporation, 2020).

El Administrador de tareas de windows muestra esta información de una forma clara. Por ejemplo, en la Fig. 12 se puede ver que el sistema tiene una CPU real (socket) y cuatro núcleos. *Hyper-threading* hace que cada núcleo parezca dos CPU para el sistema operativo, por lo que muestra 8 procesadores lógicos.

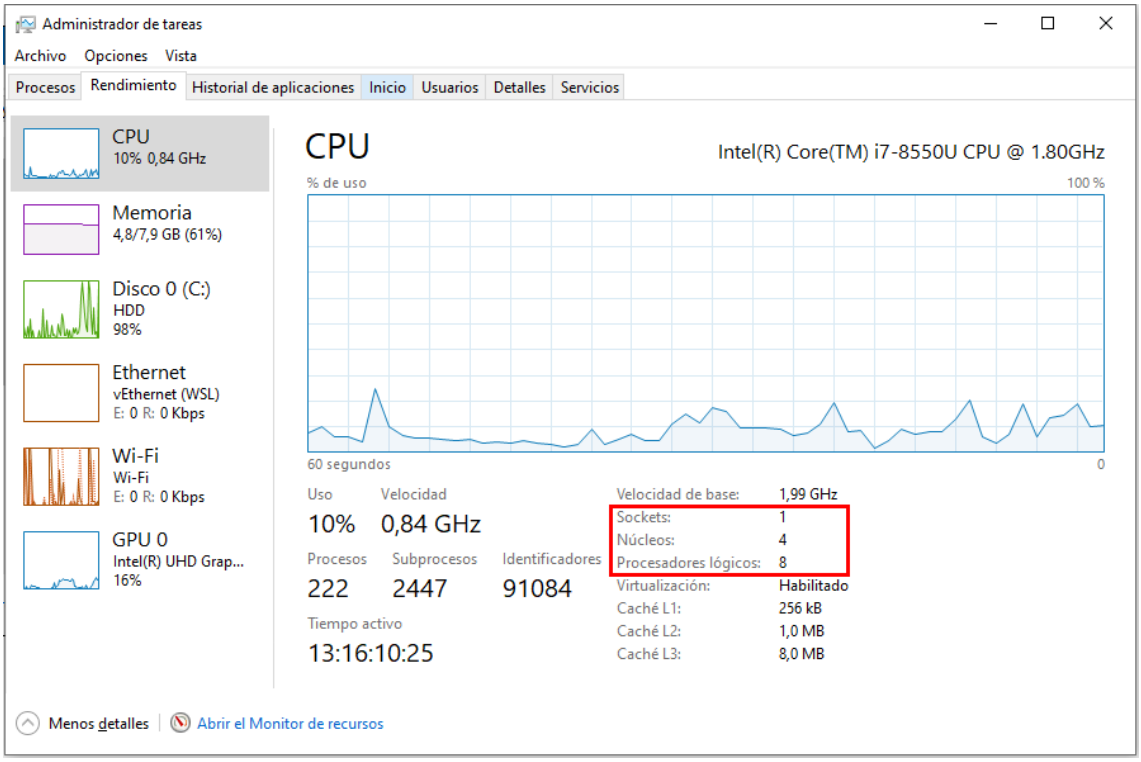

Fig. 12. Característica de la CPU
Fuente: Elaboración propia

1.2.4. Ventajas y desventajas del procesamiento multinúcleo

El procesamiento multinúcleo ofrece las siguientes ventajas:

- **Ahorro de energía:** Al utilizar procesadores multinúcleo, los arquitectos pueden reducir el número de equipos integrados. Superan el aumento de la generación de calor debido a la Ley de Moore (es decir, los circuitos más pequeños aumentan la

resistencia eléctrica, lo que crea más calor), lo que a su vez disminuye la necesidad de enfriamiento (Firesmith, 2017).

- **Concurrencia verdadera:** Al asignar aplicaciones a diferentes núcleos, el procesamiento multinúcleo aumenta el soporte intrínseco para el procesamiento paralelo real (en oposición al virtual) dentro de aplicaciones de software individuales en múltiples aplicaciones (Firesmith, 2017).
- **Rendimiento:** El procesamiento multinúcleo puede aumentar el rendimiento al ejecutar varias aplicaciones al mismo tiempo (Surakhi et al., 2018).

Del otro lado, se puntualizan algunas desventajas del procesamiento multinúcleo:

- **Recursos compartidos:** Los núcleos en el mismo procesador comparten recursos internos del procesador (caché L3, bus del sistema, controlador de memoria, controladores de Entrada/Salida e interconexiones) y recursos externos del procesador (memoria principal, dispositivos de Entrada/Salida y redes) (Surakhi et al., 2018).
- **Defectos de concurrencia:** Los núcleos se ejecutan al mismo tiempo, creando la posibilidad de defectos de concurrencia que incluyen interbloqueo, bloqueo activo, suspensión, condiciones de carrera (de datos), inversión de prioridad, violaciones de orden y violaciones de atomicidad. Teniendo en cuenta que estos son esencialmente los mismos tipos de defectos de concurrencia que pueden ocurrir cuando el software se asigna a varios subprocesos en un solo núcleo (Firesmith, 2017).

1.3. Herramientas para programación en procesadores multinúcleo.

En este apartado se presentará los lenguajes de programación más populares en el desarrollo de sistemas multinúcleos en la actualidad.

1.3.1. Lenguaje C

El lenguaje C no tiene sintaxis nativa compatible con subprocesos múltiples. Existen dos API estándar abiertas, *POSIXthreads y OpenMP*. POSIXthreads, comúnmente conocidos como *Pthreads*, es una API de bajo nivel, mientras que OpenMP depende de las directivas del compilador (Embedded Staff, 2008).

Una implementación de Pthreads está vinculada como una biblioteca normal, por lo que no se requiere soporte de compilador especial. La implementación de bajo nivel permite un control preciso sobre el hilo y la sincronización, pero la necesidad de llamadas a funciones

explícitas requiere una modificación significativa del código secuencial para expresar el paralelismo (Embedded Staff, 2008).

OpenMP es una API de alto nivel. OpenMP intenta expresar el paralelismo usando directivas del compilador para anotar el código fuente sin modificarlo. Debido a que se basa en las directivas del compilador, se requiere soporte específico para el compilador (Embedded Staff, 2008).

El enfoque de alto nivel de OpenMP puede resultar en una menor eficiencia que Pthreads en algunos casos, pero la sintaxis es más concisa y fácil de aprender. Una ventaja significativa es que la anotación OpenMP puede estar desactivada para ejecutar el código secuencialmente (Embedded Staff, 2008).

1.3.2. Java

Java es un lenguaje de alto nivel que permite escribir programas que se pueden ejecutar en una variedad de plataformas. Java fue diseñado para producir código que sea más simple de escribir y más fácil de mantener que C o C11 (Embedded Staff, 2008).

Las aplicaciones de Java normalmente se compilan en *bytecode* (archivos con una extensión .class) que pueden ejecutarse en cualquier arquitectura de computadora con la ayuda de un intérprete Java y un entorno de tiempo de ejecución llamado Java Virtual Machine (JVM). El código de bytes también se puede convertir directamente en instrucciones de lenguaje de máquina mediante un compilador Just-In-Time (JIT).

Java es uno de los lenguajes de programación más populares en la actualidad y se ha vuelto popular en sistemas integrados de alta gama como teléfonos inteligentes, PDA y consolas de juegos (Embedded Staff, 2008).

En Java existen dos formas en las que se puede crear un nuevo hilo de ejecución: se puede implementar un objeto *Runnable (Fig.13), o realizar una extensión de la clase Thread (Fig. 14).*

```
public class IamRunnable Implements Runnable {
      public void run() {
            System.out.println("Hello I am a Runnable Thread.");
      }
      public static void main(String args[]) {
            (new Thread(new IamRunnable())).start();
      }
};
```

Fig. 13. Implementación de la interfaz Runnable

Fuente: (Embedded Staff, 2008)

```
public class IamAThread extends Thread {
      public void run() {
            System.out.println("I am a Thread.");
      }
      public static void main(String args[]) {
            (new IamAThread()).start();
      }
};
```

Fig. 14. Extensión de la clase Thread

Fuente: (Embedded Staff, 2008)

1.3.3. Python

Python es un lenguaje de programación de alto nivel, altamente flexible y de propósito general que ha ganado una gran popularidad debido a su facilidad de uso y capacidad para crear código personalizado rápidamente. Admite múltiples paradigmas de programación, desde estilos orientados a objetos hasta su uso como lenguaje de programación para aplicaciones web. Python es mucho más lento que los lenguajes de programación como C (Sagar, 2019).

Python es lento. Y para aumentar la velocidad de procesamiento en Python, el código puede ser hecho para ejecutarse en múltiples procesos. Esta paralelización permite la distribución del trabajo en todos los núcleos de CPU disponibles. Cuando se ejecutan en varios núcleos, los trabajos de ejecución prolongada se pueden dividir en fragmentos manejables más pequeños. Una vez que los trabajos individuales se ejecutan en paralelo, se devuelven los resultados y, de esta manera, el tiempo de procesamiento se reduce drásticamente (Sagar, 2019).

Un punto importante que se debe tomar en consideración es el hecho de que Python utiliza de manera predeterminada un solo núcleo y esto se debe al GIL (Global Interpreter Lock), que permite que solo un hilo lleve el intérprete de Python en un momento dado. El GIL se implementó para manejar un problema de administración de memoria, pero como resultado, Python se limita a usar un solo procesador (Data@Urban, 2018).

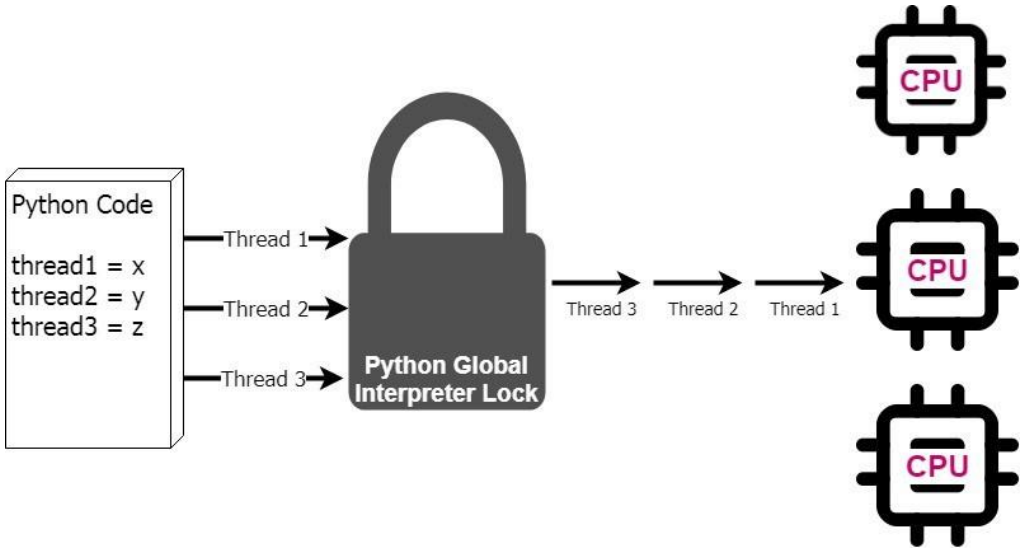

Fig. 15. Bloqueo del intérprete global en Python
Fuente: (Data@Urban, 2018)

En la Fig. 15, suponiendo que los tres subprocesos tuvieran tiempos de ejecución idénticos, la solución de multiprocesamiento reduciría el tiempo total de ejecución en un tercio. Pero esta reducción no es exactamente proporcional a la cantidad de procesadores disponibles, debido a la sobrecarga involucrada en la creación de procesos de multiprocesamiento, pero las ganancias representan una mejora significativa sobre las operaciones de un solo núcleo (Data@Urban, 2018).

1.3.4. Go

Go es un nuevo lenguaje de programación de sistemas concurrentes. Uno de sus objetivos es afrontar el desafío de la programación paralela de múltiples núcleos (Tang, 2012).

Go es expresivo, conciso, limpio y eficiente. Sus mecanismos de concurrencia facilitan la escritura de programas que aprovechan al máximo las máquinas multinúcleo y en red, mientras que su novedoso sistema de tipos permite la construcción de programas flexible y modular. Go compila rápidamente en código de máquina, pero tiene la conveniencia de la recolección de basura (garbage collection) y el poder de la reflexión en tiempo de ejecución. Es un lenguaje compilado rápido, de tipado estático que se siente como un lenguaje interpretado de tipado dinámico (GoLang, s.f.).

1.4. Buenas prácticas de programación

Las buenas prácticas de desarrollo de software son a menudo una combinación de experiencia personal, conocimiento corporativo, pensamiento grupal e incluso requisitos del cliente en algunos casos. A medida que los desarrolladores trabajan para crear nuevo software o migrar software existente para explotar el paralelismo en arquitecturas multinúcleo, identificar esas mejores prácticas es un desafío importante, un desafío que la MCA (Multicore Association) ha abordado con su lanzamiento de la guía MPP (Multicore Programming Practices) (Evanczuk, 2013).

Un conjunto conciso de mejores prácticas es una colección de los métodos más conocidos para realizar la tarea de desarrollo con programación multinúcleo. La intención es compartir técnicas de desarrollo que se sabe que funcionan de manera efectiva para procesadores multinúcleo, lo que resulta en costos de desarrollo reducidos a través de un tiempo de comercialización más corto y un ciclo de desarrollo más eficiente para quienes emplean estas técnicas (Evanczuk, 2013).

Las fases del desarrollo de software discutidas en la guía Prácticas de Programación Multinúcleo se detalla a continuación:

- Análisis y el diseño de alto nivel es un estudio de una aplicación para determinar dónde agregar la concurrencia y una estrategia para modificar la aplicación para admitir la concurrencia (Multicore Association, 2020).
- Implementación y el diseño de bajo nivel es la selección de patrones de diseño, algoritmos y estructuras de datos y la posterior codificación del software que maneja la concurrencia (Multicore Association, 2020).
- La depuración comprende la implementación de la concurrencia de una manera que minimiza los problemas de concurrencia latentes, lo que permite que una aplicación sea examinada fácilmente en busca de problemas de concurrencia y técnicas para encontrar problemas de concurrencia. El rendimiento se refiere a mejorar el tiempo de respuesta o el rendimiento de la aplicación al encontrar y abordar los efectos de cuellos de botella relacionados con la comunicación, sincronización, bloqueos, equilibrio de carga y localidad de datos (Multicore Association, 2020).
- El software existente, también conocido como software heredado, es la aplicación utilizada actualmente representada por su código de software. Los clientes que utilizan software existente han optado por evolucionar la implementación para el desarrollo de nuevos productos en lugar de volver a implementar la aplicación completa para habilitar procesadores multinúcleo (Multicore Association, 2020).

CAPÍTULO 2

Desarrollo

La fase de desarrollo de este proyecto de trabajo de grado consta de la implementación de un algoritmo de alto costo computacional en tres lenguajes de programación. En primera instancia se implementó el algoritmo sin aplicar ninguna técnica de optimización y continuando con el proceso, se aplicó las técnicas de mejora como paralelismo y vectorización, para obtener un mejor rendimiento en el tiempo de ejecución.

2.1. Selección del algoritmo

A continuación, se presentará el proceso utilizado para seleccionar el algoritmo de alto costo computacional, el cual se utilizará para realizar las comparativas de rendimiento en los diferentes lenguajes de programación, sobre un computador personal con arquitectura multinúcleo y sobre el DevCloud de Intel (Intel, 2021).

2.1.1. Selección del algoritmo de alto costo computacional

Para la selección del algoritmo de alto costo computacional se realizó en 3 fases. En la primera fase se analizó la complejidad algorítmica de los algoritmos, luego se analizó el tiempo de ejecución en los 3 lenguajes y finalmente se aplicó las técnicas de optimización para evaluar los resultados.

En primera instancia se realizó la implementación del algoritmo del problema de los **N-Body** en forma secuencial en los lenguajes de Programación C, Python y Java, donde se evaluó el tiempo de ejecución obtenido sobre el HPC de Intel.

Y en última instancia se analizó el algoritmo de la **Distancia Euclidiana** en los lenguajes de programación Python y C, del cual se tomó como referencia su tiempo de ejecución inicial, frente al tiempo de ejecución obtenido luego de aplicar las técnicas de optimización, y además se realizó la comparativa de rendimiento con el Lenguaje de programación C. Adicionalmente de aplicó técnicas de optimización como el paralelismo y el código de máquina optimizado en tiempo de ejecución.

2.2. Selección de los lenguajes de programación

Los lenguajes de programación seleccionados se escogieron en base a su popularidad según los datos presentados por PYPL Index (PopularitY of Programming Language Index), TIOBE Index y Stack Overflow's en el año 2020 y 2021.

El índice de popularidad de los lenguajes de programación según PYPL index, se realiza analizando la frecuencia con la que se buscan los tutoriales de programación en Google (PYPL index, 2022).

A continuación, se detalla la clasificación de los lenguajes de programación presentada por PYPL Index en el año 2020.

Tabla 3. Ranking PYPL Index 2020.

Posición	Lenguaje
1	**Python**
2	**Java**
3	JavaScript
4	C#
5	PHP
6	**C/C++**
7	R
8	Objective-C
9	Swift
10	TypeScript

Fuente: basado en datos de (PYPL index, 2022)

En el año 2021, la tabla de clasificación presentada por PYPL Index se posicionó de la siguiente manera.

Tabla 4. Ranking PYPL Index 2021.

Posición	Lenguaje	Cuota
1	**Python**	**30.21 %**
2	**Java**	**17.82 %**
3	JavaScript	9.16 %
4	C#	7.53 %
5	PHP	6.82 %

6	C/C++	5.84 %
7	R	3.81 %
8	TypeScript	2.03 %
9	Swift	2.02 %
10	Objective-C	1.73 %

Fuente: basado en datos de (PYPL index, 2022)

Los tres lenguajes de programación seleccionados en base a su popularidad, su facilidad de aprendizaje y por su eficiencia de código fueron: Java, Python y C.

2.3. Algoritmo del problema de los N-Body

El problema de los *n-body*, donde n es mayor que tres, ha sido atacado enérgicamente con técnicas numéricas en computadoras de alto rendimiento. La mecánica celeste en el sistema solar es, en última instancia, un problema de $n - body$, pero las configuraciones especiales y la relativa pequeñez de las perturbaciones han permitido descripciones bastante precisas de los movimientos (válidas para períodos de tiempo limitados) con varias aproximaciones y procedimientos sin ningún intento de resolver el problema completo (Peale, 2009).

Este algoritmo tiene una complejidad computacional $O(N^2)$ (Martín, 2017). Por este motivo los científicos han buscado durante años una solución al problema de los n-body, pero sólo se ha llegado a aproximaciones.

La implementación de este algoritmo se la realizó en los lenguajes de programación Python, C y Java, para comparar el tiempo de procesamiento de forma secuencial.

2.3.1. Algoritmo del problema de los N-Body en Python

Para la implementación en Python del algoritmo se hizo uso de las estructuras de datos nativas del lenguaje, para hacer una comparación equitativa entre los lenguajes. Además, las pruebas fueron realizadas sobre el HPC de Intel.

El algoritmo que se presenta a continuación fue tomado como referencia del sitio (Doug Bagley, 2021). Donde se realiza una comparativa exhaustiva de los principales lenguajes de programación.

```
1 import sys
```

```
import time

def combinations(l):
    result = []
    for x in range(len(l) - 1):
        ls = l[x+1:]
        for y in ls:
            result.append((l[x],y))
    return result

PI = 3.14159265358979323
SOLAR_MASS = 4 * PI * PI
DAYS_PER_YEAR = 365.24

BODIES = {
    'sun': ([0.0, 0.0, 0.0], [0.0, 0.0, 0.0], SOLAR_MASS),

    'jupiter': ([4.8414314424647209e+00,
                 -1.1603200440274284e+00,
                 -1.0362204447112310e-01],
                [1.6600766427440369e-03 * DAYS_PER_YEAR,
                 7.6990111841974042e-03 * DAYS_PER_YEAR,
                 -6.9046001697206302e-05 * DAYS_PER_YEAR],
                9.5479193842432661e-04 * SOLAR_MASS),

    'saturn': ([8.3433667182445799e+00,
                4.1247985641243048e+00,
                -4.0352341711432138e-01],
               [-2.7674251072686241e-03 * DAYS_PER_YEAR,
                4.9985280123491724e-03 * DAYS_PER_YEAR,
                2.3041729757376393e-05 * DAYS_PER_YEAR],
               2.8588598066613081e-04 * SOLAR_MASS),

    'uranus': ([1.2894369562139131e+01,
                -1.5111151401698631e+01,
                -2.2330757889265573e-01],
               [2.9646013756476162e-03 * DAYS_PER_YEAR,
                2.3784717395948095e-03 * DAYS_PER_YEAR,
                -2.9658956854023756e-05 * DAYS_PER_YEAR],
               4.3662440433515630e-05 * SOLAR_MASS),

    'neptune': ([1.5379697114850917e+01,
                 -2.5919314609987964e+01,
                 1.7925877295037118e-01],
                [2.6806777249038932e-03 * DAYS_PER_YEAR,
                 1.6282417003824230e-03 * DAYS_PER_YEAR,
                 -9.5159225451971587e-05 * DAYS_PER_YEAR],
                5.1513890204661145e-05 * SOLAR_MASS) }


SYSTEM = list(BODIES.values())
PAIRS = combinations(SYSTEM)


def advance(dt, n, bodies=SYSTEM, pairs=PAIRS):

```

```
    for i in range(n):
        for (([x1, y1, z1], v1, m1),
             ([x2, y2, z2], v2, m2)) in pairs:
            dx = x1 - x2
            dy = y1 - y2
            dz = z1 - z2
            mag = dt * ((dx * dx + dy * dy + dz * dz) ** (-1.5))
            b1m = m1 * mag
            b2m = m2 * mag
            v1[0] -= dx * b2m
            v1[1] -= dy * b2m
            v1[2] -= dz * b2m
            v2[0] += dx * b1m
            v2[1] += dy * b1m
            v2[2] += dz * b1m
        for (r, [vx, vy, vz], m) in bodies:
            r[0] += dt * vx
            r[1] += dt * vy
            r[2] += dt * vz


def report_energy(bodies=SYSTEM, pairs=PAIRS, e=0.0):

    for (((x1, y1, z1), v1, m1),
         ((x2, y2, z2), v2, m2)) in pairs:
        dx = x1 - x2
        dy = y1 - y2
        dz = z1 - z2
        e -= (m1 * m2) / ((dx * dx + dy * dy + dz * dz) ** 0.5)
    for (r, [vx, vy, vz], m) in bodies:
        e += m * (vx * vx + vy * vy + vz * vz) / 2.
    print("%.9f" % e)

def offset_momentum(ref, bodies=SYSTEM, px=0.0, py=0.0, pz=0.0):

    for (r, [vx, vy, vz], m) in bodies:
        px -= vx * m
        py -= vy * m
        pz -= vz * m
    (r, v, m) = ref
    v[0] = px / m
    v[1] = py / m
    v[2] = pz / m

def main(n, ref='sun'):
    tic = time.time()
    offset_momentum(BODIES[ref])
    report_energy()
    advance(0.01, n)
    report_energy()
    toc = time.time()
    print(f"Run time: {toc - tic:0.4f} seconds")

if __name__ == '__main__':
    main(int(sys.argv[1]))

```

2.3.2. Algoritmo del problema de los N-Body en C

Para la implementación y ejecución en el lenguaje C el algoritmo fue testeado sobre el HPC de Intel.

De igual forma que el caso anterior, el algoritmo que se presenta a continuación fue tomado como referencia del sitio (Doug Bagley, 2021).

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <math.h>
 4 #include <x86intrin.h>
 5 #include <time.h>
 6
 7 #define N 5
 8 #define PI 3.141592653589793
 9 #define SOLAR_MASS (4 * PI * PI)
10 #define DAYS_PER_YEAR 365.24
11 #define PAIRS (N * (N - 1) / 2)
12
13 static inline __m256d _mm256_rsqrt_pd(__m256d s)
14 {
15     __m128 q = _mm256_cvtpd_ps(s);
16     q = _mm_rsqrt_ps(q);
17     __m256d x = _mm256_cvtps_pd(q);
18     __m256d y = s * x * x;
19     __m256d a = _mm256_mul_pd(y, _mm256_set1_pd(0.375));
20     a = _mm256_mul_pd(a, y);
21     __m256d b = _mm256_mul_pd(y, _mm256_set1_pd(1.25));
22     b = _mm256_sub_pd(b, _mm256_set1_pd(1.875));
23     y = _mm256_sub_pd(a, b);
24     x = _mm256_mul_pd(x, y);
25     return x;
26 }
27
28 // compute rsqrt of distance between each pair of bodies
29 static inline void kernel(__m256d *r, double *w, __m256d *p)
30 {
31     for (int i = 1, k = 0; i < N; i++)
32         for (int j = 0; j < i; j++, k++)
33             r[k] = _mm256_sub_pd(p[i], p[j]);
34
35     for (int k = 0; k < PAIRS; k += 4)
36     {
37         __m256d x0 = _mm256_mul_pd(r[k], r[k]);
38         __m256d x1 = _mm256_mul_pd(r[k + 1], r[k + 1]);
39         __m256d x2 = _mm256_mul_pd(r[k + 2], r[k + 2]);
40         __m256d x3 = _mm256_mul_pd(r[k + 3], r[k + 3]);
41
42         __m256d t0 = _mm256_hadd_pd(x0, x1);
```

```
43          __m256d t1 = _mm256_hadd_pd(x2, x3);
44          __m256d y0 = _mm256_permute2f128_pd(t0, t1, 0x21);
45          __m256d y1 = _mm256_blend_pd(t0, t1, 0b1100);
46
47          __m256d z = _mm256_add_pd(y0, y1);
48          z = _mm256_rsqrt_pd(z);
49          _mm256_store_pd(w + k, z);
50      }
51  }
52
53  static double energy(double *m, __m256d *p, __m256d *v)
54  {
55      double e = 0.0;
56
57      __m256d r[PAIRS + 3];
58      double w[PAIRS + 3] __attribute__((aligned(sizeof(__m256d))));
59
60      r[N] = _mm256_set1_pd(0.0);
61      r[N + 1] = _mm256_set1_pd(0.0);
62      r[N + 2] = _mm256_set1_pd(0.0);
63
64      for (int k = 0; k < N; k++)
65          r[k] = _mm256_mul_pd(v[k], v[k]);
66
67      for (int k = 0; k < N; k += 4)
68      {
69          __m256d t0 = _mm256_hadd_pd(r[k], r[k + 1]);
70          __m256d t1 = _mm256_hadd_pd(r[k + 2], r[k + 3]);
71          __m256d y0 = _mm256_permute2f128_pd(t0, t1, 0x21);
72          __m256d y1 = _mm256_blend_pd(t0, t1, 0b1100);
73
74          __m256d z = _mm256_add_pd(y0, y1);
75          _mm256_store_pd(w + k, z);
76      }
77
78      for (int k = 0; k < N; k++)
79          e += 0.5 * m[k] * w[k];
80
81      r[PAIRS] = _mm256_set1_pd(1.0);
82      r[PAIRS + 1] = _mm256_set1_pd(1.0);
83      r[PAIRS + 2] = _mm256_set1_pd(1.0);
84
85      kernel(r, w, p);
86
87      for (int i = 1, k = 0; i < N; i++)
88          for (int j = 0; j < i; j++, k++)
89              e -= m[i] * m[j] * w[k];
90
91      return e;
92  }
93
94  static void advance(int n, double dt, double *m, __m256d *p, __m256d
95  *v)
96  {
97      __m256d r[PAIRS + 3];
98      double w[PAIRS + 3] __attribute__((aligned(sizeof(__m256d))));
```

```
 99
100     r[PAIRS] = _mm256_set1_pd(1.0);
101     r[PAIRS + 1] = _mm256_set1_pd(1.0);
102     r[PAIRS + 2] = _mm256_set1_pd(1.0);
103
104     __m256d rt = _mm256_set1_pd(dt);
105
106     __m256d rm[N];
107     for (int i = 0; i < N; i++)
108         rm[i] = _mm256_set1_pd(m[i]);
109
110     for (int s = 0; s < n; s++)
111     {
112         kernel(r, w, p);
113
114         for (int k = 0; k < PAIRS; k += 4)
115         {
116             __m256d x = _mm256_load_pd(w + k);
117             __m256d y = _mm256_mul_pd(x, x);
118             __m256d z = _mm256_mul_pd(x, rt);
119             x = _mm256_mul_pd(y, z);
120             _mm256_store_pd(w + k, x);
121         }
122
123         for (int i = 1, k = 0; i < N; i++)
124             for (int j = 0; j < i; j++, k++)
125             {
126                 __m256d t = _mm256_set1_pd(w[k]);
127                 t = _mm256_mul_pd(r[k], t);
128                 __m256d x = _mm256_mul_pd(t, rm[j]);
129                 __m256d y = _mm256_mul_pd(t, rm[i]);
130
131                 v[i] = _mm256_sub_pd(v[i], x);
132                 v[j] = _mm256_add_pd(v[j], y);
133             }
134
135         for (int i = 0; i < N; i++)
136         {
137             __m256d t = _mm256_mul_pd(v[i], rt);
138             p[i] = _mm256_add_pd(p[i], t);
139         }
140     }
141 }
142
143 int main(int argc, char *argv[])
144 {
145     int n = atoi(argv[1]);
146
147     clock_t tic = clock();
148
149     double m[N];
150     __m256d p[N], v[N];
151
152     // sun
153     m[0] = SOLAR_MASS;
154     p[0] = _mm256_set1_pd(0.0);
```

```
155     v[0] = _mm256_set1_pd(0.0);
156
157     // jupiter
158     m[1] = 9.54791938424326609e-04 * SOLAR_MASS;
159     p[1] = _mm256_setr_pd(0.0,
160                           4.84143144246472090e+00,
161                           -1.16032004402742839e+00,
162                           -1.03622044471123109e-01);
163     v[1] = _mm256_setr_pd(0.0,
164                           1.66007664274403694e-03 * DAYS_PER_YEAR,
165                           7.69901118419740425e-03 * DAYS_PER_YEAR,
166                           -6.90460016972063023e-05 * DAYS_PER_YEAR);
167
168     // saturn
169     m[2] = 2.85885980666130812e-04 * SOLAR_MASS;
170     p[2] = _mm256_setr_pd(0.0,
171                           8.34336671824457987e+00,
172                           4.12479856412430479e+00,
173                           -4.03523417114321381e-01);
174     v[2] = _mm256_setr_pd(0.0,
175                           -2.76742510726862411e-03 * DAYS_PER_YEAR,
176                           4.99852801234917238e-03 * DAYS_PER_YEAR,
177                           2.30417297573763929e-05 * DAYS_PER_YEAR);
178
179     // uranus
180     m[3] = 4.36624404335156298e-05 * SOLAR_MASS;
181     p[3] = _mm256_setr_pd(0.0,
182                           1.28943695621391310e+01,
183                           -1.51111514016986312e+01,
184                           -2.23307578892655734e-01);
185     v[3] = _mm256_setr_pd(0.0,
186                           2.96460137564761618e-03 * DAYS_PER_YEAR,
187                           2.37847173959480950e-03 * DAYS_PER_YEAR,
188                           -2.96589568540237556e-05 * DAYS_PER_YEAR);
189
190     // neptune
191     m[4] = 5.15138902046611451e-05 * SOLAR_MASS;
192     p[4] = _mm256_setr_pd(0.0,
193                           1.53796971148509165e+01,
194                           -2.59193146099879641e+01,
195                           1.79258772950371181e-01);
196     v[4] = _mm256_setr_pd(0.0,
197                           2.68067772490389322e-03 * DAYS_PER_YEAR,
198                           1.62824170038242295e-03 * DAYS_PER_YEAR,
199                           -9.51592254519715870e-05 * DAYS_PER_YEAR);
200
201     // offset momentum
202     __m256d o = _mm256_set1_pd(0.0);
203     for (int i = 0; i < N; i++)
204     {
205         __m256d t = _mm256_mul_pd(_mm256_set1_pd(m[i]), v[i]);
206         o = _mm256_add_pd(o, t);
207     }
208     v[0] = _mm256_mul_pd(o, _mm256_set1_pd(-1.0 / SOLAR_MASS));
209
210     printf("%.9f\n", energy(m, p, v));
```

```
211     advance(n, 0.01, m, p, v);
212     printf("%.9f\n", energy(m, p, v));
213     clock_t toc = clock();
214     printf("Run time: %f seconds\n", (double)(toc - tic) /
215 CLOCKS_PER_SEC);
216 
217     return 0;
218 }
219 
```

2.3.3. Algoritmo del problema de los N-Body en Java

Para la implementación y ejecución en el lenguaje Java fue necesario instalar el jdk de java sobre el HPC y luego se hizo las respectivas ejecuciones.

El algoritmo que se presenta a continuación también fue tomado como referencia del sitio (Doug Bagley, 2021).

```
 1 public final class nbody {
 2     public static void main(String[] args) {
 3         int n = Integer.parseInt(args[0]);
 4         long start_time = System.currentTimeMillis();
 5         NBodySystem bodies = new NBodySystem();
 6         System.out.printf("%.9f\n", bodies.energy());
 7         for (int i=0; i<n; ++i)
 8            bodies.advance(0.01);
 9         System.out.printf("%.9f\n", bodies.energy());
10         long end_time = System.currentTimeMillis();
11         System.out.println("Run time "+((double)(end_time-
12 start_time)/1000) + " seconds.");
13     }
14 }
15 
16 final class NBodySystem {
17    private static final int LENGTH = 5;
18 
19    private Body[] bodies;
20 
21    public NBodySystem(){
22       bodies = new Body[]{
23             Body.sun(),
24             Body.jupiter(),
25             Body.saturn(),
26             Body.uranus(),
27             Body.neptune()
28          };
29 
30       double px = 0.0;
31       double py = 0.0;
32       double pz = 0.0;
33       for(int i=0; i < LENGTH; ++i) {
34          px += bodies[i].vx * bodies[i].mass;
35          py += bodies[i].vy * bodies[i].mass;
36          pz += bodies[i].vz * bodies[i].mass;
```

```
37      }
38      bodies[0].offsetMomentum(px,py,pz);
39    }
40
41    public void advance(double dt) {
42      Body[] b = bodies;
43      for(int i=0; i < LENGTH-1; ++i) {
44        Body iBody = b[i];
45        double iMass = iBody.mass;
46        double ix = iBody.x, iy = iBody.y, iz = iBody.z;
47
48        for(int j=i+1; j < LENGTH; ++j) {
49          Body jBody = b[j];
50          double dx = ix - jBody.x;
51          double dy = iy - jBody.y;
52          double dz = iz - jBody.z;
53
54          double dSquared = dx * dx + dy * dy + dz * dz;
55          double distance = Math.sqrt(dSquared);
56          double mag = dt / (dSquared * distance);
57
58          double jMass = jBody.mass;
59
60          iBody.vx -= dx * jMass * mag;
61          iBody.vy -= dy * jMass * mag;
62          iBody.vz -= dz * jMass * mag;
63
64          jBody.vx += dx * iMass * mag;
65          jBody.vy += dy * iMass * mag;
66          jBody.vz += dz * iMass * mag;
67        }
68      }
69
70      for(int i=0; i < LENGTH; ++i) {
71        Body body = b[i];
72        body.x += dt * body.vx;
73        body.y += dt * body.vy;
74        body.z += dt * body.vz;
75      }
76    }
77
78    public double energy(){
79      double dx, dy, dz, distance;
80      double e = 0.0;
81
82      for (int i=0; i < bodies.length; ++i) {
83        Body iBody = bodies[i];
84        e += 0.5 * iBody.mass *
85            ( iBody.vx * iBody.vx
86                + iBody.vy * iBody.vy
87                + iBody.vz * iBody.vz );
88
89        for (int j=i+1; j < bodies.length; ++j) {
90          Body jBody = bodies[j];
91          dx = iBody.x - jBody.x;
92          dy = iBody.y - jBody.y;
```

```
            dz = iBody.z - jBody.z;

            distance = Math.sqrt(dx*dx + dy*dy + dz*dz);
            e -= (iBody.mass * jBody.mass) / distance;
         }
      }
      return e;
   }
}


final class Body {
   static final double PI = 3.141592653589793;
   static final double SOLAR_MASS = 4 * PI * PI;
   static final double DAYS_PER_YEAR = 365.24;

   public double x, y, z, vx, vy, vz, mass;

   public Body(){}

   static Body jupiter(){
      Body p = new Body();
      p.x = 4.8414314424647209e+00;
      p.y = -1.1603200440274284e+00;
      p.z = -1.0362204447112310e-01;
      p.vx = 1.6600766427440369e-03 * DAYS_PER_YEAR;
      p.vy = 7.6990111841974042e-03 * DAYS_PER_YEAR;
      p.vz = -6.9046001697206302e-05 * DAYS_PER_YEAR;
      p.mass = 9.5479193842432661e-04 * SOLAR_MASS;
      return p;
   }

   static Body saturn(){
      Body p = new Body();
      p.x = 8.3433667182445798e+00;
      p.y = 4.1247985641243048e+00;
      p.z = -4.0352341711432138e-01;
      p.vx = -2.7674251072686241e-03 * DAYS_PER_YEAR;
      p.vy = 4.9985280123491724e-03 * DAYS_PER_YEAR;
      p.vz = 2.3041729757376393e-05 * DAYS_PER_YEAR;
      p.mass = 2.8588598066613081e-04 * SOLAR_MASS;
      return p;
   }

   static Body uranus(){
      Body p = new Body();
      p.x = 1.2894369562139131e+01;
      p.y = -1.5111151401698631e+01;
      p.z = -2.2330757889265573e-01;
      p.vx = 2.9646013756476162e-03 * DAYS_PER_YEAR;
      p.vy = 2.3784717395948095e-03 * DAYS_PER_YEAR;
      p.vz = -2.9658956854023756e-05 * DAYS_PER_YEAR;
      p.mass = 4.3662440433515630e-05 * SOLAR_MASS;
      return p;
   }

```

```
149    static Body neptune(){
150       Body p = new Body();
151       p.x = 1.5379697114850916e+01;
152       p.y = -2.5919314609987964e+01;
153       p.z = 1.7925877295037118e-01;
154       p.vx = 2.6806777249038932e-03 * DAYS_PER_YEAR;
155       p.vy = 1.6282417003824229e-03 * DAYS_PER_YEAR;
156       p.vz = -9.5159225451971587e-05 * DAYS_PER_YEAR;
157       p.mass = 5.1513890204661145e-05 * SOLAR_MASS;
158       return p;
159    }
160
161    static Body sun(){
162       Body p = new Body();
163       p.mass = SOLAR_MASS;
164       return p;
165    }
166
167    Body offsetMomentum(double px, double py, double pz){
168       vx = -px / SOLAR_MASS;
169       vy = -py / SOLAR_MASS;
170       vz = -pz / SOLAR_MASS;
171       return this;
172    }
```

2.4. Algoritmo de la Matriz de Distancia Euclidiana

La matriz de distancia euclidiana (MDE), es una matriz de distancias al cuadrado entre puntos. Que se utiliza muy a menudo para medir la distancia en el plano cartesiano (Dokmanic et al., 2015).

Según la fórmula de la distancia euclidiana, la distancia entre dos puntos del plano cartesiano **(x1, y1)** y **(x2, y2)** se expresa de la siguiente forma:

$$\boldsymbol{distancia} = \sqrt{(\boldsymbol{x1} - \boldsymbol{x2})^2 + (\boldsymbol{y1} - \boldsymbol{y2})^2}$$

La fórmula anterior también se puede extender a N dimensiones:

$$\boldsymbol{distancia} = \sqrt{\sum_{n}^{n} (\boldsymbol{p_i} - \boldsymbol{q_i})^2}$$

2.4.1. Matriz de Distancia Euclidiana en Python

Python es un lenguaje de programación interpretado y el más popular según las encuestas, ver Tabla. 3. Para implementar la matriz distancia y realizar el análisis de rendimiento en el lenguaje Python, se realizó la implementación de 5 variaciones a nivel de código del algoritmo de la matriz distancia.

2.4.1.1. Matriz distancia con listas

La lista es una de las estructuras de datos más versátiles que tiene Python. Una lista se puede identificar fácilmente mediante el uso de corchetes ([]). Misma que es utilizada para almacenar los elementos de un conjunto de datos, donde cada elemento está separado por una coma (,). Además, una de las principales características de la lista se debe a que puede tener elementos de cualquier tipo de datos, ya sea un tipo entero o un tipo booleano (Shah, 2021).

Otra de las características que diferencian a una lista de una tupla y por la cual se utilizan ampliamente, es que las listas son mutables. Y al ser un medio mutable, cualquier elemento de una lista puede ser reemplazado por cualquier otro elemento de datos (Shah, 2021).

La Fig. 16 hace una representación gráfica de una lista en Python, con diferentes tipos de datos.

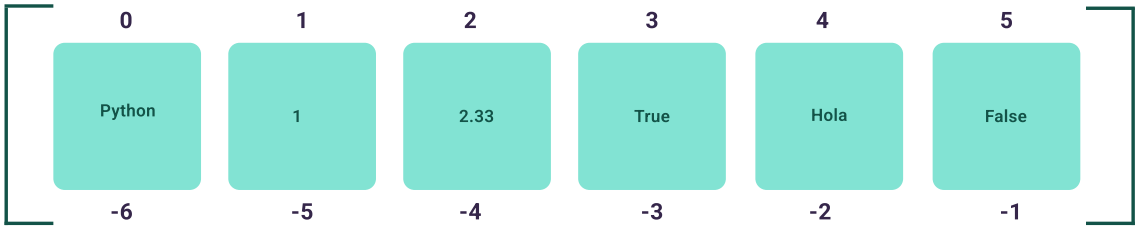

Fig. 16. Ejemplo de una lista en Python
Fuente: (Shah, 2021)

En la primera implementación del algoritmo de la matriz distancia, se hizo uso de las listas para analizar el rendimiento del algoritmo con este tipo de estructura de datos nativa de Python.

En la Fig.17 se presenta el código de la matriz distancia. La función recibe como parámetro la variable *data* que contiene la matriz original de los datos de NxM. Luego se procede a obtener el número de filas de la matriz de datos y se guarda en la variable *N;*

seguido de esto se crea una lista vacía llamada $distances$ donde se almacenará el resultado del cálculo de la matriz distancia.

Luego de definir las variables, el algoritmo hace uso de dos sentencias de tipo for donde se recorre el total de filas (N) en ambos casos. El primer for es utilizado para agregar una nueva lista (fila) vacía a la lista de resultados $distances$. Y el segundo for que se encuentra anidado dentro del primero, es utilizado para hacer el cálculo del valor de la distancia entre cada punto elevado al cuadrado.

Y finalmente, el algoritmo retorna como respuesta una matriz de $(N\ x\ N)$, con el cálculo de la distancia entre cada punto, a la que se conoce como matriz distancia.

```
"""
 Matriz de distancia utilizando listas
"""
def euclidean_distances_with_list(data):
    N = len(data) # Tamaño de los datos
    distances = [] # Matriz de distancia
    for i in range(N): # Recorre todos los datos
        distances.append([])  # Crea una lista para cada fila
        for j in range(N):
            distances[i].append(sum((data[i]-data[j])**2))   # Calcula la distancia entre cada dato
    return distances
```

Fig. 17. Matriz distancia con listas.
Fuente: Elaboración propia

Si bien es cierto, las listas son uno de los tipos de datos más utilizados en Python por su facilidad de uso; éstas no son el tipo de datos más eficientes para realizar operaciones matriciales. Esto se debe a que Python es lento, debido a dos razones principales. La primera se debe que, al ser un lenguaje de tipo dinámico no tiene una declaración de variables como Java y C, esto hace que la compilación sea bastante larga y, en ocasiones, las variables se modifican durante la ejecución sin un conocimiento del desarrollador. La otra razón es que Python usa un solo núcleo de CPU para ejecutar un subproceso en un intérprete, sin importar cuántos subprocesos haya creado en el programa (Barik, 2021).

2.4.1.2. Matriz distancia con Numpy

En la sección anterior se hizo uso de las listas que cumplen el propósito de matrices, con el detalle de que son lentas de procesar. Pero Python cuenta con un amplio ecosistema de librerías de terceros que están optimizadas para mejorar el rendimiento en cálculos matriciales; siendo una de ellas la librería Numpy.

NumPy también conocido como el paquete fundamental para la computación científica en Python. Es una librería de Python que proporciona un objeto de matriz multidimensional y una variedad de funciones que permiten realizar operaciones rápidas en matrices, que incluyen manipulación matemática, lógica, de tamaño, clasificación, selección y mucho más (Numpy, 2022).

En la Fig. 18 se presenta la implementación del código de la matriz distancia haciendo uso de la librería Numpy. La función recibe como parámetro la variable *data* que contiene la matriz original de los datos de MxN. Luego se procede a obtener el número de filas y columnas de la matriz de datos y se guardan en las variables $M\ y\ N$ respectivamente*;* seguido de esto, se crea una matriz de ceros llamado $distances$ haciendo uso de la función $zeros(M, M)$ donde se almacenará el resultado del cálculo de la matriz distancia.

Luego de definir las variables, el algoritmo hace uso de tres sentencias de tipo for . El primer y segundo for son utilizados para hacer el recorrido de la matriz resultante de (MxM). Y el tercer for es utilizado para recorrer el número de columnas de los datos N y hacer el cálculo del valor de la distancia entre cada punto elevado al cuadrado que son almacenados en la variable $result$.

```
"""
Matriz de distancia utilizando Numpy
"""
def euclidean_distances_numpy(data):
    M, N = data.shape  # Tamaño de los datos
    distances = np.zeros((M, M))  # Matriz de distancia
    for i in range(M):
        for j in range(M):
            result = 0.0
            for k in range(N):
                result += (data[i][k] - data[j][k])**2
            distances[i][j] = result
    return distances
```

Fig. 18. Matriz distancia con Numpy

Fuente: Elaboración propia

En esta variación, aunque se hace uso de la librería de Numpy, su rendimiento no presenta una gran diferencia con relación al uso de las listas. Esto se debe a la forma en la que se hizo la implementación. A diferencia del algoritmo anterior, éste utiliza tres sentencias tipo for aumentando su complejidad algorítmica.

2.4.1.3. Matriz distancia con memoria preasignada

La implementación del algoritmo con memoria preasignada representa una mejora significativa con respecto a las implementaciones anteriores. Esto se debe a que hace uso de la librería Numpy de una forma más avanzada, pero con menos líneas de código.

En la Fig.19 se presenta la implementación del código de la matriz distancia con memoria preasignada. La función recibe como parámetro la variable *data* que contiene la matriz original de los datos de NxM. Luego se procede a obtener el número de filas de la matriz de datos y se guarda en la variable $N;$ seguido de esto se crea una matriz vacía llamada $distances$ haciendo uso de la función $empty(N,N)$ donde se almacenará el resultado del cálculo de la matriz distancia. Lo que se realiza básicamente es reservar espacio de memoria de tamaño de la matriz.

Luego de definir las variables, el algoritmo hace uso de una sola sentencia de tipo for donde se recorre el total de filas (N). Pero la magia sucede en el cálculo de las distancias

entre cada punto; debido a que se hace el cálculo de fila en fila, valiéndose de la función de Numpy $sum(axis = 1)$.

```
"""
Matriz de distancia con Memoria Pre-Allocada
"""
def euclidean_distances_pre_allocated(data):
    N = len(data) # Tamaño de los datos

    distances = np.empty((N, N)) # Matriz de distancia

    for i in range(N):

        distances[i, :] = ((data-data[i])**2).sum(axis=1) # Calcula la distancia entre cada dato

    return distances
```

Fig. 19. Matriz distancia con memoria preasignada

Fuente: Elaboración propia

Esta es una de las implementaciones más eficiente de la matriz distancia, y su tiempo de procesamiento es significativamente menor que el de las dos variaciones anteriores.

2.4.1.4. Matriz distancia con vectorización

La implementación del algoritmo con vectorización también representa una mejora significativa para el cálculo de la matriz distancia.

La vectorización describe la ausencia de bucles explícitos, indexación, etc., en el código; estas cosas están ocurriendo, por supuesto, solo "detrás de escena" en código C optimizado y precompilado (Numpy, 2022).

El código vectorizado tiene muchas ventajas, entre las que se encuentran:

- El código vectorizado es más conciso y fácil de leer.
- Menos líneas de código generalmente significan menos errores.
- El código se parece más a la notación matemática estándar (lo que facilita, por lo general, codificar correctamente las construcciones matemáticas)
- La vectorización da como resultado código más $Pythonico$. Sin la vectorización, el código estaría plagado de bucles for ineficientes y difíciles de leer.

Por lo tanto, para esta implementación del algoritmo se necesita hacer una expansión de la fórmula de la distancia euclidiana (Dey, 2016). Esta se expresaría de la siguiente forma:

$$(x - y)^2 = x^2 + y^2 - 2xy$$

En Fig. 20 se muestra la implementación de la expansión de la fórmula; donde es posible notar que no se hace uso de la sentencia de tipo for. Para esto se utilizó la vectorización que es una de las optimizaciones que brinda Numpy.

```
"""
Matriz de distancia con vectorización
"""
def euclidean_distances_vectorized(data):
    x = np.sum(data**2, axis=1)[:, np.newaxis]
    y = np.sum(data**2,    axis=1)
    xy = np.dot(data, data.T)
    distances = x + y - 2*xy
    return distances
```

Fig. 20. Matriz distancia con vectorización
Fuente: Elaboración propia

Sin duda, esta es una implementación muy eficiente en términos de rendimiento y desarrollo. Debido a que no se realiza uso de bucles for de forma explícita.

2.4.1.5. Matriz distancia con Numba

La implementación de este algoritmo con la librería Numba, causa leves modificaciones a la implementación común del mismo.

Numba convierte las funciones de Python a código de máquina optimizado en tiempo de ejecución utilizando la biblioteca de compilación LLVM. Los algoritmos numéricos compilados por Numba en Python pueden acercarse a las velocidades de C o FORTRAN aplicando decoradores de Numba a la función (Numba, 2022).

En la Fig. 21 se presenta la implementación del código de la matriz distancia haciendo uso de la librería Numba. Este algoritmo es muy parecido a la implementación con Numpy (2.3.1.2).

En este punto se hace uso de procesamiento en paralelo (procesamiento multinúcleo) con el decorador $parallel = True$, el cual le indica a Numba que ejecute el código de forma paralela.

Adicional a esto, en las sentencias de tipo for se hace uso de la función $prange$ de Numba, para indicar que dicha sentencia, debe ejecutarse en paralelo.

```
"""
Matriz de distancia con Numba
"""
@numba.jit(nopython=True, cache=True, parallel=True, fastmath=True, nogil=True)
def euclidean_numba(data):
    M, N = data.shape
    distances = np.zeros((M, M))
    for i in numba.prange(M):
        for j in numba.prange(M):
            r = 0.0
            for k in numba.prange(N):
                r += (data[i][k] - data[j][k])**2
            distances[i][j] = r
    return distances
```

Fig. 21. Matriz distancia con Numba
Fuente: Elaboración propia

Sin lugar a duda, hacer uso de Numba mejora significativamente el rendimiento del algoritmo, sin aplicar mayores alteraciones al mismo.

2.4.2. Matriz de Distancia Euclidiana en C

La implementación del algoritmo de la distancia euclidiana en el lenguaje de programación C, hace uso de los tres bucles for para hacer el recorrido de todos los datos y así poder calcular la distancia entre cada punto.

En la Fig.22 la función recibe como parámetro la variable *data* de tipo $double$ que contiene la matriz original de los datos de NxM. También recibe como parámetros las variables num_rows y num_colums que indican la cantidad de filas y columnas de los datos. Adicionalmente, recibe como parámetro la variable $distances$ que es un puntero a la matriz donde se almacenará el resultado del cálculo de la matriz distancia.

Luego de definir las variables que se utilizan como contadores, el algoritmo hace uso de tres sentencias de tipo for . El primer y segundo for son utilizados para hacer el recorrido de la matriz resultante de $(num\_rows\ X\ num\_rows)$. Y el tercer for es utilizado para recorrer el número de columnas de los datos num_colums y hacer el cálculo del valor de la distancia entre cada punto elevado al cuadrado que son almacenados en la variable $temp$.

```
static int euclidean_distances(double **data, int num_rows, int num_columns, double **distances)
{
    int i, j,k;
    #pragma vector aligned
    for (i = 0; i < num_rows; i++) {
        for (j = 0; j < num_rows; j++) {
            double temp = 0.0f;
            for (k = 0; k < num_columns; k++) {
                temp += (double) pow(data[i][k] - data[j][k], 2);
            }
            distances[i][j] = temp;
        }
    }
    return 0;
}
```

Fig. 22. Matriz distancia con C

Fuente: Elaboración propia

2.5. Resultados de procesamiento

Los datos presentados a continuación, son el resultado de las pruebas de rendimiento entre los tres lenguajes de programación: Python, C y Java. Cada tabla muestra las variaciones de los algoritmos en los diferentes lenguajes. Los datos son una ponderación del promedio de los tiempos de procesamiento por cada algoritmo en su respectivo entorno. Adicionalmente, todos los datos en crudo recopilados en las pruebas de rendimiento pueden ser encontrados en el Anexo A.

Para las pruebas de rendimiento con el algoritmo del problema de los *N-Body*, se definió la variable **N** que representa el número de épocas para la cuales se va a simular el problema de los N-Body.

Tabla 5. Número de épocas N-Body

Número de Épocas	
N	1’ 000.000
N	10’000.000
N	40’000.000
N	100’000.000

Además, para las pruebas de rendimiento con el algoritmo de la *Distancia Euclidiana*, se utilizó cinco datasets (conjunto de datos) con tamaños diferentes; mismos que contienen 3 columnas de datos en todos los casos y la cantidad de filas varían dependiendo del dataset como se muestra en la Tabla 6.

Tabla 6. Datasets

Datasets	
dataset_1	700
dataset_2	3500
dataset_3	7000
dataset_4	10500
dataset_5	14000

Las pruebas de rendimiento se realizaron sobre un Ordenador Personal con el sistema operative Windows 10 con las siguientes características: Intel(R) Core (TM) i7-8750H CPU @ 2.20GHz, 16 GB de RAM, 6 núcleos físicos y 12 procesadores lógicos.

Adicionalmente, se realizó las pruebas sobre el HPC de Intel, con un nodo de computo que tenía las siguientes características: Intel(R) Xeon(R) Gold 6128 CPU @ 3.40GHz, 32 GB de RAM y 6 núcleos físicos.

2.5.1. Resultado de los tiempos de procesamiento del algoritmo N-Body

Los datos que se presentarán a continuación son el promedio en $segundos$ de los tiempos de procesamiento del algoritmo N-Body sobre el HPC de Intel. En este caso se tomó los resultados con las cinco épocas diferentes como se indica en la Tabla 5.

La Tabla 7. Muestra el promedio haciendo uso de *N (1'000.000)* donde el lenguaje C presenta el tiempo óptimo.

Tabla 7. Promedio de procesamiento N-BODY (1'000.000)

	N-BODY
Python	7,2219
C	0,0477
Java	0,0918

Fuente: Propio estudio

La Tabla 8. Muestra el promedio haciendo uso de N (10'000.000).

Tabla 8. Promedio de procesamiento N-BODY (10'000.000)

	N-BODY
Python	71,7327
C	0,3823
Java	0,8596

Fuente: Propio estudio

La Tabla 9. Muestra el promedio haciendo de N (20’000.000).

Tabla 9. Promedio de procesamiento N-BODY (20’000.000)

	N-BODY
Python	144,6161
C	0,7576
Java	1,7218

Fuente: Propio estudio

La Tabla 10. Muestra el promedio haciendo uso de *N (40’000.000).*

Tabla 10. Promedio de procesamiento N-BODY (40’000.000)

	N-BODY
Python	286,8264
C	1,5048
Java	3,4368

Fuente: Propio estudio

La Tabla 11. Muestra el promedio haciendo uso de N (100’000.000).

Tabla 11. Promedio de procesamiento N-BODY (100’000.000)

	N-BODY
Python	718,6575
C	3,7529
Java	8,6354

Fuente: Propio estudio

2.5.2. Resultado de los tiempos de procesamiento de la Distancia Euclidiana en Python (Ordenador Personal)

Los datos que se presentarán a continuación son el promedio en $segundos$ de los tiempos de procesamiento de cada variación del algoritmo de la distancia euclidiana en Python con los cinco dataset diferentes.

La Tabla 12. Muestra el promedio haciendo uso del *dataset_1*, donde la implementación con Memoria Preasignada presenta un tiempo óptimo frente a las otras implementaciones.

Tabla 12. Promedio de procesamiento MDE (dataset_1)

Distancia Euclidiana en Python	
Lista	2,0638
Numpy	2,4972
Memoria Preasignada	0,0135
Vectorización	0,0139
Numba	0,5960

Fuente: Propio estudio

La Tabla 13. Muestra el promedio haciendo uso del *dataset_2*, donde la implementación con Memoria Preasignada sigue presentando el tiempo óptimo.

Algo relevante en este análisis, se presenta en la implementación del algoritmo con la librería Numba, debido a que presenta una ligera mejora con el *dataset_2*, en relación con su promedio presentado con el ***dataset_1*** (Tabla 12).

Tabla 13. Promedio de procesamiento MDE (dataset_2)

Distancia Euclidiana en Python	
Lista	80,9742
Numpy	82,5821
Memoria Preasignada	0,1798

Vectorización	0,2829
Numba	0,5919

Fuente: Propio estudio

La Tabla 14. Muestra el promedio haciendo uso del *dataset_3*, igual que en los dos casos anteriores la implementación con Memoria Preasignada presenta el tiempo óptimo en este escenario.

Tabla 14. Promedio de procesamiento MDE (dataset_3)

Distancia Euclidiana en Python	
Lista	279,1429
Numpy	323,3746
Memoria Preasignada	0,4978
Vectorización	0,9623
Numba	0,6634

Fuente: Propio estudio

La Tabla 15. Muestra el promedio haciendo uso del *dataset_4*. En este escenario la implementación con Numba presenta el tiempo óptimo.

Tabla 15. Promedio de procesamiento MDE (dataset_4)

Distancia Euclidiana en Python	
Lista	741,0650
Numpy	748,0127
Memoria Preasignada	1,1451
Vectorización	2,6194
Numba	0,9678

Fuente: Propio estudio

La Tabla 16. Muestra el promedio haciendo uso del *dataset_5*. En este escenario la implementación con Numba sigue presentando el tiempo óptimo.

Tabla 16. Promedio de procesamiento MDE (dataset_5)

Distancia Euclidiana en Python	
Lista	853,4156
Numpy	837,5902
Memoria Preasignada	1,5949
Vectorización	6,1360
Numba	1,4825

Fuente: Propio estudio

2.5.3. Resultado de los tiempos de procesamiento de la Distancia Euclidiana en Python (HPC Intel)

Los datos que se presentarán a continuación son el promedio en $segundos$ de los tiempos de procesamiento de cada variación del algoritmo de la distancia euclidiana en Python con los cinco dataset diferentes ejecutados sobre el HPC del Intel.

La Tabla 17. Muestra el promedio haciendo uso del *dataset_1*, donde la implementación con Memoria Preasignada presenta un tiempo óptimo frente a las otras implementaciones.

Tabla 17. Promedio de procesamiento MDE (dataset_1) HPC

Distancia Euclidiana en Python	
Lista	1,2496
Numpy	1,2560
Memoria Preasignada	0,0127
Vectorización	0,0183
Numba	0,4540

Fuente: Propio estudio

La Tabla 18. Muestra el promedio haciendo uso del *dataset_2*, donde la implementación con Memoria Preasignada sigue presentando el tiempo óptimo.

Tabla 18. Promedio de procesamiento MDE (dataset_2) HPC

Distancia Euclidiana en Python	
Lista	31,3301
Numpy	31,3835
Memoria Preasignada	0,1307
Vectorización	0,1399
Numba	0,4843

Fuente: Propio estudio

La Tabla 19. Muestra el promedio haciendo uso del *dataset_3*, igual que en los dos casos anteriores la implementación con Memoria Preasignada presenta el tiempo óptimo en este escenario.

Tabla 19. Promedio de procesamiento MDE (dataset_3) HPC

Distancia Euclidiana en Python	
Lista	130,6598
Numpy	125,4298
Memoria Preasignada	0,3532
Vectorización	0,4217
Numba	0,5332

Fuente: Propio estudio

La Tabla 20. Muestra el promedio haciendo uso del *dataset_4*. En este escenario la implementación con Numba presenta el tiempo óptimo.

Tabla 20. Promedio de procesamiento MDE (dataset_4) HPC

Distancia Euclidiana en Python

Lista	291,0490
Numpy	282,0604
Memoria Preasignada	0,7098
Vectorización	0,8915
Numba	0,6055

Fuente: Propio estudio

La Tabla 21. Muestra el promedio haciendo uso del *dataset_5*. En este escenario la implementación con Numba sigue presentando el tiempo óptimo.

Tabla 21. Promedio de procesamiento MDE (dataset_5) HPC

Distancia Euclidiana en Python	
Lista	525,2495
Numpy	502,7344
Memoria Preasignada	1,2170
Vectorización	1,6408
Numba	0,7470

Fuente: Propio estudio

2.5.4. Resultado de los tiempos de procesamiento de la Distancia Euclidiana en C

El lenguaje C es reconocido por su velocidad de procesamiento y por ser el padre de los leguajes de programación orientados a objetos como Java y C++.

Para analizar el tiempo de procesamiento del algoritmo de la distancia euclidiana en C, se realizó pruebas de rendimiento del algoritmo de forma secuencial con el compilador gcc y de forma optimizada con el compilador icc de Intel.

De la misma forma que en Python, para C también se utilizó los mismos datasets.

Los datos que se presentarán a continuación son el promedio en $segundos$ de los tiempos de procesamiento de cada variación del algoritmo de la distancia euclidiana en C sobre el HPC del Intel.

La Tabla 22. Muestra el promedio haciendo uso del *dataset_1*.

Tabla 22. Promedio de procesamiento MDE C (dataset_1) HPC

	Distancia Euclidiana en C
Serial	0,0426
Optimizada	0,0023

Fuente: Propio estudio

La Tabla 23. Muestra el promedio haciendo uso del *dataset_2*.

Tabla 23. Promedio de procesamiento MDE C (dataset_2) HPC

	Distancia Euclidiana en C
Serial	1,0019
Optimizada	0,0913

Fuente: Propio estudio

La Tabla 24. Muestra el promedio haciendo uso del *dataset_3*.

Tabla 24. Promedio de procesamiento MDE C (dataset_3) HPC

	Distancia Euclidiana en C
Serial	4,7858
Optimizada	0,3746

Fuente: Propio estudio

La Tabla 25. Muestra el promedio haciendo uso del *dataset_4*.

Tabla 25. Promedio de procesamiento MDE C (dataset_4) HPC

	Distancia Euclidiana en C
Serial	12,6885
Optimizada	0,8605

Fuente: Propio estudio

La Tabla 26. Muestra el promedio haciendo uso del *dataset_5*.

Tabla 26. Promedio de procesamiento MDE C (dataset_5) HPC

Distancia Euclidiana en C	
Serial	23,5687
Optimizada	1,5381

Fuente: Propio estudio

2.5.5. Comparativa entre el Ordenador Personal y el HPC de Intel con el algoritmo de la distancia euclidiana (dataset_1).

Para este escenario se muestra una comparación detalla de algoritmo de la distancia euclidiana haciendo uso del datase_1, con la finalidad de demostrar las ventajas del HPC frente al Ordenador Personal.

La Fig. 23 presenta la comparativa de los tiempos de procesamiento del algoritmo de la distancia euclidiana con listas, ejecutado desde el Ordenador Personal frente al HPC. En este caso la aceleración obtenida en el HPC es de $2{,}56\ x$ veces más rápido que el ordenador personal.

Fig. 23. Implementación con listas del algoritmo MDE en PC y HPC

Fuente: Propia del estudio

La Fig. 24 muestra una aceleración en el HPC de $1{,}99x$ veces más rápido que el ordenador personal, tomando con referencia los tiempos de procesamiento del algoritmo de la distancia euclidiana con Numpy.

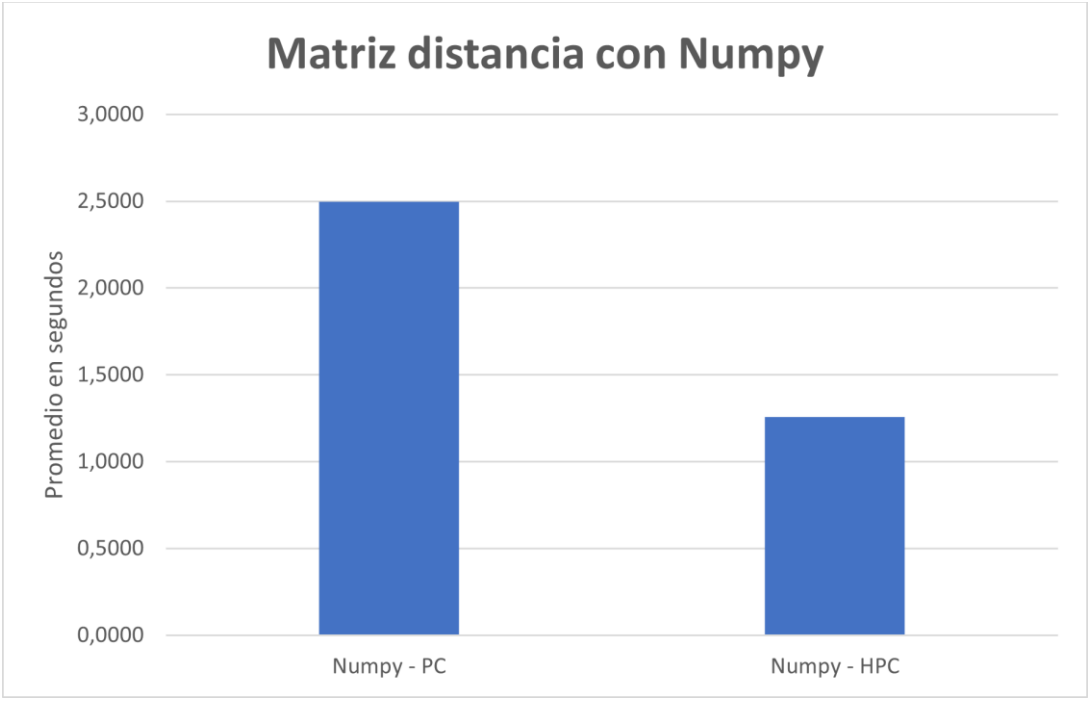

Fig. 24. Implementación con Numpy del algoritmo MDE en PC y HPC
Fuente: Propia del estudio

La Fig. 25 muestra una aceleración en el HPC de $1{,}06x$ veces más rápido que el ordenador personal, tomando con referencia los tiempos de procesamiento del algoritmo de la distancia euclidiana con Memoria Preasignada.

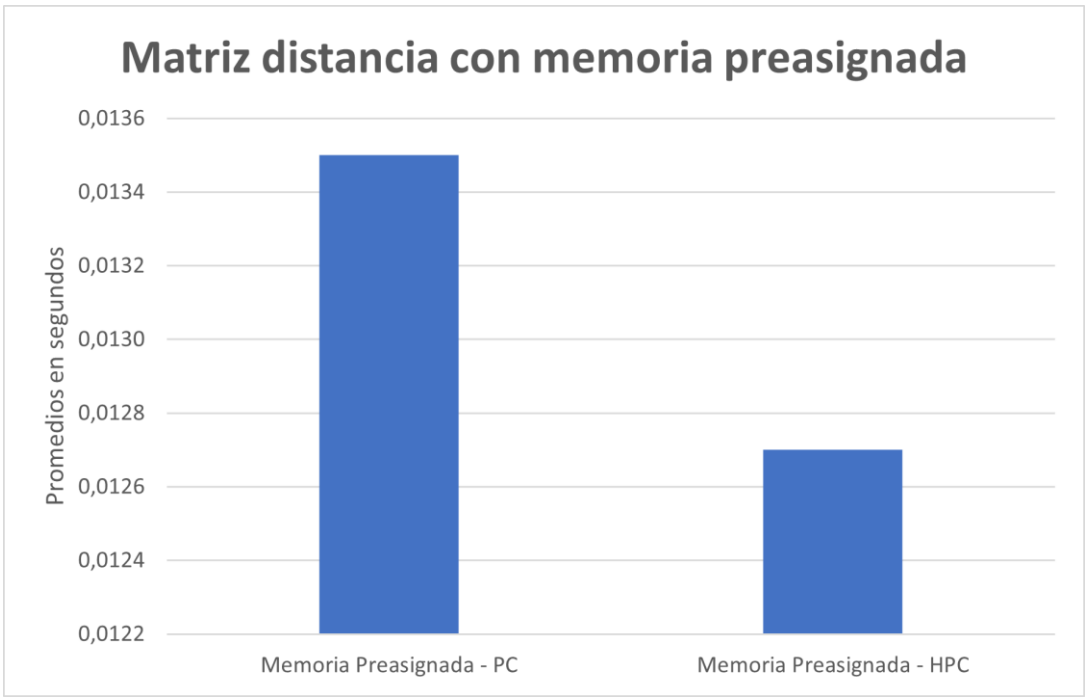

Fig. 25. Implementación con Memoria Preasignada del algoritmo MDE en PC y HPC

Fuente: Propia del estudio

La Fig. 25 muestra una situación particular, debido a que el tiempo de procesamiento en el Ordenador Personal es de $0{,}76\ x$ veces más rápido que el tiempo de procesamiento en el HPC.

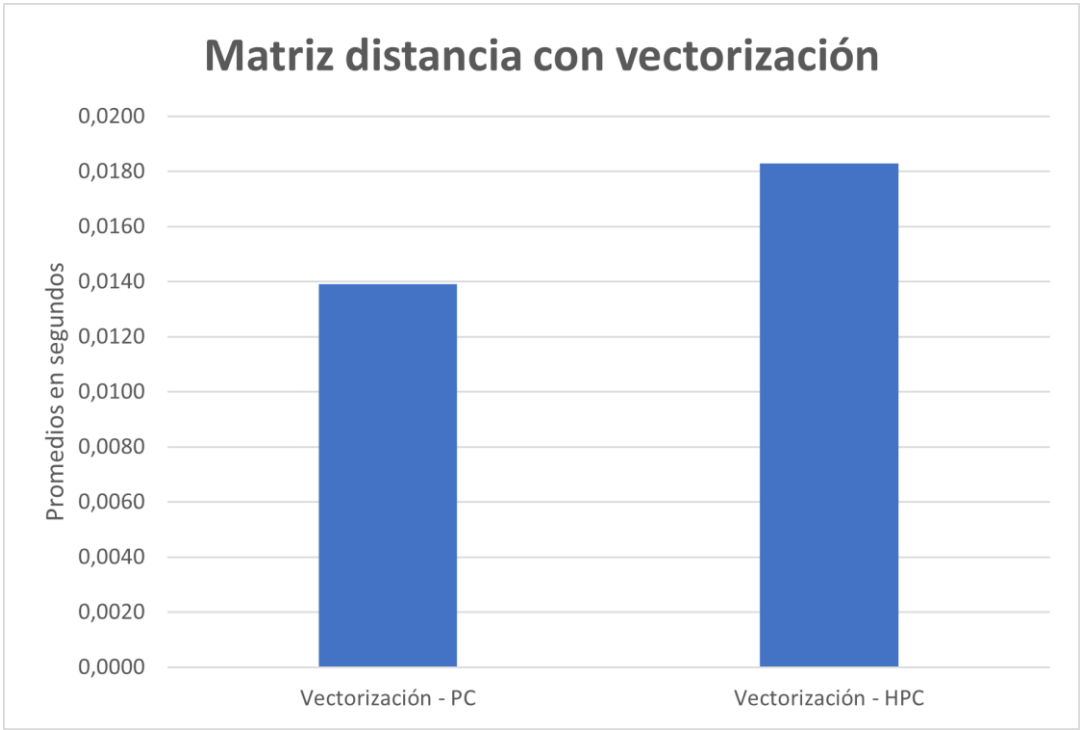

Fig. 26. Implementación con Vectorización del algoritmo MDE en PC y HPC

Fuente: Propia del estudio

La Fig. 27 muestra una aceleración del HPC de $1{,}31x$ veces más rápido que el ordenador personal, tomando con referencia los tiempos de procesamiento del algoritmo de la distancia euclidiana con Numba.

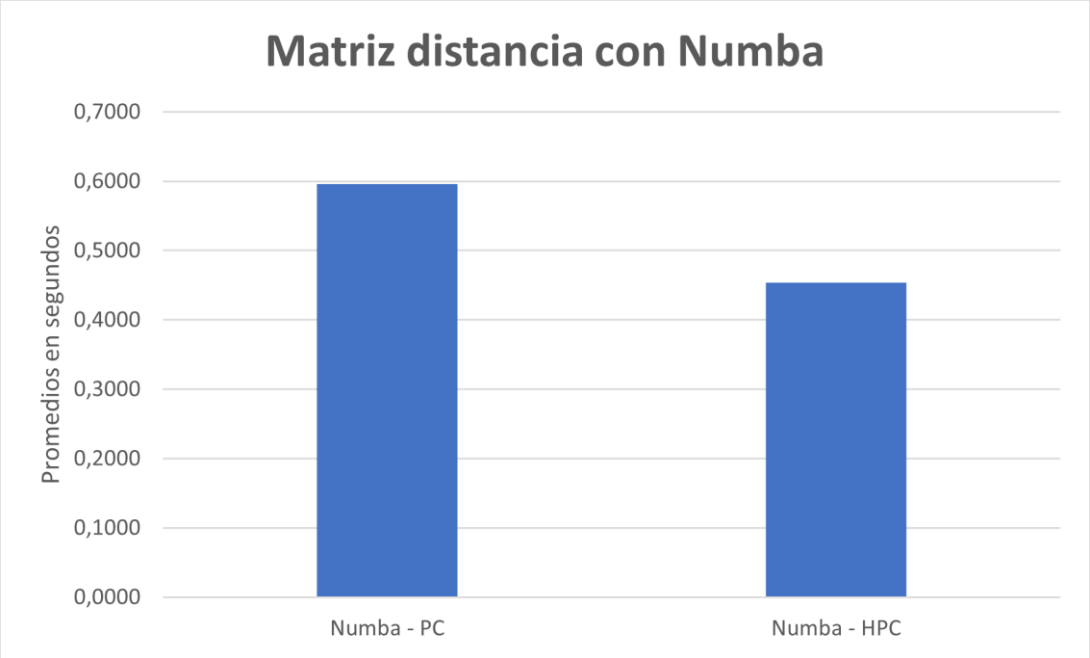

Fig. 27. Implementación con Numba del algoritmo MDE en PC y HPC
Fuente: Propia del estudio

2.5.6. Comparativa entre el Ordenador Personal y el HPC de Intel con el algoritmo de la distancia euclidiana (dataset_2).

La Fig. 28. presenta la comparativa de los tiempos de procesamiento del algoritmo de la distancia euclidiana con listas, ejecutado desde el Ordenador Personal frente al HPC. En este caso la aceleración obtenida en el HPC es de $2{,}58\ x$ veces más rápido que el ordenador personal.

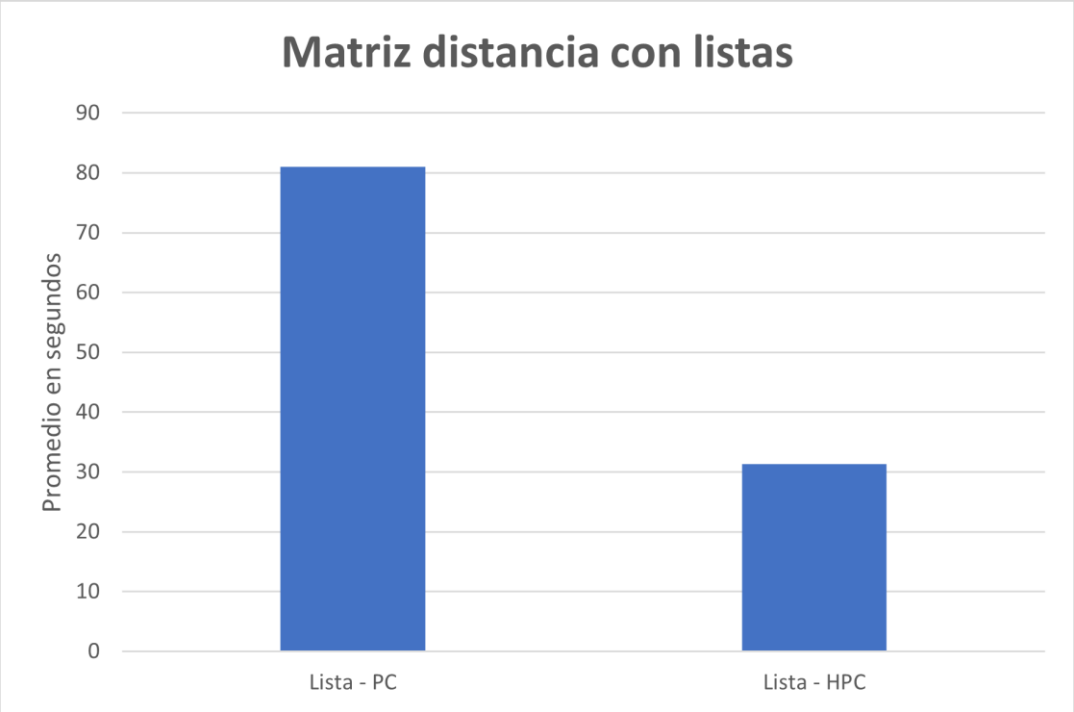

Fig. 28. Implementación con listas del algoritmo MDE en PC y HPC (dataset largo)
Fuente: Propia del estudio

La Fig. 29. muestra una aceleración del HPC de $2{,}63x$ veces más rápido que el ordenador personal, tomando con referencia los tiempos de procesamiento del algoritmo de la distancia euclidiana con Numpy.

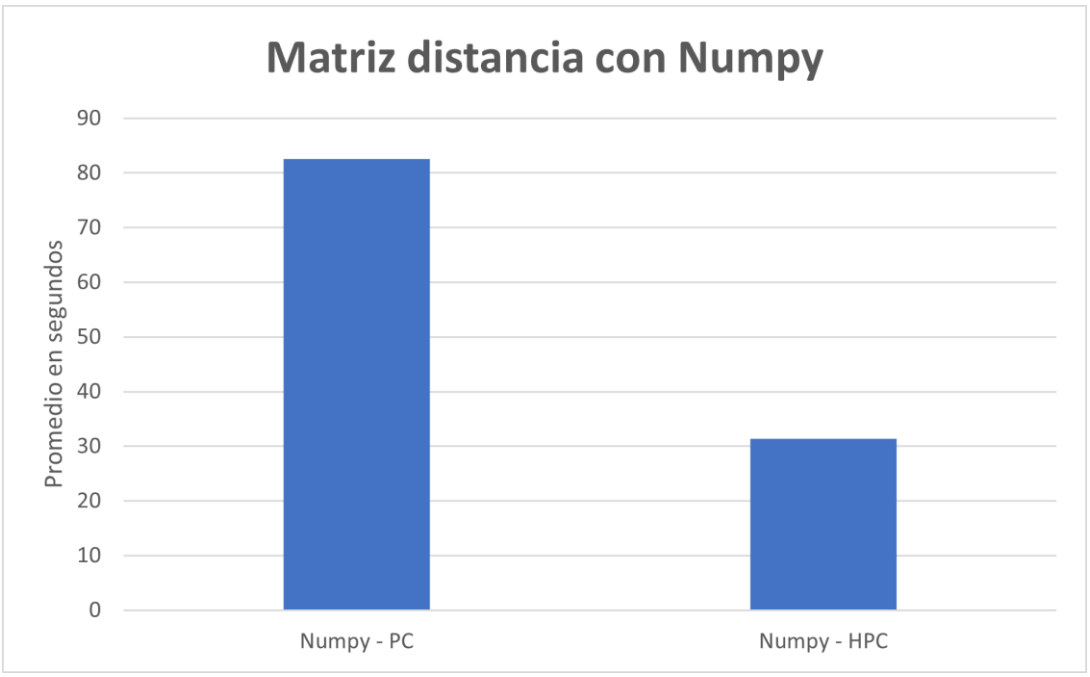

Fig. 29. Implementación con Numpy del algoritmo MDE en PC y HPC (dataset largo)

Fuente: Propia del estudio

La Fig. 30. muestra una aceleración del HPC de $2{,}38x$ veces más rápido que el ordenador personal, tomando con referencia los tiempos de procesamiento del algoritmo de la distancia euclidiana con Memoria Preasignada.

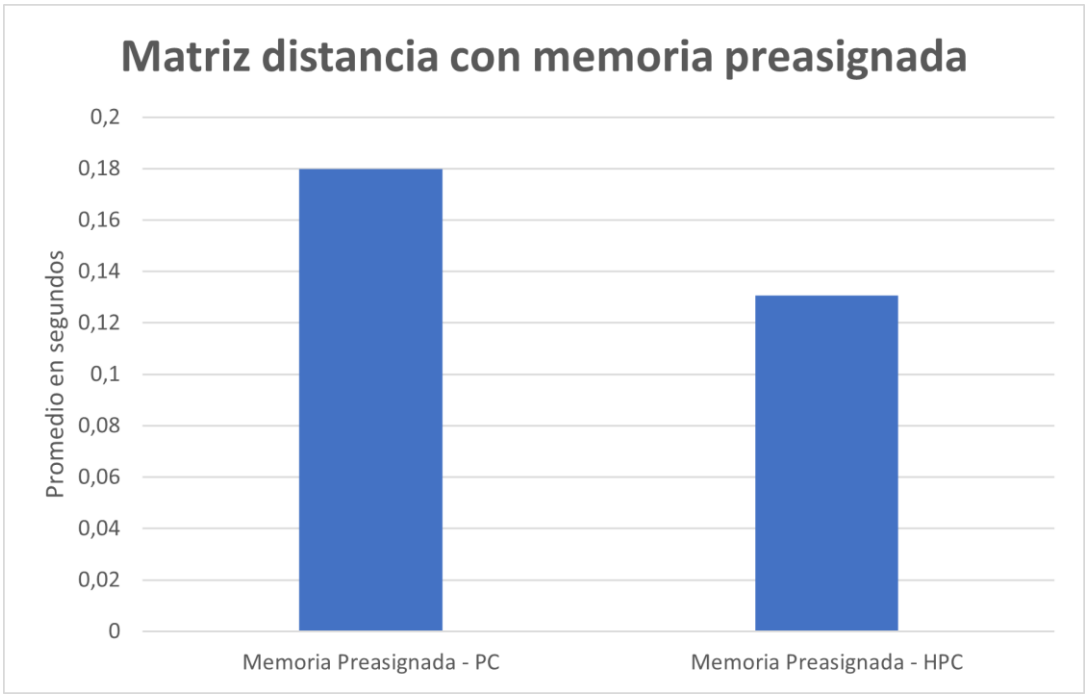

Fig. 30. Implementación con Memoria Preasignada del algoritmo MDE en PC y HPC (dataset largo)

Fuente: Propia del estudio

A diferencia de los resultados con dataset corto, en este caso en la Fig. 31 se muestra una aceleración de $2{,}02x$ veces en el HPC frente a los resultados obtenidos en el ordenador personal.

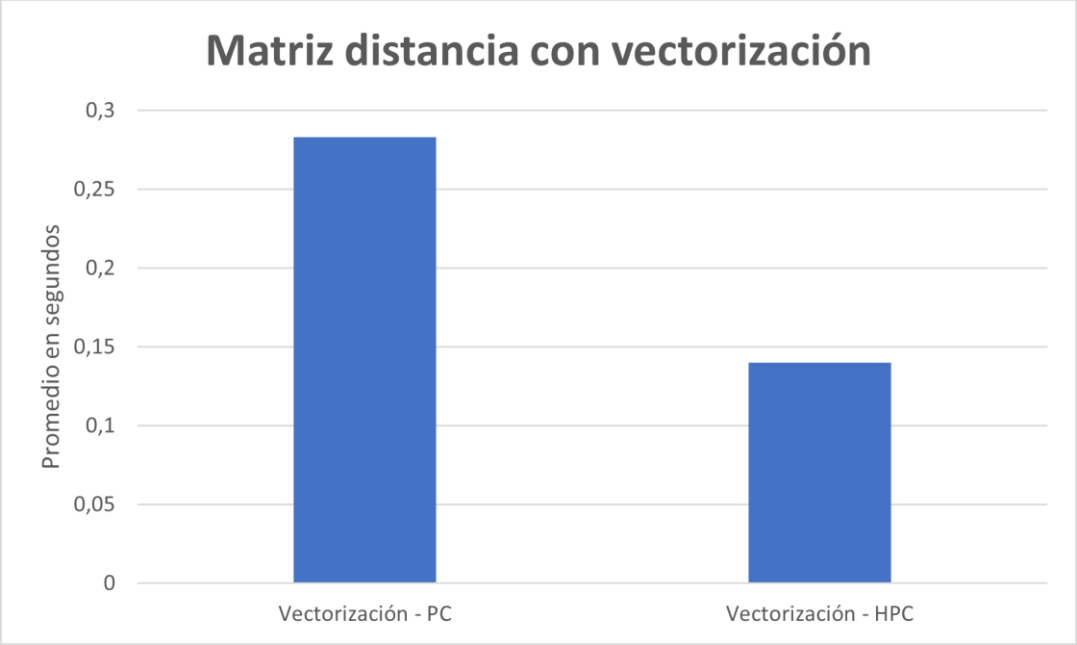

Fig. 31. Implementación con Vectorización del algoritmo MDE en PC y HPC (dataset largo)
Fuente: Propia del estudio

La Fig. 32. muestra una aceleración del HPC de $1{,}22x$ veces más rápido que el ordenador personal, tomando con referencia los tiempos de procesamiento del algoritmo de la distancia euclidiana con Numba.

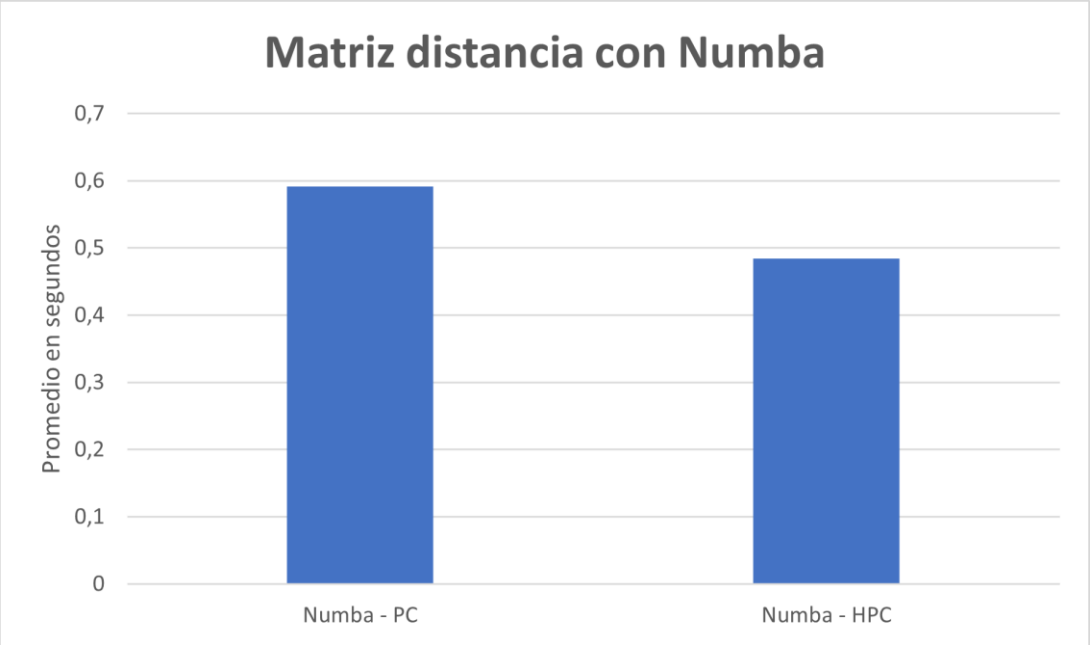

Fig. 32. Implementación con Numba del algoritmo MDE en PC y HPC (dataset largo)
Fuente: Propia del estudio

CAPÍTULO 3

Validación de resultados

3.1 Definición de métricas y factores de evaluación

Para evaluar la calidad y el rendimiento de los algoritmos existen diversos criterios. En este trabajo se utilizó los criterios recomendados por (Danko et al., 2015).

Además, se define que dos de los objetivos claves para lograr una aplicación óptima son:

- Rendimiento (Performance): que hace referencia a la capacidad de reducir el tiempo necesario para resolver un problema a medida que aumentan los recursos informáticos.
- Escalabilidad: que representa la capacidad de aumentar el rendimiento a medida que aumenta el tamaño del problema.

3.1.1. Tiempo de ejecución

El tiempo de ejecución de un algoritmo hace referencia al tiempo que tarda en ejecutarse o el tiempo que le toma en finalizar una tarea específica.

El tiempo de ejecución es un buen indicador de la funcionalidad de un algoritmo secuencial o paralelo. La medición del tiempo de ejecución utilizada está representada por el conteo de las unidades de segundos comprendidas entre el momento de inicio y termino de la ejecución de un algoritmo cualquiera.

3.1.2. Aceleración

Es una métrica muy habitual para expresar como de rápido o lento se ejecuta un algoritmo de forma paralela. Para la determinar la aceleración se utilizó la rapidez absoluta ($\boldsymbol{Sp}$); la misma que representa la razón de los tiempos entre el algoritmo serial ($\boldsymbol{Ts}$) y el tiempo del algoritmo paralelizado u optimizado ($\boldsymbol{Tp}$).

La fórmula de la aceleración de representa de la siguiente manera:

$$\boldsymbol{Sp} = \frac{\boldsymbol{Ts}}{\boldsymbol{Tp}}$$

3.2. Análisis e interpretación de resultados

3.2.1. Análisis del tiempo de procesamiento secuencial del algoritmo del problema de los N-Body

La implementación del algoritmo N-Body de forma secuencial se la realizó con el objetivo de hacer una comparativa de los tiempos de procesamiento de los lenguajes de programación Python, C y Java.

En la sección 2.5.1 se muestra el promedio de los tiempos de procesamiento para los diferentes valores que toma $\boldsymbol{N}$ de acuerdo con la Tabla 5.

En la Fig. 33 muestra que el lenguaje C presenta el tiempo de procesamiento óptimo frente a los otros lenguajes. Adicional a esto, el lenguaje Python tiene una curva de crecimiento muy alta en su tiempo de procesamiento a medida que aumenta el tamaño de $\boldsymbol{N}$.

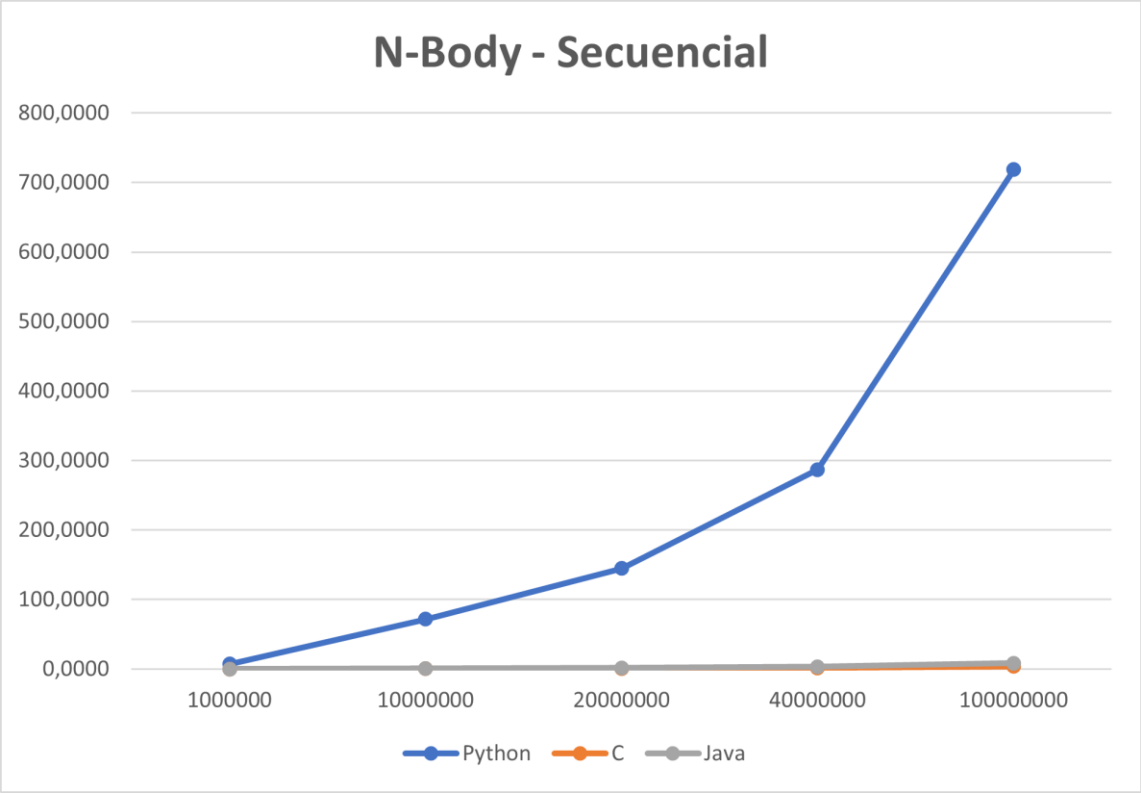

Fig. 33. Análisis del algoritmo N-Body

Fuente: Propia del estudio

3.2.2. Análisis del tiempo de procesamiento del algoritmo de la Distancia Euclidiana en Python (Ordenador Personal).

La Fig. 34 representa de forma gráfica el promedio de los tiempos de procesamiento de las cinco implementaciones del algoritmo de la distancia euclidiana con Python sobre el ordenador personal.

Inicialmente es posible identificar que las implementaciones con Lista y Numpy, presentan una mayor curva de crecimiento en el tiempo de procesamiento debido a que se ejecutan de forma secuencial.

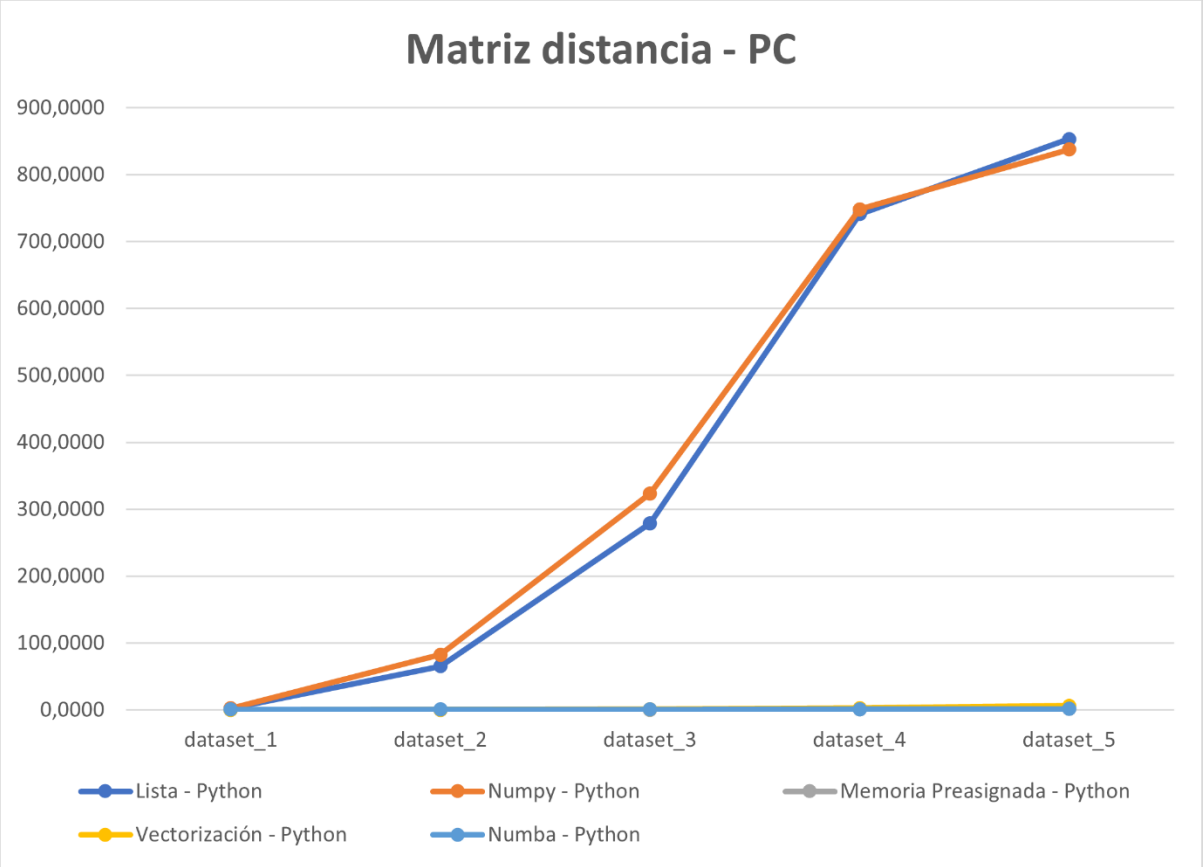

Fig. 34. Análisis del algoritmo de la MDE – PC

Fuente: Propia del estudio

La Fig. 35 muestra un análisis más cercano de los tiempos de procesamiento de las implementaciones optimizadas del algoritmo de la distancia euclidiana.

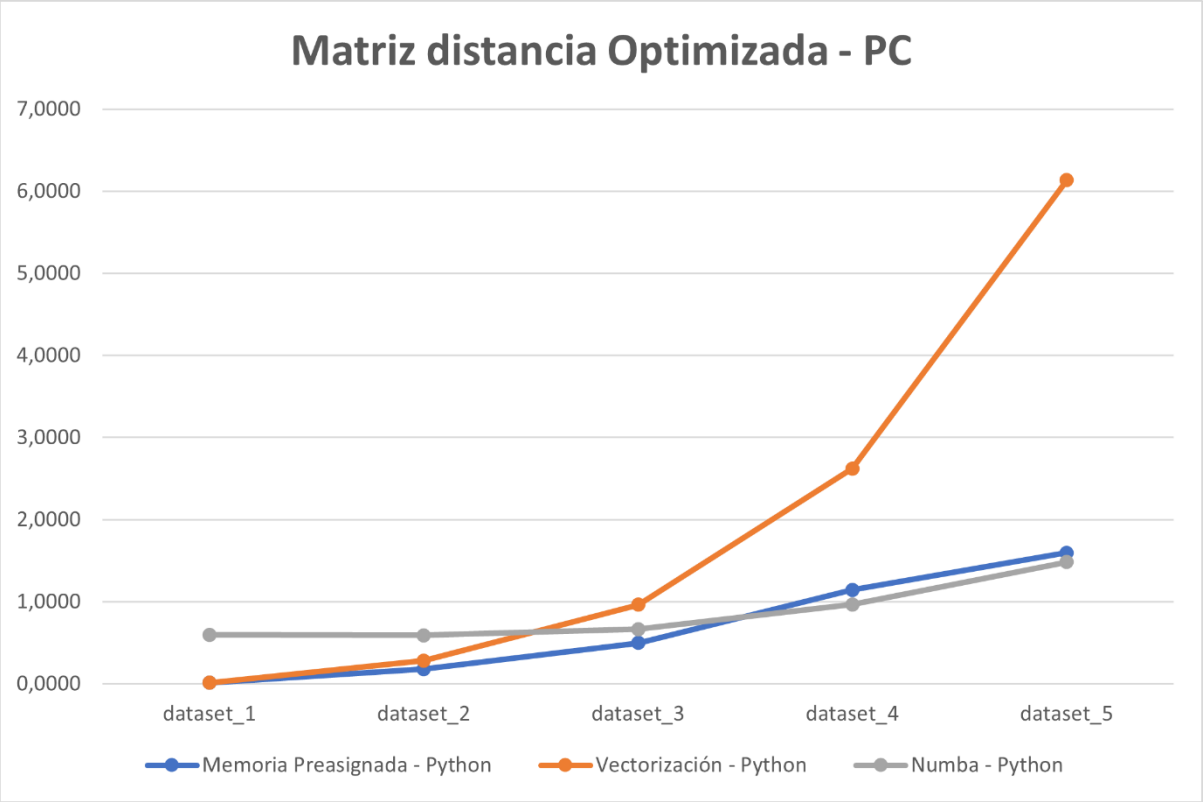

Fig. 35. Análisis del algoritmo de la MDE Optimizado – PC

Fuente: Propia del estudio

De forma más detallada se muestra que con el $dataset\_1$ la implementación que presentó el tiempo óptimo fue el algoritmo con Memoria Preasignada, con una aceleración de $1{,}03x\ veces$ más rápido que el algoritmo con Vectorización y de $44{,}15x\ veces$ que la implementación con Numba.

Pero, también es posible apreciar que con el $dataset\_4$ la implementación que presentó el tiempo óptimo fue el algoritmo con Numba, con una aceleración de $1{,}08x\ veces$ más rápido que el algoritmo con Memoria Preasignada y de $4{,}14x\ veces$ que la implementación con Vectorización.

3.2.3. Análisis del tiempo de procesamiento del algoritmo de la Distancia Euclidiana en Python (HPC).

La Fig. 35 representa de forma gráfica el promedio de los tiempos de procesamiento de las cinco implementaciones del algoritmo de la distancia euclidiana con Python sobre el HPC de Intel.

Es posible identificar que las implementaciones con Lista y Numpy, presentan una mayor curva de crecimiento en el tiempo de procesamiento debido a que se ejecutan de forma secuencial igual que en el Ordenador Personal.

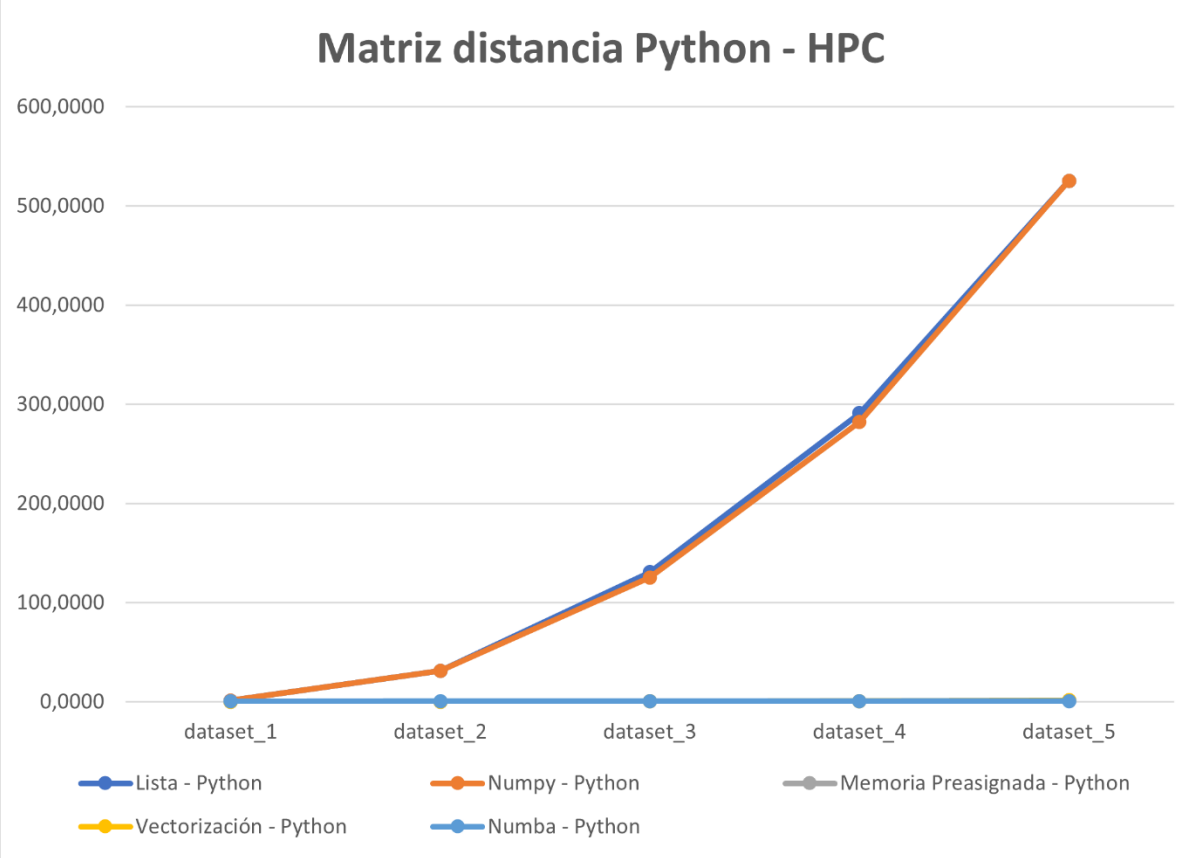

Fig. 36. Análisis del algoritmo de la MDE – HPC

Fuente: Propia del estudio

La Fig. 37 muestra un análisis más cercano de los tiempos de procesamiento de las implementaciones optimizadas del algoritmo de la distancia euclidiana.

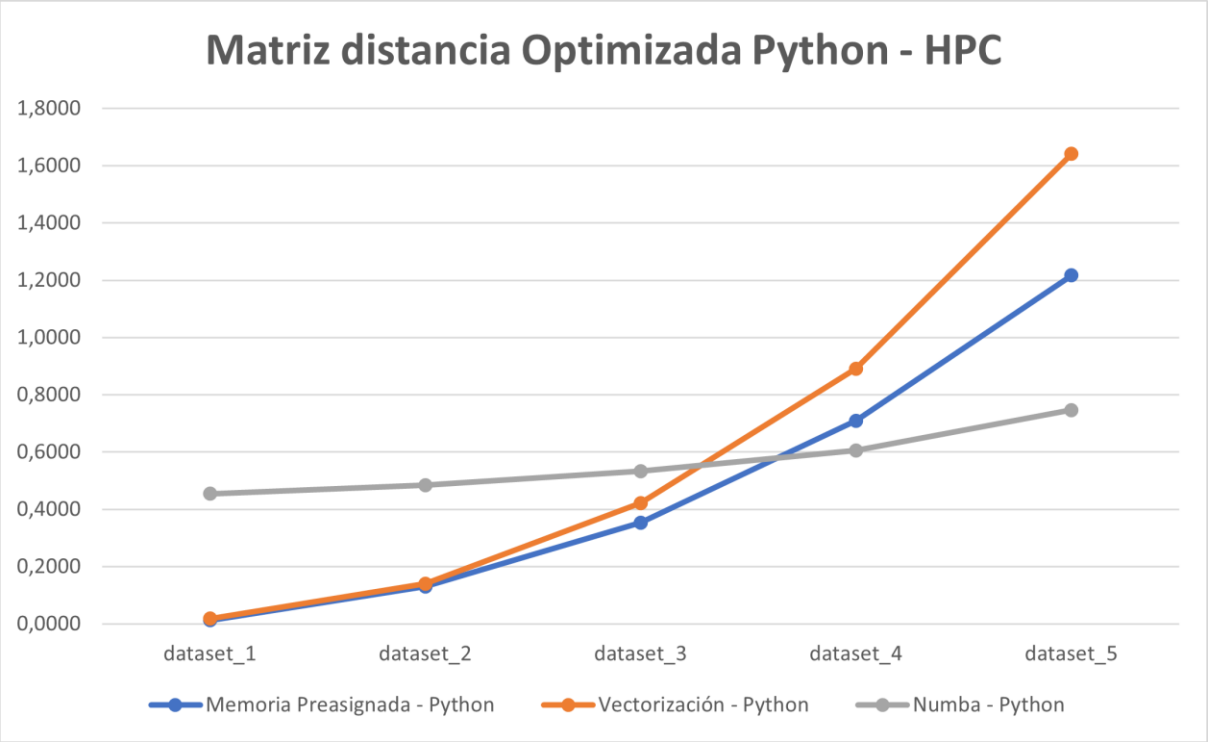

Fig. 37. Análisis del algoritmo de la MDE Optimizado – HPC

Fuente: Propia del estudio

De forma más detallada se muestra que con el $dataset\_1$ la implementación que presentó el tiempo óptimo fue el algoritmo con Memoria Preasignada, con una aceleración de $1{,}44x\ veces$ más rápido que el algoritmo con Vectorización y de $35{,}75x\ veces$ que la implementación con Numba.

Adicionalmente, con el $dataset\_4$ la implementación que presentó el tiempo óptimo fue el algoritmo con Numba, con una aceleración de $1{,}63x\ veces$ más rápido que el algoritmo con Memoria Preasignada y de $2{,}20x\ veces$ que la implementación con Vectorización.

3.2.4. Análisis del tiempo de procesamiento del algoritmo de la Distancia Euclidiana en C (HPC).

La Fig. 38 representa de forma gráfica el promedio de los tiempos de procesamiento de la implementación en secuencial y paralela del algoritmo de la distancia euclidiana con C sobre el HPC de Intel.

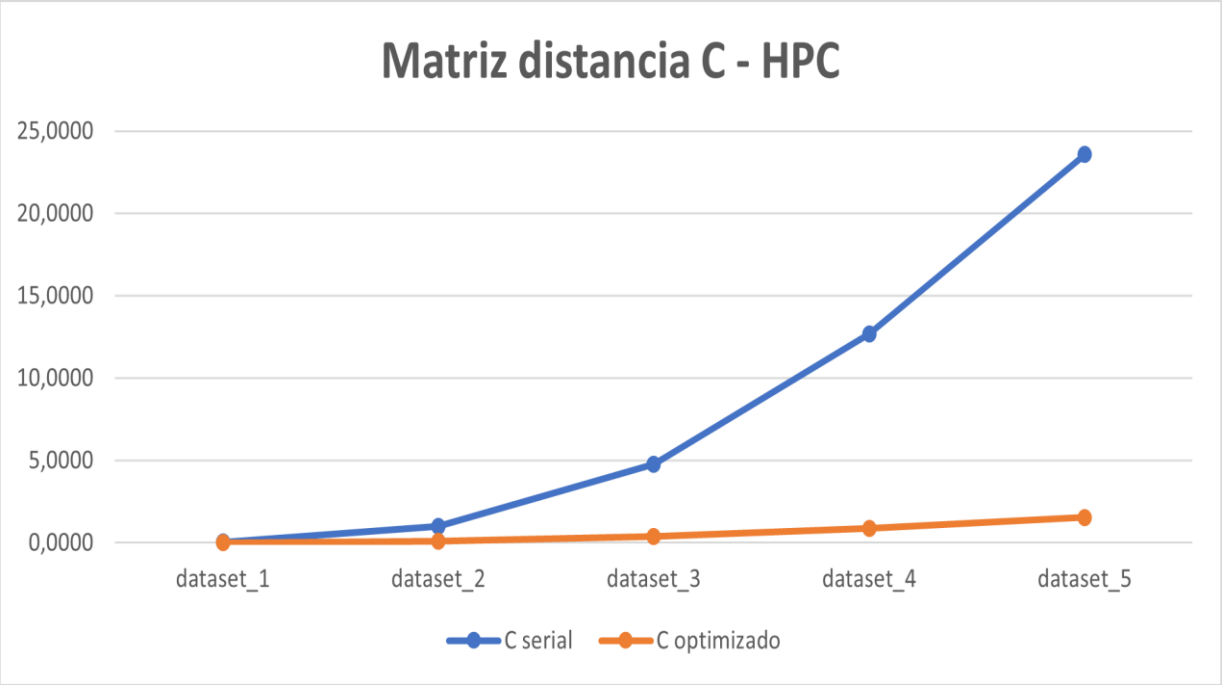

Fig. 38. Análisis del algoritmo de la MDE en C – HPC

Fuente: Propia del estudio

En C, con el $dataset\_1$ la implementación optimizada presentó una aceleración de $18{,}58x\ veces$ frente a la implementación en secuencial.

Adicionalmente, con el $dataset\_5$ la implementación optimizada presentó una aceleración de $15{,}32x\ veces$ frente a la implementación en secuencial.

3.2.5. Análisis de las optimizaciones en Python vs C.

En este punto se realizó un análisis de las versiones optimizadas del algoritmo de la distancia euclidiana entre Python y C.

Como se presentó en la Fig. 35 y Fig. 37 la implementación de Python utilizando Numba, inicialmente con los datasets 1, 2,3 y 4 no fue la solución óptima. Pero con el dataset_5 si lo fue, además tuvo la curva de crecimiento más baja con respecto a las otras implementaciones. Por este motivo, se utilizó dos datasets adicionales (Tabla 27) para demostrar cómo la implementación con Numba mejora a medida que se aumenta el tamaño de los datos.

Tabla 27. Datasets adicionales

Datasets	
dataset_6	35000
dataset_7	52500

En la Tabla 28. y Tabla 29 se muestra el promedio de las diferentes implementaciones con los datasets 6 y 7.

Tabla 28. Promedio de procesamiento MDE (dataset_6) HPC

Distancia Euclidiana Python y C	
Memoria Preasignada	7,8755
Vectorización	14,7322
Numba	2,0651
C optimizada	10,0361

Tabla 29. Promedio de procesamiento MDE (dataset_7) HPC

Distancia Euclidiana Python y C	
Memoria Preasignada	18,4885
Vectorización	35,2441
Numba	4,4388
C optimizada	22,6401

Finalmente, se pudo identificar que la implementación con Numba muestra una mejor aceleración para volúmenes de datos más grades, mientras que la implentación Optimizada de C muestra ser la óptima para volúmenes de datos más pequeños.

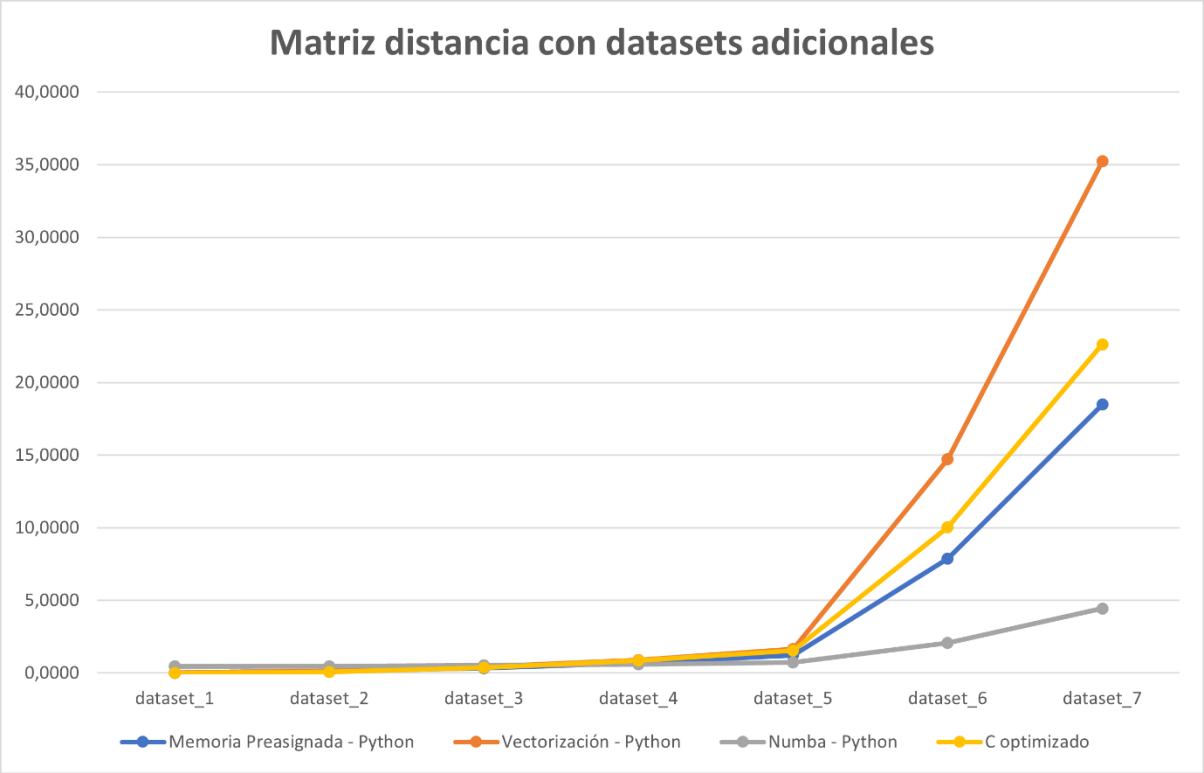

Fig. 39. Análisis de la MDE con los datasets adicionales

Fuente: Propia del estudio

Conclusiones

En el ámbito de la investigación científica C es el lenguaje de programación más utilizado para realizar tareas de alto procesamiento.

El compilador de C mantenido por Intel (icc), está mejor optimizado para realizar tareas de procesamiento en paralelo. Además, proporciona varias directivas de compilador para realizar optimizaciones y transformaciones de bucles más agresivas.

La librería de terceros Numba desarrollada para Python, brinda un alto nivel de optimización en tiempo de procesamiento para algoritmos que utilizan la librería Numpy, sin hacer cambios significativos en el código.

El lenguaje de programación Python por sí sólo, presentó un tiempo de procesamiento mucho más alto en comparación con los lenguajes Java y C. Esto se debe a que es un lenguaje de programación interpretador y puede ejecutarse en un solo hilo, lo que hace que no pueda aprovecharse de todas las características del hardware por sí solo.

Python tiene una extensa cantidad de librería de terceros y una gran comunidad de desarrolladores activa, que buscan proporcionar mejoras de rendimiento y optimizaciones de código para este lenguaje.

Recomendaciones

Si se busca implementar algoritmos donde su tiempo de ejecución es primordial, usar el lenguaje de programación C sería la mejor opción, por su alto rendimiento.

El lenguaje de programación Python está presentando de forma continua, alternativas para ayudar a mejorar los tiempos de procesamiento que podrían ayudarnos a desarrollar soluciones en un menor tiempo.

C es un lenguaje de programación de nivel medio, que produce código de máquina que se ejecuta mucho más rápido que otros lenguajes de programación como Java, Python Javascript, etc.

Si necesita implementar algoritmos o aplicaciones donde su tiempo de ejecución no es la prioridad sino su tiempo de desarrollo y fácil implementación, el lenguaje de programación Python sería una muy buena opción. Además, es un lenguaje de programación fácil de aprender para aquellas personas que no tienen mucha experiencia en el campo de la programación.

Se recomienda utilizar el Intel Devcloud para realizar tareas que requieran un alto nivel de procesamiento y un largo tiempo de ejecución. Este proporciona diferentes nodos con diferentes características a nivel de hardware que podrían ayudar en la mejora del rendimiento.

Referencias

Barik, S. (14 de 03 de 2021). *Medium*. Obtenido de https://medium.com/analytics-vidhya/is-python-really-very-slow-2-major-problems-to-know-which-makes-python-very-slow-9e92653265ea#:~:text=In%20a%20nutshell,the%20run%20without%20our%20knowledge.

Britannica T. (23 de Noviembre de 2011). *Complejidad Computacional*. Obtenido de https://www.britannica.com/topic/computational-complexity

Data@Urban. (18 de Septiembre de 2018). *Using Multiprocessing to Make Python Code Faster*. Obtenido de https://urban-institute.medium.com/using-multiprocessing-to-make-python-code-faster-23ea5ef996ba

Dey, S. (15 de 08 de 2016). *Medium*. Obtenido de https://medium.com/@souravdey/l2-distance-matrix-vectorization-trick-26aa3247ac6c

Doug Bagley. (2021). *Benchmarks Game*. Obtenido de https://benchmarksgame-team.pages.debian.net/benchmarksgame/

Embedded Staff. (9 de Julio de 2008). *Getting started with multicore programming: Part 2 – Multithreading in C*. Obtenido de https://www.embedded.com/getting-started-with-multicore-programming-part-2-multithreading-in-c/

Escobar, G. (2019). *Complejidad (Big-O)*. Obtenido de https://guias.makeitreal.camp/algoritmos/complejidad#:~:text=Complejidad%20lineal%20%2D%20O(n),operaciones%20aumenta%20de%20forma%20proporcional.&text=En%20este%20caso%20la%20complejidad,longitud%20de%20la%20segunda%20lista.

Evanczuk, S. (19 de Febrero de 2013). *Best practices for multicore programming*. Obtenido de https://www.edn.com/best-practices-for-multicore-programming/

Firesmith, D. (21 de Agosto de 2017). *Multicore Processing*. Obtenido de https://insights.sei.cmu.edu/sei_blog/2017/08/multicore-processing.html

GoLang. (s.f.). *Documentation*. Obtenido de https://golang.org/doc/

Hoffman, C. (12 de Octubre de 2018). *CPU Basics: Multiple CPUs, Cores, and Hyper-Threading Explained*. Obtenido de https://www.howtogeek.com/194756/cpu-basics-multiple-cpus-cores-and-hyper-threading-explained/

Intel. (2021). *Intel*. Obtenido de Intel: https://devcloud.intel.com/oneapi/get_started/

Intel Corporation. (2020). *¿Qué es Hyper-Threading?* Obtenido de https://www.intel.la/content/www/xl/es/gaming/resources/hyper-threading.html

Martín, L. E. (2017). *Algoritmo para la resolución física en tiempo lineal del problema de los n-cuerpos en colonias de bacterias.* Madrid.

Millar, D. (2020). *How Many Cores Is More Always Better* . Obtenido de https://www.newcmi.com/blog/how-many-cores

Multicore Association. (31 de Diciembre de 2020). *What Is Multicore*. Obtenido de https://www.multicore-association.org/

Numba. (01 de 01 de 2022). *Numba*. Obtenido de https://numba.pydata.org/

Numpy. (11 de 01 de 2022). *Numpy*. Obtenido de https://numpy.org/devdocs/user/whatisnumpy.html

Padmanabhan, A. (05 de Junio de 2020). *Algorithmic Complexity* . Obtenido de https://devopedia.org/algorithmic-complexity

Peale, S. J. (12 de 02 de 2009). *Britannica*. Obtenido de https://www.britannica.com/science/ultraviolet-astronomy

PYPL index. (2022). *PYPL PopularitY of Programming Language*. Obtenido de PYPL PopularitY of Programming Language: https://pypl.github.io/PYPL.html

Sagar, A. (03 de Noviembre de 2019). *A Hands on Guide to Multiprocessing in Python*. Obtenido de https://towardsdatascience.com/a-hands-on-guide-to-multiprocessing-in-python-48b59bfcc89e

Shah, R. (30 de 04 de 2021). *Analytics Vidhya*. Obtenido de Analytics Vidhya: https://www.analyticsvidhya.com/blog/2021/04/python-list-programs-for-absolute-beginners/#:~:text=Lists%20are%20used%20to%20store,is%20that%20Lists%20are%20mutable.

Anexos

Anexo A: Conjunto de datos analizados en la comparativa de rendimiento de los lenguajes de programación.

Se utilizaron 7 datasets para las pruebas de rendimiento, que se encuentran disponibles en el siguiente git:

https://github.com/NeiderRequene/ProcesamientoMulticore