UNIVERSIDAD DE GRANADA

TRABAJO FIN DE GRADO

INGENIERÍA INFORMÁTICA

Dockerización de Aplicación Paralela y Distribuida para Clasificación de EEGs: Análisis de Viabilidad y Rendimiento

DockEEG

Autor
Fernando Cuesta Bueno

Director
Juan José Escobar Pérez

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Granada, septiembre de 2025

UNIVERSIDAD
DE GRANADA

DockEEG

Dockerización de Aplicación Paralela y Distribuida para Clasificación de EEGs: Análisis de Viabilidad y Rendimiento

Autor
Fernando Cuesta Bueno

Director
Juan José Escobar Pérez

DockEEG - Dockerización de Aplicación Paralela y Distribuida para Clasificación de EEGs: Análisis de Viabilidad y Rendimiento

Fernando Cuesta Bueno

Palabras clave: palabra_clave1, palabra_clave2, palabra_clave3, ...

Resumen

Poner aquí el resumen.

DockEEG - Dockerization of a Parallel and Distributed Application for EEG Classification: Feasibility and Performance Analysis

Fernando Cuesta Bueno

Keywords: Keyword1, Keyword2, Keyword3, ...

Abstract

Write here the abstract in English.

Yo, **Fernando Cuesta Bueno**, alumno de la titulación Graduado en Ingeniería Informática de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 77150866B, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Fernando Cuesta Bueno

Granada a 5 de septiembre de 2025.

D. **Juan José Escobar Pérez**, Profesor del Área de XXXX del Departamento de Lenguajes y Sistemas Informáticos de la Universidad de Granada.

Informa:

Que el presente trabajo, titulado ***Dockerización de Aplicación Paralela y Distribuida para Clasificación de EEGs: Análisis de Viabilidad y Rendimiento, DockEEG***, ha sido realizado bajo su supervisión por **Fernando Cuesta Bueno**, y autorizo la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a 5 de septiembre de 2025.

El director:

Juan José Escobar Pérez

Agradecimientos

Poner aquí agradecimientos...

Índice general

Acrónimos **1**

1. Introducción **3**
1.1. Motivación 5
1.2. Objetivos 5
1.2.1. Objetivos específicos 5

2. Gestión del Proyecto **7**
2.1. Tareas 7
2.2. Planificación temporal 11
2.3. Estimación de costes 12

3. Estado del arte **15**
3.1. Computación de alto rendimiento (HPC) 15
3.1.1. Virtualización vs contenerización 17
3.1.2. Ecosistema de contenedores en HPC 20
3.1.3. Limitaciones de la contenerización en macOS 23
3.2. Portabilidad y reproducibilidad científica 24
3.2.1. Cómo los contenedores facilitan la portabilidad de aplicaciones HPC 24
3.2.2. Reproducibilidad de experimentos científicos usando contenedores 25

4. Propuesta principal **27**
4.1. Introducción de la propuesta 27
4.2. Aplicación seleccionada: HPMoon 27
4.2.1. Origen y contexto 27
4.2.2. Objetivo de HPMoon 28
4.2.3. Arquitectura y funcionamiento 28
4.2.4. Justificación de la elección para este TFG 29
4.2.5. Configuración y parámetros de compilación y ejecución 30
4.2.6. Selección de parámetros de estudio 31
4.2.7. Diseño de los experimentos 34
4.3. Herramientas y scripts utilizados 36

4.3.1. Compilación y preparación 36
4.3.2. Pruebas de escalabilidad 36
4.3.3. Automatización total y orquestación 37
4.3.4. Valor añadido e innovación 37
4.4. Repositorio del proyecto 38

5. Experimentación **39**
5.1. Experimentos preliminares 39
5.1.1. Determinación del número óptimo de subpoblaciones y hebras 39
5.1.2. Estudio del rendimiento al requerir más hebras de las disponibles 41
5.1.3. Estudio del rendimiento al utilizar la misma GPU en distintos nodos 45
5.1.4. Análisis de los experimentos preliminares 47
5.2. Pruebas mononodo 48
5.2.1. Ejecución en Ubuntu en nativo 48
5.2.2. Ejecución en contenedores de Ubuntu 52
5.2.3. Contenedores en contenedores de Windows 58
5.2.4. Contenedores en contenedores de Mac 60
5.3. Pruebas multinodo 65
5.3.1. Ejecución en Ubuntu en nativo 65
5.3.2. Ejecución en contenedores de Ubuntu 65
5.3.3. Contenedores en contenedores de Windows 65
5.3.4. Contenedores en contenedores de Mac 65
5.4. Pruebas de barrido de hebras 65
5.4.1. Ejecución en Ubuntu en nativo 65
5.4.2. Ejecución en contenedores de Ubuntu 65
5.4.3. Contenedores en contenedores de Windows 65
5.4.4. Contenedores en contenedores de Mac 65

6. Conclusiones y trabajo futuro **67**
6.1. Contribuciones 67
6.2. Retos y trabajo futuro 67

Bibliografía **71**

Índice de figuras

2.1. Diagrama de Gantt del proyecto 11

3.1. Comparación entre la arquitectura de una máquina virtual y la de un contenedor. Fuente: `https://electronicaonline.net/software-y-apps/windows/maquina-virtual/` 19

5.1. Gráficas de ejecución de las pruebas variando el número de subpoblaciones y hebras. 40
5.2. Gráfica de tiempo de ejecución en función del número de hebras por nodo, con límite de 16 hebras. 42
5.3. Gráfica de uso de CPU en función del número de hebras por nodo, con límite de 16 hebras. 42
5.4. Gráfica de tiempo de ejecución en función del número de hebras por nodo, sin límite en el número de hebras. 43
5.5. Gráfica de uso de CPU en función del número de hebras por nodo, sin límite en el número de hebras. 43
5.6. Gráfica de tiempo de ejecución en función del número de hebras por nodo, con la GPU limitada a un único nodo. 45
5.7. Gráfica de tiempo de ejecución en función del número de hebras por nodo, permitiendo el uso de la GPU en todos los nodos. 46
5.8. Tiempo de ejecución en función del número de hebras en Ubuntu nativo (CPU). 48
5.9. Uso de CPU en función del número de hebras en Ubuntu nativo (CPU). 49
5.10. Tiempo de ejecución en función del número de hebras en Ubuntu nativo (CPU + GPU). 50
5.11. Comparativa de tiempo de ejecución entre CPU y CPU + GPU en función del número de hebras en Ubuntu nativo. . . 51
5.12. Tiempo de ejecución en un único nodo con contenedores de Ubuntu (CPU). 53
5.13. Tiempo de ejecución en un único nodo con contenedores de Ubuntu gestionados por Podman (CPU). 54

5.14. Tiempo de ejecución en un único nodo con contenedores de Ubuntu (CPU + GPU). 55
5.15. Tiempo de ejecución en un único nodo con contenedores de Ubuntu gestionados por Podman (CPU + GPU). 56
5.16. Comparativa de tiempo de ejecución entre nativo y contenedores de Ubuntu en función del número de hebras, para CPU. 57
5.17. Tiempo de ejecución en un único nodo con Docker en Windows (CPU). 58
5.18. Tiempo de ejecución en un único nodo con Podman en Windows (CPU). 59
5.19. Tiempo de ejecución en un único nodo con Docker en Mac (CPU). 61
5.20. Tiempo de ejecución en un único nodo con Podman en Mac (CPU). 62

Índice de tablas

2.1. Planificación temporal de tareas y horas estimadas 11
2.2. Costes estimados de hardware para el proyecto 13
2.3. Costes estimados de recursos humanos para el proyecto . . . 13
2.4. Coste total estimado del proyecto 13

3.1. Comparativa entre máquinas virtuales y contenedores en entornos HPC, incluyendo resultados de Sharma et al. (2016) [1]. 19

4.1. Rango de valores de los parámetros de entrada y su uso desde la línea de argumentos (si está disponible). 32

5.1. Tiempos de ejecución en segundos de las pruebas variando el número de subpoblaciones y hebras. 40
5.2. Porcentaje de reducción del tiempo de ejecución respecto a la configuración base para distintas combinaciones de hebras y subpoblaciones 40
5.3. Resumen de configuraciones de nodos, hebras y uso de CPU . 44
5.4. Resumen de tiempos de ejecución para distintas combinaciones de nodos y hebras, comparando el uso de la GPU limitada a un nodo frente a su uso en todos los nodos. 46
5.5. Tiempos de ejecución y reducción porcentual respecto a una hebra en Ubuntu nativo (CPU). 48
5.6. Porcentaje de uso de CPU y eficiencia en función del número de hebras en Ubuntu nativo (CPU). 49
5.7. Tiempos de ejecución y reducción porcentual respecto a una hebra en Ubuntu nativo (CPU + GPU). 51
5.8. Comparativa de tiempos de ejecución y variación porcentual entre CPU y CPU+GPU en Ubuntu nativo. 52
5.9. Tiempos de ejecución y reducción porcentual respecto a una hebra en contenedores de Ubuntu (CPU). 53
5.10. Tiempos de ejecución y reducción porcentual respecto a una hebra en contenedores de Ubuntu gestionados por Podman (CPU). 54

5.11. Tiempos de ejecución y reducción porcentual respecto a una hebra en contenedores de Ubuntu (CPU + GPU). 55
5.12. Tiempos de ejecución y reducción porcentual respecto a una hebra en contenedores de Ubuntu gestionados por Podman (CPU + GPU). 57
5.13. Comparativa de tiempos de ejecución entre nativo, Docker y Podman (CPU+GPU) y variación porcentual respecto a nativo. 58
5.14. Tiempos de ejecución y reducción porcentual respecto a una hebra en Docker sobre Windows (CPU). 59
5.15. Tiempos de ejecución y reducción porcentual respecto a una hebra en Podman sobre Windows (CPU). 60
5.16. Tiempos de ejecución y reducción porcentual respecto a una hebra en Docker sobre Mac (CPU). 61
5.17. Tiempos de ejecución y reducción porcentual respecto a una hebra en Podman sobre Mac (CPU). 62

Acrónimos

PLC	Programmable Logic Controller
HPC	High-Performance Computing
EEG	Electroencephalogram
BCI	Brain-Computer Interface
MOFS	Multi-Objective Feature Selection
MPI	Message Passing Interface
OpenMP	Open Multi-Processing
OpenCL	Open Computing Language
MOGA	Multi-Objective Genetic Algorithm
NSGA-II	Non-dominated Sorting Genetic Algorithm II
WCSS	Within-Cluster Sum of Squares
BCSS	Between-Cluster Sum of Squares

Capítulo 1

Introducción

En el ámbito biomédico y de la bioingeniería, el análisis de señales de electroencefalograma (EEG) para el desarrollo de Interfaces Cerebro-Computadora (Brain-Computer Interfaces, BCI) constituye un reto de gran relevancia. Las BCI permiten a personas con discapacidades motoras recuperar capacidades de comunicación y control sobre dispositivos externos, facilitando su autonomía y mejorando su calidad de vida. Asimismo, estas tecnologías se están explorando para aplicaciones médicas avanzadas, como la rehabilitación neurológica, la detección temprana de trastornos cerebrales y el control de prótesis inteligentes, lo que convierte el procesamiento eficiente de datos EEG en un requisito crítico para traducir los avances científicos en beneficios tangibles para la sociedad [2, 3].

La gran cantidad de datos generados por los EEG y la necesidad de procesarlos en tiempo real requieren infraestructuras de cómputo altamente eficientes. En este contexto, la Computación de Alto Rendimiento (High Performance Computing, HPC) constituye un pilar fundamental. Gracias a su capacidad para ejecutar cálculos masivos en tiempos reducidos, HPC ha permitido abordar problemas antes inabordables, desde la simulación de fenómenos climáticos hasta el entrenamiento de modelos de inteligencia artificial de gran escala, y en particular el procesamiento de datos EEG para BCI [4].

Sin embargo, la utilización de HPC para procesar datos EEG no está exenta de desafíos. El análisis de señales cerebrales de gran volumen y alta complejidad requiere paralelismo multinivel, arquitecturas heterogéneas y escalabilidad, además de una cuidadosa optimización de recursos para mantener la precisión en la clasificación mientras se minimizan los tiempos de procesamiento. Trabajos recientes han demostrado que la combinación de técnicas de reducción de dimensionalidad con infraestructuras HPC puede acelerar el procesamiento entre 1.5 y 4 veces sin sacrificar la precisión [2, 3, 4], lo que evidencia la necesidad de entornos HPC eficientes y configu-

rados adecuadamente.

Esta complejidad en la gestión de recursos y entornos de ejecución se ve agravada por dos problemas adicionales que limitan la eficacia de la ciencia computacional moderna: la creciente heterogeneidad de las arquitecturas y la falta de portabilidad y reproducibilidad de las aplicaciones. En numerosos casos, un software científico que funciona correctamente en un sistema falla en otro debido a diferencias en el hardware (CPU, GPU) o en las configuraciones de software (versiones de librerías, compiladores, dependencias del sistema). Esta situación no solo dificulta la verificación de resultados, sino que también limita la colaboración y el avance científico a gran escala.

En este contexto, la tecnología de contenedores ha emergido como una solución prometedora. Los contenedores permiten encapsular aplicaciones junto con todas sus dependencias en unidades portables y aisladas. De este modo, se garantiza que el software se ejecute de manera consistente en cualquier entorno, reduciendo la complejidad de despliegue y mejorando la reproducibilidad. Frente a las máquinas virtuales tradicionales, los contenedores presentan una sobrecarga mínima y ofrecen un rendimiento cercano al nativo, lo que los convierte en una alternativa atractiva para entornos HPC. En particular, *Docker* se reconoce como la tecnología líder en contenerización debido a su baja sobrecarga, flexibilidad, portabilidad y capacidad de garantizar reproducibilidad [5].

No obstante, diversos estudios han puesto de manifiesto que, en escenarios de computación de alto rendimiento, Docker todavía presenta ciertas limitaciones relacionadas con la seguridad y la latencia de red [6, 7]. Estas restricciones abren un espacio de investigación particularmente relevante: evaluar hasta qué punto los contenedores pueden ser adoptados en entornos HPC sin comprometer el rendimiento ni la eficiencia, y bajo qué condiciones pueden convertirse en un elemento clave para mejorar la portabilidad y la reproducibilidad científica en sistemas heterogéneos.

Para poner a prueba esta propuesta, se ha seleccionado como caso de estudio el software *HPMoon*, desarrollado en el marco de una tesis doctoral para la clasificación no supervisada de señales EEG. *HPMoon* constituye un escenario experimental idóneo al incorporar múltiples niveles de paralelismo (MPI, OpenMP, OpenCL), estar diseñado para arquitecturas heterogéneas con CPU y GPU, y contar con una base científica consolidada. Evaluar su ejecución tanto en entornos nativos como contenerizados permite analizar de manera rigurosa las ventajas, limitaciones y retos que supone la contenerización en aplicaciones HPC reales.

Este trabajo se enmarca, por tanto, en el estudio de la viabilidad y el impacto del uso de contenedores en aplicaciones científicas de alto rendimiento. La investigación realizada pretende contribuir a la mejora de la portabilidad, reproducibilidad y adopción de este tipo de aplicaciones en

la comunidad científica, sin comprometer su rendimiento en arquitecturas modernas y heterogéneas.

1.1. Motivación

La motivación principal de este trabajo surge de la necesidad de conciliar dos objetivos que, en ocasiones, parecen contrapuestos en la investigación científica computacional: por un lado, maximizar el rendimiento mediante arquitecturas HPC cada vez más complejas y heterogéneas, y por otro, garantizar la portabilidad y reproducibilidad de las aplicaciones en entornos diversos.

La contenerización representa una oportunidad única para cerrar esta brecha. Su adopción en entornos de HPC aún no es generalizada, en parte debido a la percepción de que puede introducir sobrecargas o limitar el acceso eficiente a los recursos de hardware, especialmente en configuraciones multinodo con GPU. Este trabajo busca arrojar luz sobre esta problemática mediante un análisis experimental detallado, aplicando la contenerización a un caso de uso real y exigente como *HPMoon*.

En este sentido, el proyecto no se centra únicamente en medir tiempos de ejecución, sino en evaluar de manera integral la escalabilidad, la eficiencia en arquitecturas heterogéneas y la capacidad de reproducir resultados en múltiples plataformas. Con ello, se pretende ofrecer una contribución práctica y útil para la comunidad científica, ayudando a sentar las bases para una adopción más amplia y fundamentada de los contenedores en HPC.

De este modo, los apartados siguientes se centran en definir los objetivos concretos que guían esta investigación, tanto generales como específicos, y que permitirán estructurar el análisis y validar las hipótesis planteadas.

1.2. Objetivos

Analizar la viabilidad y las limitaciones del uso de contenedores, concretamente Docker, para encapsular y ejecutar aplicaciones de alto rendimiento (HPC) en arquitecturas heterogéneas modernas —como big.LITTLE— y entornos multiplataforma, con el fin de facilitar su portabilidad, uso y adopción por parte de la comunidad científica.

1.2.1. Objetivos específicos

- **OB1.** Investigar el estado del arte en el ámbito de la tecnología de contenedores y su aplicación en entornos de computación de alto ren-

dimiento.

- **OB2.** Diseñar e implementar un conjunto de experimentos para evaluar el rendimiento de aplicaciones HPC contenerizadas en diferentes arquitecturas y plataformas.
- **OB3.** Comparar el rendimiento de las aplicaciones contenerizadas en diferentes entornos y arquitecturas, identificando las ventajas y desventajas de cada enfoque.
- **OB4.** Analizar los resultados obtenidos en los experimentos para identificar las limitaciones y desafíos asociados al uso de contenedores en entornos HPC, así como establecer futuras líneas de investigación.

Capítulo 2

Gestión del Proyecto

2.1. Tareas

Tareas de planificación del proyecto

- **Definición de objetivos y alcance del proyecto**
  Establecer claramente los objetivos principales y específicos del proyecto, así como el alcance y las limitaciones.
- **Planificación temporal**
  Desarrollar un cronograma detallado que incluya todas las fases del proyecto, desde la investigación inicial hasta la redacción final del informe.
- **Asignación de recursos**
  Identificar y asignar los recursos necesarios, tanto humanos como materiales, para llevar a cabo el proyecto de manera eficiente.
- **Gestión de riesgos**
  Identificar posibles riesgos que puedan afectar al desarrollo del proyecto y establecer planes de contingencia para mitigarlos.
- **Comunicación y seguimiento**
  Establecer canales de comunicación efectivos entre todos los miembros del equipo y realizar un seguimiento regular del progreso del proyecto para asegurar que se cumplen los plazos establecidos.

Tareas del OB1 (Estudio del estado del arte)

- **Revisión del estado del arte en HPC**
  Estudiar la evolución histórica y tendencias actuales en la computación de alto rendimiento.

- **Análisis del uso de contenedores en HPC**
  Revisar tecnologías de contenedores aplicadas a entornos científicos y de alto rendimiento. Comparar contenedores frente a máquinas virtuales en cuanto a eficiencia, overhead y portabilidad en HPC.

- **Estudio del uso de GPU en aplicaciones HPC**
  Revisar el papel de las GPUs en la aceleración de aplicaciones científicas y de ingeniería. Identificar librerías y frameworks para programación en GPU. Analizar casos de éxito en la integración de GPU en entornos HPC.

- **Investigación sobre el soporte de GPU en contenedores**
  Revisar soluciones actuales para ejecutar GPUs dentro de contenedores. Analizar el grado de compatibilidad con diferentes sistemas operativos y arquitecturas. Estudiar el impacto en rendimiento del uso de GPUs en entornos contenerizados en comparación con la ejecución nativa.

Tareas del OB2 (Diseño e implementación de la propuesta)

- **Selección de la aplicación o problema HPC a estudiar**
  Se seleccionará una aplicación representativa del ámbito HPC, justificando su elección en función de su relevancia científica, disponibilidad de código abierto y viabilidad técnica para su ejecución en diferentes plataformas y entornos contenerizados.

- **Preparar entornos y dependencias**
  Se identificarán y documentarán las librerías y herramientas necesarias, incluyendo MPI, OpenMP y CUDA. Se garantizará la homogeneidad de las configuraciones en todos los sistemas de prueba y se detallarán los requisitos específicos para cada plataforma (Linux, Windows, macOS).

- **Diseñar y construir imágenes de contenedor**
  Se desarrollarán Dockerfiles reproducibles que incluyan todas las dependencias necesarias, asegurando soporte para GPU mediante NVIDIA Container Toolkit. Las imágenes serán versionadas y almacenadas en un registro para facilitar su reutilización y trazabilidad.

- **Definir casos de prueba y parámetros de ejecución**
  Se establecerán experimentos mononodo variando el número de hebras, experimentos multinodo con diferentes cantidades de nodos y casos combinados que exploren el espacio de parámetros hebras $\times$ nodos.

- **Automatización y orquestación**
  Se implementarán scripts en para automatizar la ejecución de lotes de pruebas, así como la recogida y almacenamiento de logs y resultados.
- **Instrumentación y métricas**
  Se instrumentará la aplicación para medir tiempos totales de ejecución, uso de CPU y otros recursos. Se calcularán métricas como aceleración, eficiencia, throughput y overhead comparando la ejecución en contenedor frente a la nativa. Se generarán gráficos comparativos para el análisis de resultados.
- **Reproducibilidad y trazabilidad**
  Se mantendrá un repositorio con los Dockerfiles, scripts y documentación del proyecto. Se etiquetarán las versiones de las imágenes y dependencias para asegurar la reproducibilidad de los experimentos.

Tareas del OB3 (Evaluación de rendimiento)

- **Definición de criterios de comparación**
  Se establecerán las métricas principales para la comparación de rendimiento y se garantizará la paridad de configuraciones (versiones de compiladores, librerías, drivers).
- **Ejecución de pruebas comparativas**
  Se ejecutarán las mismas baterías de experimentos tanto en modo nativo como en contenedor. Se registrarán logs completos de cada ejecución.
- **Recopilación y organización de resultados**
  Se guardarán los tiempos de ejecución y métricas de uso de recursos, clasificando los datos según plataforma (Linux, Windows, macOS) y tipo de acelerador (CPU, GPU). Se establecerá un formato homogéneo para los ficheros de resultados (CSV o base de datos).
- **Visualización de resultados comparativos**
  Se generarán gráficos y tablas que destaquen los casos extremos (mejores y peores comportamientos), facilitando la interpretación de los resultados.

Tareas del OB4 (Análisis de resultados)

- **Revisión sistemática de los resultados experimentales**
  Se analizarán de manera estructurada los datos obtenidos en las pruebas, comparando el rendimiento entre ejecución nativa y contenerizada en las distintas plataformas (Linux, Windows, macOS) y ante el uso o

no de aceleradores (CPU, GPU). Se identificarán tendencias generales, anomalías y comportamientos consistentes.

- **Análisis cuantitativo del rendimiento**
  Se calcularán diferencias absolutas y relativas entre ejecución nativa y contenerizada, estimando overheads medios y por caso. Se evaluará la escalabilidad en cada escenario y se aplicará análisis estadístico para validar la significancia de las diferencias observadas.

- **Análisis cualitativo**
  Se identificarán ventajas no estrictamente de rendimiento, como portabilidad, reproducibilidad y facilidad de despliegue, así como limitaciones observadas relacionadas con drivers de GPU, gestión de red en contenedores y problemas de compatibilidad.

- **Detección de desafíos en la adopción de contenedores en HPC**
  Se evaluará la complejidad asociada a la configuración, despliegue y mantenimiento de entornos contenerizados en HPC, incluyendo la integración de aceleradores como GPU y la gestión de dependencias específicas.

- **Propuesta de líneas de investigación futura**
  A partir de los resultados y desafíos identificados, se propondrán posibles líneas de trabajo futuro.

Tareas de redacción del informe final

- **Estructuración del informe**
  Se definirá un esquema claro y lógico para el informe, asegurando que cada sección fluya de manera coherente hacia la siguiente.

- **Redacción de secciones técnicas**
  Se describirán detalladamente los métodos, experimentos y resultados obtenidos, utilizando un lenguaje técnico preciso y adecuado para la audiencia objetivo.

- **Incorporación de gráficos y tablas**
  Se incluirán visualizaciones que apoyen y clarifiquen los puntos clave del informe, asegurando que estén correctamente etiquetadas y referenciadas en el texto.

- **Revisión y edición**
  Se realizará una revisión exhaustiva del informe para corregir errores gramaticales, mejorar la claridad y coherencia del texto, y asegurar que todas las referencias bibliográficas estén correctamente citadas.

- **Formato y presentación**
  Se aplicarán las normas de formato establecidas por la institución o publicación a la que se destina el informe, asegurando una presentación profesional y uniforme.

2.2. Planificación temporal

En la tabla 2.1 se presenta una estimación del tiempo necesario para completar cada una de las tareas principales del proyecto, desglosado en horas dedicadas por el desarrollador y el tutor.

Tarea	**Desarrollador (h)**	**Tutor (h)**
Planificación	20	5
Estado del arte	40	10
Implementación	85	8
Evaluación	55	7
Análisis	30	5
Informe	30	15
Total	**260**	**50**

Tabla 2.1: Planificación temporal de tareas y horas estimadas

En la imagen 2.1 se muestra un diagrama de Gantt que ilustra la distribución temporal de las tareas a lo largo del proyecto.

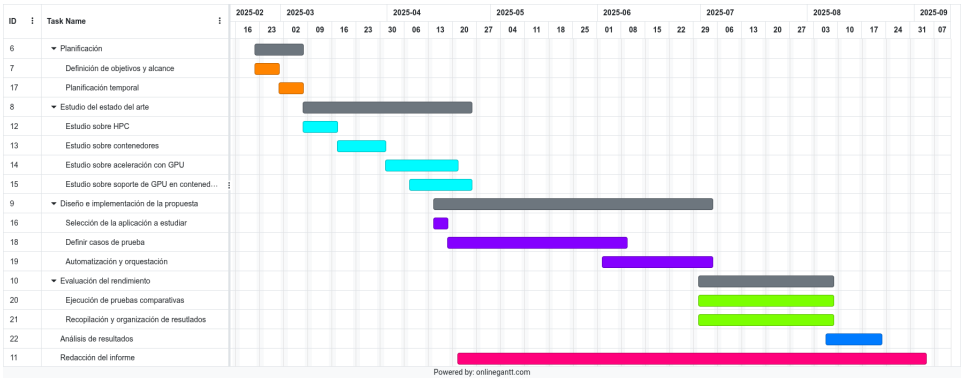

Figura 2.1: Diagrama de Gantt del proyecto

2.3. Estimación de costes

A continuación, se detallan los recursos necesarios para llevar a cabo el proyecto, incluyendo hardware, software y recursos humanos, junto con una estimación de los costes asociados.

Hardware

- Ordenador portátil LG Gram 14Z90Q-G.AA75B, este equipo se utilizará para el desarrollo general del trabajo: creación del código para las pruebas y creación de la memoria. Cuenta con un procesador Intel Core i7-1260P, 16 GB de RAM y 512 GB de almacenamiento SSD.
- Ordenador portátil Lenovo Legion 5, utilizado como plataforma principal para la ejecución de pruebas de rendimiento. Está equipado con un procesador AMD Ryzen 7 4800H (8 núcleos y 16 hilos), 16 GB de memoria RAM DDR4 y 512 GB de almacenamiento SSD NVMe. La presencia de la GPU NVIDIA permite la ejecución y evaluación de aplicaciones HPC que hacen uso de CUDA, así como la integración y validación del soporte de GPU en entornos contenerizados mediante NVIDIA Container Toolkit. Además, este equipo facilita la comparación de resultados entre ejecución nativa y contenerizada en escenarios reales de computación de alto rendimiento.
- Ordenador Apple Mac Mini M4, utilizado como plataforma de pruebas para la ejecución y validación de aplicaciones HPC en entornos macOS y arquitectura ARM. Este equipo incorpora un procesador Apple M4, 10 núcleos de CPU divididos en 4 de alto rendimiento y 6 de eficiencia, 10 núcleos de GPU integrada de última generación y 16 GB de memoria unificada. El Mac Mini M4 permitirá evaluar la compatibilidad y el rendimiento de contenedores en sistemas Apple Silicon.

Software

- Sistema operativo Ubuntu 24.04 LTS. Será la distribución Linux principal con la que vamos a trabajar, tanto en forma nativa, como en los contenedores.
- Sistema operativo Microsoft Windows 11 Home. Será la distribución con la que se ejecutarán las pruebas de compatibilidad y rendimiento en entornos Windows.
- Sistema operativo macOS Sequoia 15.5. Será la distribución con la que se ejecutarán las pruebas de compatibilidad y rendimiento en entornos Apple.

Dispositivo	**Descripción**	**Coste (€)**
LG Gram 14Z90Q-G.AA75B	Portátil principal de desarrollo	1 000
Lenovo Legion 5	Portátil de pruebas	500
Apple Mac Mini M4	Dispositivo de pruebas Apple	599
Total		**2 099**

Tabla 2.2: Costes estimados de hardware para el proyecto

Recursos humanos.

En cuanto al software, los sistemas operativos Microsoft Windows 11 Home y macOS Sequoia 15.5 vienen incluidos en los dispositivos correspondientes, por lo que no se ha considerado un coste adicional. El sistema operativo Ubuntu 24.04 LTS es de código abierto y gratuito, por lo que tampoco se ha considerado un coste adicional.

Recursos humanos

En la tabla 2.3 se detalla el coste por hora, las horas estimadas y el coste total de los recursos humanos necesarios para llevar a cabo el proyecto.

Recurso	**Puesto**	**€/h**	**Horas**	**Total (€)**
Fernando Cuesta Bueno	Desarrollador	20	260	5 200
Juan José Escobar Pérez	Tutor/Supervisor	50	50	2 500
Total				**7 700**

Tabla 2.3: Costes estimados de recursos humanos para el proyecto

Coste total del proyecto

El coste total del proyecto se calcula sumando los costes de hardware, software y recursos humanos. En la tabla 2.4 se detalla el coste total estimado.

Concepto	**Coste (€)**
Hardware	2 099
Software	0
Recursos humanos	7 700
Total	**9 799**

Tabla 2.4: Coste total estimado del proyecto

Capítulo 3

Estado del arte

3.1. Computación de alto rendimiento (HPC)

La computación de alto rendimiento (HPC, por sus siglas en inglés) se refiere a la práctica de agregar poder de cómputo para lograr un rendimiento mucho mayor que el que se podría obtener con una computadora convencional, con el objetivo de resolver problemas complejos en ciencias, ingeniería o negocios [8].

Objetivos principales de HPC

El propósito fundamental de HPC es acelerar la resolución de problemas complejos, alcanzando resultados en tiempos factibles que de otra manera requerirían semanas o meses. Para ello, HPC se apoya en dos conceptos centrales: paralelización y escalabilidad.

Paralelización La paralelización consiste en descomponer un problema en múltiples tareas que puedan ejecutarse simultáneamente en distintos núcleos de procesamiento, ya sean CPUs o GPUs. Este enfoque permite aprovechar todos los recursos del sistema para reducir significativamente el tiempo de ejecución de las aplicaciones. Existen distintos niveles de paralelización:

- **Paralelización a nivel de instrucción**: el procesador ejecuta varias instrucciones de manera simultánea mediante pipelines y unidades vectoriales.
- **Paralelización a nivel de hilo o thread**: diferentes hilos de ejecución procesan tareas concurrentes dentro de un mismo núcleo o CPU.

- **Paralelización a nivel de proceso o nodo**: tareas completas se distribuyen entre múltiples nodos de un clúster, cada uno con su propio conjunto de recursos.

Escalabilidad La escalabilidad se refiere a la capacidad de un sistema para mejorar su rendimiento al añadir más recursos de cómputo. Se distingue entre:

- **Escalabilidad fuerte**: mejora del tiempo de ejecución de un problema de tamaño fijo al incrementar el número de recursos.
- **Escalabilidad débil**: capacidad de mantener constante el tiempo de ejecución al aumentar simultáneamente el tamaño del problema y los recursos de manera proporcional.

Lograr buena escalabilidad es crítico, especialmente en entornos multinodo, donde la comunicación entre nodos y la sincronización de tareas pueden generar cuellos de botella.

Arquitecturas comunes en HPC

Los sistemas HPC modernos pueden clasificarse en función de su arquitectura:

- **Sistemas homogéneos**: utilizan múltiples CPUs idénticas interconectadas. Esto simplifica la planificación de tareas y el balance de carga, aunque no aprovecha posibles ventajas de eficiencia energética de núcleos especializados.
- **Sistemas heterogéneos**: combinan distintos tipos de núcleos de procesamiento o aceleradores especializados, como GPUs, FPGAs o núcleos de eficiencia energética tipo *big.LITTLE*. Estas arquitecturas permiten un mayor rendimiento por watt y mejor aprovechamiento de recursos, pero requieren técnicas avanzadas de programación y planificación.
- **Clústeres y supercomputadores**: integran múltiples nodos interconectados mediante redes de alta velocidad (Infiniband, Omni-Path). Soportan aplicaciones distribuidas que requieren comunicación intensiva entre nodos, siendo los supercomputadores clústeres de alto rendimiento con características avanzadas de interconexión, memoria y almacenamiento.

3.1.1. Virtualización vs contenerización

La virtualización y la contenerización representan dos enfoques distintos para la abstracción y gestión de recursos computacionales, cuyo propósito es ejecutar aplicaciones de manera aislada del sistema operativo anfitrión. Ambas tecnologías han sido ampliamente utilizadas en entornos de servidores y, más recientemente, en la computación de alto rendimiento (HPC), aunque presentan diferencias fundamentales que influyen en el rendimiento, la eficiencia y la portabilidad.

Máquinas virtuales (VMs)

Las máquinas virtuales permiten la ejecución de sistemas operativos completos sobre un hipervisor, el cual actúa como intermediario entre el hardware físico y el sistema operativo invitado. Cada VM incorpora su propio kernel, librerías y aplicaciones, proporcionando un aislamiento fuerte respecto al host y a otras VMs.

Entre las ventajas más destacadas se encuentran:

- **Aislamiento robusto**: garantiza una separación completa de procesos y aplicaciones, reduciendo riesgos de interferencia.
- **Compatibilidad multiplataforma**: posibilita ejecutar sistemas operativos diferentes al del host.
- **Seguridad**: el aislamiento a nivel de kernel refuerza la protección frente a vulnerabilidades.

Sin embargo, en entornos HPC las VMs presentan ciertas limitaciones:

- **Sobrecarga de recursos**: cada VM requiere un kernel completo y librerías redundantes, incrementando el consumo de memoria y almacenamiento.
- **Overhead de rendimiento**: la capa de virtualización introduce latencias que afectan especialmente a cargas intensivas en cómputo y al acceso a hardware especializado como GPUs.
- **Gestión compleja**: operaciones como la migración, el clonado o la actualización son más costosas en comparación con los contenedores.

Contenedores

Los contenedores constituyen una forma de virtualización a nivel de sistema operativo. Comparten el kernel del host, pero mantienen un entorno

aislado con librerías y dependencias específicas para cada aplicación. Este enfoque reduce significativamente la sobrecarga respecto a las VMs, permitiendo una mayor eficiencia en la ejecución de aplicaciones HPC.

Sus principales ventajas son:

- **Ligereza y eficiencia**: requieren menos recursos al no replicar un sistema operativo completo, lo que disminuye el overhead en tiempo de ejecución.
- **Portabilidad**: las imágenes de contenedores pueden ejecutarse en distintos sistemas operativos y arquitecturas, facilitando la distribución en entornos heterogéneos.
- **Rapidez de despliegue**: iniciar o actualizar contenedores es mucho más ágil que en las VMs.
- **Compatibilidad con bibliotecas y frameworks HPC**: herramientas como Docker, Singularity/Apptainer o Podman permiten empaquetar dependencias científicas y garantizar la reproducibilidad de los experimentos.

Comparativa entre máquinas virtuales y contenedores en HPC

Diversos estudios han comparado en detalle las diferencias entre la virtualización a nivel de hardware y la virtualización a nivel de sistema operativo. En el desarrollado en Sharma et al. (2016) [1] concluyen que los contenedores ofrecen un rendimiento cercano al nativo, con una sobrecarga mínima en cargas de CPU y red (menor al 3 %), mientras que las VMs introducen una latencia mayor en operaciones intensivas de memoria (hasta un 10 %) y presentan un rendimiento significativamente inferior en operaciones de E/S de disco (hasta un 80 % peor que contenedores). En cuanto al aislamiento de recursos, las VMs proporcionan mayor robustez en escenarios de multi-tenencia, especialmente frente a cargas adversarias en CPU y memoria, mientras que los contenedores son más susceptibles a interferencias en estos casos, como se resume en la Tabla 3.1. Además, en la Figura 3.1 se muestran las diferencias arquitectónicas entre una máquina virtual y un contenedor, ilustrando cómo se gestionan los recursos y el aislamiento en cada enfoque.

Desde el punto de vista de la gestión, los contenedores ofrecen despliegues más rápidos (menos de un segundo frente a decenas de segundos en VMs) y una mayor flexibilidad en la asignación de recursos. Sin embargo, la migración en vivo está más desarrollada en VMs, mientras que en contenedores se encuentra en una etapa menos madura. Asimismo, el desarrollo de software se ve favorecido en los contenedores gracias a la construcción

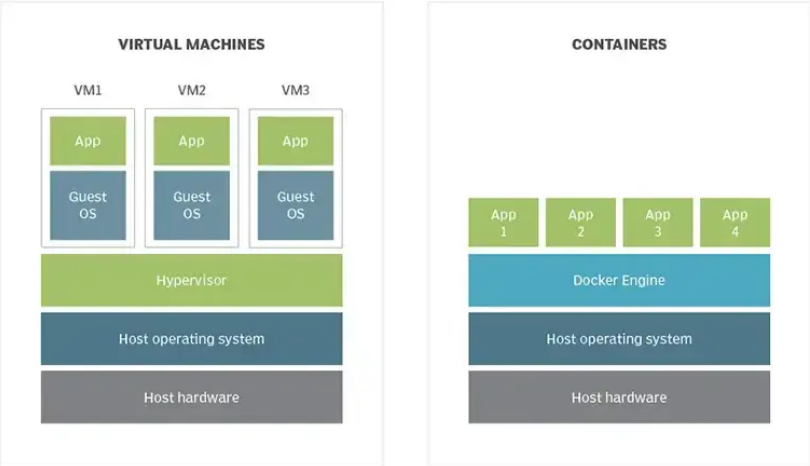

Figura 3.1: Comparación entre la arquitectura de una máquina virtual y la de un contenedor. Fuente: `https://electronicaonline.net/software-y-apps/windows/maquina-virtual/`

más rápida de imágenes, menor tamaño de almacenamiento y capacidades integradas de control de versiones [1].

Característica	**Máquinas virtuales**	**Contenedores**
Aislamiento	Completo	Parcial (kernel compartido)
Consumo de recursos	Alto	Bajo
Tiempo de arranque	Largo	Corto
Portabilidad	Alta	Muy alta
Acceso a hardware especializado	Limitado	Directo con soporte adecuado
Mantenimiento y despliegue	Complejo	Simple
Rendimiento en CPU	Casi nativo	Casi nativo
Rendimiento en memoria	Degradación ∼10 %	Mejor que VMs
Rendimiento en E/S de disco	Hasta 80 % peor	Cercano a nativo

Tabla 3.1: Comparativa entre máquinas virtuales y contenedores en entornos HPC, incluyendo resultados de Sharma et al. (2016) [1].

En entornos HPC, la contenerización se ha consolidado como la opción preferida para aplicaciones que requieren eficiencia, portabilidad y reproducibilidad.

3.1.2. Ecosistema de contenedores en HPC

El ecosistema de contenedores en HPC ha experimentado una evolución acelerada en los últimos años, motivado por la necesidad de incrementar la portabilidad, reproducibilidad y facilidad de despliegue de aplicaciones científicas y de ingeniería. Han surgido múltiples tecnologías y frameworks que ofrecen soluciones adaptadas, en mayor o menor medida, a los entornos HPC. Cada uno presenta ventajas, limitaciones y un grado diferente de adopción dentro de la comunidad científica.

Docker

Docker es la plataforma de contenedores más extendida a nivel global, reconocida por su capacidad para empaquetar aplicaciones junto con todas sus dependencias en imágenes portables, ejecutables en diferentes sistemas operativos y arquitecturas. Esta característica ha sido clave para su éxito, ya que permite reproducir entornos de software de manera consistente y simplifica considerablemente el despliegue y la gestión de aplicaciones.

Entre sus principales ventajas se encuentra la virtualización ligera a nivel de sistema operativo, que permite ejecutar contenedores de forma rápida y eficiente, sin la sobrecarga que implican las máquinas virtuales tradicionales. Esto conlleva un menor consumo de recursos y un arranque mucho más rápido, al compartir el kernel y las bibliotecas del host. Además, Docker facilita la portabilidad extrema, permitiendo mover aplicaciones entre distintos sistemas Linux sin inconsistencias, y asegura aislamiento y seguridad, evitando que contenedores independientes interfieran entre sí y eliminando residuos tras su eliminación.

Docker también contribuye a la reducción de costos y tiempos de despliegue, al eliminar configuraciones complejas y ofrecer entornos listos para ejecutar en cuestión de segundos. Su naturaleza de código abierto y gratuita elimina la necesidad de licencias comerciales, y su compatibilidad con arquitecturas de microservicios y herramientas de orquestación, como Kubernetes, simplifica la escalabilidad y la gestión de aplicaciones distribuidas [9].

A pesar de estas ventajas, su uso en entornos HPC presenta ciertas limitaciones. La necesidad de permisos de root para ejecutar contenedores y las estrictas políticas de seguridad en clústeres multiusuario restringen su adopción, motivando el uso de alternativas como Podman o Singularity en contextos de computación de alto rendimiento.

Podman

Podman se presenta como una alternativa a Docker, compatible con la mayoría de sus comandos y formatos de imagen, pero con una diferencia fundamental: no requiere privilegios de root para la ejecución de contenedores. Esta característica lo hace especialmente adecuado para entornos HPC multiusuario, donde las políticas de seguridad son estrictas.

Entre sus ventajas, destaca el modo rootless, que permite ejecutar y construir contenedores sin privilegios administrativos, aislando el entorno del usuario y reduciendo los riesgos de escalada de privilegios. Podman también soporta múltiples runtimes, como `runc` y `crun`, y es compatible con el estándar OCI, lo que facilita la creación y utilización de imágenes en distintos entornos.

Otra característica importante es su modelo fork-exec sin daemon, que prescinde de un demonio central. Esto simplifica la auditoría y el control de recursos, alineándose con la filosofía de HPC. Además, en pruebas con aplicaciones reales y benchmarks estándar de CPU y memoria, Podman muestra un overhead muy bajo, comparable al de Singularity y Shifter [10, 11], lo que garantiza un rendimiento cercano al de ejecuciones *bare-metal*, es decir, directamente sobre el hardware sin capas de virtualización adicionales.

Podman permite agrupar contenedores mediante Pods, compartiendo namespaces y facilitando la ejecución de aplicaciones complejas en paralelo. También optimiza el uso de recursos a gran escala, ya que el modo *exec* para MPI minimiza el overhead de arranque y mejora el rendimiento en aplicaciones intensivas en metadatos. Asimismo, integra soporte para GPU y redes HPC avanzadas mediante hooks OCI y wrappers, permitiendo configurar librerías, variables de entorno y acceso a interconexiones de alta velocidad, como Cray Slingshot u otras equivalentes.

Por último, Podman facilita la gestión de usuarios y almacenamiento, ofreciendo subuid/subgid persistentes y almacenamiento local en los nodos de login, lo que mejora los tiempos de construcción de imágenes y la productividad de los usuarios. En conjunto, estas características hacen de Podman una herramienta versátil y eficiente para entornos HPC modernos.

Gracias a estas capacidades, Podman se posiciona como una alternativa robusta y escalable para la contenerización en HPC. Su modo rootless, rendimiento competitivo y soporte para entornos distribuidos lo convierten en una opción cada vez más valorada en centros de supercomputación y laboratorios de investigación [10, 11].

Comparativa y adopción en la comunidad científica

En función de su adopción y casos de uso, destacan los siguientes puntos:

Comparativa y adopción en la comunidad científica

En la práctica, Docker y Podman se utilizan en contextos distintos dentro de la comunidad científica. Docker es ampliamente empleado en desarrollo y pruebas debido a su ecosistema maduro, la virtualización ligera, la portabilidad y la eficiencia en el uso de recursos [9]. Por su parte, Podman ha ganado popularidad en entornos HPC, gracias a su ejecución rootless, su rendimiento a escala comparable al de ejecuciones *bare-metal* y su compatibilidad con *runtimes* modernos [10, 11].

Esta diferenciación refleja cómo la comunidad científica selecciona la herramienta más adecuada según el tipo de carga de trabajo y las restricciones de seguridad de cada entorno.

Retos principales en la contenerización en HPC

La computación de alto rendimiento presenta desafíos específicos que afectan tanto a la eficiencia como a la adopción de tecnologías emergentes como la contenerización [12]. Uno de los principales obstáculos es la compatibilidad de software y librerías. La coexistencia de distintas versiones de librerías, como *glibc*, o diferencias en el ABI del kernel pueden generar fallos o comportamientos inesperados. Además, no todas las imágenes de Docker son directamente portables a otros motores de contenedores utilizados en HPC, debido a la falta de estandarización completa en los hooks de ejecución y en la gestión de privilegios.

La seguridad constituye otro reto importante. En entornos HPC, los contenedores comparten el mismo kernel, lo que hace que vulnerabilidades en uno de ellos puedan comprometer la estabilidad del sistema completo. Amenazas como la escalada de privilegios, ataques de denegación de servicio (DoS) o fugas de información son preocupantes. Aunque mecanismos como los *cgroups* y los *user namespaces* ayudan a mitigar estos riesgos, muchos centros de supercomputación optan por deshabilitarlos parcialmente, lo que limita la adopción de contenedores modernos.

Otro desafío es la posible degradación del rendimiento. La ejecución de aplicaciones dentro de contenedores puede afectar la eficiencia, especialmente cuando se utilizan librerías optimizadas para GPUs o interconexiones de alta velocidad, como InfiniBand. Asimismo, al incrementar el número de procesos MPI en un mismo nodo, los costes de comunicación interna aumentan, afectando tanto a operaciones punto a punto como colectivas.

Finalmente, los contenedores presentan limitaciones en la personalización del kernel. A diferencia de los entornos nativos, restringen la instalación de módulos de kernel para preservar el aislamiento, lo que impide realizar ajustes de bajo nivel que podrían mejorar el rendimiento en escenarios HPC. Aunque existen propuestas experimentales como *Xcontainer*, estas todavía no están maduras para un uso generalizado.

3.1.3. Limitaciones de la contenerización en macOS

El uso de contenedores en macOS presenta desafíos específicos, especialmente en lo que respecta al acceso y aprovechamiento de la GPU. A diferencia de los entornos Linux, donde los contenedores pueden habilitar el acceso directo a dispositivos de hardware, en macOS existen restricciones derivadas de la arquitectura del sistema y de la infraestructura de virtualización utilizada por Docker. Estas limitaciones afectan de manera significativa a aplicaciones de HPC y de inteligencia artificial que requieren aceleración por GPU, dificultando su ejecución eficiente y restringiendo el rendimiento alcanzable en este sistema operativo.

Uno de los principales obstáculos es la ausencia de soporte para GPU passthrough. Docker en macOS no permite el acceso directo a la GPU del host, lo que impide que los contenedores aprovechen la aceleración por hardware y limita el rendimiento de aplicaciones que dependen de procesamiento gráfico intensivo o cómputo acelerado. Esta restricción está estrechamente vinculada a la dependencia de la infraestructura de virtualización de Apple, la cual no proporciona acceso nativo a la GPU, afectando negativamente tanto el rendimiento como la compatibilidad de aplicaciones científicas y de ingeniería.

Aunque existen alternativas como Vulkan o MoltenVK que permiten cierto grado de aceleración gráfica en contenedores sobre macOS, el rendimiento obtenido es notablemente inferior al que se logra en entornos Linux o Windows. Además, la configuración de estas tecnologías suele ser compleja y no garantiza compatibilidad total con todas las aplicaciones. En general, habilitar el acceso a la GPU en contenedores sobre macOS enfrenta desafíos técnicos importantes, derivados de la infraestructura de virtualización y de la falta de soporte oficial para GPU passthrough.

En conjunto, estos retos muestran que, si bien la contenerización aporta ventajas claras en términos de reproducibilidad, portabilidad y flexibilidad, su implementación en macOS requiere soluciones específicas para gestionar dependencias, garantizar la seguridad en entornos compartidos y mantener la eficiencia en hardware y comunicaciones paralelas. Muchas de estas cuestiones siguen siendo áreas activas de investigación dentro de la comunidad HPC.

3.2. Portabilidad y reproducibilidad científica

La portabilidad y la reproducibilidad son aspectos fundamentales en la computación científica moderna. La creciente heterogeneidad de plataformas, que abarca desde supercomputadores tradicionales hasta infraestructuras en la nube o clústeres híbridos CPU-GPU, dificulta que una aplicación pueda ejecutarse sin modificaciones en distintos entornos. Además, la reproducibilidad de resultados experimentales se ha convertido en un reto central, ya que incluso pequeñas diferencias en el entorno de ejecución pueden conducir a variaciones significativas en los resultados obtenidos.

En este contexto, la contenerización emerge como una solución tecnológica que no solo aporta ventajas en términos de gestión e infraestructura, sino que también permite garantizar que las aplicaciones sean fácilmente portables y que los experimentos científicos puedan reproducirse bajo condiciones controladas y consistentes.

3.2.1. Cómo los contenedores facilitan la portabilidad de aplicaciones HPC

En el contexto de HPC, la portabilidad se refiere a la capacidad de ejecutar una misma aplicación en diferentes sistemas de cómputo sin necesidad de modificar su código fuente ni su configuración. Los contenedores facilitan esta portabilidad al encapsular la aplicación junto con todas sus dependencias, incluyendo bibliotecas, compiladores, intérpretes, librerías de comunicación como MPI y entornos de ejecución específicos. De esta manera, se reduce significativamente la fricción que surge cuando un código funciona en un sistema pero falla en otro.

Mediante la creación de una única imagen de contenedor, es posible desplegar la misma aplicación en múltiples entornos, siempre que exista soporte para la tecnología de contenedores utilizada. Esto asegura consistencia entre plataformas, ya que el mismo contenedor puede ejecutarse en distintos sistemas operativos y arquitecturas, minimizando incompatibilidades. Asimismo, disminuye el tiempo de despliegue, ya que los usuarios no necesitan adaptar ni recompilar el software para cada infraestructura, y facilita el acceso a entornos heterogéneos que combinan CPUs y GPUs, donde la gestión de drivers y librerías suele ser crítica. Otro beneficio es la compatibilidad con diversos entornos, desde clusters virtuales autoescalables en la nube hasta infraestructuras HPC tradicionales o equipos personales, garantizando un entorno reproducible y consistente [13].

Además, el uso de gestores de paquetes como Nix dentro de contenedores permite fijar versiones exactas de bibliotecas y dependencias, asegurando lo que se denomina *pureza funcional*. Esto garantiza que cualquier ejecución

posterior del contenedor produzca resultados idénticos, independientemente de la plataforma en la que se ejecute [13].

3.2.2. Reproducibilidad de experimentos científicos usando contenedores

La reproducibilidad científica implica que los resultados de un experimento puedan replicarse bajo las mismas condiciones. En computación científica, factores como la evolución de bibliotecas, diferencias entre compiladores o cambios en sistemas operativos suelen comprometer esta reproducibilidad, dificultando la verificación de resultados a lo largo del tiempo.

Los contenedores abordan estos problemas al encapsular no solo el código de la aplicación, sino también todo el contexto de ejecución. De este modo, se puede garantizar que los experimentos se ejecuten en entornos idénticos, independientemente de la infraestructura subyacente. Por ejemplo, mediante tecnologías como Nix dentro de Docker o Singularity, es posible fijar versiones exactas de paquetes y librerías, asegurando reproducibilidad incluso años después [13].

Otra ventaja es la transparencia: al compartir la imagen del contenedor junto con el código y los datos, otros investigadores pueden verificar que los resultados se obtuvieron en condiciones equivalentes. Esto facilita también la colaboración eficiente entre equipos distribuidos geográficamente, que pueden trabajar sobre un mismo entorno sin necesidad de configurar individualmente cada sistema.

Asimismo, los contenedores reducen la carga administrativa. El uso de plantillas de contenedores (*Container Template Library - CTL*) permite a los investigadores adaptar construcciones de software a sus necesidades, manteniendo la consistencia y la reproducibilidad sin depender de la intervención de administradores de sistemas [13]. Además, la metodología de contenerización ofrece escalabilidad y flexibilidad, permitiendo ejecutar los mismos experimentos desde computadoras personales hasta clusters HPC o nubes públicas, garantizando que los resultados sigan siendo reproducibles en cualquier infraestructura [13].

En conjunto, los contenedores, especialmente cuando se integran con gestores de paquetes como Nix y tecnologías HPC como Singularity, proporcionan un marco sólido para asegurar la reproducibilidad de resultados científicos y habilitar la computación científica en entornos diversos de manera eficiente y segura.

Capítulo 4

Propuesta principal

4.1. Introducción de la propuesta

En esta sección se presenta la propuesta general del trabajo, explicando la metodología seguida para evaluar el uso de contenedores en entornos HPC. Se justifica la elección de los experimentos en función de los objetivos del TFG y del estado del arte, y se define el alcance de los mismos, indicando qué se medirá y evaluará.

4.2. Aplicación seleccionada: HPMoon

4.2.1. Origen y contexto

HPMoon es un software desarrollado por el investigador Juan José Escobar Pérez en el marco de la tesis doctoral *"Energy-Efficient Parallel and Distributed Multi-Objective Feature Selection on Heterogeneous Architectures"*. Su nombre proviene del proyecto *High Performance Multi-Objective Optimization for Neuroengineering and Rehabilitation Technologies* (HPMoon), reflejando su enfoque en optimización de alto rendimiento y aplicaciones biomédicas.

El programa se concibe como una herramienta del proyecto *e-hpMOBE* y está disponible públicamente en GitHub[1]. Su desarrollo fue apoyado por proyectos de investigación nacionales financiados por el Ministerio de Ciencia, Innovación y Universidades (MICIU) y fondos FEDER, así como por una beca de NVIDIA.

Esta aplicación se utiliza como caso de estudio en este TFG debido a su relevante aportación científica y tecnológica en el ámbito de la computación

[1] `https://github.com/hpmoon/`

de alto rendimiento (HPC).

4.2.2. Objetivo de HPMoon

El objetivo principal de HPMoon es abordar la clasificación no supervisada de señales de Electroencefalograma (EEG) en tareas de Interfaz Cerebro-Computadora (BCI). Este es un problema de alta dimensionalidad y computacionalmente costoso debido a las características de las señales EEG y al gran número de características que pueden contener. La aplicación combina la selección de características multiobjetivo (MOFS) con algoritmos paralelos y energéticamente eficientes, buscando minimizar el tiempo de ejecución y el consumo de energía, cruciales en problemas de *Machine Learning* y bioingeniería que requieren plataformas de alto rendimiento.

4.2.3. Arquitectura y funcionamiento

HPMoon implementa un procedimiento paralelo multinivel y energéticamente eficiente para la selección de características multiobjetivo (MOFS). La versión más avanzada, conocida como ODGA, ha sido descrita como la "más eficiente desarrollada hasta la fecha". Su arquitectura combina técnicas de algoritmos evolutivos, paralelismo en múltiples niveles y optimización de recursos para abordar problemas complejos de selección de características.

El enfoque principal de HPMoon es de tipo wrapper, utilizando un Algoritmo Genético Multiobjetivo (MOGA), basado en una adaptación de NSGA-II, para seleccionar las características más relevantes. Cada individuo en la población representa una combinación posible de características y se evalúa mediante un procedimiento de fitness basado en el algoritmo K-means para clasificación no supervisada. La evaluación se realiza considerando dos funciones de coste: f_1, que minimiza la suma de distancias dentro de los clústeres (WCSS), y f_2, que maximiza la distancia entre centroides (BCSS). Ambas funciones se normalizan en el intervalo (0,1) para garantizar comparabilidad.

HPMoon explota hasta cuatro niveles de paralelismo en arquitecturas heterogéneas. A nivel de clúster, se distribuyen las tareas entre nodos utilizando MPI. Dentro de cada nodo, OpenMP gestiona la distribución de trabajo entre dispositivos CPU y GPU, mientras que a nivel de CPU se aprovechan múltiples hilos o unidades de cómputo (CUs). En la GPU, el paralelismo de datos se aplica al algoritmo K-means, acelerando el cálculo de distancias y la actualización de centroides.

El software está desarrollado principalmente en C++ e integra tecnologías HPC estándar: MPI para comunicación entre nodos, OpenMP para paralelismo en CPU y OpenCL para ejecutar los kernels de K-means en

GPU. Además, incorpora esquemas master-worker con balanceo dinámico de carga, tanto entre CPU y GPU como entre nodos del clúster, optimizando la distribución de trabajo y minimizando desequilibrios, especialmente en versiones avanzadas como ODGA.

La configuración y ejecución de HPMoon se realiza desde la línea de comandos, permitiendo ajustar parámetros como el número de subpoblaciones, tamaño de la población, migraciones, generaciones, número de características y uso de CPU o GPU mediante un archivo XML. Esta flexibilidad facilita la adaptación del programa a distintos entornos de HPC y diferentes conjuntos de datos, maximizando eficiencia y reproducibilidad.

4.2.4. Justificación de la elección para este TFG

HPMoon se ha seleccionado como caso de estudio por varias razones que lo convierten en un candidato ideal para analizar aspectos como portabilidad, rendimiento y overhead de contenedores en entornos HPC. En primer lugar, su relevancia científica y tecnológica es clara: aborda un problema intensivo en cómputo, la clasificación de señales EEG de alta dimensionalidad, común en bioinformática e ingeniería biomédica. La complejidad de estas tareas permite generar métricas significativas de rendimiento en HPC.

Otro factor clave es su diseño para paralelización multinivel y arquitecturas heterogéneas. HPMoon explota múltiples niveles de paralelismo en CPUs multi-núcleo, GPUs y sistemas distribuidos en clústeres, lo que permite realizar análisis exhaustivos en distintas configuraciones. A esto se suma un énfasis en la eficiencia energética, optimizando tanto el consumo de energía como el tiempo de ejecución, un aspecto crítico en la computación de alto rendimiento moderna. Asimismo, incorpora estrategias de balanceo de carga dinámico entre CPU y GPU, lo que facilita manejar las diferencias de capacidad y consumo energético de dispositivos heterogéneos.

La disponibilidad y madurez del software también influyen en su elección. HPMoon es un proyecto robusto derivado de tesis doctoral y publicaciones científicas, con múltiples versiones evolutivas (SGA, PGA, OPGA, MDGA, MPGA, DGA, DGA-II, ODGA, GAAM) y documentación completa. Además, incluye modelos de energía-tiempo que permiten predecir el comportamiento de los algoritmos en sistemas monocomputador y distribuidos, proporcionando una base sólida para comparar resultados obtenidos en contenedores. Por último, la complejidad y los desafíos de optimización que presenta —como el balanceo de carga en entornos heterogéneos, irregularidades en accesos a memoria y la gestión de comunicación entre nodos y dispositivos— lo hacen ideal para evaluar el impacto de los contenedores en la eficiencia y optimización del código.

4.2.5. Configuración y parámetros de compilación y ejecución

La aplicación HPMoon requiere una fase previa de compilación y, posteriormente, una configuración en tiempo de ejecución a través de un fichero XML (con soporte parcial mediante parámetros de línea de comandos).

Compilación

La compilación de HPMoon se realiza a través de un *Makefile*, que permite configurar los parámetros más importantes antes de generar el ejecutable. Entre estos parámetros destacan el número de características de la base de datos (`N_FEATURES` o NF), que debe definirse entre 4 y el máximo disponible, y el compilador MPI a utilizar (`COMP` o COMPILER), que por defecto es `mpic++`.

Estos valores se procesan en tiempo de compilación, lo que permite evitar el uso de memoria dinámica innecesaria y maximizar el rendimiento del programa. Una vez completada la compilación, el ejecutable resultante, llamado `hpmoon`, se encuentra disponible en el directorio `bin`, listo para su ejecución en los distintos entornos de HPC.

Configuración en tiempo de ejecución

La configuración en tiempo de ejecución se realiza principalmente mediante un fichero XML, que permite ajustar tanto los parámetros de los algoritmos evolutivos como la gestión de recursos computacionales. Los parámetros más relevantes son:

- **NSubpopulations**: número total de subpoblaciones (modelo de islas).
- **SubpopulationSize**: tamaño de cada subpoblación (número de individuos).
- **NGlobalMigrations**: número de migraciones entre subpoblaciones en diferentes nodos.
- **NGenerations**: número de generaciones a simular.
- **MaxFeatures**: número máximo de características permitidas.
- **DataFileName**: fichero de salida con la aptitud de los individuos del primer frente de Pareto.
- **PlotFileName**: fichero con el código de gnuplot para visualización.

- **ImageFileName**: fichero de salida con la gráfica generada.
- **TournamentSize**: número de individuos en el torneo de selección.
- **NInstances**: número de instancias (filas) de la base de datos a utilizar.
- **FileName**: nombre del fichero de la base de datos de entrada.
- **Normalize**: indica si la base de datos debe ser normalizada.
- **NDevices**: número de dispositivos OpenCL empleados en el nodo.
- **Names**: lista de nombres de dispositivos OpenCL, separados por comas.
- **ComputeUnits**: unidades de cómputo por dispositivo OpenCL, en el mismo orden que `Names`.
- **WiLocal**: número de *work-items* por unidad de cómputo, alineado con los dispositivos.
- **CpuThreads**: número de hilos de CPU para la evaluación de aptitud. Si es 1 y `NDevices=0`, la ejecución es secuencial.
- **KernelsFileName**: fichero con los kernels de OpenCL.

Muchos de estos parámetros pueden especificarse también mediante opciones en la línea de comandos al ejecutar `hpmoon`, aunque no todos están disponibles de esta forma. Esta circunstancia se refleja en la columna `CMD OPTION` de la Tabla 4.1, donde un guion (`-`) indica ausencia de soporte por línea de comandos.

4.2.6. Selección de parámetros de estudio

La elección de los parámetros para este estudio se centró en mantener la mayoría de los valores por defecto y analizar únicamente la relación entre el número de subpoblaciones y el número de hilos. Esta decisión se basa en la arquitectura de paralelismo multinivel de HPMoon y su impacto directo en la distribución de la carga de trabajo y el rendimiento. A continuación, se detalla la justificación de esta selección.

Paralelismo multinivel del programa

HPMoon está diseñado como un algoritmo evolutivo que utiliza subpoblaciones y explota paralelismo en múltiples niveles. A nivel de clúster, MPI (Message Passing Interface) distribuye las subpoblaciones entre los distintos

PARÁMETRO	RANGO	CMD OPTION
N_FEATURES	$4 \leq$ NF $\leq$ Número de características de la base de datos	-
NSubpopulations	$1 \leq$ NP	-ns
SubpopulationSize	$4 \leq$ PS	-ss
NGlobalMigrations	$1 \leq$ NM	-ngm
NGenerations	$0 \leq$ NG	-g
MaxFeatures	$1 \leq$ MaxF	-maxf
DataFileName	Nombre de fichero válido	-plotdata
PlotFileName	Nombre de fichero válido	-plotsrc
ImageFileName	Nombre de fichero válido	-plotimg
TournamentSize	$2 \leq$ TS	-ts
NInstances	$4 \leq$ NI $\leq$ Número de instancias de la base de datos	-trni
FileName	Base de datos de entrenamiento existente	-trdb
Normalize	1 ó 0	-trnorm
NDevices	$0 \leq$ ND	-
Names	Nombre de dispositivo existente	-
ComputeUnits	$1 \leq$ CU	-
WiLocal	$1 \leq$ WL $\leq$ Máx. work-items locales del dispositivo	-
CpuThreads	$0 \leq$ CT	-
KernelsFileName	Fichero de kernels existente	-ke
Display usage	-	-h
List OpenCL devices	-	-l

Tabla 4.1: Rango de valores de los parámetros de entrada y su uso desde la línea de argumentos (si está disponible).

nodos. Dentro de cada nodo, OpenMP se encarga de repartir dinámicamente el trabajo entre los dispositivos disponibles, incluyendo CPU y GPU. La evaluación de la aptitud de los individuos combina OpenMP en la CPU y OpenCL en la GPU, ofreciendo hasta tres niveles de paralelismo en la CPU y cuatro en la GPU. Esta arquitectura multinivel es la que determina la elección de parámetros clave para el estudio.

Número de subpoblaciones

El parámetro `NSubpopulations` representa el número total de subpoblaciones y es fundamental para el modelo de islas del algoritmo. Incrementar este número puede mejorar la calidad de los resultados más que simplemente aumentar la población de individuos, según se indica en estudios previos [14]. Además, define cómo se distribuye el trabajo a nivel de MPI entre los nodos y cómo estas unidades se gestionan posteriormente con OpenMP dentro de cada nodo.

Número de hilos en CPU y GPU

El programa hace un uso intensivo de hilos para la evaluación de la aptitud. En la CPU, el parámetro `CpuThreads` indica cuántos hilos se utilizan, y se recomienda que coincida con el número de núcleos lógicos para obtener un buen rendimiento. En la GPU, parámetros como `NDevices` (número de dispositivos OpenCL), `ComputeUnits` (unidades de cómputo) y `WiLocal` (hilos por unidad de cómputo) son esenciales. Ajustar `WiLocal` como múltiplo de 32 o 64 mejora la eficiencia de OpenCL, y la combinación óptima de estos valores depende del problema y del dispositivo. Estos parámetros controlan la paralelización de la función de evaluación en la GPU y la distribución dinámica de individuos mediante OpenMP [14].

Número de nodos

HPMoon está preparado para ejecutarse en sistemas distribuidos. En la versión DGA (Distributed Genetic Algorithm), las subpoblaciones se reparten entre nodos usando MPI, agregando un cuarto nivel de paralelismo. El nodo maestro (MPI 0) gestiona esta distribución de manera dinámica. La escalabilidad y el rendimiento dependen directamente del número de nodos: aumentar su cantidad puede reducir significativamente el tiempo de ejecución y el consumo energético, alcanzando speedups de hasta 83 veces y reduciendo el consumo de energía a menos del 5 % en el mejor de los casos [14, 15]. Además, la heterogeneidad de los nodos y su configuración de CPU/GPU influyen en la eficiencia de la distribución de carga.

Razones para mantener otros parámetros por defecto

Se decidió no modificar el resto de los parámetros por varias razones. Primero, la prioridad del estudio es entender cómo el paralelismo y la asignación de recursos influyen en el rendimiento, por lo que se enfocó en `NSubpopulations` y en los hilos de CPU/GPU. Segundo, algunos parámetros como `N_FEATURES` se fijan en tiempo de compilación, lo que dificulta su ajuste en pruebas de rendimiento. Tercero, la función de evaluación (K-means) consume más del 99 % del tiempo de ejecución para poblaciones moderadas, por lo que optimizar su paralelización es crítico [14]. Otros parámetros secundarios relacionados con el algoritmo evolutivo o la gestión de datos no afectan de manera tan directa al comportamiento fundamental del paralelismo. Por último, estudiar bases de datos muy grandes podría provocar problemas de memoria en la GPU, por lo que es más prudente analizar estos factores una vez comprendido el paralelismo básico.

En resumen, al concentrarse en el número de subpoblaciones, el número de hebras y el número de nodos, se aborda directamente la capacidad del

programa para aprovechar arquitecturas paralelas y la distribución de la carga de trabajo en todos los niveles, sentando las bases para análisis más detallados posteriormente [14].

4.2.7. Diseño de los experimentos

Los experimentos se estructuran en varias fases con el fin de determinar los parámetros óptimos y evaluar la escalabilidad y portabilidad de HPMoon en distintos entornos y plataformas.

Determinación del número óptimo de subpoblaciones

Se realizarán ejecuciones exploratorias con distintas configuraciones de subpoblaciones y hebras, para definir el parámetro de `NSubpopulations`, así como el de `CpuThreads`, a usar en los tests posteriores:

- Se lanzarán ejecuciones con 1, 2, 4, 8 y 16 subpoblaciones.
- Para cada configuración de subpoblaciones, se evaluará con 1, 2, 4, 8 y 16 hebras por subpoblación.

Estudio del rendimiento al requerir más hebras de las disponibles

Uno de los experimentos planteados consiste en evaluar el rendimiento de la aplicación realizando un barrido completo de todas las combinaciones posibles de número de nodos y de hebras por nodo. Dado que el dispositivo en el que se realizarán los tests cuenta con un máximo de 16 hebras, resulta especialmente interesante analizar cómo se comporta la aplicación cuando se solicita un número de hebras superior al disponible. Este estudio permitirá determinar si es necesario imponer un límite en el número de hebras para los tests posteriores.

En las ejecuciones multinodo, cada nodo se configurará con un número de hebras tal que el total de hebras activas sea igual al producto del número de nodos por el número de hebras por nodo. Para explorar el comportamiento de la aplicación en escenarios límite, se plantean dos variantes de prueba: en la primera, si el número total de hebras requerido supera el máximo del dispositivo (16), el test no se ejecuta; en la segunda, se permiten ejecuciones aunque la cantidad de hebras solicitadas exceda el límite del hardware. Con esto se pretende identificar si el rendimiento se degrada o si el sistema puede manejar de manera eficiente solicitudes superiores a la capacidad real, estableciendo así las condiciones óptimas para los experimentos posteriores.

Estudio del rendimiento al utilizar la misma GPU en todos los nodos

En un escenario multinodo local, donde los nodos corresponden a procesos distribuidos dentro de un mismo dispositivo, resulta importante analizar cómo el uso de la GPU afecta al rendimiento de HPMoon. El dispositivo disponible cuenta con una única tarjeta gráfica, por lo que se consideran dos configuraciones posibles.

En la primera, todos los nodos comparten la misma GPU, lo que permite evaluar el impacto de la contención del recurso gráfico cuando varios procesos intentan ejecutar cálculos en paralelo sobre la misma tarjeta. En la segunda configuración, la GPU se asigna únicamente a un nodo, mientras que el resto de nodos realiza los cálculos exclusivamente con la CPU. Este enfoque permite medir si centralizar el uso de la GPU en un solo nodo mejora la eficiencia global frente al acceso compartido.

El objetivo de este estudio es identificar cuál de estas estrategias ofrece un mejor rendimiento en configuraciones multinodo locales. Para ello se analizará la eficiencia en términos de tiempo total de cálculo y utilización de recursos, se evaluará la escalabilidad de la aplicación al incrementar el número de nodos y se extraerán recomendaciones sobre la asignación de la GPU en futuras pruebas multinodo. De esta manera, se podrá determinar si resulta más eficiente que todos los nodos compartan la GPU o si es preferible que solo uno de ellos la utilice, teniendo en cuenta tanto el rendimiento como la complejidad de gestión de los recursos.

Experimentos de escalabilidad

Una vez definido el número óptimo de subpoblaciones, el rango válido de hebras y la estrategia de uso de la GPU en entornos multinodo, se realizarán los siguientes experimentos:

- **Escalabilidad mononodo:** con un único nodo fijo, se escala el número de hebras.
- **Escalabilidad multinodo:** con una única hebra por nodo, se escala el número de nodos.
- **Barrido de hebras:** se ejecutan todas las combinaciones posibles de número de nodos y hebras por nodo.

Plataformas y entornos de ejecución

Los experimentos se realizarán en distintas plataformas y entornos, tanto con uso único de CPU como la combinación con uso de GPU (salvo en el caso de Mac, donde el uso de GPU en contenedores es objeto de estudio):

- Ubuntu 24.04: ejecución nativa, y en contenedores Docker y Podman.
- Windows 11 Home: ejecución en contenedores Docker y Podman.
- MacOS Sequoia 15.5: ejecución en contenedores Docker y Podman.

Esta estructura permitirá evaluar la escalabilidad, portabilidad y rendimiento de HPMoon en entornos contenerizados y nativos, así como en arquitecturas heterogéneas con CPU y GPU.

4.3. Herramientas y scripts utilizados

La carpeta de scripts está organizada en subcarpetas y archivos que se pueden agrupar según su función principal en cuatro bloques:

4.3.1. Compilación y preparación

Incluye scripts como *build_hpmoon.sh* y *clean_system.sh* dentro de la carpeta `utils`. Su objetivo es automatizar la compilación del software HPMoon y preparar el entorno, asegurando que el binario esté actualizado y que el sistema esté limpio antes de cada experimento.

4.3.2. Pruebas de escalabilidad

En esta categoría se agrupan los scripts que permiten evaluar cómo HPMoon se comporta frente a diferentes configuraciones de hardware y parámetros del algoritmo, así como explorar de manera sistemática todas las combinaciones posibles de nodos y hebras. Los scripts se organizan en subcarpetas como *single-node*, *multi-node*, *experiments* y *thread-sweep*, e incluyen ejemplos como *run_ubuntu_native.sh*, *run_ubuntu_native_limit.sh* o *run_ubuntu_container.sh*.

El objetivo de estos scripts es doble: por un lado, analizar la escalabilidad y el rendimiento de la aplicación en entornos nativos y contenerizados; por otro, generar un panorama completo de cómo diferentes parámetros afectan la ejecución, permitiendo realizar barridos sistemáticos que produzcan matrices completas de resultados.

De manera más concreta, los scripts permiten:

- Variar parámetros críticos como el número de nodos, el número de hebras por nodo y el número de subpoblaciones, observando cómo influyen en el tiempo de ejecución y en la eficiencia general.
- Comparar el comportamiento de la aplicación en entornos nativos frente a entornos contenerizados, evaluando el overhead introducido por la virtualización.
- Realizar barridos exhaustivos de parámetros, registrando resultados para todas las combinaciones posibles de nodos y hebras, de manera que se puedan identificar configuraciones óptimas y detectar posibles cuellos de botella.

Gracias a esta organización y automatización, se puede estudiar de forma sistemática la eficiencia, la escalabilidad y el comportamiento de HPMoon en distintas plataformas, garantizando resultados reproducibles y comparables.

4.3.3. Automatización total y orquestación

Incluye scripts de alto nivel como *run_ubuntu_all.sh* o *run_windows_all.sh*, que coordinan compilación, ejecución y recolección de resultados de forma secuencial.

4.3.4. Valor añadido e innovación

Los scripts desarrollados aportan un valor significativo al estudio al combinar automatización, flexibilidad y reproducibilidad. Permiten ejecutar campañas de pruebas complejas con un solo comando, minimizando errores humanos y optimizando el tiempo de ejecución. Al mismo tiempo, facilitan la modificación rápida de parámetros clave sin intervención manual, lo que permite explorar distintas configuraciones de nodos, hebras y subpoblaciones de manera eficiente. Su estructura también posibilita comparar de forma sistemática el rendimiento entre ejecuciones nativas y contenerizadas, así como entre distintas plataformas, asegurando resultados homogéneos y fácilmente interpretables. Además, estos scripts soportan escenarios multinodo y multiplataforma, validando la escalabilidad y la portabilidad de HPMoon en distintos entornos de HPC. Finalmente, la gestión organizada de logs y resultados garantiza que los experimentos sean completamente reproducibles y auditables, alineándose con las buenas prácticas científicas.

4.4. Repositorio del proyecto

Todo el trabajo desarrollado en este TFG, incluyendo scripts, configuraciones, documentación y resultados de los experimentos, está recogido y disponible públicamente en un repositorio de GitHub[2].

[2] `https://github.com/FerniCuesta/DockEEG`

Capítulo 5

Experimentación

5.1. Experimentos preliminares

En esta primera fase de experimentacion se realizarán pruebas para determinar los parámetros óptimos de ejecución y estudiar el rendimiento de la aplicación en distintos entornos y plataformas.

5.1.1. Determinación del número óptimo de subpoblaciones y hebras

Como se ha comentado en la sección 4.2.7, se realizarán ejecuciones exploratorias con distintas configuraciones de subpoblaciones y hebras, para definir el parámetro de `NSubpopulations` a usar en los tests posteriores.

En la figura 5.1 se muestran las gráficas de estas ejecuciones exploratorias. Se observa que para cualquier número de subpoblaciones, el comportamiento es similar. En la tabla 5.1 se presentan los tiempos de ejecución en segundos para cada combinación de hebras y subpoblaciones, mostrando cómo el incremento en el número de subpoblaciones aumenta significativamente el tiempo de ejecución, especialmente cuando se utilizan pocas hebras.

En la tabla 5.2 se presentan los porcentajes de reducción del tiempo de ejecución respecto a la configuración base. En cada columna, la ejecución con una sola hebra y una sola subpoblación se toma como referencia (0 % de reducción). Estos resultados permiten cuantificar de manera objetiva el beneficio relativo derivado del incremento del paralelismo y de la subdivisión de la población, proporcionando una base empírica para identificar configuraciones óptimas de ejecución.

Del análisis de esta tabla pueden extraerse varias conclusiones de interés para la definición de los parámetros en los experimentos posteriores. En

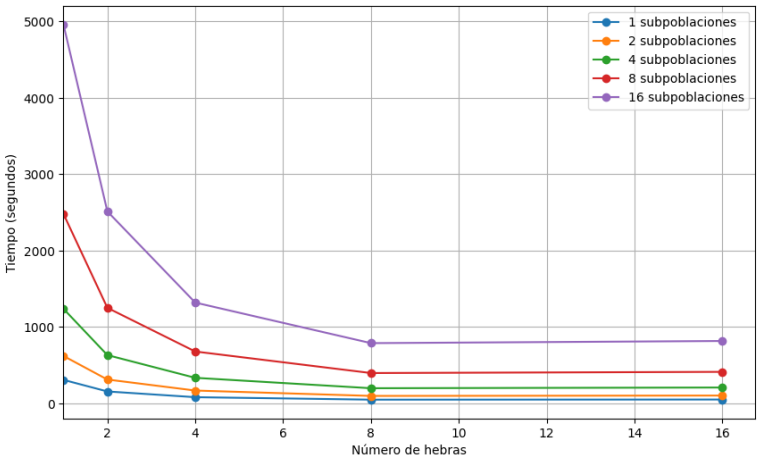

Figura 5.1: Gráficas de ejecución de las pruebas variando el número de subpoblaciones y hebras.

Hebras	**Subpoblaciones**				
	1	**2**	**4**	**8**	**16**
1	309.43	622.63	1239.73	2481.89	4960.00
2	157.79	313.86	633.66	1252.12	2513.72
4	82.73	169.76	336.25	680.46	1320.66
8	50.58	99.37	200.01	398.66	790.04
16	52.32	104.06	209.17	414.11	818.17

Tabla 5.1: Tiempos de ejecución en segundos de las pruebas variando el número de subpoblaciones y hebras.

Hebras	**Subpoblaciones**				
	1	**2**	**4**	**8**	**16**
1	0.00	0.00	0.00	0.00	0.00
2	-49.01	-49.59	-48.89	-49.55	-49.32
4	-73.26	-72.74	-72.88	-72.58	-73.37
8	-83.65	-84.04	-83.87	-83.94	-84.07
16	-83.09	-83.29	-83.13	-83.31	-83.50

Tabla 5.2: Porcentaje de reducción del tiempo de ejecución respecto a la configuración base para distintas combinaciones de hebras y subpoblaciones

primer lugar, se observa que, para un número fijo de hebras, la variación en el porcentaje de reducción del tiempo es prácticamente inexistente al modificar el número de subpoblaciones, lo que indica que el comportamiento es proporcional con independencia de este parámetro. En segundo lugar, el mayor beneficio en términos de reducción del tiempo de ejecución se obtiene

al incrementar el número de hebras de 1 a 8, alcanzando valores en torno al 83–84 %. Sin embargo, al pasar de 8 a 16 hebras, los tiempos de ejecución se incrementan en todos los casos, lo que revela que se ha sobrepasado el punto de paralelismo óptimo para la arquitectura hardware utilizada. Este resultado sugiere que, en el entorno experimental considerado, el uso de más de 8 hebras no proporciona mejoras de rendimiento adicionales e, incluso, puede resultar contraproducente debido a la sobrecarga y la contención de recursos.

No obstante, se ha decidido mantener configuraciones con 16 hebras en los experimentos posteriores con el fin de analizar el comportamiento bajo condiciones de sobrecarga, dado que el objetivo del estudio trasciende la mera optimización de un caso específico y busca evaluar la escalabilidad y el rendimiento en un rango amplio de escenarios. En cuanto al número de subpoblaciones, se constata que con 16 subpoblaciones los tiempos de ejecución pueden alcanzar hasta 1 hora y 20 minutos en configuraciones con una única hebra, lo que resulta excesivo para los objetivos de este trabajo. Por el contrario, con 8 subpoblaciones se obtienen tiempos de ejecución más razonables, se alcanza un equilibrio adecuado entre eficiencia y aprovechamiento de recursos, y se garantiza una buena escalabilidad.

En consecuencia, se justifica la decisión metodológica de fijar el número de subpoblaciones en 8 para los experimentos posteriores, al representar un compromiso óptimo entre eficiencia, utilización de recursos y escalabilidad en el entorno analizado.

5.1.2. Estudio del rendimiento al requerir más hebras de las disponibles

En esta sección se presentan los resultados de las ejecuciones exploratorias realizadas para analizar el rendimiento de la aplicación al variar el número de nodos y hebras, incluyendo configuraciones que superan el límite físico de hebras de la CPU (16 hebras). El objetivo es comprender cómo afecta esta variación al tiempo de ejecución y al uso efectivo de la CPU, proporcionando una base para la selección de parámetros en estudios posteriores.

En la figura 5.2 se muestra la evolución del tiempo de ejecución en función del número de hebras asignadas por nodo, para configuraciones que van desde 1 hasta 16 nodos, pero limitando el número máximo de hebras a 16.

La figura 5.3 muestra el porcentaje de uso total de CPU (donde 100 % equivale al uso completo de una hebra) para las distintas configuraciones de hebras por nodo analizadas.

Por otro lado, en la figura 5.4 se presentan los resultados de las ejecucio-

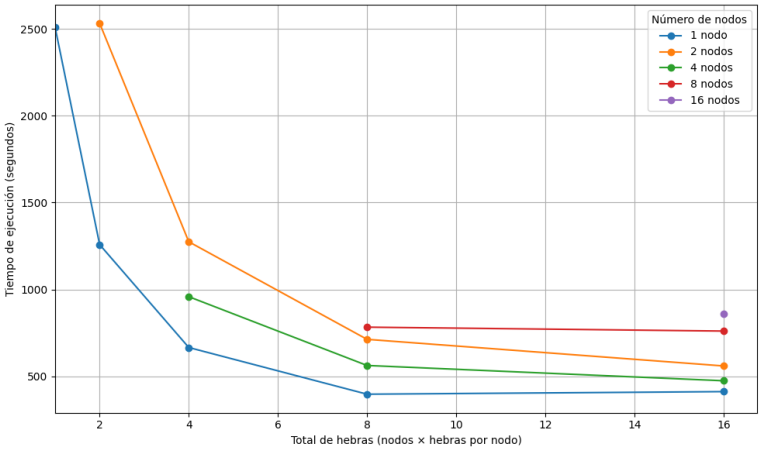

Figura 5.2: Gráfica de tiempo de ejecución en función del número de hebras por nodo, con límite de 16 hebras.

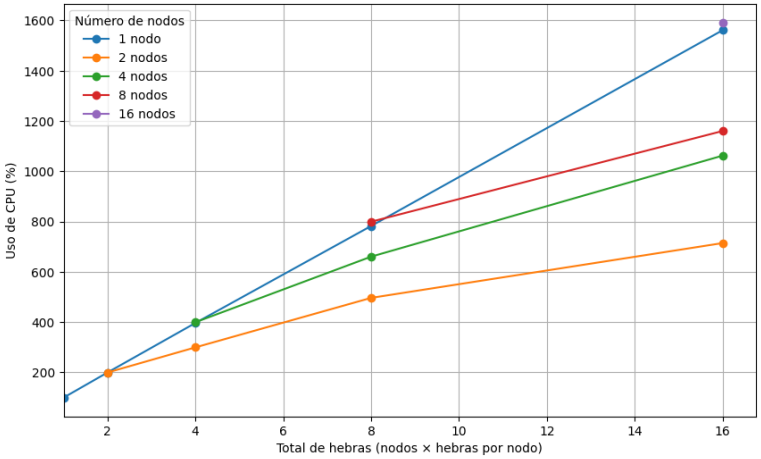

Figura 5.3: Gráfica de uso de CPU en función del número de hebras por nodo, con límite de 16 hebras.

nes exploratorias sin límite en el número de hebras, permitiendo así evaluar el comportamiento del sistema al requerir más hebras de las disponibles.

La figura 5.5 muestra el porcentaje de uso total de CPU para las distintas configuraciones de hebras por nodo sin límite, permitiendo observar cómo varía el aprovechamiento de los recursos en función del número total de hebras asignadas.

Las conclusiones que se pueden extraer de estas gráficas se pueden ver de manera más clara en la tabla 5.3, donde se resumen los tiempos de ejecución y el uso de CPU para todas las combinaciones de nodos y hebras analizadas,

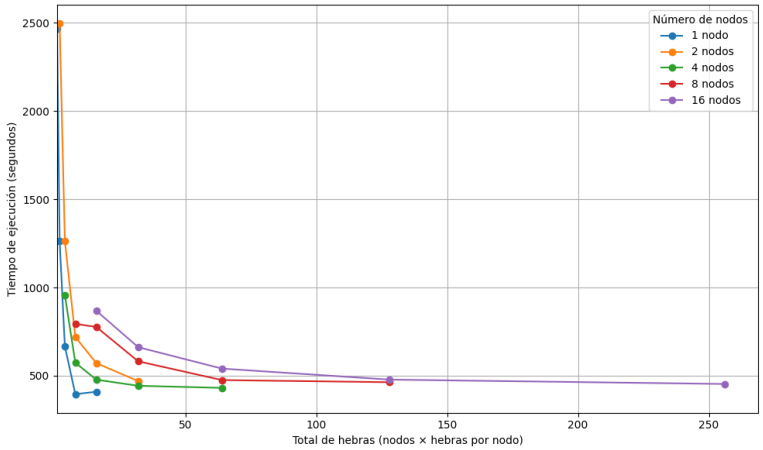

Figura 5.4: Gráfica de tiempo de ejecución en función del número de hebras por nodo, sin límite en el número de hebras.

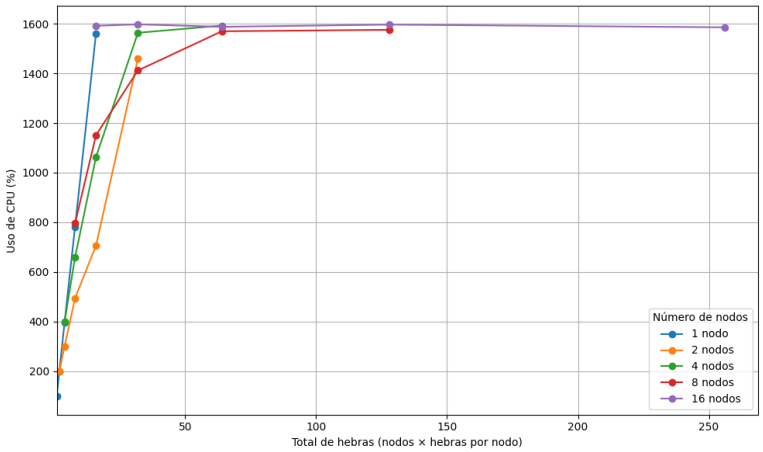

Figura 5.5: Gráfica de uso de CPU en función del número de hebras por nodo, sin límite en el número de hebras.

tanto con límite como sin límite en el número de hebras.

Los resultados muestran que el mejor rendimiento se alcanza empleando un menor número de nodos con un mayor número de hebras por nodo, siendo la configuración óptima la de un único nodo y ocho hebras. Aunque el límite físico de hebras de la CPU es de 16, se observa que, entre los diez mejores resultados, solo tres respetan este límite. En los demás casos, incrementar el número de hebras más allá de la capacidad física sigue proporcionando mejoras en el rendimiento, un comportamiento que, aunque inicialmente contraintuitivo, puede explicarse analizando el uso efectivo de la CPU.

Nodos	Hebras por nodo	Hebras totales	Tiempo (s)	Uso de CPU (%)
1	8	8	395.63	782
1	16	16	409.51	1561
4	16	64	431.30	1593
4	8	32	443.23	1564
16	16	256	453.48	1586
8	16	128	463.45	1576
2	16	32	469.88	1460
8	8	64	475.59	1570
4	4	16	477.90	1063
16	8	128	478.23	1597
16	4	64	540.54	1588
2	8	16	571.65	706
4	2	8	573.94	659
8	4	32	581.91	1412
16	2	32	662.10	1598
1	4	4	666.51	396
2	4	8	719.43	494
8	2	16	776.44	1151
8	1	8	794.09	796
16	1	16	868.28	1592
4	1	4	956.38	399
2	2	4	1263.74	299
1	2	2	1264.94	199
1	1	1	2467.76	99
2	1	2	2497.06	199

Tabla 5.3: Resumen de configuraciones de nodos, hebras y uso de CPU

El porcentaje de utilización de la CPU refleja el grado de aprovechamiento de las hebras disponibles: por ejemplo, con una hebra se alcanza un uso máximo de 100 %, con dos hebras el 200 %, y así sucesivamente hasta un máximo teórico de 1600 % (16 hebras $\times$ 100 %). En configuraciones de un único nodo, aumentar el número de hebras se traduce en un incremento proporcional del uso de CPU, lo que explica las mejoras de rendimiento observadas.

En contraste, cuando el número total de hebras se distribuye entre varios nodos, incluso si no se supera el límite físico de la CPU, el rendimiento no mejora de la misma manera. Esto se debe a la sobrecarga inherente a la gestión de múltiples nodos, que puede contrarrestar los beneficios de disponer de más hebras, haciendo que el uso efectivo de la CPU no alcance los valores esperados y resultando en un rendimiento inferior respecto a configuraciones mononodo equivalentes. Por tanto, para configuraciones multinodo es

necesario incrementar el número de hebras más allá de la capacidad física de cada CPU para acercarse al uso máximo teórico, lo que explica por qué los mejores resultados se obtienen bajo estas condiciones.

5.1.3. Estudio del rendimiento al utilizar la misma GPU en distintos nodos

En esta sección se presentan los resultados de las ejecuciones exploratorias realizadas para analizar el rendimiento de la aplicación al variar el número de nodos y hebras, considerando dos enfoques distintos respecto al uso de la GPU: uno en el que se limita su uso a un único nodo y otro en el que se permite su uso en todos los nodos. El objetivo es comprender cómo afecta esta variación al tiempo de ejecución y al uso efectivo de la CPU, proporcionando una base para la selección de parámetros en estudios posteriores.

En la figura 5.6 se muestra la evolución del tiempo de ejecución en función del número de hebras asignadas por nodo, para configuraciones que van desde 1 hasta 16 nodos, limitando el uso de la GPU a un único nodo.

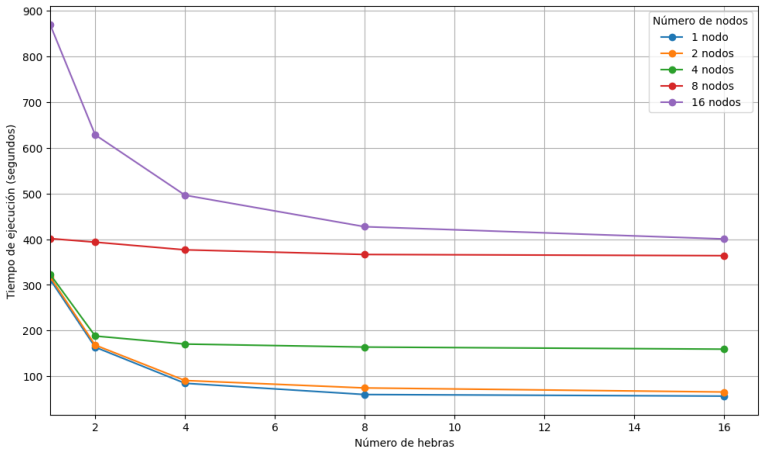

Figura 5.6: Gráfica de tiempo de ejecución en función del número de hebras por nodo, con la GPU limitada a un único nodo.

La figura 5.7 presenta los resultados de las ejecuciones exploratorias permitiendo el uso de la GPU en todos los nodos, evaluando así el comportamiento del sistema bajo esta configuración.

Las conclusiones que se pueden extraer de estas gráficas se pueden ver de manera más clara en la tabla 5.4, donde se resumen los tiempos de ejecución para todas las combinaciones de nodos y hebras analizadas, tanto con la GPU limitada a un nodo como permitiendo su uso en todos los nodos.

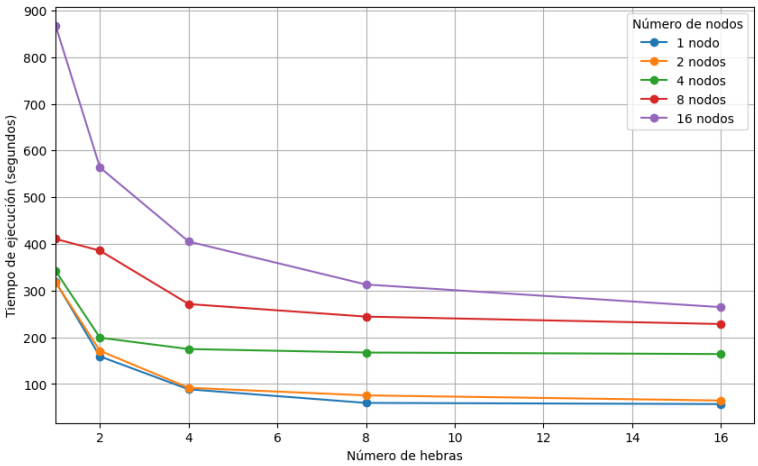

Figura 5.7: Gráfica de tiempo de ejecución en función del número de hebras por nodo, permitiendo el uso de la GPU en todos los nodos.

Nodos	Hebras	GPU 1 nodo (s)	GPU todos (s)	Var. (%)
1	1	311.68	319.01	2.35
1	2	163.59	158.94	-2.84
1	4	84.49	88.52	4.77
1	8	59.91	59.41	-0.83
1	16	56.45	56.91	0.81
2	1	316.92	317.35	0.14
2	2	168.14	170.99	1.70
2	4	90.58	91.78	1.32
2	8	74.30	75.53	1.66
2	16	65.40	64.51	-1.36
4	1	322.77	341.95	5.94
4	2	187.98	199.04	5.88
4	4	170.38	174.74	2.56
4	8	163.80	167.31	2.14
4	16	159.22	164.07	3.05
8	1	401.33	410.54	2.29
8	2	393.50	385.47	-2.04
8	4	376.68	271.10	-28.03
8	8	366.47	244.26	-33.35
8	16	363.94	228.34	-37.26
16	1	869.40	867.50	-0.22
16	2	628.49	563.51	-10.34
16	4	496.28	404.98	-18.40
16	8	427.30	312.94	-26.76
16	16	400.48	264.49	-33.96

Tabla 5.4: Resumen de tiempos de ejecución para distintas combinaciones de nodos y hebras, comparando el uso de la GPU limitada a un nodo frente a su uso en todos los nodos.

Para configuraciones con pocos nodos (1, 2 o 4), la diferencia entre utilizar la GPU en un único nodo o en todos los nodos resulta pequeña y variable.

Las variaciones porcentuales oscilan entre valores positivos y negativos, pero en general se mantienen por debajo del 6 %. Esto indica que, en estos escenarios, no existe una ventaja clara ni consistente de emplear la GPU de forma compartida en todos los nodos.

A partir de 8 nodos, la ejecución con la GPU disponible en todos los nodos comienza a mostrar mejoras significativas, especialmente al incrementar el número de hebras. Por ejemplo, con 8 nodos y 16 hebras se observa una reducción del tiempo de ejecución del 37.26 %, mientras que con 16 nodos y 16 hebras la mejora alcanza el 33.96 %. Estas diferencias son consistentes y tienden a aumentar conforme crece el número de nodos y hebras.

En configuraciones con un número elevado de nodos y hebras, la opción de permitir el acceso a la GPU en todos los nodos se presenta como claramente superior, ya que maximiza el aprovechamiento de la capacidad de cómputo distribuido y reduce de manera significativa los tiempos de ejecución.

En resumen, para experimentos pequeños o con pocos nodos ambas opciones resultan comparables. Sin embargo, en experimentos de mayor escala y con configuraciones que involucran un alto número de nodos y hebras, la estrategia más eficiente y recomendable es habilitar el uso de la GPU en todos los nodos.

5.1.4. Análisis de los experimentos preliminares

A partir de los resultados obtenidos, se recomienda fijar el número de subpoblaciones en 8, ya que este valor representa un equilibrio óptimo entre eficiencia, utilización de recursos y escalabilidad en el entorno analizado.

En cuanto al número de hebras, los datos muestran que imponer un límite estricto puede restringir el aprovechamiento total de los recursos, especialmente en configuraciones multinodo. Por ello, se recomienda no limitar el número de hebras, permitiendo que el sistema utilice tantas como sean necesarias para maximizar el rendimiento.

Respecto al uso de la GPU, los experimentos indican que en configuraciones pequeñas las diferencias entre habilitarla en todos los nodos o solo en uno son mínimas. Sin embargo, en escenarios de mayor escala, habilitar su uso en todos los nodos aporta mejoras sustanciales en el tiempo de ejecución y en la eficiencia global. Por tanto, se establece como criterio general que la GPU esté disponible en todos los nodos.

En conjunto, estas decisiones proporcionan una base sólida y coherente para el diseño de los experimentos posteriores, asegurando que se exploten al máximo los recursos disponibles sin comprometer la comparabilidad de los resultados.

5.2. Pruebas mononodo

5.2.1. Ejecución en Ubuntu en nativo

CPU

En la figura 5.8 se muestra el tiempo de ejecución para la configuración de CPU en un único nodo con Ubuntu nativo.

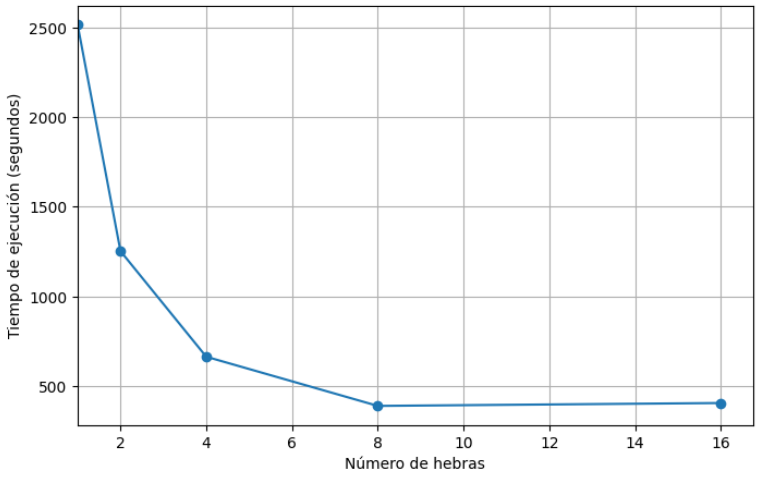

Figura 5.8: Tiempo de ejecución en función del número de hebras en Ubuntu nativo (CPU).

En la tabla 5.5 se presentan los tiempos de ejecución y la reducción porcentual respecto a una hebra.

Hebras	Tiempo (s)	Δ % vs 1 hebra
1	2515.21	0.00
2	1253.18	-50.18
4	664.69	-73.57
8	390.72	-84.47
16	406.76	-83.83

Tabla 5.5: Tiempos de ejecución y reducción porcentual respecto a una hebra en Ubuntu nativo (CPU).

El tiempo de ejecución disminuye drásticamente al aumentar el número de hebras, especialmente en el rango de 1 a 8 hebras. Con 2 hebras, el tiempo se reduce prácticamente a la mitad ($-50.18\,\%$), y con 4 hebras, a casi una cuarta parte del tiempo original ($-73.57\,\%$). El mayor beneficio se observa al pasar de 4 a 8 hebras, alcanzando una reducción del $-84.47\,\%$ respecto a

una hebra. Sin embargo, al incrementar a 16 hebras, el tiempo de ejecución es ligeramente superior al obtenido con 8 hebras, lo que sugiere la aparición de saturación o sobrecarga en el sistema. Por tanto, el óptimo se alcanza con 8 hebras, coincidiendo con el número de núcleos físicos disponibles en el sistema.

En la figura 5.9 se muestra el porcentaje de uso de CPU en función del número de hebras.

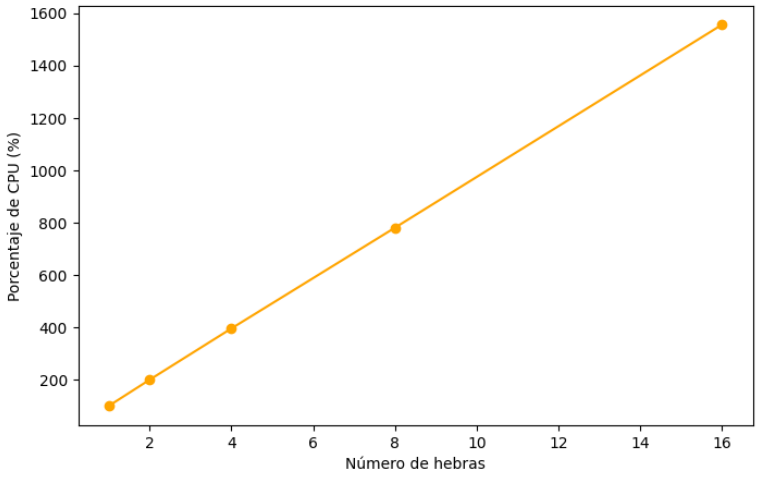

Figura 5.9: Uso de CPU en función del número de hebras en Ubuntu nativo (CPU).

La tabla 5.6 muestra el porcentaje de uso de CPU alcanzado para cada número de hebras, el máximo teórico posible (calculado como número de hebras por 100 %), y la eficiencia relativa obtenida.

Hebras	**Porcentaje de CPU (%)**	**Max posible CPU (%)**	**Eficiencia CPU (%)**
1	99.00	100.00	99.00
2	199.00	200.00	99.50
4	395.00	400.00	98.75
8	780.00	800.00	97.50
16	1556.00	1600.00	97.25

Tabla 5.6: Porcentaje de uso de CPU y eficiencia en función del número de hebras en Ubuntu nativo (CPU).

El uso de CPU aumenta de manera casi lineal conforme se incrementa el número de hebras, lo que evidencia un excelente escalado del paralelismo en la ejecución. La eficiencia de uso de la CPU se mantiene muy elevada en todos los casos, superando el 97 %, lo que indica que prácticamente se está aprovechando todo el potencial de cómputo disponible. Aunque la eficiencia

disminuye ligeramente al aumentar el número de hebras (desde un máximo del 99 % hasta un mínimo del 97.25 %), esta reducción es mínima y esperable, ya que responde a la sobrecarga asociada a la gestión de un mayor número de hilos y a posibles contenciones internas. En conjunto, el sistema demuestra un escalado eficiente hasta 16 hebras, con una pérdida de eficiencia muy reducida.

Estos resultados evidencian que el entorno mononodo nativo en Ubuntu gestiona eficazmente el paralelismo y aprovecha bien los recursos de la CPU, mostrando un buen escalado del rendimiento al aumentar el número de hebras.

CPU + GPU

En la figura 5.10 se muestra el tiempo de ejecución para la configuración de CPU + GPU en un único nodo con Ubuntu nativo.

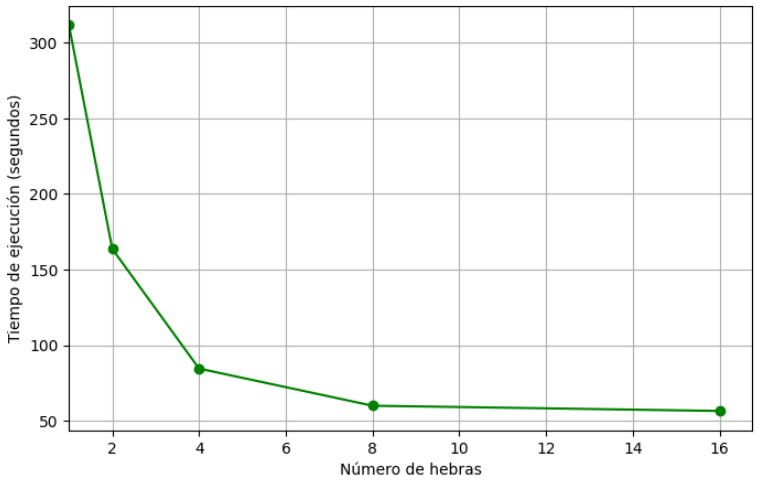

Figura 5.10: Tiempo de ejecución en función del número de hebras en Ubuntu nativo (CPU + GPU).

En la tabla 5.7 se presentan los tiempos de ejecución y la reducción porcentual respecto a una hebra.

El análisis de los resultados muestra que el tiempo de ejecución disminuye de forma significativa al incrementar el número de hebras, especialmente en el rango de 1 a 4 hebras, donde se alcanza una reducción del 72.89 %. La mayor ganancia relativa se produce al pasar de 1 a 2 hebras (−47.51 %) y de 2 a 4 hebras (−25.38 % adicional). Sin embargo, a partir de 8 hebras, la mejora es marginal: el tiempo de ejecución solo disminuye de 59.91 s a 56.45 s al pasar de 8 a 16 hebras, lo que supone apenas un 1.11 % adicional.

Hebras	Tiempo (s)	Δ % vs 1 hebra
1.00	311.68	0.00
2.00	163.59	-47.51
4.00	84.49	-72.89
8.00	59.91	-80.78
16.00	56.45	-81.89

Tabla 5.7: Tiempos de ejecución y reducción porcentual respecto a una hebra en Ubuntu nativo (CPU + GPU).

Esto indica que el sistema alcanza un punto óptimo de rendimiento con 8 hebras, y que a partir de ese valor se observa una saturación en la eficiencia de la paralelización. En conjunto, la combinación CPU + GPU proporciona una aceleración notable, pero la eficiencia se estabiliza a partir de cierto número de hebras, reflejando el límite práctico de paralelismo efectivo en el entorno analizado.

Comparativa CPU vs CPU + GPU

En la figura 5.11 se muestra una comparativa del tiempo de ejecución entre las configuraciones de CPU y CPU + GPU en un único nodo con Ubuntu nativo.

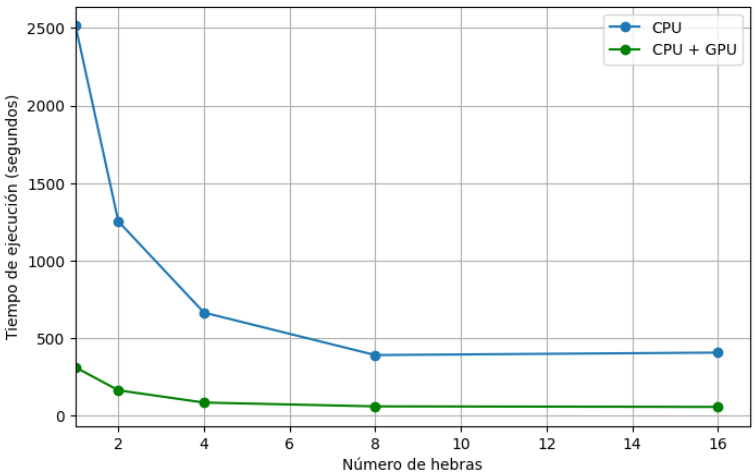

Figura 5.11: Comparativa de tiempo de ejecución entre CPU y CPU + GPU en función del número de hebras en Ubuntu nativo.

En la tabla 5.8 se presentan los tiempos de ejecución para ambas configuraciones y la variación porcentual entre ellas.

El análisis de los resultados muestra que el uso de GPU reduce el tiempo

Hebras	Tiempo CPU (s)	Tiempo CPU+GPU (s)	Variación (%)
1	2515.21	311.68	-87.61
2	1253.18	163.59	-86.95
4	664.69	84.49	-87.29
8	390.72	59.91	-84.67
16	406.76	56.45	-86.12

Tabla 5.8: Comparativa de tiempos de ejecución y variación porcentual entre CPU y CPU+GPU en Ubuntu nativo.

de ejecución entre un 84 % y un 88 % respecto a la ejecución únicamente en CPU, independientemente del número de hebras empleadas. La mejora relativa se mantiene estable al aumentar el paralelismo, lo que indica que la GPU aporta un beneficio constante y no dependiente del número de hilos de CPU utilizados. Además, mientras que en la configuración solo CPU el tiempo de ejecución deja de mejorar al pasar de 8 a 16 hebras (e incluso empeora ligeramente), en la configuración CPU+GPU la mejora es marginal pero consistente. En conjunto, la incorporación de GPU resulta altamente beneficiosa, acelerando el procesamiento en torno al 85 % en todos los escenarios analizados.

5.2.2. Ejecución en contenedores de Ubuntu

CPU

En la figura 5.12 se muestra el tiempo de ejecución para la configuración de CPU en un único nodo con contenedores de Ubuntu.

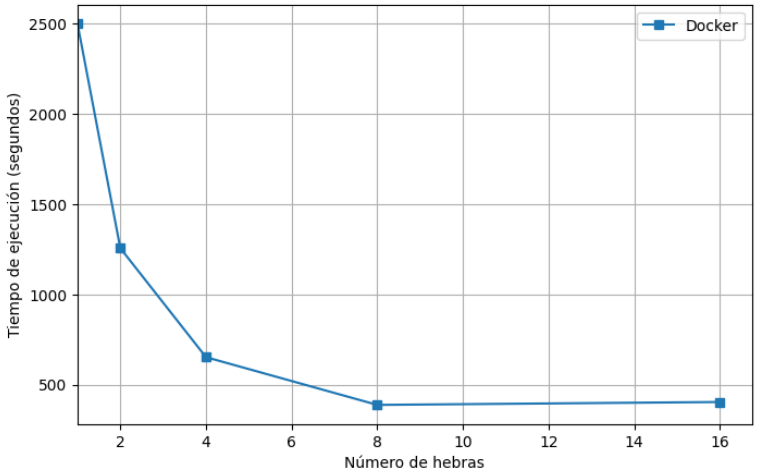

Figura 5.12: Tiempo de ejecución en un único nodo con contenedores de Ubuntu (CPU).

En la tabla 5.9 se presentan los tiempos de ejecución y la reducción porcentual respecto a una hebra.

Hebras	**Tiempo (s)**	**Δ % vs 1 hebra**
1	2499.09	0.00
2	1255.95	-49.74
4	652.33	-73.90
8	388.04	-84.47
16	404.09	-83.83

Tabla 5.9: Tiempos de ejecución y reducción porcentual respecto a una hebra en contenedores de Ubuntu (CPU).

El tiempo de ejecución disminuye significativamente al aumentar el número de hebras, especialmente al pasar de 1 a 8 hebras, donde la reducción alcanza el 84.47 %.

La mayor ganancia relativa se obtiene al pasar de 1 a 2 hebras (-49.74 %) y de 2 a 4 hebras (-48.16 % adicional), mostrando una buena escalabilidad inicial.

A partir de 8 hebras, la mejora se estabiliza y el rendimiento apenas varía, e incluso con 16 hebras el tiempo de ejecución es ligeramente superior al de 8 hebras, lo que indica que se alcanza un límite de paralelización eficiente.

Estos resultados sugieren que, en este entorno, el uso de más de 8 hebras no aporta beneficios significativos y puede incluso generar sobrecarga, por lo que 8 hebras representa el punto óptimo de eficiencia para la ejecución en contenedores de Ubuntu con CPU.

En la figura 5.13 se muestra el tiempo de ejecución para la configuración de CPU en un único nodo con contenedores de Ubuntu gestionados por Podman.

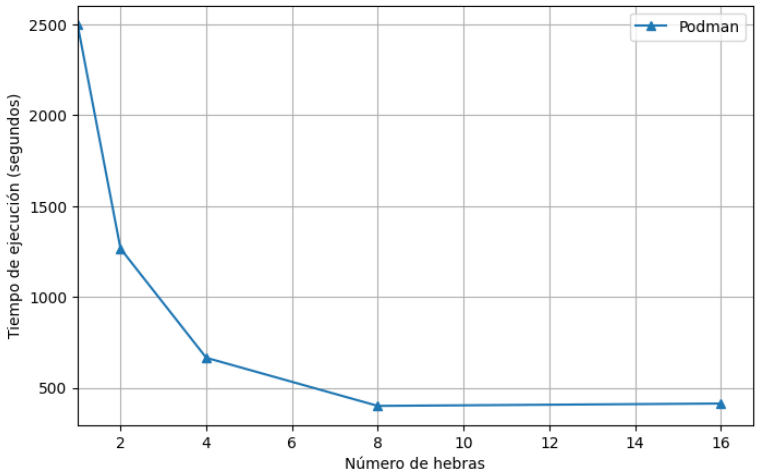

Figura 5.13: Tiempo de ejecución en un único nodo con contenedores de Ubuntu gestionados por Podman (CPU).

En la tabla 5.10 se presentan los tiempos de ejecución y la reducción porcentual respecto a una hebra.

Hebras	**Tiempo (s)**	**Δ % vs 1 hebra**
1	2499.16	0.00
2	1266.51	-49.32
4	665.38	-73.38
8	400.51	-83.97
16	413.53	-83.45

Tabla 5.10: Tiempos de ejecución y reducción porcentual respecto a una hebra en contenedores de Ubuntu gestionados por Podman (CPU).

El tiempo de ejecución disminuye de forma considerable al aumentar el número de hebras, especialmente entre 1 y 8 hebras, donde la reducción alcanza el 83.97 %.

La mayor mejora relativa se observa al pasar de 1 a 2 hebras (-49.32 %) y de 2 a 4 hebras (-47.97 % adicional), lo que indica una buena escalabilidad inicial.

A partir de 8 hebras, la reducción en el tiempo de ejecución se estabiliza y el beneficio adicional es mínimo; con 16 hebras, el tiempo incluso aumenta ligeramente respecto a 8 hebras, lo que sugiere que se alcanza el límite de

paralelización eficiente.

Estos resultados muestran que, en este entorno con Podman y CPU, el uso óptimo se encuentra en torno a 8 hebras, ya que aumentar más allá de este valor no aporta mejoras significativas y puede generar sobrecarga.

CPU + GPU

En la figura 5.14 se muestra el tiempo de ejecución para la configuración de CPU + GPU en un único nodo con contenedores de Ubuntu.

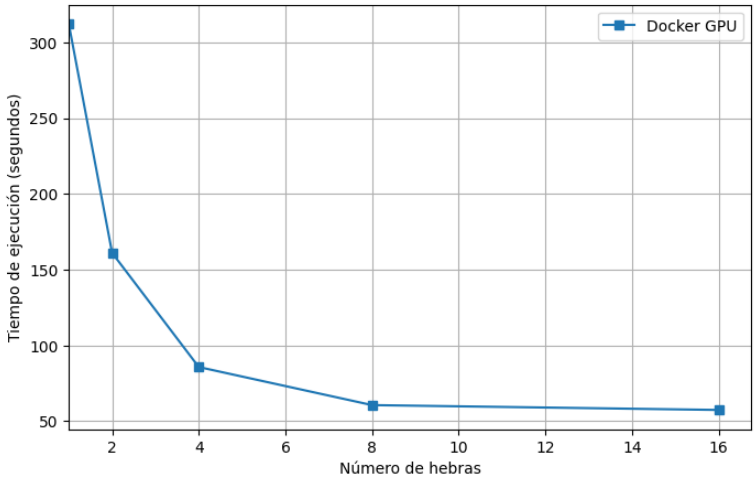

Figura 5.14: Tiempo de ejecución en un único nodo con contenedores de Ubuntu (CPU + GPU).

En la tabla 5.11 se presentan los tiempos de ejecución y la reducción porcentual respecto a una hebra.

Hebras	**Tiempo (s)**	**Δ % vs 1 hebra**
1	312.18	0.00
2	161.10	-48.40
4	85.78	-72.52
8	60.65	-80.57
16	57.45	-81.60

Tabla 5.11: Tiempos de ejecución y reducción porcentual respecto a una hebra en contenedores de Ubuntu (CPU + GPU).

El uso combinado de CPU y GPU en contenedores de Ubuntu permite una reducción muy significativa en los tiempos de ejecución al aumentar el

número de hebras, alcanzando una disminución del 81.60 % con 16 hebras respecto a la ejecución con una sola hebra.

La mayor ganancia relativa se observa al pasar de 1 a 2 hebras (-48.40 %) y de 2 a 4 hebras (-24.12 % adicional), lo que indica una buena escalabilidad inicial.

A partir de 8 hebras, la mejora se estabiliza y el beneficio adicional es marginal; el tiempo de ejecución con 16 hebras es solo ligeramente menor que con 8 hebras.

Estos resultados muestran que, en este entorno, la combinación de CPU y GPU permite aprovechar eficientemente el paralelismo hasta 8 hebras, siendo el punto óptimo de eficiencia, ya que aumentar más allá de este valor aporta mejoras muy pequeñas.

En la figura 5.15 se muestra el tiempo de ejecución para la configuración de CPU + GPU en un único nodo con contenedores de Ubuntu gestionados por Podman.

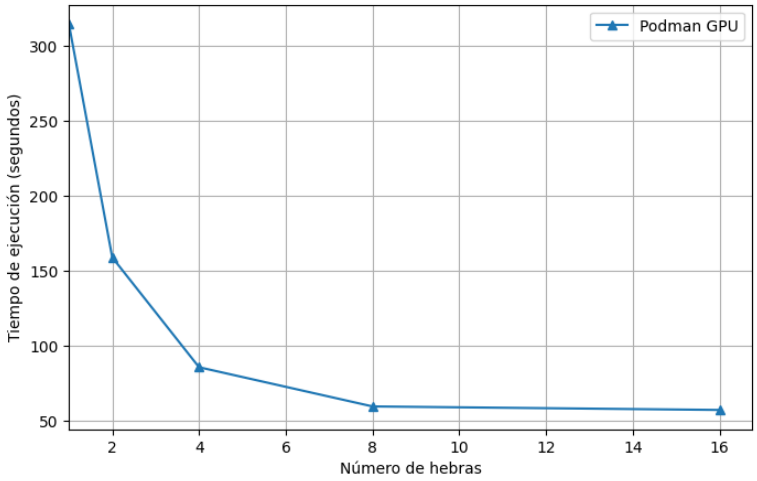

Figura 5.15: Tiempo de ejecución en un único nodo con contenedores de Ubuntu gestionados por Podman (CPU + GPU).

En la tabla 5.12 se presentan los tiempos de ejecución y la reducción porcentual respecto a una hebra.

El tiempo de ejecución disminuye drásticamente al aumentar el número de hebras, alcanzando una reducción del 81.84 % con 16 hebras respecto a la ejecución con una sola hebra.

La mayor mejora relativa se observa al pasar de 1 a 2 hebras (-49.50 %) y de 2 a 4 hebras (-23.28 % adicional), lo que indica una excelente escalabilidad inicial.

Hebras	Tiempo (s)	Δ % vs 1 hebra
1	314.51	0.00
2	158.84	-49.50
4	85.62	-72.78
8	59.47	-81.09
16	57.12	-81.84

Tabla 5.12: Tiempos de ejecución y reducción porcentual respecto a una hebra en contenedores de Ubuntu gestionados por Podman (CPU + GPU).

A partir de 8 hebras, la reducción en el tiempo de ejecución se estabiliza y el beneficio adicional es muy pequeño; el tiempo con 16 hebras es solo ligeramente menor que con 8 hebras.

Estos resultados muestran que, en este entorno con Podman y CPU+GPU, el uso óptimo se encuentra en torno a 8 hebras, ya que aumentar más allá de este valor no aporta mejoras significativas y puede generar sobrecarga.

Comparativa contenedores vs nativo

En la figura 5.16 se muestra una comparativa del tiempo de ejecución entre las configuraciones nativas y en contenedores de Ubuntu para CPU.

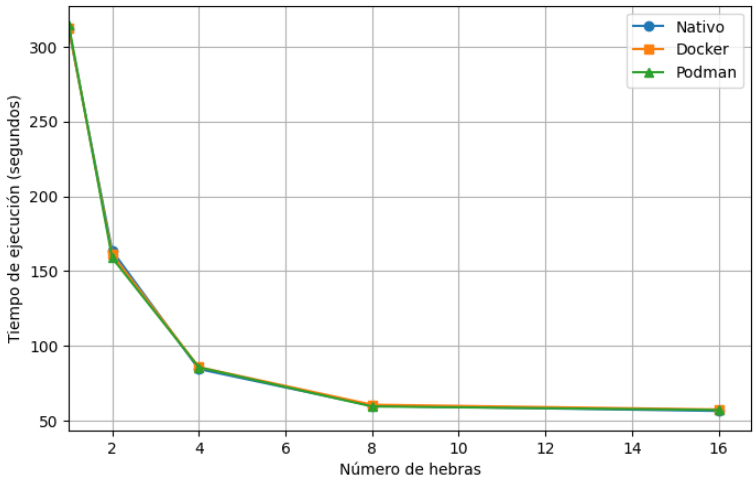

Figura 5.16: Comparativa de tiempo de ejecución entre nativo y contenedores de Ubuntu en función del número de hebras, para CPU.

En la tabla 5.13 se presentan los tiempos de ejecución para ambas configuraciones y la variación porcentual entre ellas.

Hebras	Nativo (s)	Docker (s)	Docker Δ %	Podman (s)	Podman Δ %
1	311.68	312.18	0.16	314.51	0.91
2	163.59	161.10	-1.52	158.84	-2.90
4	84.49	85.78	1.53	85.62	1.34
8	59.91	60.65	1.24	59.47	-0.73
16	56.45	57.45	1.77	57.12	1.19

Tabla 5.13: Comparativa de tiempos de ejecución entre nativo, Docker y Podman (CPU+GPU) y variación porcentual respecto a nativo.

5.2.3. Contenedores en contenedores de Windows

CPU

En la figura 5.17 se muestra el tiempo de ejecución para la configuración de CPU en un único nodo con Docker en Windows.

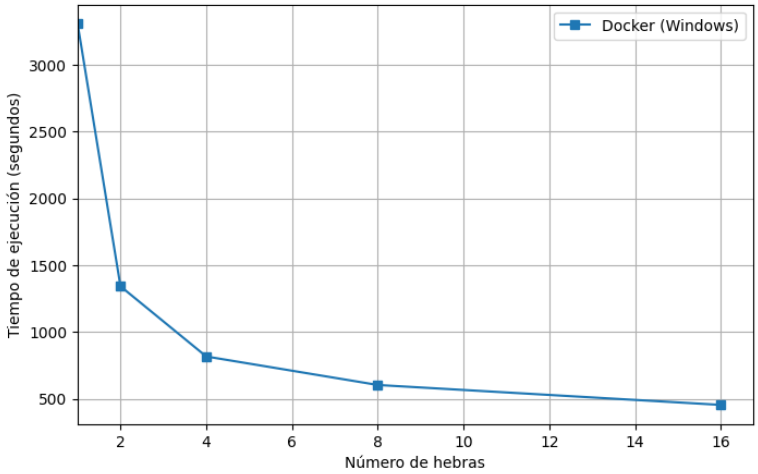

Figura 5.17: Tiempo de ejecución en un único nodo con Docker en Windows (CPU).

En la tabla 5.14 se presentan los tiempos de ejecución y la reducción porcentual respecto a una hebra.

El tiempo de ejecución disminuye de forma significativa al aumentar el número de hebras, alcanzando una reducción del 86.29 % con 16 hebras respecto a la ejecución con una sola hebra.

La mayor mejora relativa se observa al pasar de 1 a 2 hebras (-59.45 %) y de 2 a 4 hebras (-39.46 % adicional), lo que indica una excelente escalabilidad inicial.

Hebras	Tiempo (s)	Δ % vs 1 hebra
1.00	3308.08	0.00
2.00	1341.41	-59.45
4.00	816.06	-75.33
8.00	602.60	-81.78
16.00	453.44	-86.29

Tabla 5.14: Tiempos de ejecución y reducción porcentual respecto a una hebra en Docker sobre Windows (CPU).

A medida que se incrementa el número de hebras, la reducción en el tiempo de ejecución se mantiene, aunque con beneficios marginales decrecientes a partir de 8 hebras.

Estos resultados muestran que, en Docker sobre Windows (CPU), el uso de múltiples hebras es muy eficiente y permite aprovechar el paralelismo, siendo recomendable utilizar el mayor número de hebras posible para minimizar el tiempo de ejecución.

En la figura 5.18 se muestra el tiempo de ejecución para la configuración de CPU en un único nodo con Podman en Windows.

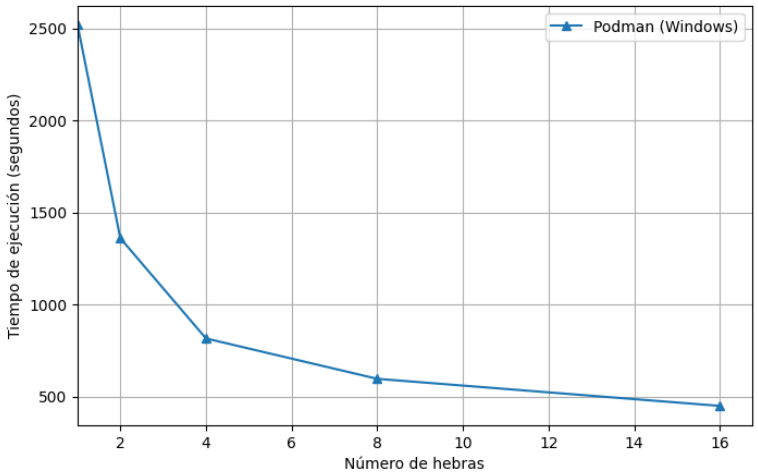

Figura 5.18: Tiempo de ejecución en un único nodo con Podman en Windows (CPU).

En la tabla 5.15 se presentan los tiempos de ejecución y la reducción porcentual respecto a una hebra.

El tiempo de ejecución disminuye notablemente al aumentar el número de hebras, alcanzando una reducción del 82.21 % con 16 hebras respecto a una sola hebra.

Hebras	Tiempo (s)	Δ % vs 1 hebra
1.00	2520.21	0.00
2.00	1359.69	-46.05
4.00	815.04	-67.66
8.00	595.54	-76.37
16.00	448.43	-82.21

Tabla 5.15: Tiempos de ejecución y reducción porcentual respecto a una hebra en Podman sobre Windows (CPU).

La mayor mejora relativa se observa al pasar de 1 a 2 hebras (-46.05 %) y de 2 a 4 hebras (-21.61 % adicional), lo que indica una buena escalabilidad inicial.

A partir de 8 hebras, la reducción en el tiempo de ejecución continúa, aunque los beneficios adicionales son menores, mostrando una tendencia a estabilizarse.

Estos resultados indican que, en Podman sobre Windows (CPU), el uso de múltiples hebras es eficiente y permite aprovechar el paralelismo, siendo recomendable utilizar el mayor número de hebras posible para reducir el tiempo de ejecución, aunque las ganancias adicionales disminuyen a partir de 8 hebras.

CPU + GPU

5.2.4. Contenedores en contenedores de Mac

CPU

En la figura 5.19 se muestra el tiempo de ejecución para la configuración de CPU en un único nodo con Docker en Mac.

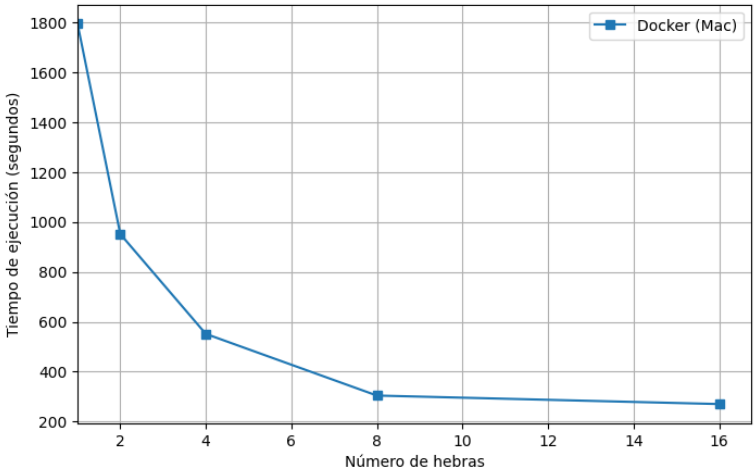

Figura 5.19: Tiempo de ejecución en un único nodo con Docker en Mac (CPU).

En la tabla 5.16 se presentan los tiempos de ejecución y la reducción porcentual respecto a una hebra.

Hebras	**Tiempo (s)**	**Δ % vs 1 hebra**
1.00	1794.98	0.00
2.00	951.56	-46.99
4.00	551.41	-69.28
8.00	304.35	-83.04
16.00	270.25	-84.94

Tabla 5.16: Tiempos de ejecución y reducción porcentual respecto a una hebra en Docker sobre Mac (CPU).

El tiempo de ejecución disminuye considerablemente al aumentar el número de hebras, alcanzando una reducción del 84.94 % con 16 hebras respecto a una sola hebra.

La mayor mejora relativa se observa al pasar de 1 a 2 hebras (-46.99 %) y de 2 a 4 hebras (-22.29 % adicional), lo que indica una buena escalabilidad inicial.

A partir de 8 hebras, la reducción en el tiempo de ejecución se estabiliza, con beneficios adicionales menores al incrementar a 16 hebras.

Estos resultados muestran que, en Docker sobre Mac (CPU), el uso de múltiples hebras es eficiente y permite aprovechar el paralelismo, siendo recomendable utilizar el mayor número de hebras posible para minimizar el tiempo de ejecución, aunque las ganancias adicionales disminuyen a partir de 8 hebras.

En la figura 5.20 se muestra el tiempo de ejecución para la configuración de CPU en un único nodo con Podman en Mac.

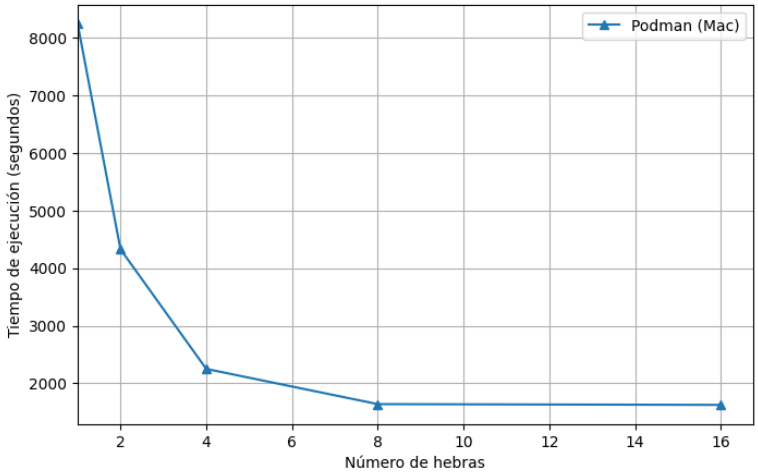

Figura 5.20: Tiempo de ejecución en un único nodo con Podman en Mac (CPU).

En la tabla 5.17 se presentan los tiempos de ejecución y la reducción porcentual respecto a una hebra.

Hebras	**Tiempo (s)**	**Δ % vs 1 hebra**
1.00	8247.00	0.00
2.00	4328.00	-47.52
4.00	2250.20	-72.71
8.00	1638.79	-80.13
16.00	1625.19	-80.29

Tabla 5.17: Tiempos de ejecución y reducción porcentual respecto a una hebra en Podman sobre Mac (CPU).

El tiempo de ejecución disminuye de forma muy significativa al aumentar el número de hebras, alcanzando una reducción del 80.29 % con 16 hebras respecto a una sola hebra.

La mayor mejora relativa se observa al pasar de 1 a 2 hebras (-47.52 %) y de 2 a 4 hebras (-25.19 % adicional), lo que indica una buena escalabilidad inicial.

A partir de 8 hebras, la reducción en el tiempo de ejecución se estabiliza, con beneficios adicionales mínimos al incrementar a 16 hebras (solo -0.16 % respecto a 8 hebras).

Estos resultados muestran que, en Podman sobre Mac (CPU), el uso de

múltiples hebras es eficiente hasta cierto punto, pero las ganancias adicionales más allá de 8 hebras son muy limitadas, sugiriendo que el paralelismo óptimo se alcanza alrededor de ese valor.

5.3. Pruebas multinodo

5.3.1. Ejecución en Ubuntu en nativo

CPU

CPU + GPU

5.3.2. Ejecución en contenedores de Ubuntu

CPU

CPU + GPU

5.3.3. Contenedores en contenedores de Windows

CPU

CPU + GPU

5.3.4. Contenedores en contenedores de Mac

CPU

5.4. Pruebas de barrido de hebras

5.4.1. Ejecución en Ubuntu en nativo

CPU

CPU + GPU

5.4.2. Ejecución en contenedores de Ubuntu

CPU

CPU + GPU

5.4.3. Contenedores en contenedores de Windows

CPU

CPU + GPU

5.4.4. Contenedores en contenedores de Mac

CPU

Capítulo 6

Conclusiones y trabajo futuro

[En este capítulo se presentan las conclusiones obtenidas al llevar a cabo el presente trabajo]

6.1. Contribuciones

[En esta sección se presentan las principales contribuciones del trabajo realizado.]

- Contribución 1 ...
- Contribución 2 ...

6.2. Retos y trabajo futuro

[Exponer aquí los retos y trabajos futuros.]

Bibliografía

[1] P. Sharma, L. Chaufournier, P. Shenoy, and Y. Tay, "Containers and virtual machines at scale: A comparative study," in *Proceedings of the 17th International Middleware Conference*, ACM. Trento, Italy: ACM, 2016, pp. 1–13.

[2] D. McFarland and J. Wolpaw, "Eeg-based brain-computer interfaces," *Current Opinion in Biomedical Engineering*, vol. 4, pp. 194–200, 2017, author manuscript; available in PMC 2018 December 01. Published in final edited form as: Curr Opin Biomed Eng. 2017 December ; 4: 194–200.

[3] F. Lotte, L. Bougrain, and M. Clerc, "Electroencephalography (eeg)-based brain–computer interfaces," 2015. [Online]. Available: https://api.semanticscholar.org/CorpusID:60672758

[4] I. M. Stefanyshyn and O. A. Pastukh, "High-performance computing for machine learning and artificial intelligence in brain-computer interfaces with big data," *COMPUTER-INTEGRATED TECHNOLOGIES: EDUCATION, SCIENCE, PRODUCTION*, 2025. [Online]. Available: https://api.semanticscholar.org/CorpusID:279622759

[5] P. Saha, A. Beltre, P. Uminski, and M. Govindaraju, "Evaluation of docker containers for scientific workloads in the cloud," in *Proceedings of the 2016 IEEE 8th International Conference on Cloud Computing Technology and Science (CloudCom)*, IEEE. Luxembourg City, Luxembourg: IEEE, 2016, pp. 797–802.

[6] G. R. Alles, A. da Silva Carissimi, and L. M. Schnorr, "Assessing the computation and communication overhead of linux containers for hpc applications," *2018 Symposium on High Performance Computing Systems (WSCAD)*, pp. 116–123, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:195774456

[7] F. Medrano-Jaimes, J. E. Lozano-Rizk, S. Castañeda-Ávila, and R. Rivera-Rodríguez, "Use of containers for high-performance

computing," *Communications in Computer and Information Science*, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:59283860

[8] G. Sravanthi, B. Grace, and V. Kamakshamma, "A review of high performance computing," *IOSR Journal of Computer Engineering (IOSR-JCE)*, vol. 16, no. 1, Ver. VII, pp. 36–43, Feb 2014, e-ISSN: 2278-0661, p-ISSN: 2278-8727. Department of Computer Science, PBR Visvodaya Institute of Science And Technology, India. [Online]. Available: http://www.iosrjournals.org

[9] G. Bhatia, "The road to docker: A survey," *International Journal of Advanced Research in Computer Science*, vol. 8, pp. 83–87, 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:188398107

[10] H. Gantikow, "Rootless containers with podman for hpc," in *Lecture Notes in Computer Science*, 2020. [Online]. Available: https://doi.org/10.1007/978-3-030-59851-8_23

[11] L. Stephey, S. Canon, A. Gaur, D. Fulton, and A. J. Younge, "Scaling podman on perlmutter: Embracing a community-supported container ecosystem," in *2022 IEEE/ACM 4th International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*, IEEE/ACM. Berkeley, CA, USA: IEEE, 2022, laurie Stephey, Shane Canon, Aditi Gaur (now at Microsoft Azure), Daniel Fulton (NERSC, Lawrence Berkeley National Laboratory), Andrew J. Younge (Center for Computing Research, Sandia National Laboratories). [Online]. Available: https://doi.org/10.1109/CANOPIE-HPC56686.2022.00009

[12] N. Zhou, H. Zhou, and D. Hoppe, "Containerisation for high performance computing systems: Survey and prospects," *Journal of Software Engineering*, vol. X, no. X, May 2022, this is the authors' version.

[13] P. Vaillancourt, J. E. Coulter, R. Knepper, and B. Barker, "Self-scaling clusters and reproducible containers to enable scientific computing," *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–8, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:220128332

[14] J. J. Escobar, "Energy-efficient parallel and distributed multi-objective feature selection on heterogeneous architectures," Ph.D. dissertation, Universidad de Granada, 2020.

[15] J. J. Escobar, J. Ortega, A. F. Díaz, J. González, and M. Damas, "Time-energy analysis of multilevel parallelism in heterogeneous

clusters: the case of eeg classification in bci tasks," *The Journal of Supercomputing*, vol. 75, no. 7, pp. 3397–3425, 2019. [Online]. Available: https://doi.org/10.1007/s11227-019-02908-4