Escuela de Doctorado
y Estudios de Posgrado
Universidad de La Laguna

Trabajo de Fin de Máster

Computación de Altas Prestaciones en entornos contenerizados

High Performance Computing in containerized environments

Omar Patricio Pérez Znakar

La Laguna, 15 de marzo de 2021

D. **Vicente José Blanco Pérez**, con N.I.F. 42171808C profesor Titular de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor.

D. **Francisco Carmelo Almeida Rodriguez**, con N.I.F. 42831571M Catedrático de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como cotutor

C E R T I F I C A (N)

Que la presente memoria titulada:

”*Computación de Altas Prestaciones en entornos contenerizados*”

Ha sido realizada bajo su dirección por D. **Omar Patricio Pérez Znakar**, con N.I.F. 79062976Q.

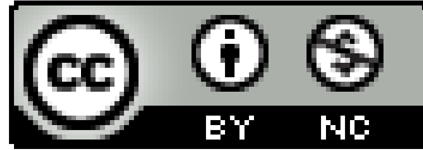
Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 15 de marzo de 2021

Agradecimientos

En este documento me gustaría hacer mención a mis dos tutores Vicente Blanco y Francisco Almeida, sin los cuales no habría podido realizar este TFM. Quiero agradecer su apoyo, comprensión , colaboración, dedicación y, sobre todo, la confianza que han depositado en mi a lo largo del curso. Asimismo, me gustaría destacar que me han obsequiado con una disponibilidad impecable, lo que me ha permitido poder aumentar mi confianza con respecto al proyecto que abarcamos hoy y, para proyectos futuros.

Por otra parte, no podría terminar este apartado sin dedicar unas palabras de agradecimiento a mis padres, quienes me han apoyado a lo largo de toda mi vida y, quienes han dispuesto a mi alcance todos los recursos que he necesitado tanto para la realización del grado, como para el estudio de este máster.

Licencia



© Esta obra está bajo una licencia de Creative Commons Reconocimiento 4.0 Internacional.

Resumen

En la actualidad los despliegues con Kubernetes han tenido un gran impacto en las aplicaciones web basadas en arquitecturas de microservicios. Su potencialidad más destacable es la capacidad para escalar y administrar los servicios asociados a estas aplicaciones que se ejecutan en la nube, normalmente con tecnología basada en contenedores. Se dispone de una gran variedad de herramientas y entornos que posibilitan mejoras de los despliegues con Kubernetes. Sin embargo, uno de los inconvenientes que surgen en este ámbito es que, no se conoce de manera precisa qué software posee las características idóneas para proporcionar un mejor rendimiento en problemáticas específicas o, si en su defecto, es mejor la ejecución sobre un sistema sin contenerización. Este proyecto se plantea el objetivo de desarrollar una comparativa de entornos de ejecución con y sin contenedores con el fin de analizar si Kubernetes es una alternativa viable frente a la ejecución en un servidor sin contenerización para cargas de trabajo en computación de altas prestaciones.

Con el fin de llevar a cabo este objetivo, se ha creado para este proyecto la aplicación ”GraphCloud” que, consiste en un servicio de cómputo que permite poder lanzar aplicaciones de manera remota desde clientes autorizados. Este aplicativo, se encuentra bajo el stack de desarrollo denominado ”MEVN” (Mongodb, Express, Vue.js y node.js) permitiendo, además, aplicar diversas tecnologías en el ámbito de la Computación de Altas Prestaciones. Al mismo tiempo, debemos destacar que se va a realizar un exhaustivo análisis de rendimiento sobre algoritmos de Machine Learning y filtro Gaussiano sobre imágenes, ejecutando cada uno tanto de forma secuencial como de forma paralela.

El proyecto se ha desarrollado haciendo uso de la tecnología en tres sistemas, en un ordenador personal (procesador con 4 cores), en el IaaS de la universidad de la Laguna (correspondiente a una estructura de máquinas virtualizadas de un core) y, posteriormente, en el servidor privado (Verode) del grupo de investigación ”Computación de Altas Prestaciones” (servidor con 40 cores integrados). El análisis de rendimiento que hemos realizado se centra en este últimos sistema.

Palabras clave: Kubernetes, Docker Hub, Quay, Machine Learning, Filtro Gaussiano, Python, C, MPI, OpenMP, Computación de Altas Prestaciones.

Abstract

Today Kubernetes deployments have had a major impact on web applications based on microservices architectures. Its most notable potential is the ability to scale and manage the services associated with these applications that run in the cloud, usually with container based technology. A wide variety of tools and environments are available that enable enhancements to Kubernetes deployments. However, one of the drawbacks that arise in this area is that it is not precisely known which software has the ideal characteristics to provide better performance in specific problems or, if not, the execution is better on a system without containerization. The objective of this project is to develop a comparison of runtime environments with and without containers in order to analyze whether Kubernetes is a viable alternative to running on a server without containerization for High Performance Computing (HPC) workloads.

In order to carry out this objective, the "GraphCloud " application has been created for this project, which consists of a computer service that allows applications to be launched remotely from authorized clients. This application is under the development stack called " MEVN " (Mongodb, Express, Vue.js and node.js) allowing, in addition, to apply various technologies in the field of High Performance Computing. At the same time, we must emphasize that an exhaustive performance analysis will be carried out on Machine Learning algorithms and Gaussian filter on images, executing each one both sequentially and in parallel.

The project has been developed using technology in three systems, in a personal computer (processor with 4 cores), in the IaaS of the University of La Laguna (corresponding to a structure of virtualized machines of a core) and, later, in the private server (Verode) of the research group "High Performance Computing"(server with 40 integrated cores). The performance analysis we have conducted focuses on the latter system.

Keywords: Kubernetes, Docker Hub, Quay, Machine Learning, Gaussian Filter, Python, C, MPI, OpenMP, High Performance Computing.

Índice general

1. Introducción **1**

1.1. Descripción y objetivos . . . 1

1.2. Justificación del proyecto . . . 3

1.3. Antecedentes y Estado Actual . . . 3

2. Herramientas y tecnologías de virtualizacón para HPC **4**

2.1. Docker . . . 4

2.2. Kubernetes . . . 5

2.2.1. Administradores de clúster de Kubernetes . . . 6

2.2.2. Kubectl . . . 6

2.2.3. Despliegue sencillo de prueba en Linux . . . 7

2.2.4. Despliegue sencillo de prueba en Windows . . . 11

2.3. Selección y justificación . . . 18

2.4. Tecnología basadas en contenedores para HPC . . . 18

2.4.1. Docker Hub . . . 18

2.4.2. Quay (Bioinformatics) . . . 19

3. GraphCloud: una aplicación para cómputo en la nube **21**

3.1. Tecnología y arquitectura utilizada . . . 21

3.2. Algoritmos de Scikit-Learn para Machine Learning . . . 24

3.2.1. Estructura general . . . 24

3.2.2. Estructura de ficheros . . . 25

3.2.3. Estructura de algoritmos . . . 26

3.2.4. Calsificación, Regresión y Clustering 27
3.2.5. Ejecución Paralela 31
3.3. Filtro Gaussiano sobre imágenes 32
3.3.1. Secuencial 32
3.3.2. MPI 34
3.3.3. OpenMP 37

4. Desplegando GraphCloud en un clúster de Kubernetes 40
4.1. Creando y registrando imágenes de contenedores con Docker 40
4.2. Diseñando la arquitectura de despliegue 42
4.3. Prueba de concepto en un despliegue local 43
4.3.1. Implementación de la configuración de contenedores 43
4.3.2. Despliegue de la prueba de concepto 46
4.4. Despliegue en el IAAS de la ULL 49
4.4.1. Despliegue 49
4.5. Despliegue en un nodo HPC 52
4.5.1. Docker Hub 52
4.5.2. Quay (Bioinformatics) 53

5. Análisis de rendimiento de algunos algoritmos en GraphCloud 57
5.1. Introducción 57
5.2. Random Forest Regressor. 58
5.2.1. Análisis de rendimiento para Random Forest Regression 59
5.2.2. Comparativa de los entornos de ejecución 61
5.3. Algoritmos de filtro Gaussiano sobre imágenes. 63
5.3.1. Análisis de rendimiento de la implementación MPI 63
5.3.2. Análisis de rendimiento de la implementación OpenMP 65
5.3.3. Comparativa de los entornos de ejecución 67

6. Conclusiones y líneas futuras. 71
6.1. Conclusiones 71

6.2. Líneas futuras 72

7. Summary and Conclusions. 73

7.1. Conclusions 73

7.2. Future Work 74

8. Presupuesto 75

8.1. Justificación del presupuesto 75

Índice de Figuras

2.1. Salida comando ”minikube status” 8
2.2. Salida comando ”minikube dashboard” 8
2.3. Salida comando ”kubectl get pods” 9
2.4. Salida comando ”kubectl get service” 10
2.5. Despliegue programa ”Hello Kubernetes” 11
2.6. Salida comando ”kubectl get service” 11
2.7. Distribución de Linux en tienda de Windows 12
2.8. Windows Terminal en tienda de Windows 12
2.9. Comprobación despliegue de Kubernetes en Windows 13
2.10. Versiones de los paquetes ”Kubectl” y ”Docker” 14
2.11. Crear clúster mediante el paquete denominado ”Kind” 15
2.12. Ejecución del comando ”Kubectl cluster-info” 15
2.13. Ejecución del comando ”Kubectl get nodes” 16
2.14. Servicios creados en Kubernetes 16
2.15. Comprobación despliegue con Kind 17
2.16. Información adicional en Docker para Windows 17
2.17. Repositorio de Hello-Kubernetes en Docker Hub 19

3.1. Diagrama de la Arquitectura de GraphCloud 22
3.2. Ejemplo de ejecución de el algoritmo Random Forest Regressor . . . 24
3.3. Patrón de diseño para los algoritmos de Machine Learning 25
3.4. Estructura de ficheros de la sección de código Python 25
3.5. Estructura de los algoritmos del código Python 26

3.6. Algoritmos de Clasificación implementados 27
3.7. Resultados obtenidos de aplicar los algoritmos de Clasificación sobre el Conjunto de Datos ”The Iris Dataset”. 28
3.8. Algoritmos de Regresión implementados 29
3.9. Representación de los algoritmos de Regresión implementados . . . 29
3.10Algoritmos Clustering implementados 30
3.11Algoritmos de Clustering implementados 31
3.12Comprobación ejecución único Core en código secuencial 31
3.13Comprobación ejecución multi-core en código paralelo (Joblib) . . . 32
3.14Ejemplo de ejecución del código secuencial 34
3.15Ejemplo de ejecución de MPI con kernel 10x10 36
3.16Ejemplo de ejecución de MPI con kernel 5x5 37
3.17Comprobación hilos en ejecución del código MPI mediante HTOP . . 37
3.18Comprobación hilos en ejecución del código OpenMP 38
3.19Ejemplo de ejecución de OpenMP con kernel 5x5 39
3.20Ejemplo de ejecución de OpenMP con kernel 10x10 39

4.1. Comprobación Imagen Frontend en Docker Hub 42
4.2. Comprobación Imagen Backend en Docker Hub 42
4.3. Estructura de despliegue de Kubernetes en el IAAS 43
4.4. Comprobación de Pods en despliegue Kubernetes en local 47
4.5. Comprobación del Ingress Controller en despliegue Kubernetes local 47
4.6. Comprobación despliegue Kubernetes local 48
4.7. Comprobación logs Kubernetes local 48
4.8. Comprobación del funcionamiento del Ingress Controller 51
4.9. Comprobación de despliegue en el IAAS 51
4.10Subiendo las imágenes de los contenedores a Quay 54

5.1. Tiempos de Random Forest Regressor junto con Docker Hub, Quay.io y Slurm. 60

5.2. Random Forest Regressor. Comparativa de los tres entornos de ejecución 62
5.3. Random Forest Regressor. Speedup y eficiencia 63
5.4. Filtro Gaussiano sobre una imagen. Implementación MPI. Tiempos de ejecución 64
5.5. Filtro Gaussiano sobre una imagen. Implementación OpenMP. Tiempos de ejecución 66
5.6. Filtro Gaussiano sobre una imagen. Implementación MPI. Comparativa 67
5.7. Filtro Gaussiano sobre una imagen. Implementación OpenMP. Comparativa 68
5.8. Filtro Gausiano sobre una imágen. Implmentación MPI. Speedup y eficiencia 69
5.9. Filtro Gausiano sobre una imágen. Implmentación OpenMP. Speedup y eficiencia 70

Índice de Tablas

5.1. Datos de entrada (TomeCano) para el análisis de rendimiento del Random Forest Regression (RFR) 59

8.1. Presupuesto 75

Capítulo 1

Introducción

La utilización de tecnologías de virtualización y contenerización en el campo de la computación de altas prestaciones, ha supuesto un cambio de paradigma en la explotación de los supercomputadores. Una tecnología que surge en el contexto de la computación web y la nube, que se ha adaptado a este contexto de forma natural. No en vano, los centros de procesamiento de datos para aplicaciones web se asemejan a un supercomputador. El hecho de poder aislar la configuración y dependencias de software de una aplicación en un contenedor, facilita enormemente la explotación de estos sistemas. La instalación de software y aplicaciones se hace más modular y es automatizable, lo que permite simplificar el mantenimiento de los supercomputadores.

En este capítulo, abordaremos la exploración sobre tecnologías de virtualización y contenerización que, puedan ser utilizadas como entornos de ejecución para aplicaciones de computación de altas prestaciones (High Performance Computing o HPC, en inglés). Analizaremos las distintas herramientas disponibles y comprobaremos su idoneidad a la hora de ser integradas en un clúster de cómputo sobre HPC.

1.1. Descripción y objetivos

En la actualidad existen diversas tecnologías que nos permiten poder contenerizar aplicaciones, normalmente aplicaciones web. Nos centraremos en aquellas herramientas y tecnologías que podamos aplicar a un clúster de computación de altas prestaciones. Aprovecharemos las ventajas que ofrecen los desarrollos web en cuanto a accesibilidad, a cómputo en remoto (para poder ejecutar aplicaciones HPC desde un navegador) y, la flexibilidad de configuraciones, que permiten los entornos de orquestación de contenedores para aprovechar al máximo las capacidades de cómputo del clúster.

En este trabajo se propone hacer un análisis de todas estas tecnologías, estudiar las funcionalidades que cada una nos ofrece y realizar un análisis de rendimiento de aplicaciones HPC, que podamos ejecutar en el clúster utilizando estos entornos.

Sumado a la idea anterior, debemos también tener en cuenta las siguientes consideraciones, que podemos plantear como objetivos para este proyecto:

- **Sistemas operativos y virtualización**: si bien tenemos varias opciones de S.O. en el equipamiento de escritorio (Linux, Windows, IOs, . . .) los sistemas HPC, al igual que los clústeres de cómputo en la nube utilizan principalmente un S.O Linux, que puede estar virtualizado utilizando alguna tecnología de hipervisor para abstraer capas de hardware. Analizaremos el impacto que puede tener el uso de esta capa de virtualización.

- **Contenerización de aplicaciones HPC**: en paralelo a la idea de la virtualización del sistema operativo y la abstracción del hardware respecto a este, surge la idea de desacoplar las aplicaciones y sus dependencias de software del sistema operativo que las sustenta. Este concepto es lo que se conoce como "contenerizar" las aplicaciones, de forma que ese "contenedor" sea trasladable de un sistema a otro sin necesidad de más mantenimiento del software que el propio sistema operativo. Valoraremos en este aspecto, herramientas que nos permitan construir contenedores que puedan ejecutarse en un sistema HPC.

- **Herramientas de orquestación**: un clúster HPC normalmente está construido con nodos de cómputo (que pueden llegar a ser miles) e incorporan procesadores de varios núcleos (*multicore*). La planificación de la ejecución de aplicaciones HPC permite optimizar estos recursos en los supercomputadores, por ejemplo, mediante el uso de un sistema de gestión de colas que, permite la asignación de elementos de proceso según las necesidades que la aplicación requiera. Esta misma tarea se puede realizar con herramientas de orquestación de contenedores que habitualmente se utilizan en el ámbito del desarrollo web. Valoraremos las tecnologías disponibles y que puedan adaptarse a este contexto de computación de altas prestaciones.

- **Contenerización y selección de recursos**: uno de los aspectos importantes en la ejecución de aplicaciones HPC es la utilización del recurso computacional que mejor se adapte a la aplicación desde el punto de vista de su rendimiento. Los clústeres HPC actuales son heterogéneos. Disponen de diferentes tipos de procesadores: procesadores multinúcleo de alto rendimiento, aceleradores de cómputo tipo GPU o manycore, procesadores de bajo consumo, ... Seleccionar el adecuado para cada aplicación es clave para obtener un rendimiento óptimo. Los sistemas de gestión de colas proporcionan mecanismos que permiten seleccionar los elementos de proceso adecuados para la aplicación a ejecutar. Debemos reproducir estos mecanismos en el sistema de orquestación de contenedores, de forma que podamos "dirigir" la ejecución del contenedor al recurso computacional adecuado.

Para poder responder a todas estas consideraciones, tendremos que investigar las posibilidades de las tecnologías y herramientas disponibles, labor que describiremos a lo largo de este documento.

1.2. Justificación del proyecto

Actualmente las tecnologías de virtualización y contenerización permiten realizar la tarea de despliegue de aplicaciones mediante diversos mecanismos. Existen varios frameworks que implementan estas funcionalidades, elegir el más adecuado depende del tipo de algoritmo o aplicación que queramos desplegar y del hardware sobre el que se realice esta operación. No siempre las soluciones más populares son las más adecuadas, por lo que procede una evaluación de las diferentes herramientas disponibles e identificar aquellas con más potencial para estos casos de uso. Un factor determinante en el esquema de despliegue elegido será conseguir el mejor rendimiento posible para las aplicaciones desplegadas.

1.3. Antecedentes y Estado Actual

A largo de muchos años, ha existido un gran abanico de formas de desplegar una aplicación, cada una nos proporcionaba ciertas ventajas sobre las otras, pero también presentan algunos inconvenientes. En una primera etapa, se disponía de servidores que se configuraban de forma manual, esto generaba el hándicap de una configuración más lenta y, donde ante un fallo, no era posible volver hacia atrás, ergo, se debía comenzar el proceso nuevamente. De igual forma, el despliegue se caracterizaba por esa lentitud, era necesario el uso de máquinas potentes que se podían aprovechar al máximo y casi no tenían disponibilidad. Posteriormente, se introdujeron los contenedores, que intentaban resolver la problemática anterior, aportando mayor sencillez y mejor aprovechamiento, pero con un inconveniente, la baja disponibilidad. Poco después, se popularizaron estas nuevas tecnologías y, surgió la aparición Docker, la cual hacia posible el despliegue de contenedores de una forma mucho más sencilla. Finalmente, se comenzó a pensar en resolver algunos de los problemas anteriores y, es aquí, donde se proponen los orquestadores de contenedores, entre los que figuran Kubernetes o Docker Swarm. Estos, posibilitan el despliegue de muchos contenedores de una vez e intentan resolver la problemática de la alta disponibilidad, incorporan equilibrado de carga y generan la reconfiguración de peticiones ante el fallo de una máquina.

En la literatura reciente podemos encontrar algunos casos de uso donde se utilizan entornos de orquestación especializados en sistemas de cómputo HPC [1, 2, 3]. Estos nos relatan las mejoras que dispone Kubernetes frente a la contenerización de aplicativos relacionados con HPC (High Performance Computing).

En [1] Zhou et al describen la necesidad de incorporar un nuevo sistema denominado ”Torque-Operator” , que permite realizar una conexión entre un administrador de cargas de trabajo HPC (Torque) y el orquestador de contenedores (en este caso Kubernetes). En [2], López-Huguet nos muestra una nueva herramienta denominada ”HPC-connector” , que posibilita gestionar el ciclo de vida de un contenedor de Kubernetes. Por último, el trabajo de Takahashi et al [3], nos describe la información relacionada con un nuevo sistema de balanceo de carga enfocado en la mejora de rendimiento para contenedores basados en HPC.

Capítulo 2

Herramientas y tecnologías de virtualizacón para HPC

Esta sección recoge uno de los objetivos del proyecto, conocer, analizar y realizar estudios comparativos de herramientas para la administración de contenedores como son Minicube, Kind, Kuberspray y K3sup entre otras, desarrollando pruebas sobre diferentes sistemas operativos (Linux y Windows). Introduciremos previamente los conceptos de Docker y Kubernetes con el fin de establecer el contexto de trabajo.

2.1. Docker

Para describir Docker es necesario previamente introducir el concepto de contenedor [4, 5, 6]. Un contenedor hace referencia a una unidad básica que nos permite ejecutar una aplicación. Una unidad es un sistema operativo independiente del de la máquina que dispone del software mínimo necesario para que la ejecución de una aplicación (código,...) sea funcional. A diferencia de una máquina virtual que es un sistema operativo completo. A su vez, un contenedor es portable [7]. Si se produce un fallo en un contenedor es posible crear otro de forma automática con las mismas características iniciales, mientras que, en el caso de una máquina virtual debe ser creada de forma manual.

Docker [8, 9] es una tecnología que permite crear y usar contenedores Linux con mayor sencillez. A continuación describimos algunas de las ventajas de su uso:

- **Modularidad:** esta tecnología incorpora la capacidad de poder dividir un proyecto en varias secciones y que estas se comuniquen entre sí. Permite la creación microservicios [10] (pequeña sección de una aplicación).

- **Control de versiones y restauración:** Docker dispone de un mecanismo para el control de versiones con los diferentes cambios realizados en un contenedor específico, pudiendo volver a una versión antigua si se requiere.

- **Implementación rápida:** permite desarrollar el despliegue de una aplicación en cuestión de segundos. Este hecho constituye una importante ventaja respecto a un despliegue tradicional en el que se podrían tardar días.

Docker constituye, por tanto, una opción viable desde el punto de vista de la facilidad de despliegue y mantenimiento de una aplicación. Las siguientes secciones analizarán su viabilidad desde la perspectiva del rendimiento.

2.2. Kubernetes

Kubernetes [11, 12, 13, 14] es una plataforma orientada a la orquestación de contenedores. Mientras que Docker permite crear un nodo único, que puede alojar una herramienta de interés (página web, base de datos,...). Kubernetes [15], sin embargo, se orienta a la creación de uno o varios clústeres (conjunto de contenedores) de nodos únicos, lo que introduce algunas ventajas:

- **Escalabilidad y balanceo de cargas:** se trata de una de las ventajas más importantes de esta tecnología que, permite crear contenedores de forma sencilla y además, escalar aplicaciones de forma automática. Asimismo, es posible realizar un equilibrado de cargas entre los nodos existentes para mejorar la experiencia del usuario.
- **Aislamiento de procesos y aplicaciones:** permite aislar una aplicación de forma sencilla en un clúster.
- **Facilidad de despliegue:** Docker constituyó un cambio respecto a la facilidad para el despliegue de una aplicación, Kubernetes, por su parte, mejora estas facilidades estableciendo una capa superior con la que crear y usar contenedores de un manera más rápida, a través de un orquestador relativamente sencillo.
- **Alta disponibilidad:** esta tecnología dispone de una alta capacidad para mantener una aplicación desplegada. Si un nodo falla deja de enviarle peticiones e incluso admite configuraciones para la eliminación y creación del nodo si fuera necesario.
- **Posibilidad de desplegar tanto en nubes privadas como en públicas:** una de las ventajas más destacadas es que posibilita su uso tanto de forma local (un ordenador convencional) como en servidores, tanto de forma pública como privada.
- **Comunidad amplia:** Kubernetes es una tecnología que está en auge y dispone de una comunidad muy amplia. Grandes empresas del sector tecnológico (Microsoft, Google,...) usan esta tecnología en estos momentos.

Kubernetes se ha convertido, por tanto, en una de las opciones más viables a la hora de desplegar una aplicación o herramienta en un servidor, ya sea en la nube o privado.

2.2.1. Administradores de clúster de Kubernetes

Un administrador de clúster de Kubernetes [16, 17] nos permite usar Kubernetes con el mínimo esfuerzo. Este nos permite crear clústeres de forma sencilla y rápida. A continuación, podremos ver algunos de los tipos más usados:

- **Minikube [18, 19]:** es un método minimalista para la gestión de Kubernetes, se trata de una buena herramienta para iniciarse en este tipo de gestión, aunque no es la mejor herramienta desde el punto de vista de las funcionalidades que ofrece. Para su uso es necesario disponer de una herramienta de virtualización (Virtualbox,...) y de permisos de virtualización del procesador.
- **Kind [20, 21]:** esta herramienta se creó originalmente para probar Kubernetes, pero ha evolucionado y cada vez es más potente permitiendo la gestión. No necesita más herramientas para su uso ni permisos de virtualización. Es la opción recomendada en la página de Kubernetes para desplegar un clúster mediante Windows y WSL2 (Windows Server for Linux) [22].
- **Kubespray [23, 24]:** está orientada a la gestión de Kubernetes y está basada en Ansible [25], un sistema que permite automatizar las tareas a realizar. Para su correcto uso es necesario tener instalado Ansible y, además, configurar el acceso a los nodos mediante ssh. Proporciona algunas ventajas como la posibilidad de actualizar todos los nodos de forma sencilla y simultánea. Sin embargo, requiere la instalación de software adicional, que podría influir en el rendimiento de la máquina, y la adaptación a una nueva tecnología.
- **K3sup ('Ketchup') [26, 27]:** es un paquete para la gestión de Kubernetes que facilita automatizar, escalar y administrar un despliegue de forma sencilla. Permite la virtualización de los contenedores o que estos sean reales (no virtualizados), lo que proporciona ventajas en cuanto a rendimiento si se dispone de una granja de servidores.

En las siguientes secciones se desarrollarán algunas pruebas con el objetivo de seleccionar la que se mejor se adapta a este proyecto. Asimismo, este proceso se llevará a cabo teniendo en cuenta el rendimiento, escogiendo la herramienta que proporcione mejor rendimiento en nuestro aplicativo,

2.2.2. Kubectl

Kubectl [28, 29] es un paquete que nos permitirá utilizar comandos para usar Kubernetes. Posibilitando muchas ventajas como poder crear, eliminar y actualizar componentes. La forma de instalación es muy sencilla y la podremos ver a continuación:

- Realizamos un curl de la dirección que nos proporcionan mediante “curl -LO (url ultima versión)“

- Modificamos los permisos de ejecución mediante “chmod +x ./kubectl“.
- Movemos el binario al PATH de la máquina haciendo uso de ”sudo mv ./kubectl /usr/local/bin/kubectl”.

Una vez realizados estos pasos, podremos realizar diferentes comandos de interés, entre los que encontramos:

- **”kubectl cluster-info”:** nos proporciona la información del clúster.
- **”kubectl get nodes”:** aporta información sobre los nodos del clúster.
- **”kubectl get all –all-namespaces”**: nos surte información de los servicios del clúster.

2.2.3. Despliegue sencillo de prueba en Linux

A la hora de elaborar un despliegue mediante el uso de Kubernetes en Linux, se requerirá la realización de diversos pasos, estos se podrán ver citados a continuación:

- **Actualizar la distribución:** para ello se usarán los comandos ”Sudo apt update” y ”sudo apt upgrade -y”.
- **Descargamos Kubectl.**
- **Descargamos un administrador de Kubernetes:** en este caso, se utilizará uno sencillo, concretamente, usaremos el denominado ”Minikube”.
- **Descargamos un programa de virtualización de máquinas:** en nuestro caso, se ha instalado el denominado ”Virtual Box” (aplicación que permite crear máquinas virtuales).

Una vez se poseen los paquetes necesarios, deberemos de realizar los pasos que vienen a continuación:

1. Ejecutar el comando ”minikube start”. Este nos permitirá crear una estructura inicial para Kubernetes. De igual forma, cabe resaltar que, se le podría especificar las características de la máquina. Sin embargo, si no se le especifican (como es el caso) te crea una con 2 cores y 20 gb de memoria interna. Estas características son más que suficientes para el ejemplo que se requiere desarrollar.
2. Ejecutamos el comando ”minikube status”, cuya salida debe ser satisfactoria. En nuestro caso, se ha obtenido lo que podremos divisar en la figura 2.1

```
omar@omar-B85M-D3H:\$ minikube status
minikube
type: Control Plane
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured
```

Figura 2.1: Salida comando "minikube status"

3. Ejecutamos el comando "minikube dashboard". Este nos mostrará un dashboard con la información más importante de nuestro clúster de Kubernetes. En este caso, veremos algo similar a que podremos ver en la figura 2.2.

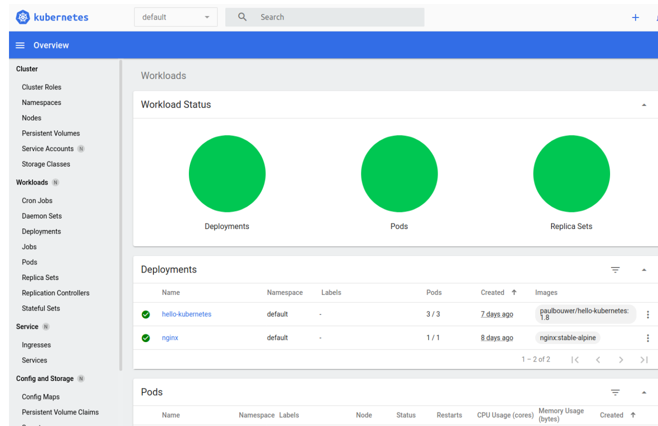


Figura 2.2: Salida comando "minikube dashboard"

4. Buscamos un ejemplo de prueba [30]. En el caso que nos ocupa, hemos cogido uno que nos permite la posibilidad de crear 3 pods (contenedor mínimo de Kubernetes) y, mediante el balanceador de carga, manda la petición a un pod aleatorio.

5. Se agrega la configuración, para ello, utilizaremos el comando "kubectl create -f hello-kubernetes-deployment.yaml". El contenido de este fichero lo podremos ver reflejado en el Listing 2.1

```yaml
# Versión de la API de Kubernetes a usar
apiVersion: apps/v1
# Clase de objeto a crear
kind: Deployment
metadata:
  # Nombre del Despliegue
  name: hello-kubernetes
spec:
  # Pods a generar con el mismo proyecto
  replicas: 3
  selector:
    matchLabels:
      app: hello-kubernetes
  template:
    metadata:
      labels:
        app: hello-kubernetes
    spec:
      containers:
      - name: hello-kubernetes
        # Imagen de Docker Hub
        image: paulbouwer/hello-kubernetes:1.8
        ports:
        - containerPort: 8080
```

Listing 2.1: Configuración del despliegue de Hello-Kubernetes

6. Se comprueba que todo se ha creado satisfactoriamente. Para ello, se hará uso del comando ”kubectl get pods” y, quedará reflejado a través de la figura 2.3:

```
omar@omar-B85M-D3H:\$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
hello-kubernetes-767d49787b-hkgmd   1/1     Running   1          8d
hello-kubernetes-767d49787b-tglcj   1/1     Running   1          8d
hello-kubernetes-767d49787b-zk89n   1/1     Running   1          8d
nginx-69c78cd8c6-txdpr              1/1     Running   1          8d
```

Figura 2.3: Salida comando ”kubectl get pods”

7. Creamos un servicio para poder mostrar el contenido de nuestro proyecto. Para ello, ejecutamos el comando ”kubectl apply -f hello-kubernetes-service.yaml”. El contenido de este fichero lo podremos ver reflejado en el Listing 2.2.

```yaml
apiVersion: v1 # API de Kubernetes a usar
kind: Service # Clase de objeto a crear
metadata:
  name: hello-kubernetes # Nombre del Despliegue
spec:
# Tipo de servicio se desea: (ClusterIP,..)
  type: LoadBalancer
  ports:
  - port: 80
    targetPort: 8080
  selector:
    app: hello-kubernetes
```

Listing 2.2: Configuración servicio del Hello-Kubernetes

8. Comprobamos que el servicio se ha creado correctamente. Para ello, ejecutamos el comando ”kubectl get service” (véase la figura 2.4):

```
omar@omar-B85M-D3H:\$ kubectl get service
NAME               TYPE          CLUSTER-IP      EXTERNAL-IP   PORT(S)
hello-kubernetes   LoadBalancer  10.104.225.189  <none>        80:31570/TCP
kubernetes         ClusterIP     10.96.0.1       <none>        443/TCP
nginx              LoadBalancer  10.100.94.60    <none>        80:30095/TCP
```

Figura 2.4: Salida comando ”kubectl get service”

Si se han realizado los pasos anteriores de forma satisfactoria, debemos de coger el puerto que nos proporciona el comando ”Kubectl get service” y agregarlo a una dirección url junto con la que nos devuelve el comando ”kubectl cluster-info”. Si lo hacemos correctamente, debemos ver lo que se representa en la figura 2.5.

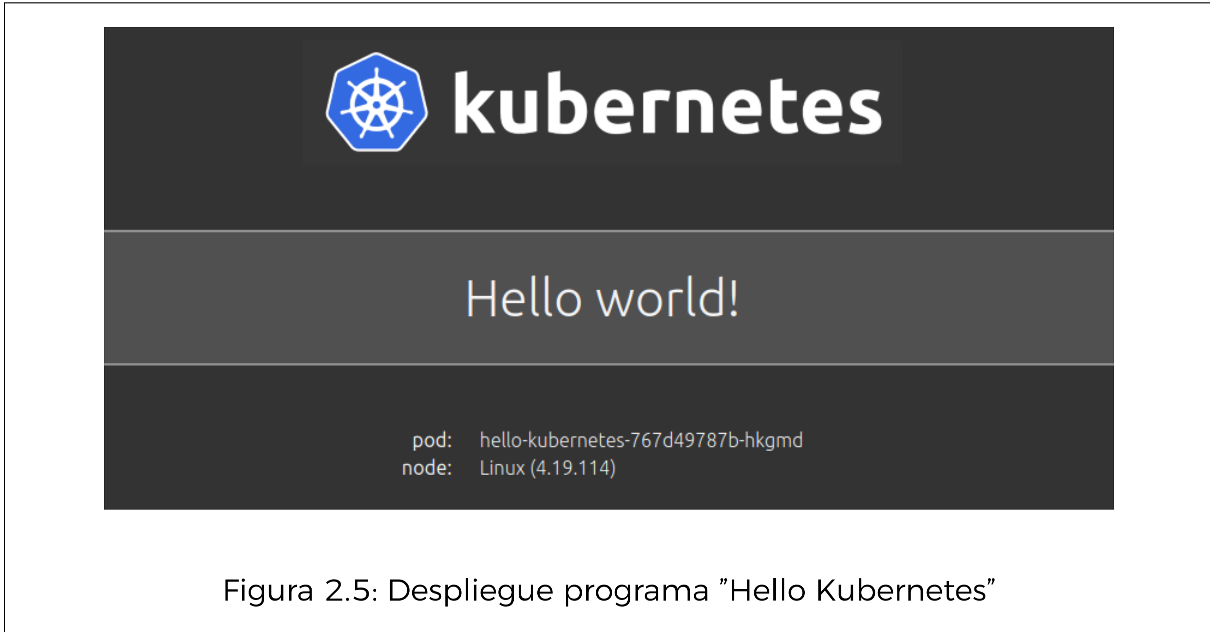


Figura 2.5: Despliegue programa "Hello Kubernetes"

Por último, si queremos que nuestro despliegue tenga dirección externa, podremos usar el comando "minikube tunnel" en una terminal nueva y, si nuestro tipo de servicio es "LoadBalancer" (nos permite proveer el servicio a un sistema de carga externo) se creará de forma automática. Cabe resaltar, que no se despliega hacia el exterior solo nos permitirá hacer una simulación de cómo se vería la aplicación desplegada. Un ejemplo de esto se podrá ver mediante la figura 2.6.

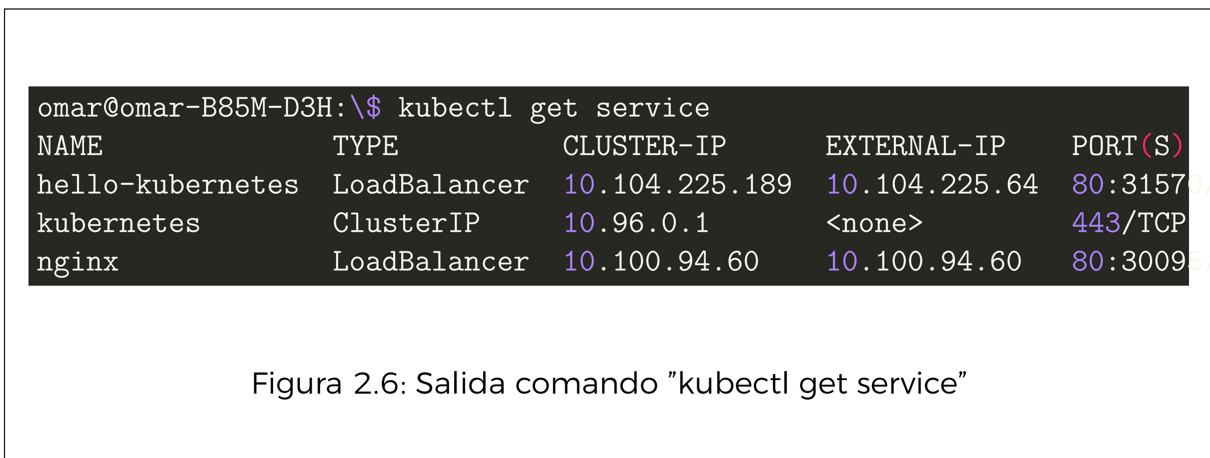


Figura 2.6: Salida comando "kubectl get service"

2.2.4. Despliegue sencillo de prueba en Windows

En este capítulo abordaremos el proceso en virtud del cual se realiza un sencillo despliegue en Windows. Para llevar a cabo un despliegue de Kubernetes dentro del sistema operativo de Windows [22] necesitaremos disponer de las ciertas características como las que se presentarán a continuación.

En prime lugar, debemos de tener descargado WSL [31, 32] (Windows Subsisten for Linux), concretamente, en su segunda versión. Esta herramienta nos permitirá poder tener un sistema Linux completo dentro de Windows. Para descárgalo seguiremos las directrices encontradas en el tutorial [33].

Asimismo, debemos Descargar una distribución de Linux de la tienda de Windows. Para ello, la buscamos, descargamos e instalamos. Se podría ver un ejemplo a través de la figura 2.7.

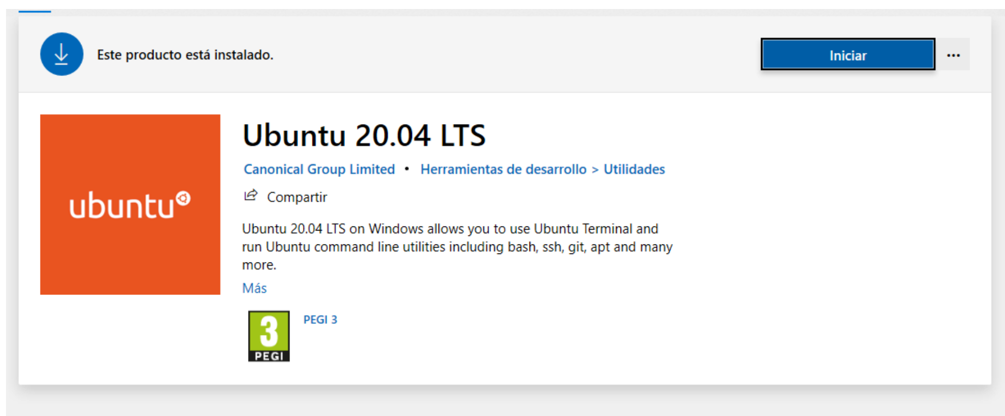


Figura 2.7: Distribución de Linux en tienda de Windows

Por otro lado, tendremos que Descargar la herramienta ”Windows Terminal” de la tienda. Esta nos ayudará a poder cambiar de terminal de Windows (WSL2, PowerShell,..) de forma sencilla. Esta extensión se aprecia en la figura 2.8.

Figura 2.8: Windows Terminal en tienda de Windows

En otra instancia, encontraremos la necesidad de Tener descargado Docker para Windows. Para ello accedemos a [34]

seguidamente, debemos de Autorizar a WSL2 el uso de Docker para Windows. Para ello, accedemos a ”Ajustes/Resources/WSL Integration” y clicamos en ”Enable integration with my default WSL distro” junto con la distribución de Linux que tengamos descargada. Una prueba de esto seria la presentación 2.9.

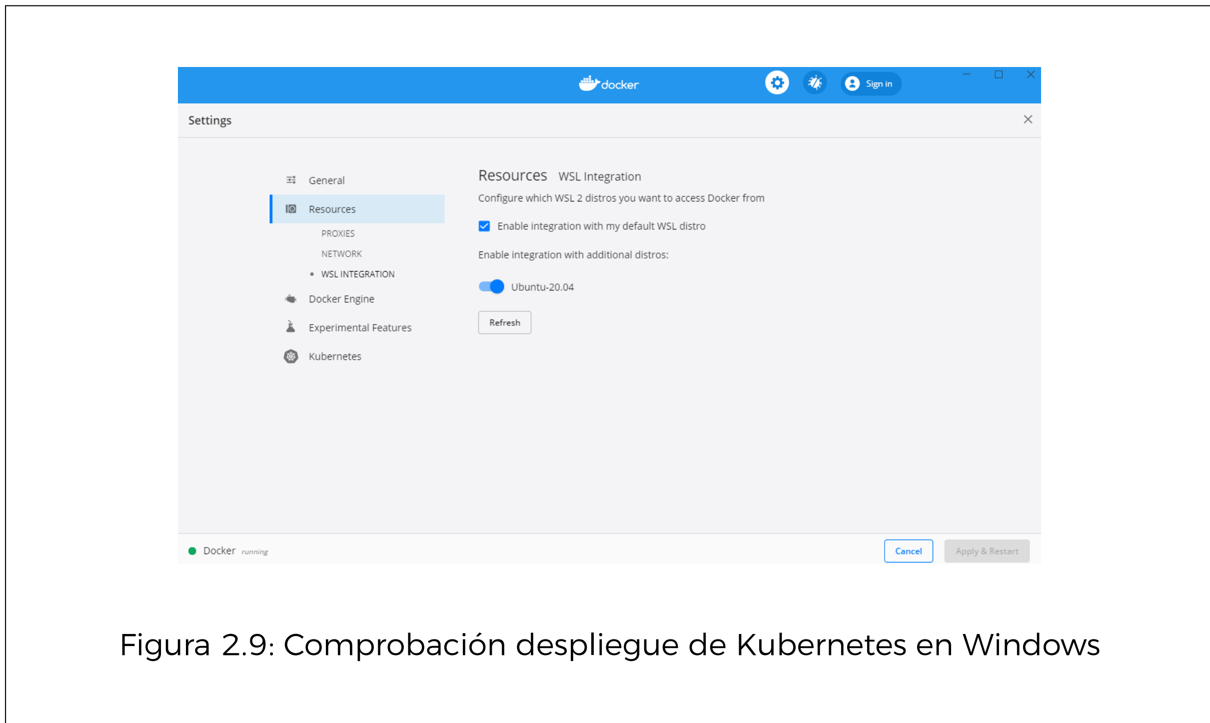


Figura 2.9: Comprobación despliegue de Kubernetes en Windows

Seguido a la ejecución de los pasos anteriores, solo debemos iniciar nuestra máquina de Linux desde la herramienta de ”Windows Terminal” y realizar los siguientes pasos:

1. Actualizar la distribución de Linux mediante los comandos ”sudo apt update” y ”sudo apt upgrade -y”.

2. Comprobar que tanto Docker como Kubectl están instalados mediante los comandos ”docker version” y ”kubectl version”. Aunque no lo tengamos instalados dentro de esta distribución, esta cogerá lo necesario de la instalación de Docker para Windows realizada con anterioridad (veáse la figura 2.10).

```
omar@DESKTOP-J8051TQ:/mnt/c/Users/Omar$ kubectl version
Client Version: version.Info{Major:"1", Minor:"19", GitVersion:"v1.19
.3",GitCommit:"1e11e4a2108024935ecfcb2912226cedeafd99df",
GitTreeState:"clean",BuildDate:"2020-10-14T12:50:19Z",
GoVersion:"go1.15.2", Compiler:"gc", Platform:"linux/amd64"}
Error from server (InternalError): an error
on the server ("") has prevented the request from succeeding

omar@DESKTOP-J8051TQ:/mnt/c/Users/Omar$ docker version
Client:
 Version:           19.03.8
 API version:       1.40
 Go version:        go1.13.8
 Git commit:        afacb8b7f0
 Built:             Wed Oct 14 19:43:43 2020
 OS/Arch:           linux/amd64
 Experimental:      false

Server: Docker Engine - Community
 Engine:
  Version:          19.03.13
  API version:      1.40 (minimum version 1.12)
  Go version:       go1.13.15
  Git commit:       4484c46d9d
  Built:            Wed Sep 16 17:07:04 2020
  OS/Arch:          linux/amd64
  Experimental:     false
```

Figura 2.10: Versiones de los paquetes "Kubectl" y "Docker"

3. Descargamos Kind mediante los comandos:
   - Realizamos un curl a la última versión existente mediante el comando "curl -Lo (url última versión)".
   - Damos permisos de ejecución mediante el domando "chmod +x ./kind".
   - Movemos el binario al path para facilitar su uso. Para ello, usaremos el comando "sudo mv ./kind /usr/local/bin/".
4. Creamos un cluster con Kind mediante el comando "kind create cluster –name (nombre cluster)". Gracias a la figura 2.11 podremos ver su representación.

```
omar@DESKTOP-J8051TQ:\$ kind create cluster --name wslkind
Creating cluster "wslkind" ...
 - Ensuring node image (kindest/node:v1.17.0)
 - Preparing nodes
 - Writing configuration
 - Starting control-plane
 - Installing CNI
 - Installing StorageClass
Set kubectl context to "kind-wslkind2"
You can now use your cluster with:

kubectl cluster-info --context kind-wslkind2

Have a nice day!
```

Figura 2.11: Crear clúster mediante el paquete denominado ”Kind”

5. Comprobamos que se ha creado correctamente mediante el comando ”kubectl cluster-info”(figura 2.12).

```
omar@DESKTOP-J8051TQ:\$ kubectl cluster-info
Kubernetes master is running at https://127.0.0.1:32770
KubeDNS is running at https://127.0.0.1:32770/api/v1/namespaces/
kube-system/services/kube-dns:dns/proxy
```

Figura 2.12: Ejecución del comando ”Kubectl cluster-info”

6. Miramos la cantidad de nodos desplegados mediante el comando ”kubectl get nodes” (figura 2.13).

```
 omar@DESKTOP-J8051TQ:\$ kubectl get nodes
NAME                       STATUS    ROLES     AGE      VERSION
wslkind-control-plane      Ready     master    6m17s    v1.17.0
```

Figura 2.13: Ejecución del comando ”Kubectl get nodes”

7. Miramos los servicios creados mediante el comando ”kubectl get all –all-namespaces” (figura 2.14).

```
omar@DESKTOP-J8051TQ:/mnt/c/Users/Omar$ kubectl get all --all-namespaces
NAMESPACE            NAME                                              READY   STATUS    RESTARTS
AGE
kube-system          pod/coredns-6955765f44-9w9j5                      1/1     Running   0
5m46s
kube-system          pod/coredns-6955765f44-w4q44                      1/1     Running   0
5m46s
kube-system          pod/etcd-wslkind-control-plane                    1/1     Running   0
5m42s
kube-system          pod/kindnet-7ts62                                 1/1     Running   0
5m46s
kube-system          pod/kube-apiserver-wslkind-control-plane          1/1     Running   0
5m42s
kube-system          pod/kube-controller-manager-wslkind-control-plane 1/1     Running   0
5m42s
kube-system          pod/kube-proxy-p6hlt                              1/1     Running   0
5m46s
kube-system          pod/kube-scheduler-wslkind-control-plane          1/1     Running   0
5m41s
local-path-storage   pod/local-path-provisioner-7745554f7f-cv5c8       1/1     Running   0
5m46s

NAMESPACE     NAME                   TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)                   AGE
default       service/kubernetes     ClusterIP   10.96.0.1     <none>        443/TCP                   6m5
s
kube-system   service/kube-dns       ClusterIP   10.96.0.10    <none>        53/UDP,53/TCP,9153/TCP    6m2
s

NAMESPACE     NAME                          DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SE
LECTOR                 AGE
kube-system   daemonset.apps/kindnet        1         1         1       1            1           <none>
                       5m57s
kube-system   daemonset.apps/kube-proxy     1         1         1       1            1           beta.ku
bernetes.io/os=linux   6m2s

NAMESPACE            NAME                                     READY   UP-TO-DATE   AVAILABLE   AGE
kube-system          deployment.apps/coredns                  2/2     2            2           6m2s
local-path-storage   deployment.apps/local-path-provisioner   1/1     1            1           5m57s

NAMESPACE            NAME                                                DESIRED   CURRENT   READY   A
GE
kube-system          replicaset.apps/coredns-6955765f44                  2         2         2       5
m46s
local-path-storage   replicaset.apps/local-path-provisioner-7745554f7f   1         1         1       5
m46s
```

Figura 2.14: Servicios creados en Kubernetes

Una vez se han llevado a cabo los pasos anteriores, se podrá comprobar que se ha desplegado correctamente la aplicación de prueba. Para ello, cogemos la url que nos proporciona el comando "kubecyl cluster-info" y se pegará en el buscador. Debería de salir lo que se refleja en la figura 2.15:

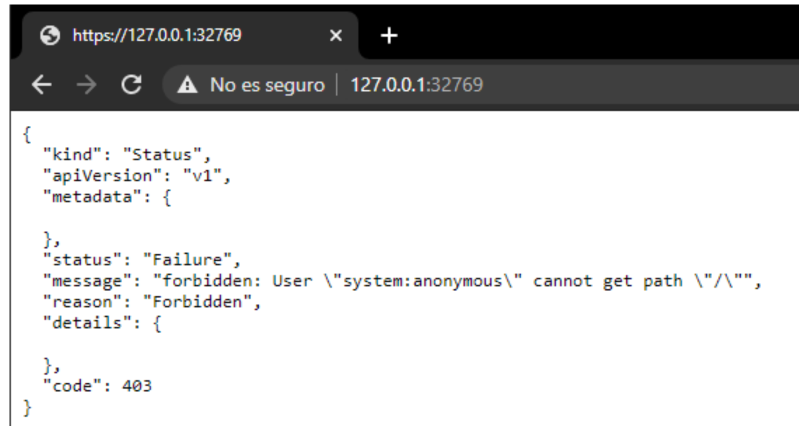


Figura 2.15: Comprobación despliegue con Kind

Si accedemos a la herramienta de Docker para Windows veremos que nuestro clúster se ha creado, junto a esta información encontraremos información sobre los clústers, pods, logs, estadísticas,... Una prueba de ello, la podremos ver en la figura 2.16.

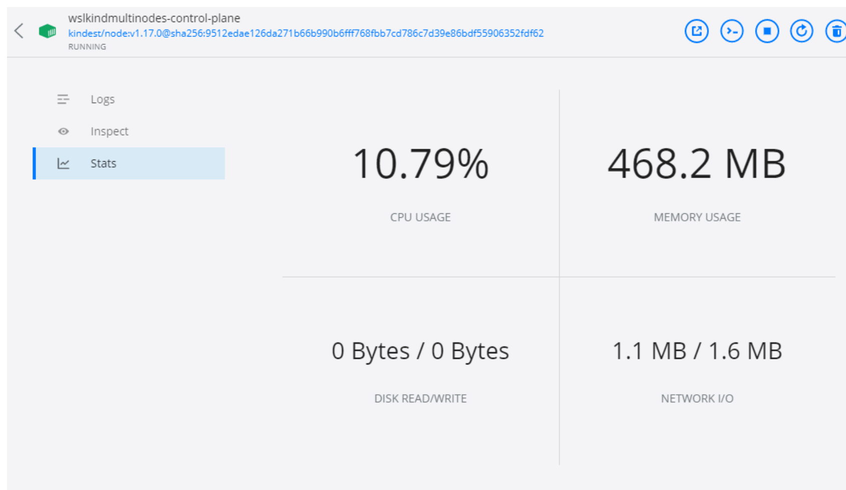


Figura 2.16: Información adicional en Docker para Windows

2.3. Selección y justificación

El objeto de este capítulo se sustenta en gran parte de las herramientas que hemos visto en apartados anteriores, debido a que, gracias a la investigación exhaustiva de diversas tecnologías y, su respectivo comportamiento, se han podido extraer las conclusiones oportunas para este apartado, concretamente, hemos podido concluir que, para el tipo de investigación que nos concierna, vamos a utilizar las siguientes tecnologías:

- **Kuberntes:** constituye la herramienta por excelencia que nos va a poder dar más posibilidad de rendimiento a la hora de desplegar una aplicación . De igual forma, otro motivo de elección es la gran comunidad que tiene detrás.
- **Linux:** a pesar de que se puede realizar en Windows, Linux es un sistema operativo más liviano, el cual podríamos configurar para que lo sea aún más. Esta ventaja nos proporciona no solo que pese menos, sino que tengamos menos recursos en segundo plano y, por tanto, que nuestro ordenador se centre más en las tareas encomendadas.
- **K3sup ('ketchup') como administrador de Kubernetes:** esta herramienta es la más idónea a la hora de desplegar un clúster en producción para nuestro tipo de ejercicio. Nosotros buscamos sacar el mayor rendimiento de una máquina y, la posibilidad de usarla sin virtualización provoca que nos decantemos con esta tecnología.

Como se puede apreciar, la investigación se a orientado hacia aquellas tecnologías que nos dan mayor facilidad, pero sobretodo, por aquellas que nos permitan sacar una mayor rendimiento con el menor consumo posible,ergo, las que nos aporten un rendimiento más óptimo.

2.4. Tecnología basadas en contenedores para HPC

El presente capítulo constará de las tecnologías basadas en contenedores Docker para Kubernetes, concretamente nos centraremos en Docker Hub y Quay.io. Estas herramientas nos posibilitan desplegar software (páginas web, bases de datos,...) tanto para uso cotidiano, como para empresas como para investigaciones, concretamente, en contenedores para HPC entre las existentes.

2.4.1. Docker Hub

El ápice de este apartado será Docker Hub [35, 36, 37], haciendo referencia a un repositorio de imágenes de Docker, en donde podremos encontrar diversos repositorios sobre diversos proyectos ya realizados. De igual forma, no solo nos tenemos que adaptar a lo que existe, sino que, nosotros también podemos publicar nuestras propias imágenes. Como se ha visto, hacemos referencia a la siguiente imagen de Docker Hub 2.17.

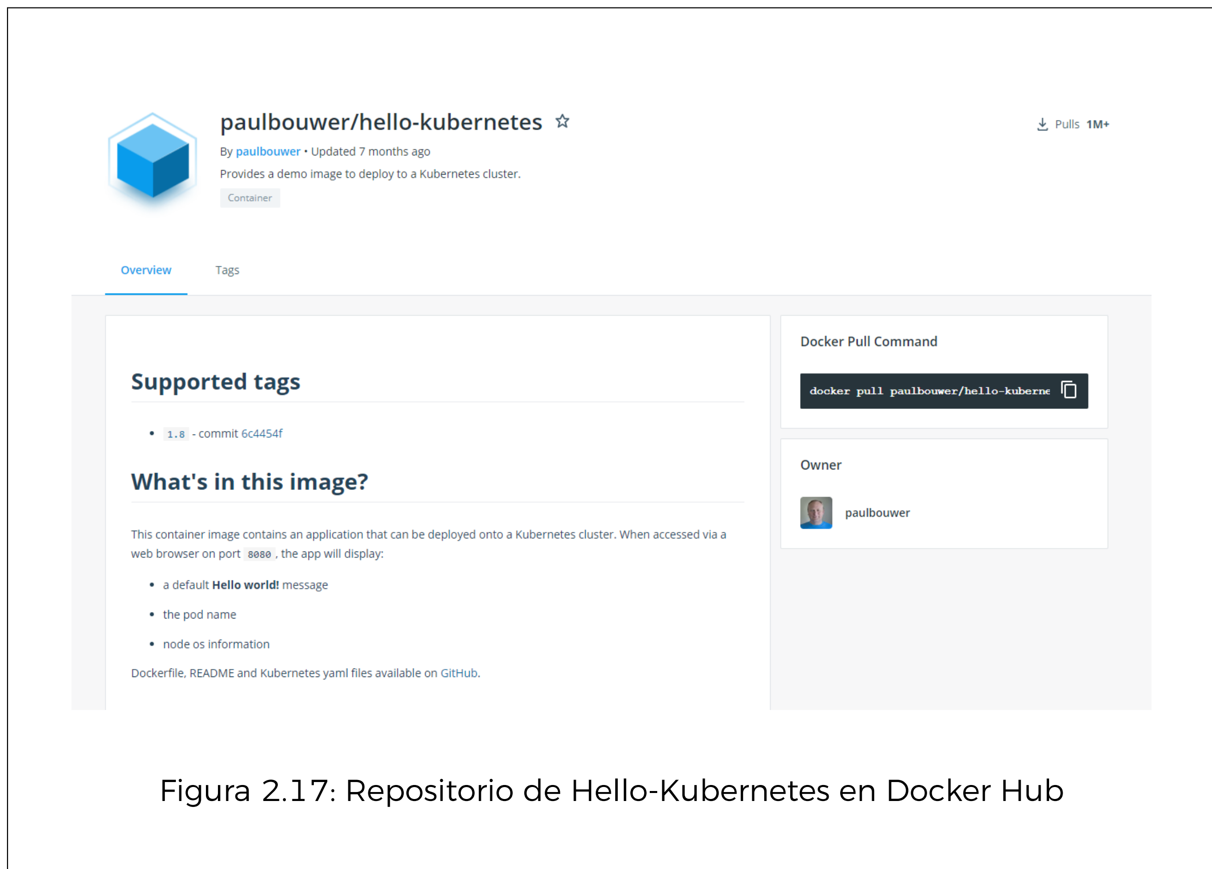


Figura 2.17: Repositorio de Hello-Kubernetes en Docker Hub

2.4.2. Quay (Bioinformatics)

En relación a esta sección, describiremos el funcionamiento de Quay [38, 39, 40] entendiendo este como un registro para imágenes Docker altamente enfocado en las empresas. Funciona de forma muy similar a Docker Hub, es decir, nos permite almacenar, diseñar, distribuir e implementar imágenes Docker. Sin embargo, dispone de diversas características que lo hacen más interesante. Algunas de ellas, las podremos ver a continuación:

- **Gestión de control de acceso:** nos permite controlar quien tiene permisos para entrar a nuestras implementaciones de forma avanzada. Con esto, conseguimos que se dispongan de diversos proveedores de autenticación y un sistema de permisos detallado. Esto nos permitirá poder dictaminar quién puede acceder a unas características y quién a otras.
- **Cifrado TSL:** este nos brinda la capacidad de aumentar la seguridad de nuestro contenido permitiendo mayor tranquilidad para los usuarios.
- **Escaneo de seguridad:** de forma automática esta herramienta detecta vulnerabilidades y las comunica al usuario.
- **Integración con software de automatización de tareas:** esta herramienta

permite el acceso a aplicaciones de integración continua para mejorar la experiencia de uso de los programadores.

- **Restauración de imágenes:** nos permite hacer un seguimiento de las diversas versiones implementadas y, si se requiere, poder volver a una versión en específico.
- **Registro y auditoría:** facilita el uso de las auditorías para la integración continua.

Como se ha podido comprobar a lo largo de este sección, esta herramienta nos brinda diversas ventajas con respecto a otros repositorios de imágenes, no obstante, su desventaja se halla en el coste de su uso. Este es superior al de otras plataformas, por tanto, es importante saber si estas características nos brindan una ventaja equivalente al precio que nos proponen.

Capítulo 3

GraphCloud: una aplicación para cómputo en la nube

En este capítulo presentamos GraphCloud, una aplicación web (desarrollada en este proyecto) con el objetivo de facilitar un servicio de cómputo. Posibilita la incorporación de nuevos algoritmos a la plataforma y que puedan ser ejecutados como un servicio de manera remota desde clientes autorizados. Este aplicativo, responde a una arquitectura cliente-servidor implementada siguiendo la tecnología "MEVN" (MongoDB, Express, Vue y Node). En estos momentos la plataforma incorpora algoritmos secuenciales y paralelos para la aplicación de Machine Learning y para el procesamiento de imágenes.

3.1. Tecnología y arquitectura utilizada

GraphCloud es una aplicación web que nos permite, de manera remota, lanzar la ejecución de aplicaciones desde clientes autorizados. Las aplicaciones se incorporan al servidor como servicios de cómputo que se ofertan a los clientes. Esta infraestructura ha sido desarrollada bajo el stack denominado "MEVN" que responde a una estructura conformada con MongoDB, Express, Vue y Node.

MongoDB [41, 42] es una base de datos no relacional que permite almacenar documentos, este almacenamiento es posible gracias a BJSON que consiste en una representación binaria de JSON. **Express** [43, 44] puede ser definido como un framework que permite la creación de una API REST de forma sencilla. **Vue** [45, 46] es un framework basado en JavaScript que posibilita crear una interfaz de usuario. Por último, **Node** [47, 48], representa una tecnología que permite ejecutar JavaScript del lado del servidor.

GraphCloud ha sido construido siguiendo una arquitectura cliente-servidor en la que hay dos módulos independientes, el módulo cliente y el módulo servidor. El cliente conforma la sección que será visible para el usuario incluyendo la interfaz gráfica con la que interactúa. Para la creación del módulo cliente se ha hecho uso de diversas herramientas como: HTML, CSS, Javascript, Boostrap o Vue.js. El

módulo servidor constituye la sección con la que comunica el cliente para ejecutar las aplicaciones disponibles en el servidor. Para construir el servidor se han utilizado tecnologías como: Node.js, Multer, Express o MongoDB. Este diseño permite la instalación de los módulos en máquinas diferentes que estén interconectadas.

GraphCloud permitiría la integración de distintos módulos cliente y servidor que podrían estar instalados en máquinas diferentes. En la figura 3.1 podremos ver el diagrama de la arquitectura de GraphCloud.

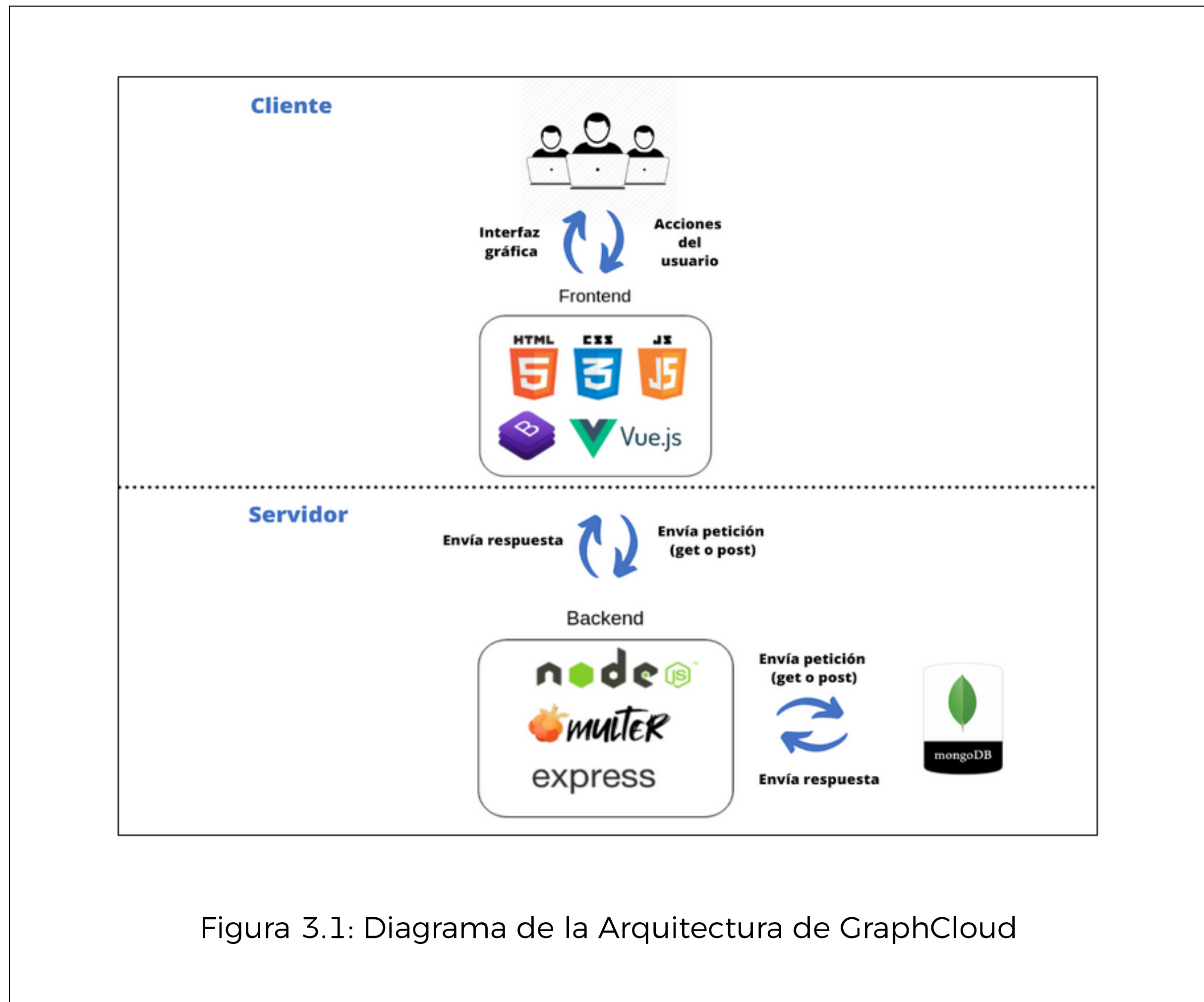


Figura 3.1: Diagrama de la Arquitectura de GraphCloud

Destacar que este aplicativo permite incorporar nuevos servicios de manera dinámica, es decir, es posible agregar nuevos algoritmos de forma sencilla y automática (sin modificar ninguna sección de código). Basta con seguir los siguientes pasos:

- Modificar el archivo de configuración que permite editar la interfaz gráfica. En el listing 3.1 puede verse un ejemplo de este archivo de configuración.

Listing 3.1: Archivo de configuración para la interfaz gráfica

```
{ "Methods":[
        {
            "Name":"MPI",
            "file":"image.png",
            "Description":"Método que permite aplicar un filtrado
                a una imagen a través de un Algoritmo usando la
              tecnología MPI. Este filtro permite poder
              enborronar una imagen dada previamente.",
            "Elements":[
              {
                  "Name":"height",
                  "value":"630",
                  "DescriptionShort":"Altura de la Imagen.",
                  "DescriptionLong":"Valor numérico que describe
                     la altura de la imagen. Con estp, provocamos
                      que nuestro algoritmos aumente su velocidad
                      en gran medida."
              },
              {
                  "Name":"width",
                  "value":"1200",
                  "DescriptionShort":"Ancho de la Imagen.",
                  "DescriptionLong":"Valor numérico que describe
                     la anchura de la imagen. Con estp,
                     provocamos que nuestro algoritmos aumente su
                      velocidad en gran medida."
              }
            ]
        },
    ]
}
```

- Introducir la aplicación en la carpeta de configuración.
- Crear un Makefile que permita recoger los datos introducidos en el archivo de configuración del cliente. Esto es, el Makefile debe contener una una instrucción denominada “all” (compilar el programa) y otra denominada “run” (ejecutar el programa).

Por último, señalar que para la incorporación de nuevos conjuntos de datos debemos, en primer lugar, acceder a la aplicación y, seguidamente señalar el algoritmo a ejecutar. Una vez estamos en esta pestaña, nos dejara introducir tanto los datos que queremos seleccionar como otros parámetros de interés (dependiendo del algoritmo). Este proceso, puede visualizarse de una manera gráfica en la figura 3.2.

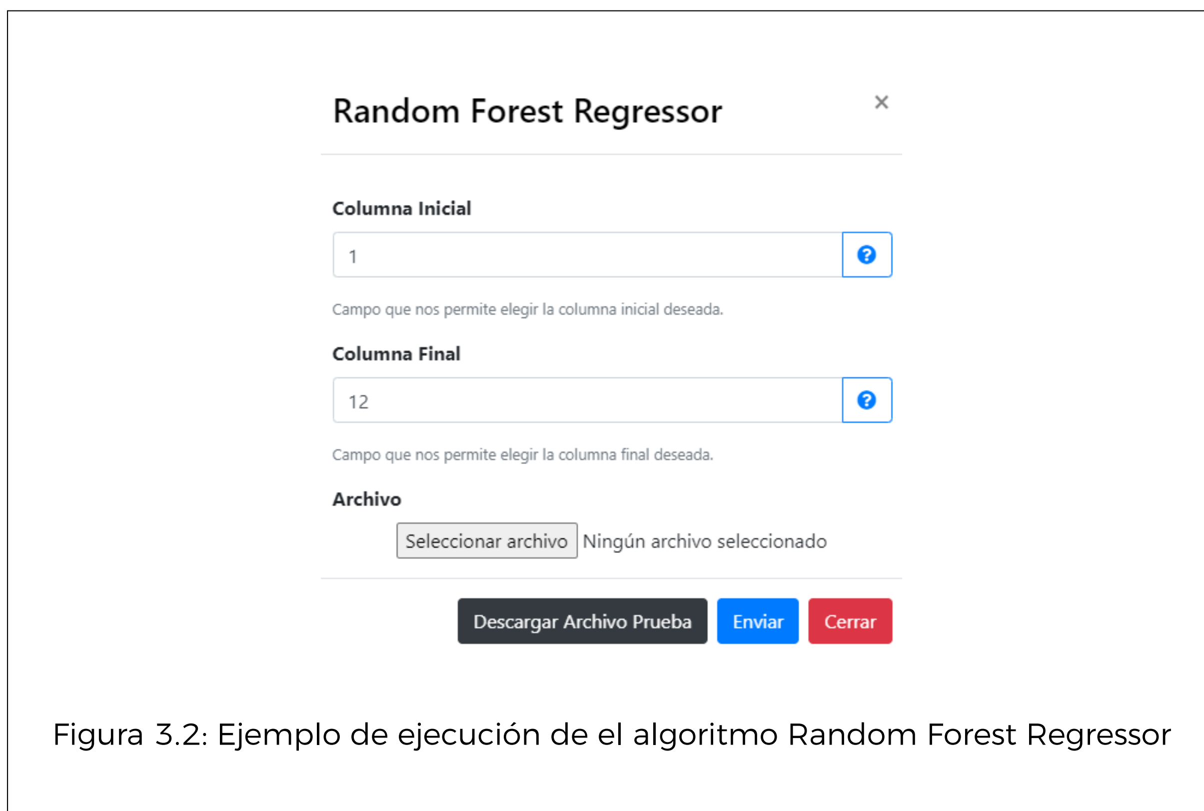


Figura 3.2: Ejemplo de ejecución de el algoritmo Random Forest Regressor

3.2. Algoritmos de Scikit-Learn para Machine Learning

3.2.1. Estructura general

Durante el desarrollo de este proyecto se ha hecho un uso intensivo del patrón de diseño "Estrategia" [49, 50]. La consecuencia del uso de estrategias es que permiten abstraer el desarrollo de familias de algoritmos diseñando cada algoritmo en una clase separada haciendo los objetos intercambiables. Esto reduce el impacto de desarrollo al modificar algoritmos ya incluidos en el framework o al incorporar nuevas variantes o algoritmos. El diseño contempla dos capas de abstracción que pasamos a describir a continuación:

En primer lugar, obtenemos la sección denominada **StrategyFile**, conformando la sección que permite entender ficheros con diferentes formatos. No obstante, todos los formatos que sean compatibles se traducen a un formato estándar (".csv"). Y, en segundo lugar, obtenemos el **StrategyAlgorithm**, que permite ejecutar los diferentes algoritmos implementados en el proyecto. La estructura general se muestra en la figura 3.3.

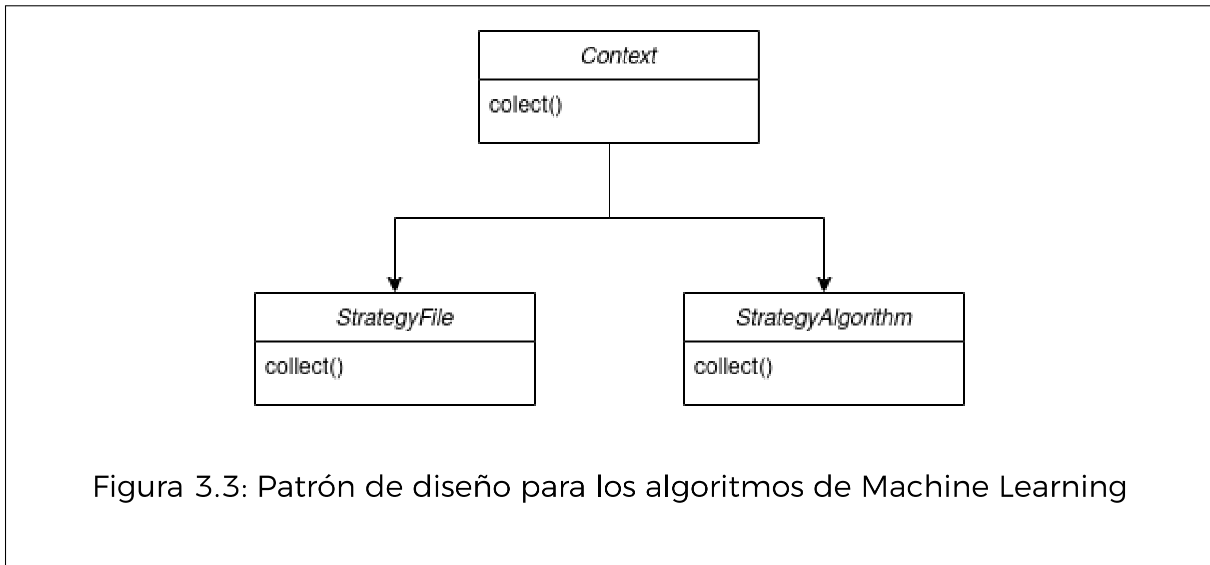


Figura 3.3: Patrón de diseño para los algoritmos de Machine Learning

3.2.2. Estructura de ficheros

La figura 3.4 muestra la estructura de clases utilizada para la gestión de ficheros en este framework. Obsérvese que se permite hacer uso de diferentes formatos, en concreto, CSV, JSON o XSLX. No obstante, a nivel interno el framework sólo hace uso del formato CSV, por lo que archivos almacenados en otro formato (JSON y XLSX) deben ser traducidos. Para la conversión de archivos se utiliza la librería "Pandas" [51]. La incorporación de ficheros con diferentes formatos al sistema resulta, de este modo, bastante sencilla.

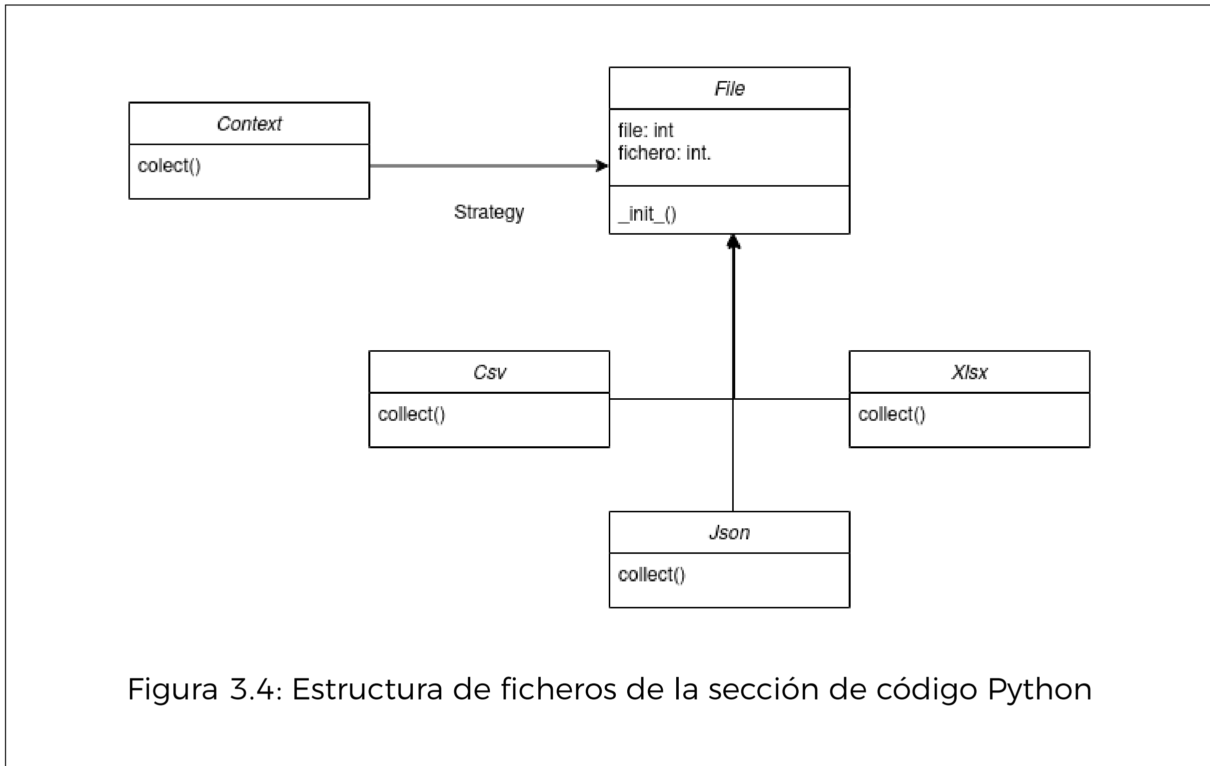


Figura 3.4: Estructura de ficheros de la sección de código Python

3.2.3. Estructura de algoritmos

La figura 3.5 muestra la estructura utilizada para la representación de los algoritmos de Machine Lerning, contemplando diferentes algoritmos relacionados con la Clasificación, la Regresión y el Clustering, integrados a través de la librería "Scikit-Learn" [52] (incorpora un conjunto amplio de algoritmos de Machine Learning). Se trata, nuevamente, del patrón estrategia que permite la ejecución de diferentes algoritmos previamente seleccionados por el usuario/a. Se ha hecho uso del lenguaje de programación "Python" junto con las librerías "Pandas" [51] (facilita el tratamiento eficiente de los datos), "Matplotlib" [53] (permite la generación de gráficas) y "Joblib Parallel" [54] (versiones paralelas de algoritmos de Machine Learning).

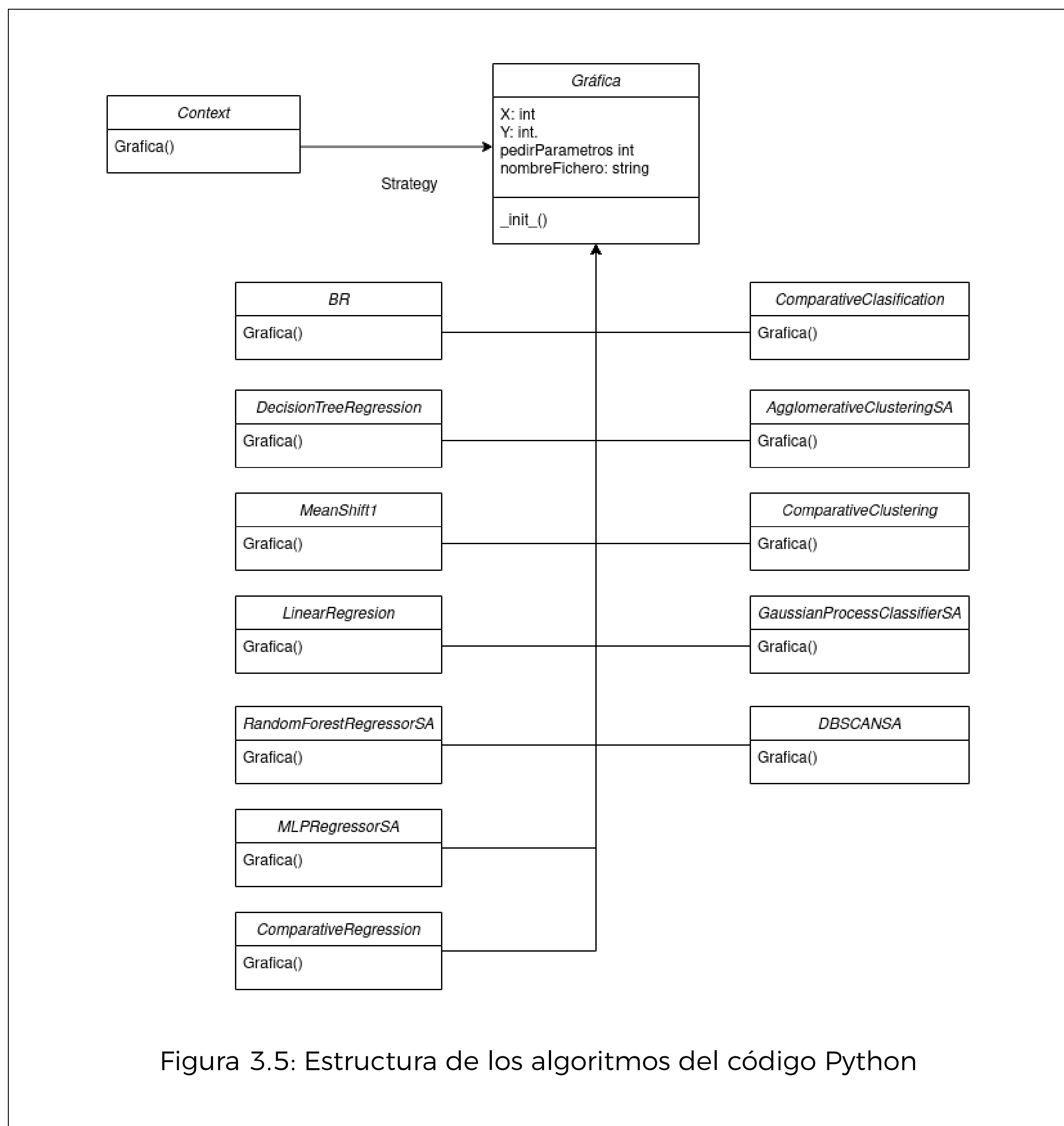


Figura 3.5: Estructura de los algoritmos del código Python

3.2.4. Calsificación, Regresión y Clustering

A continuación se describen los problemas que han sido resueltos aplicando técnicas de Machine Learning haciendo uso de la libería Scikit-Learn. Se ha hecho uso de algoritmos de clasificación, de regresión y de clustering.

3.2.4.1 Clasificación

En esta sección se muestra el uso de métodos de clasificación [55] incorporados en GraphCloud. Para ello, se ha hecho uso de un conjunto de datos denominado ”The Iris Dataset” [56, 57] que, contiene tres tipos de especies dentro de los Lirios: Versicolor, Setosa y Virginica respectivamente. Además del conjunto de etiquetas se cuenta con datos como la longitud y el ancho tanto del sépalo como del pétalo de cada flor, entre otros. Se trata de establecer una clasificación de los Lirios en función de su especie de acuerdo a sus características.

La interfaz para algoritmos de clasificación se muestra en la figura 3.6. GraphCloud en estos momentos proporciona tres algoritmos para la clasificación: Regresión Logística, Support Vector Classification y el Clasificador Gausiano respectivamente. La figura 3.7 representa gráficamente el resultado de la aplicación de estos algoritmos al conjunto de datos que contiene las especies de Lirios. La clasificación resultante contiene tres clases (Versicolor, Setosa y Virginica), los colores representan la probabilidad de pertenecer a dicha clase (representado en el inferior de la imagen).

Figura 3.6: Algoritmos de Clasificación implementados

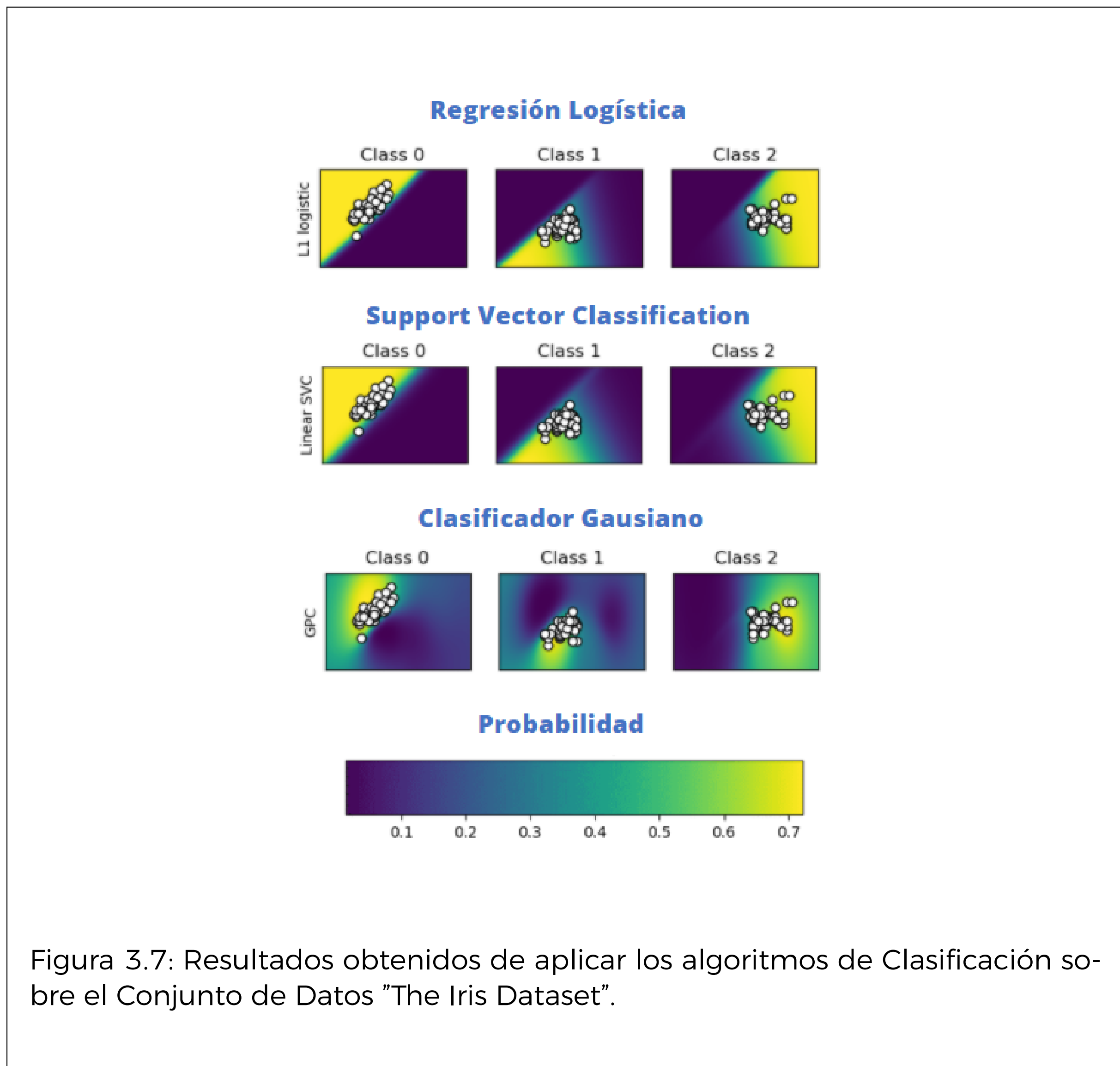


Figura 3.7: Resultados obtenidos de aplicar los algoritmos de Clasificación sobre el Conjunto de Datos ”The Iris Dataset”.

3.2.4.2 Regresión

Para ilustrar el uso de algoritmos de clasificación consideraremos un conjunto que contiene datos relacionados con la contaminación en uno de los municipios de Santa Cruz de Tenerife (Tomecano), concretamente, estos datos aportan las siguientes variables: Fecha, SO2, NO, PM10, O3, CO, Benceno, Tolueno, Xileno y NO2. Lo que se pretende predecir es si existe una alta contaminación en días concretos. Para ello, se ha utilizado una validación cruzada para obtener un modelo mejor entrenado.

GraphCloud incorpora tres algoritmos de regresión implementados en la librería Sckit-Learn [52] (incorpora un conjunto amplio de algoritmos de Machine Learning) [58]: Regresión Lineal, Randon Forest Regresor y una Red Neuronal (figura 3.8).

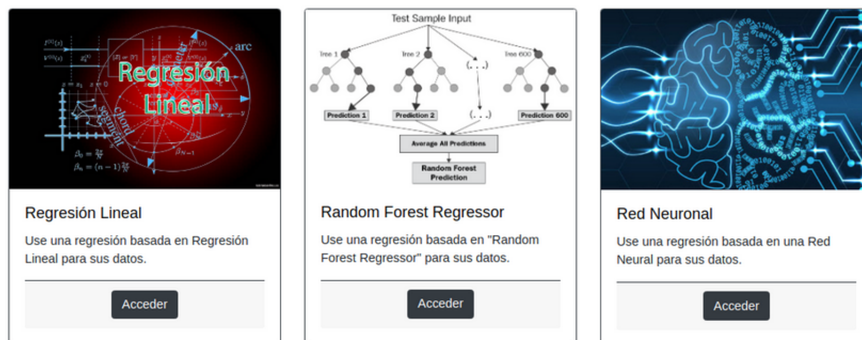


Figura 3.8: Algoritmos de Regresión implementados

La figura 3.9 muestra la ejecución de estos tres algoritmos de regresión sobre el conjunto de datos Tomecano.Se intenta predecir qué día resultaría más contaminante, como podemos observar el algoritmo que mejor predicción ofrece tiene es la Red Neuronal (81,0 %), seguido al Árbol de Decisión (70,06 %) y, por ultimo, la Regresión Lineal (69,79 %).

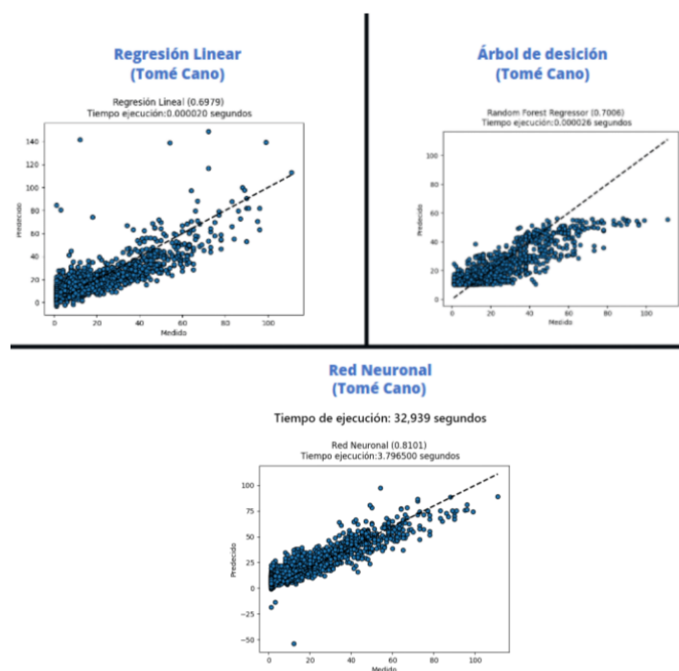


Figura 3.9: Representación de los algoritmos de Regresión implementados

3.2.4.3 Clustering

En esta sección se han utilizado los datos relacionados con Tomecano (recogiendo datos sobre la contaminación diaria) y datos relacionados con el dataset ” The iris dataset” (en relación con los diferentes tipos de Lirios que existen). En la misma instancia, presentamos los algoritmos de clustering [59] que ofrece GraphCloud en estos momentos: Mean shift, Agglomerative Clustering y DBSCANN (figura 3.10).

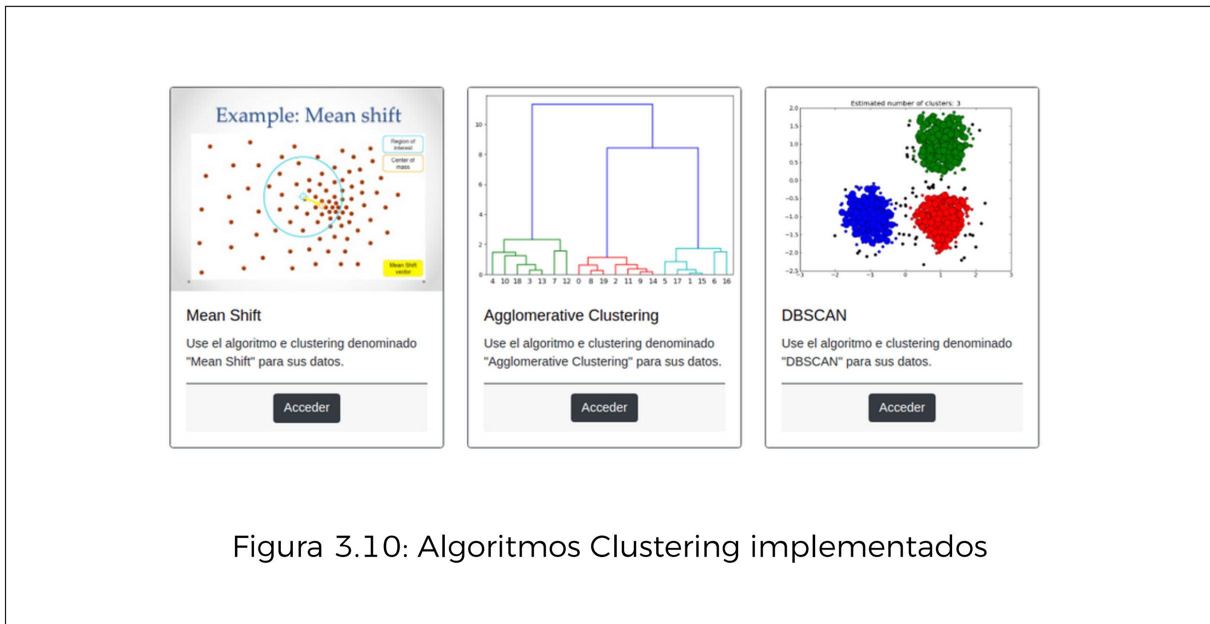


Figura 3.10: Algoritmos Clustering implementados

El resultado de la ejecución de estos tres algoritmos de clustering puede verse en la figura 3.11. Puede observarse que el algoritmos Mean Shift en relación a los datos de Tomecano nos proporciona 32 agrupaciones. No obstante, también podemos observar que para el Agglometive Clusteing y DBSCAN relacionados con el dataset ” The Iris dataset” se obtienen solo dos agrupaciones.

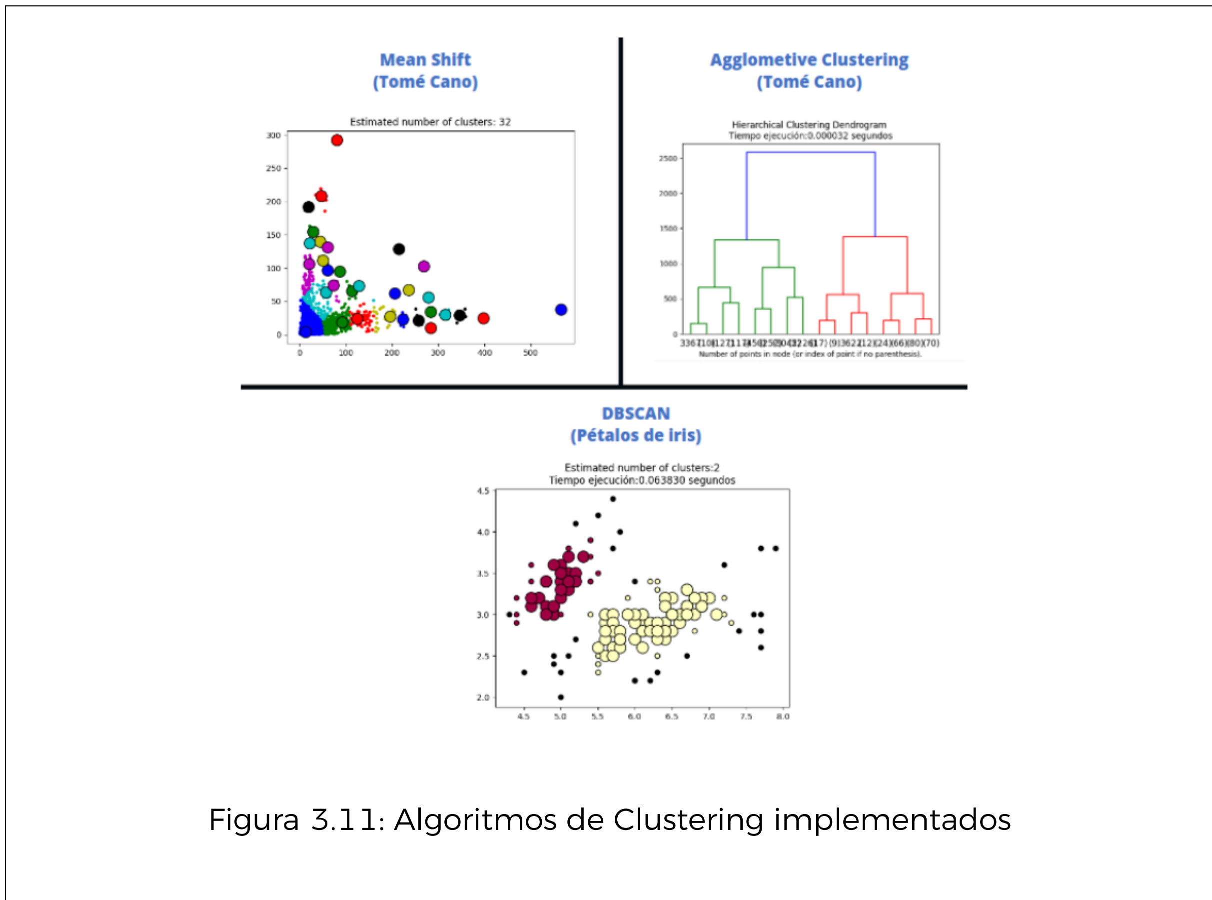


Figura 3.11: Algoritmos de Clustering implementados

3.2.5. Ejecución Paralela

En este apartado ilustramos las posibilidades de reducción de tiempo de cómputo a través del uso del paralelismo. El algoritmo **secuencial** para el análisis de regresión hace uso de la librería “Scikit-Learn” junto con “Pandas” y “Matplotlib”. La figura 3.12 recoge una captura de pantalla del programa “htop” en el que observamos la ejecución haciendo uso de un solo núcleo de cómputo.

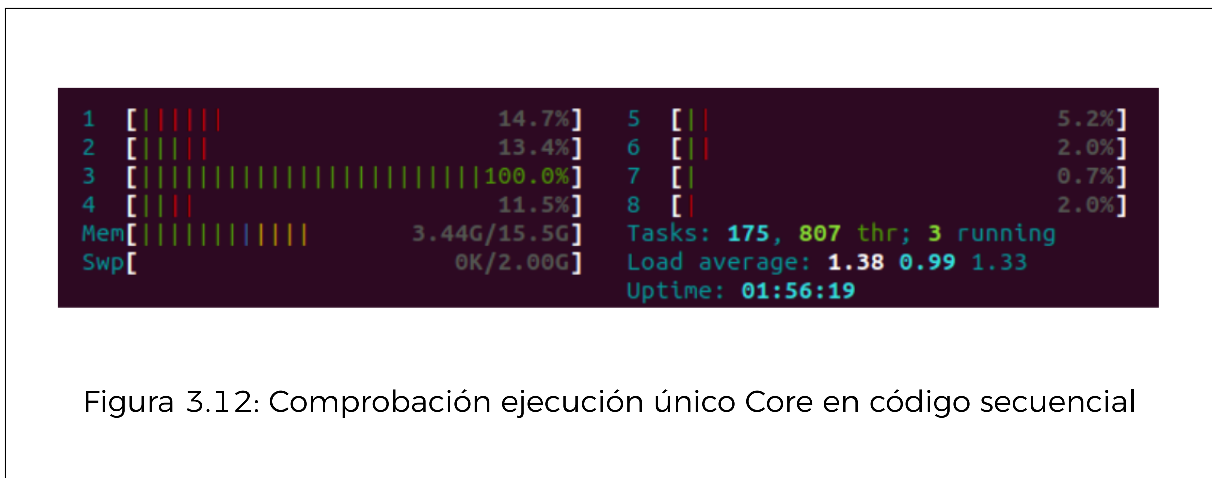


Figura 3.12: Comprobación ejecución único Core en código secuencial

La versión **paralela** de este mismo código hace uso de la librería 'Scikit-Learn"[52] (incorpora un diversos algoritmos de Machine Learning) junto con "Joblib Parallel" [54] (permite paralelizar un código de Python), "Pandas" [51] (permite tratar con mayor facilidad los datos) y "Matplotlib" [53] (posibilita la creación de gráficas).

La figura 3.13 muestra una explotación más eficiente de los núcleos de procesamiento como consecuencia de la paralelización del código, observándose también una reducción importante del tiempo de ejecución.

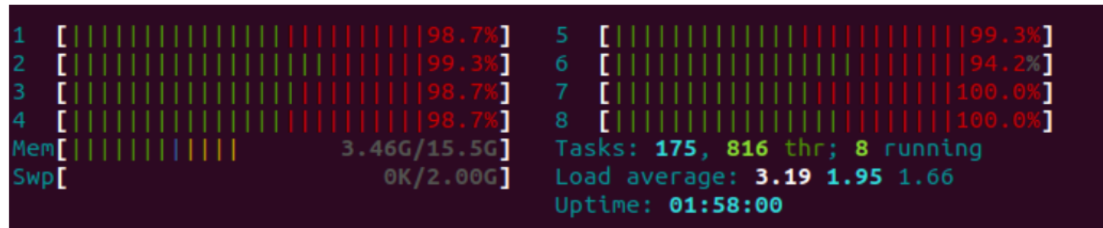


Figura 3.13: Comprobación ejecución multi-core en código paralelo (Joblib)

3.3. Filtro Gaussiano sobre imágenes

En esta sección analizamos varias (versiones secuencial y paralelas) para el procesamiento de una imagen mediante un Filtro Gaussiano. Partimos de una versión secuencial implementada en C/C++ y generamos una versión paralela con MPI y otra con OpenMP. Todas las versiones están disponibles para la ejecución como un servicio de cómputo a través de GraphCloud.

3.3.1. Secuencial

El Listing 3.2 muestra una implementación en C que aplica el filtro Gaussiano a una imagen. Este programa constituye el código base sobre el que se generarán las versiones paralelas MPI y OpenMP.

```
assert(image.size() == 3 && filter.size() != 0);

int heightFinal = image[0].size() + 1 - initHeight;

// Extraemos el tamaño de la imagen
int height = image[0].size();
int width = image[0][0].size();
// Extraemos el tamaño del filtro
int filterHeight = filter.size();
int filterWidth = filter[0].size();
// Calculamos la diferencia entre el tamaño de la imagen y del filtrado
int newImageHeight = height - filterHeight;
int newImageWidth = width - filterWidth;

Image newImage(3, Matrix(heightFinal, Array(newImageWidth)));

int x = 0;

for (int d = 0; d < 3; d++)
{
    for (int i = initHeight; i < newImageHeight; i++)
    {
        for (int j = 0; j < newImageWidth; j++)
        {
            for (int h = i; h < i + filterHeight; h++)
            {
                for (int w = j; w < j + filterWidth; w++)
                {
                    // Aplicamos la operación del filtro Gaussiano
                    newImage[d][x][j] += filter[h - i][w - j] * image[d][h][w];
                }
            }
        }
        x++;
    }
    x = 0;
}
return newImage;

```

Listing 3.2: Filtro Gausssiano en la versión secuencial.

La figura 3.14 muestra la aplicación del filtro gaussiano a una imagen. La imagen de partida se “desenfoca” como consecuencia de la aplicación del filtro.

Figura 3.14: Ejemplo de ejecución del código secuencial

3.3.2. MPI

Esta sección muestra la paralelización, sobre memoria distribuida utilizando la librería para paso de mensajes MPI [60, 61], del algoritmo secuencial que aplica el filtro Gaussiano a una imagen. El código paralelo sigue el modelo de programación SPMD en el que la imagen es distribuida (por boques) entre el conjunto de procesadores, a continuación, los trozos de imagen son computados localmente por cada procesador y por último se combinan las subimágenes obtenidas en la imagen completa. El Listing 3.3 muestra las distintas etapas de las que se compone esta versión paralela.

1. Inicializar la estructura de MPI.

```
1      # Inicializa la estructura de comunicación
2      // de MPI entre los procesos.
3      rc = MPI_Init(&argc, &argv);
4      // Determina el tamaño del grupo asociado con un comunicador
5      rc = MPI_Comm_size(MPI_COMM_WORLD, &size);
6      // Determina el rango (identificador) del proceso
7      // que lo llama dentro del comunicador seleccionado.
8      rc = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
9
```

Listing 3.3: Inicialización estructura MPI

2. Dividir el programa en dos. Uno para si el proceso es el inicial y otro para el resto de los procesos.

3. Dividimos la imagen por el número de proceso. Esto nos ayudará a saber que sección deberá realizar cada proceso. Posteriormente, se enviará la imagen para que cada proceso pueda aplicar el filtrado (Listing 3.4).

```
1      // Calculamos los valores necesarios para poder aplicar el filtrado
2      int newImageHeightNode = userHeight/size;
3      // Enviamos la sección de la imagen a todos los procesos
4      int firstHeight = 0;
5      int finalHeight = 0;
6      for(int n = 1; n < size; n++){
7          finalHeight = newImageHeightNode * n;
8          for (int j = 0; j < 3; j++){
9              for (int i = firstHeight; i < finalHeight; i++){
10                 rc = MPI_Send(&image[j][i][0], userWidth, MPI_DOUBLE,
11                               n, tag, MPI_COMM_WORLD);
12             }
13         }
14         firstHeight += newImageHeightNode;
15     }
16
```

Listing 3.4: División de la imagen

4. Cada proceso (incluido el principal) tendrá que realizar el filtrado de la sección que tiene asignada (Listing 3.5).

```
1      // Reenviamos las secciones de la imagen
2      for (int j = 0; j < 3; j++){
3          for (int i = 0; i < finalImage[0].size(); i++){
4              rc = MPI_Send(&finalImage[j][i][0], finalImage[0][0].size(),
5                            MPI_DOUBLE, 0, tag, MPI_COMM_WORLD);
6          }
7      }
8
```

Listing 3.5: Sección de código donde se le reenvía cada porción de la imagen a cada nodo

5. Cada proceso enviará cada sección de imagen calculada y el principal las unificará para formar la Imagen final (Listing 3.6).

```
// Recogemos los valores y unificamos la Imagen
for(int n = 1; n < size; n++){
    Image newImageNode(3, Matrix(recvImageHeight, Array(newImageWidth)));
    for (int j = 0; j < 3; j++){
        for (int i = 0; i < recvImageHeight; i++){
            rc = MPI_Recv(&newImageNode[j][i][0], newImageWidth,
                          MPI_DOUBLE, n, tag, MPI_COMM_WORLD, &status);
        }
    }
    if(n == 1)
    {
        finalImage = newImageNode;
    }
    else
    {
        finalImage = joinImage(finalImage,newImageNode);
    }
}

```

Listing 3.6: Unificación de la imagen final

Las figuras 3.15 y 3.16 muestran ejecuciones con la versión MPI del filtro Gaussiano variando el tamaño del kernel. Asimismo, la figura 3.17 muestra el estado de los procesos en la ejecución del código MPI mediante el el comando HTOP [62].

Figura 3.15: Ejemplo de ejecución de MPI con kernel 10x10

Programa kernel 5 x 5

Ejecución

```
[ptondreau@ptondreau-1 MPI]$ make run
mpiexec -np 4 mpi_version sample3.png

--- Información de la Imagen---
height: 2112
width: 2816
filterHeight: 5
filterWidth: 5
newImageHeight: 2108
newImageWidth: 2812
newImageHeightNode: 527

Cargando...
Tiempo de ejecución 8 sec
```

Salida

Figura 3.16: Ejemplo de ejecución de MPI con kernel 5x5

HTOP

```
P3 : htop
1  [||||||||||||||||||||||||||||||||||||||||||100.0%]   5  [||||||||||||            25.8%]
2  [||||||||||||||||||||||||||||||||||||||||||100.0%]   6  [||||||||||              21.6%]
3  [||||||||||||                              24.0%]   7  [||||||||||||||||||||||||||||||||||||||||100.0%]
4  [||||||||||||||||||||||||||||||||||||||||||100.0%]   8  [||||||||||||            24.0%]
Mem[||||||||||||||||||||||||||||||||||||5.22G/7.73G]   Tasks: 122, 1044 thr; 8 running
Swp[|                                   768K/8.80G]   Load average: 3.38 2.89 2.90
                                                      Uptime: 02:00:49
```

Figura 3.17: Comprobación hilos en ejecución del código MPI mediante HTOP

3.3.3. OpenMP

Este apartado muestra la versión paralela, sobre memoria compartida utilizando "OpenMP" [63, 64], del algoritmo secuencial que aplica el filtro Gaussiano a una imagen. El Listing 3.7 recoge esta paralelización en la que se ha incluido la correspondiente directiva de OpenMP. La figura 3.18 muestra una captura de pantalla de un instante de ejecución de esta versión paralela en la que se muestran la actividad de los núcleos de procesamiento.

```
    for (int d = 0; d < 3; d++)
    {
    // Sección de código añadida
        #pragma omp parallel for num_threads(numThreads)
        for (int i = 0; i < newImageHeight; i++)
        {
            for (int j = 0; j < newImageWidth; j++)
            {
                for (int h = i; h < i + filterHeight; h++)
                {
                    for (int w = j; w < j + filterWidth; w++)
                    {
                        newImage[d][i][j] += filter[h - i][w - j] * image[d][h
][w];
                    }
                }
            }
        }
    }

```

Listing 3.7: Sección cambiada en OpenMP

Figura 3.18: Comprobación hilos en ejecución del código OpenMP

Las figuras 3.19, 3.20 muestran ejecuciones con el código OpenMP variando el tamaño del kernel.

Ejecución

```
[ptondreau@ptondreau-1 OpenMP]$ make run
./OpenMP_version sample3.png

--- Información de la Imagen---
height: 2112
width: 2816
filterHeight: 5
filterWidth: 5

Cargando...
Tiempo de ejecución 6 sec
```

Salida

Figura 3.19: Ejemplo de ejecución de OpenMP con kernel 5x5

Ejecución

```
[ptondreau@ptondreau-1 OpenMP]$ make run
./OpenMP_version sample3.png

--- Información de la Imagen---
height: 2112
width: 2816
filterHeight: 10
filterWidth: 10

Cargando...
Tiempo de ejecución 16 sec
```

Salida

Figura 3.20: Ejemplo de ejecución de OpenMP con kernel 10x10

Capítulo 4

Desplegando GraphCloud en un clúster de Kubernetes

En este capítulo describiremos la arquitectura basada en contenedores utilizada para la implementación de este proyecto. El objetivo es desplegar los algoritmos disponibles en GraphCloud utilizando un clúster de Kubernetes. Como ya hemos comentado, esta tecnología facilita y automatiza el proceso de despliegue de aplicaciones y queremos evaluar la posible influencia en el rendimiento al utilizarla para aplicaciones HPC (Hight Performance Computing). Antes de realizar la implementación final sobre un clúster real se realizarán diversas pruebas de concepto, con el objeto de corregir detalles de implementación y posibles errores en los despliegues. Una vez solventados estos detalles, se mejorará la especificación del despliegue definitivo de forma que no sea exclusivo al sistema HPC objetivo. También realizaremos diversas pruebas en un equipo de escritorio y en la infraestructura IAAS de la ULL.

4.1. Creando y registrando imágenes de contenedores con Docker

En esta sección se describe el proceso para la creación de imágenes de contenedores con Docker. Este proceso es necesario para poder desplegar aplicaciones utilizando Kubernetes. Los elementos de cómputo esenciales dentro de un clúster de Kubernetes son los Pods, que esencialmente son contenedores similares a los de Docker. La construcción de estos contenedores se llevará a cabo a través del uso de dos ficheros de configuración (uno para el cliente y otro para el servidor en el caso de las aplicaciones de GraphCloud) para su posterior alojamiento en un registro en la nube (DockerHub o Quay.io según proceda).

En primer lugar se procederá a crear la imagen del **cliente**. Este contenedor es el que nos permitirá poder hacer el despliegue del *Frontend* de la aplicación y es el que nos proporciona la parte gráfica de esta. Por otro lado, también se deberá de crear la imagen del contenedor para el **servidor**, que implementa la *API-Rest* de la

aplicación y que permite la ejecución de los algoritmos elegidos por el usuario. A continuación se describen los pasos necesarios para construir estos contenedores:

- Crear un fichero denominado “Dockerfile” e incorporarle las características de la aplicación a desplegar. En nuestro caso, se ha creado uno para la sección del servidor (Listing 4.1) y otro para la sección del cliente (Listing 4.2). Estos ficheros nos permiten configurar nuestras máquinas virtuales dentro del despliegue de Kubernetes, para este proceso, nos basamos en una máquina ya predefinida de Node, en la que instalamos las dependencias de la aplicación (dependencias de Python,...).

```
FROM node:latest
WORKDIR /usr/src/app

COPY ./package*.json ./
COPY ./requirements.txt ./

RUN apt update
RUN apt -y install python3-pip
RUN pip3 install -r requirements.txt
RUN npm install

COPY ./ .

EXPOSE 3000
CMD ["npm", "start"]
```

Listing 4.1: Documento de configuración de Docker-Servidor

```
FROM node:latest
WORKDIR /usr/src/app

COPY ./package*.json ./

RUN npm install

COPY ./ .

EXPOSE 4100
CMD ["npm", "start"]
```

Listing 4.2: Ddocumento de configuración de Docker-Cliente

- Nos situamos dentro de la carpeta correspondiente y ejecutamos los siguientes pasos para construir las imágenes:

- **Crear la imagen:** docker build -t <nombre>.
- **Crear etiqueta:** docker build -t <username>/<nombre>.
- **Crear versión:** docker build -t <username>/<nombre>-t <username>/<nombre>.
- **Enviar nuestra imagen a Docker Hub:** docker push <username>/<nombre>.

Una vez construidas las imágenes de nuestros contenedores, el último paso las publicará en Docker Hub o Quay.io (según proceda) y podremos visualizar que están correctamente publicadas en el repositorio público o privado (según su configuración) en la nube (ver figuras 4.1 y 4.2).

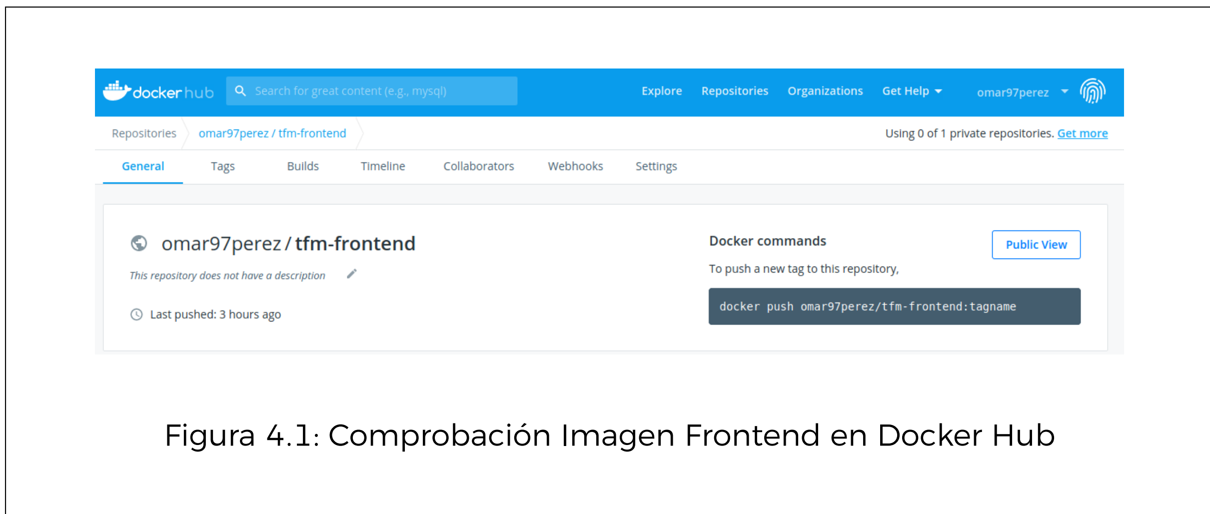


Figura 4.1: Comprobación Imagen Frontend en Docker Hub

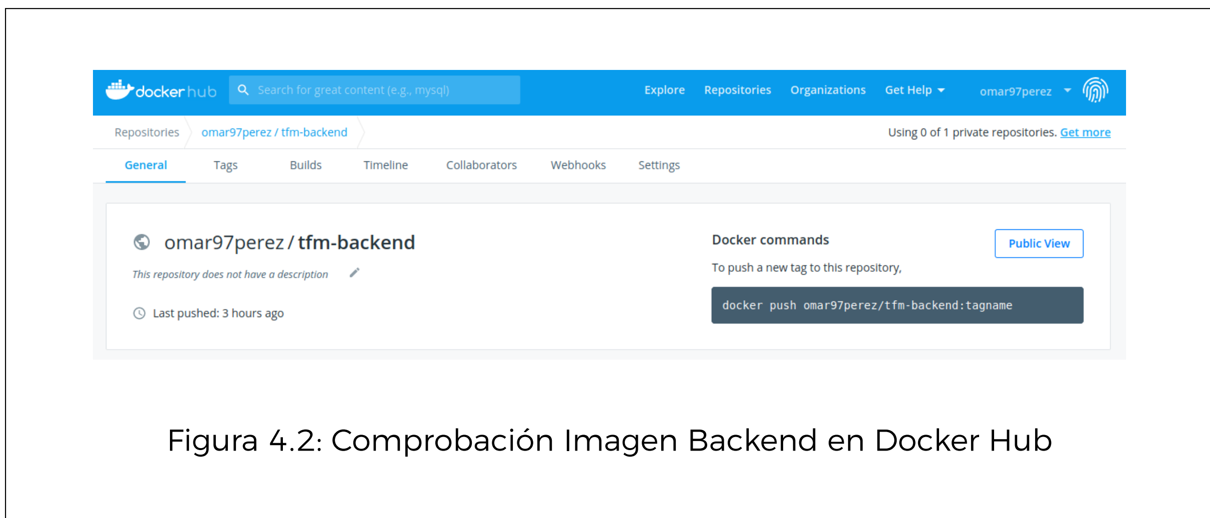


Figura 4.2: Comprobación Imagen Backend en Docker Hub

4.2. Diseñando la arquitectura de despliegue

En esta sección abordaremos el diseño de la estructura para el despliegue de nuestras aplicaciones. El objetivo a alcanzar busca compaginar la simplicidad de la arquitectura junto con otros requisitos como la seguridad de nuestro despliegue. El esquema planteado se muestra en la figura 4.3.

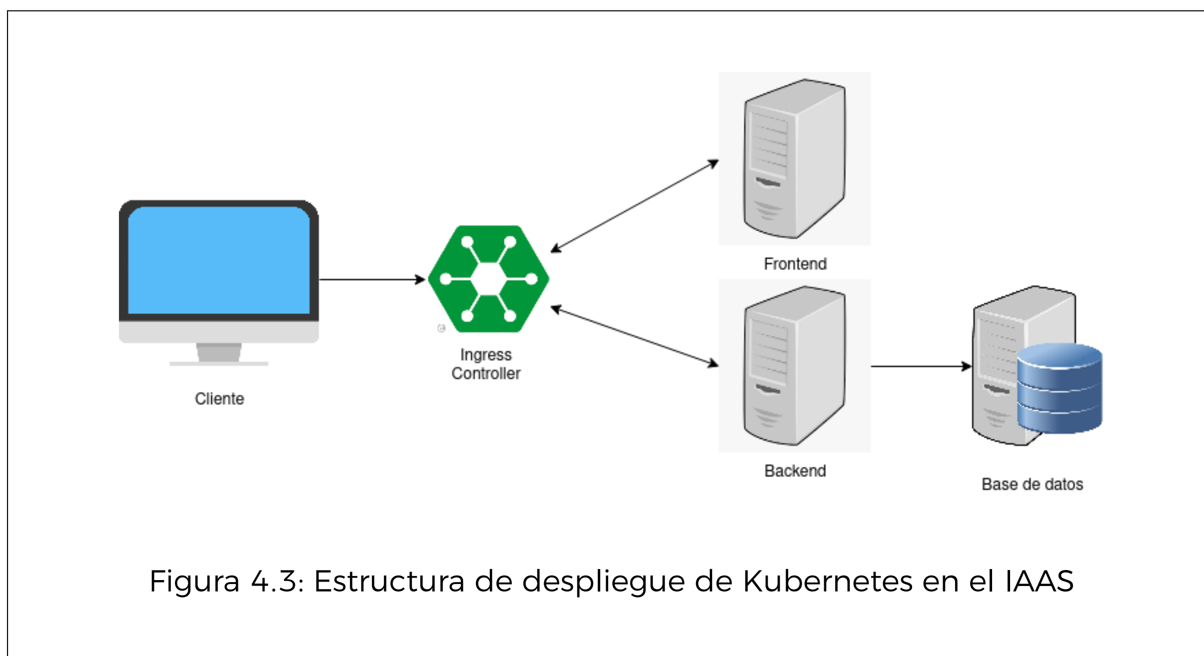


Figura 4.3: Estructura de despliegue de Kubernetes en el IAAS

4.3. Prueba de concepto en un despliegue local

Uno de los principales objetivos de esta sección consiste en realizar un despliegue en un equipo personal y comprobar su correcto funcionamiento. A esto lo denominamos ”despliegue en local”.

4.3.1. Implementación de la configuración de contenedores

Para llevar a cabo el despliegue en Kubernetes de una aplicación, es necesario realizar una serie de tareas. En primer lugar, debemos crea una o varias instancias del *Frontend* de la aplicación que se ejecutarán en el clúster. Kubernetes dispone de varios mecanismos para especificar estas cargas de trabajo (en K8s se denominan *Workloads*) y, en este caso, utilizaremos un *Deployment* [65] (Listing 4.3) que nos permite especificar la imagen del contenedor a ejecutar y cuantas réplicas se instanciarán en el clúster. Para acceder desde el exterior del clúster de K8s es necesario además definir un *Servicio* (Listing 4.4) que permitirá dirigir las peticiones recibidas hacia la instancia del contenedor correspondiente. Estas instancias en K8s se denominan *Pods* como ya hemos mencionado y hemos elegido como mecanismo de servicio para acceder a ellas una combinación de los servicios *Ingress* [66] (permite exponer una IP del clúster al exterior) y *NodePort* [67] (expone en un puerto estático para cada nodo).

En el caso del *backend* de la aplicación, utilizaremos un esquema similar al del *frontend*. De igual forma, será necesario configurar un despliegue de Pods de backend (también con *Deployment*, Listing 4.5) y un servicio con *Ingress* y *NodePort* (Listing 4.6) para que las peticiones a la API que venga de la aplicación *Frontend* ya desplegada en un navegador alcance las instancias de los Pods de Backend en

el clúster. En estos archivos de configuración en K8s, podremos ver la arquitectura a implementar, así como, la configuración general de estos componentes (cliente y servidor).

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: tfm-frontend
spec:
  replicas: 2
  strategy:
    type: Recreate
  selector:
    matchLabels:
      app: tfm-frontend
    spec:
      containers:
      - name: tfm-frontend
        image: omar97perez/tfm-frontend
        imagePullPolicy: Always
        ports:
        - containerPort: 4100
```

Listing 4.3: Fichero despliegue Frontend

```yaml
apiVersion: v1
kind: Service
spec:
    type: NodePort
    ports:
    - port: 4100
      targetPort: 4100
    selector:
      app: tfm-frontend
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: ingress-frontend
  annotations:
    kubernetes.io/ingress.class: "nginx"
spec:
  rules:
  - http:
      paths:
      - path: /
        backend:
          serviceName: tfm-frontend
          servicePort: 4100
```

Listing 4.4: Fichero servicio Frontend

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: tfm-backend
spec:
  replicas: 1
  strategy:
    type: Recreate
  selector:
    matchLabels:
      app: tfm-backend
    spec:
      containers:
      - name: tfm-backend
        image: omar97perez/tfm-backend
        imagePullPolicy: Always
        ports:
        - containerPort: 3000
```

Listing 4.5: Fichero despliegue Backend

```yaml
apiVersion: v1
kind: Service
metadata:
    name: tfm-backend
spec:
    type: NodePort
    ports:
    - port: 3000
      targetPort: 3000
    selector:
      app: tfm-backend
---
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: ingress-backend
  annotations:
    kubernetes.io/ingress.class: "nginx"
spec:
  rules:
  - http:
      paths:
      - path: /api
        backend:
          serviceName: tfm-backend
          servicePort: 3000
```

Listing 4.6: Fichero servicio Backend

4.3.2. Despliegue de la prueba de concepto

Una vez especificados en código YAML los contenedores y servicios necesarios para ejecutar una aplicación en el clúster de K8s, realizaremos los pasos necesarios para aplicarla en esta primera implementación del clúster. De esta forma, esta prueba de concepto en local nos sirve como mecanismo de verificación de las funcionalidades implementadas.

Para la implementación del clúster de K8s en local utilizaremos Minikube [18, 19],haciendo referencia a una herramienta sencilla que nos permite desplegar un clúster en Kubernetes. Asimismo, una vez se tenga instalado Minikube, los pasos a realizar para aplicar la configuración de la aplicación al clúster son:

- **Iniciar el clúster:** para este punto deberemos de ejecutar el comando ”minikube start”.

- Situarnos dentro de las carpetas que contienen la configuración de la aplicación a desplegar y seguir los siguientes pasos:
    - **Crear el despliegue:** kubectl create -f <nombre>.yaml
    - **Añadir el servicio:** kubectl apply -f <nombre>.yaml

Una vez hemos llevado a cabo los pasos requeridos, el despliegue estaría desarrollado y funcionando. Para comprobar su correcto funcionamiento, deberemos corroborar que todos los *Pods* están desplegados y sin contener ningún error (recogido en la figura 4.4).

```
omar@omar-B85M-D3H:~/server\$ kubectl get Pods
NAME                             READY   STATUS    RESTARTS   AGE
nginx-69c78cd8c6-txdpr           1/1     Running   2          54d
tfm-backend-7b745c98f4-nwrks     1/1     Running   0          162m
tfm-frontend-6566b8666d-r42rl    1/1     Running   0          3h42m
tfm-frontend-6566b8666d-w9lc7    1/1     Running   0          3h42m
```

Figura 4.4: Comprobación de Pods en despliegue Kubernetes en local

De igual forma, se debe comprobar si existe algún error en el despliegue del servicio *Ingress Controller*, revisando la salida obtenida del comando kubectl get ingress (figura 4.5).

```
omar@omar-B85M-D3H:~/\$ kubectl get ingress
NAME               CLASS    HOSTS   ADDRESS   PORTS   AGE
ingress-backend    <none>   *                 80      4h46m
ingress-frontend   <none>   *                 80      4h39m
```

Figura 4.5: Comprobación del Ingress Controller en despliegue Kubernetes local

Posteriormente, accedemos a la URL del despliegue y vemos si nuestra aplicación web se ha desplegado correctamente. El resultado se puede ver en la figura 4.6.

Figura 4.6: Comprobación despliegue Kubernetes local

Por último, se han comprobado los *logs* para verificar que todo funciona correctamente. A continuación podremos ver un ejemplo de esto en la figura 4.7.

```
omar@omar-B85M-D3H:~/\$ kubectl logs -f tfm-backend-7b745c98f4-nwrks

> server@0.0.1 start
> nodemon index.js
[nodemon] 2.0.4
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`

Server on port 3000
Conexión satisfactoria a la base de datos

make -C ./Algoritmos/ScikitLearn file=../../Archivos/
file-1609538696828.csv fileExit=../../Archivos/file-1609538696828.png
tipoGrafica=2 columnaSeleccionadaInicial=1 columnaSeleccionada=12 run
```

Figura 4.7: Comprobación logs Kubernetes local

4.4. Despliegue en el IAAS de la ULL

En esta sección describimos la experiencia de despliegue en la infraestructura IAAS de la ULL. Este proceso, nos ha ayudado a practicar con las herramientas seleccionadas y, además, a plantear el problema de forma distribuida, con varios nodos de cómputo que simulan a un sistema HPC y que nos han permitido detectar fallos y establecer una identificación de mejoras a realizar.

4.4.1. Despliegue

Para realizar la implementación del clúster sobre el IAAS de la ULL hemos utilizado una herramienta de gestión de clúster de K8s denominada *k3sup* [68]. Se ha elegido esta herramienta entre otras por su sencillez y versatilidad a la hora de realizar los despliegues. *k3sup* ha sido utilizada con éxito en el despliegue de Kubernetes tanto en un sistema basado en RPis [69] como en sistemas de cómputo de gran capacidad [70, 71].

Se han seguido las directrices de la página oficial de la herramienta para el despliegue del clúster, entre las que se encuentran los requisitos previos que deben implementarse en los nodos de cómputo. A continuación se detallan las acciones y el software que fue necesario instalar:

- Instalar curl.
- Generar claves y compartirlas entre las máquinas.
- Añadir al usuario que va a ejecutar el clúster a la lista de administradores del sistema.

Una vez realizados los pasos requeridos estamos en disposición de realizar el despliegue del clúster. En primer lugar, debemos instalar *k3sup*:

- Instalar k3sup.
- Ejecutar el instalador de k3sup. Para ello, ejecutaremos el comando “k3sup install –ip IP –user usuario”.
- Realizar la exportación del Kubeconfig generado mediante el comando “export KUBECONFIG=/home/usuario/kubeconfig”.

Una vez llevemos a cabo todas las directrices establecidas, solo deberemos crear tanto el servicio como el despliegue de la aplicación tanto en la versión de Backend (Listing 4.8) como la de Frontend (Listing 4.7). De igual forma, se ha modificado el fichero de servicio (con respecto a la versión anterior) para utilizar el Ingress que nos proporciona k3s. Los nuevos ficheros contienen la siguiente información:

```yaml
apiVersion: v1
kind: Service
spec:
    ports:
    - port: 80
      targetPort: 4100
    selector:
      app: tfm-frontend
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-frontend
spec:
  rules:
  - http:
      paths:
      - path: /
        backend:
          serviceName: tfm-frontend
          servicePort: 80
```

Listing 4.7: Fichero servicio Frontend

```yaml
apiVersion: v1
kind: Service
spec:
    ports:
    - port: 80
      targetPort: 3000
    selector:
      app: tfm-backend
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-backend
spec:
  rules:
  - http:
      paths:
      - path: /api
        backend:
          serviceName: tfm-backend
          servicePort: 80
```

Listing 4.8: Fichero servicio Backend

Seguidamente, comprobamos que los Ingress se han creado satisfactoriamente. Este procedimiento se refleja en la siguiente figura (4.8).

```
usuario@debian:~\$ kubectl get ingress
NAME               CLASS    HOSTS   ADDRESS        PORTS   AGE
ingress-frontend   <none>   *       10.6.131.243   80      21h
ingress-backend    <none>   *       10.6.131.243   80      21h
```

Figura 4.8: Comprobación del funcionamiento del Ingress Controller

Para finalizar, deberemos acceder a la IP que nos proporciona el Ingress, teniendo a su vez que observar que la aplicación web se ha desplegado correctamente (figura 4.9).

Figura 4.9: Comprobación de despliegue en el IAAS

4.5. Despliegue en un nodo HPC

El objeto principal de este apartado, consiste en observar como se han utilizado las diferentes herramientas dentro del servidor definitivo a utilizar. Este servidor es uno de los nodos de cómputo del clúster HPC Verode del grupo de investigación ("Computación de altas prestaciones") en el que están integrados los tutores de este trabajo. En concreto, se va a utilizar un nodo que dispone de 40 cores y 128Gb de RAM y que también incorpora una GPU de la que podemos hacer uso desde los contenedores desplegados. La finalidad principal de esta experiencia consiste en el análisis de rendimiento que podamos obtener al ejecutar las instancias de Backend en nodos con estas características. Podremos realizar diferentes comparaciones lanzando ejecuciones utilizando diferentes cantidades de recursos (número de cores o memoria utilizada) y comprobar que efectivamente se observan disminuciones de tiempos de cómputo cuanto más cores se utilizan si el Backend de estas aplicaciones es paralelizable.

4.5.1. Docker Hub

Como primera aproximación, desplegaremos en el clúster K8s instanciado en este nodo de cómputo con imágenes obtenidas del registro de imágenes de nuestras aplicaciones en Docker Hub. Para realizarlo, se han modificado los archivos de configuración que especifican los Pods de forma que queden definidos los recursos solicitados, en concreto el número de cores o la memoria requerida. Esto nos permitirá en el siguiente capítulo la posibilidad de poder ejecutar la aplicación con diferentes configuraciones, variando el número de cores por ejemplo. Este estudios nos permitirá analizar el rendimiento de los algoritmos implementados en los Backends y demostrar la viabilidad de utilizar K8s en un sistema HPC. Las configuraciones de los Pods modificadas las podemos ver en el listado 4.9.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: tfm-backend
spec:
  replicas: 1
  strategy:
    type: Recreate
  selector:
    matchLabels:
      app: tfm-backend
  template:
    metadata:
      labels:
        app: tfm-backend
    spec:
      containers:
      - name: tfm-backend
        image: omar97perez/tfm-backend
        imagePullPolicy: Always
        # Mejora realizada
        resources:
                requests:
                        memory: "128Mi"
                        cpu: "4"
                limits:
                        memory: "128Mi"
                        cpu: "4"
        ports:
                - containerPort: 3000
```

Listing 4.9: Nuevo fichero de despliegue del Backend

4.5.2. Quay (Bioinformatics)

Como ya se describió en la sección 2.4.2, se ha utilizado también el registro de imágenes de contenedores "Quay.io". Las imágenes almacenadas en este repositorio presentan ciertas ventajas sobre las que podemos encontrar en Docker Hub en términos de rendimiento. Suelen estar optimizadas en tamaño y las imágenes basadas en Python también han sido mejoradas.

Se describen a continuación los pasos necesario para registrar y utilizar este repositorio:

- Crear una cuenta en la plataforma de "Quay.io".

- Crear dos repositorios dentro de la plataforma. Uno para la aplicación de Frontend y otro para el Backend.
- Utilizar la interfaz de comandos de Quay para autenticarnos vía terminal de nuestro equipo. Para ello, ejecutaremos el comando ”docker login quay.io” e introduciremos tanto el usuario como la contraseña de la cuenta creada con anterioridad.
- Etiquetar ambos contenedores a ambas imágenes. Para ello, usaremos el comando ”docker tag <imageId><repositorioQuay>”.

Una vez realizados los pasos previos, solo tendremos que subir nuestras imágenes a los repositorio de Quay.io. Para ello, tendremos que utilizar el comando”sudo docker push <repositorioQuay>”(ver proceso en figura 4.10).

```
omar@DESKTOP-J8051TQ:/mnt/c/Users/Omar$ sudo docker push quay.io/omar97perez/tfm-backend:latest
The push refers to repository [quay.io/omar97perez/tfm-backend]
029ca5f6d34d: Pushed
775d2b226d6a: Pushed
688598d94791: Pushed
3227a6e34b28: Pushed
6996c37cf13a: Pushed
138c03a877bd: Pushed
04c099f9268d: Pushed
ceefa87e2985: Pushed
b5fdfe379fdc: Pushed
6f903a63aec0: Pushed
9459233b6a63: Pushed
7ed9e3d1c5f1: Pushed
fdba6a5d9dd7: Pushed
07700abd910e: Pushed
edfb8ee7c346: Pushed
aa817488a0dd: Pushed
74825a980b6d: Pushed
1fb0a31fe7c2: Pushed
latest: digest: sha256:df3c66636370831b7ab694e5bc466ae4dc4c1c37a645f1f042143093931a62a7 size: 4111
```

Figura 4.10: Subiendo las imágenes de los contenedores a Quay

Por último, deberemos modificar los archivos del despliegue, tanto los del Frontend (Listing 4.11) como los del Backend (Listing 4.10). Esta modificación, consiste simplemente en cambiar el repositorio de uso para el despliegue, eliminando la referencia al repositorio de ”Docker Hub” e introduciendo la referencia al repositorio de ”Quay.io”.

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: tfm-backend
spec:
 replicas: 1
 strategy:
  type: Recreate
 selector:
  matchLabels:
   app: tfm-backend
 template:
  metadata:
   labels:
    app: tfm-backend
  spec:
   containers:
   - name: tfm-backend
    # Modificación
    image: quay.io/omar97perez/tfm-backend
    imagePullPolicy: Always
    resources:
        requests:
            memory: "1280Mi"
            cpu: "32"
        limits:
            memory: "1280Mi"
            cpu: "32"
    ports:
        - containerPort: 3000
```
Listing 4.10: Nuevo fichero de despliegue del Backend

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: tfm-frontend
spec:
  replicas: 2
  strategy:
    type: Recreate
  selector:
    matchLabels:
      app: tfm-frontend
  template:
    metadata:
      labels:
        app: tfm-frontend
    spec:
      containers:
      - name: tfm-frontend
        # Modificación
        image: quay.io/omar97perez/tfm-frontend
        imagePullPolicy: Always
        ports:
        - containerPort: 4100
```

Listing 4.11: Nuevo fichero de despliegue del Frontend

Capítulo 5

Análisis de rendimiento de algunos algoritmos en GraphCloud

5.1. Introducción

Con el objetivo de demostrar que es factible utilizar un entorno de ejecución como Kubernetes para realizar cómputo HPC, vamos a realizar un análisis de rendimiento de algunos algoritmos en GraphCloud. Observaremos y compararemos el rendimiento de algunos códigos con cada una de las tecnologías utilizadas y también con una ejecución tradicional HPC a través de un sistemas de colas. Intentaremos medir el tiempo de ejecución para algunos de los algoritmos implementados en Graphcloud (un análisis de regresión con el algoritmo Random Forest y un computo habitual sobre imágenes como es un filtro Gaussiano). Se elaborará un estudio comparativo con los datos recogidos que precise de manera concisa cual es la mejor herramienta para cada tipo de problema y, si el resto resultan recomendables o no en cada caso.

Para cada uno de los algoritmos analizados, realizaremos una representación gráfica de métricas de rendimiento. Se han realizado medidas del tiempo de ejecución de varias partes de los algoritmos, separando por un lado a) el tiempo de cómputo específico del algoritmo estudiado y b) el tiempo total incluyendo cómputos entrada/salida (incluida carga de datos vía red). La diferencia de estas dos medidas la hemos representado como c) tiempo E/S, y nos permite separar la escalabilidad del algoritmo estudiado de otras tareas necesarias para la ejecución del problema completo. Esta métricas posibilitarán una mayor comprensión para cada una de las herramientas analizadas y, facilitará la comparación de las mismas.

Queremos detallar sobre cada uno de las métricas anteriormente citadas las siguiente consideraciones: el **Tiempo de Cómputo**, es aquel entendido como el tiempo en que tarda en ejecutar únicamente el algoritmo bajo estudio, una vez se han establecido la carga de datos y se llama a la función que lo implemente. El **Tiempo total**, recoge, como su definición indica, el tiempo en el que se tarda en ejecutar todo el proceso incluyendo la interacción con el usuario a través

de la interfaz web o el sistema de colas. Esto viene a decirnos que, si el proceso se ejecuta en una página web, el proceso acabaría cuando la imagen procesada pueda reflejarse dentro de la interfaz del frontend y el usuario pueda visualizarla.. Del mismo modo, si el proceso ocurre en una ejecución en el sistema de colas, el proceso culminaría cuando la imagen este almacenada dentro de la carpeta de salida de resultados. Por tanto, recogeremos en una métrica que denominamos **Tiempo E/S**, como el tiempo que abarca todo el proceso que no recoge el tiempo de ejecución del algoritmo. Dicho de otro modo, es el tiempo no contemplado en el tiempo de cómputo. Este tiempo está compuesto por el tiempo de lectura o escritura de ficheros, el tiempo que tarda en materializarse en la pagina web (si procediese), entre otros.

Para los tiempos de cómputo o totales, incluiremos a demás métricas de *speedup* y *eficiencia* de los diferentes tipos de algoritmos. Una métrica como el *speedup* (página 189 del libro [72]) nos proporciona la ganancia de velocidad que adquiere un algoritmo con respecto a su versión secuencial y es muy utilizada en el análisis de algoritmos paralelos. Y la *eficiencia* (página 193 del libro [72]) nos ofrece además la capacidad de saber cual es la porción de tiempo que los elementos de proceso (cores) dedican al trabajo útil de cómputo (y no a otras tareas como comunicaciones). Cuanto más cerca de 1 estemos, más estaremos aprovechando la capacidad pico del sistema paralelo.

5.2. Random Forest Regressor.

En esta sección nos planteamos el objetivo de medir, comparar y sacar conclusiones sobre el uso de diferente métodos de ejecución en el clúster HPC con uno de los algoritmos de Machine Learning implementados, en concreto, el algoritmo denominado ”Random Forest Regressor” (o en español regresión de bosques aleatorios). La idea fundamental es demostrar que el uso de una tecnología basada en contenedores como Kubernetes no supone un coste adicional en términos de rendimiento a la hora de ejecutar cargas de trabajo en un clúster HPC.

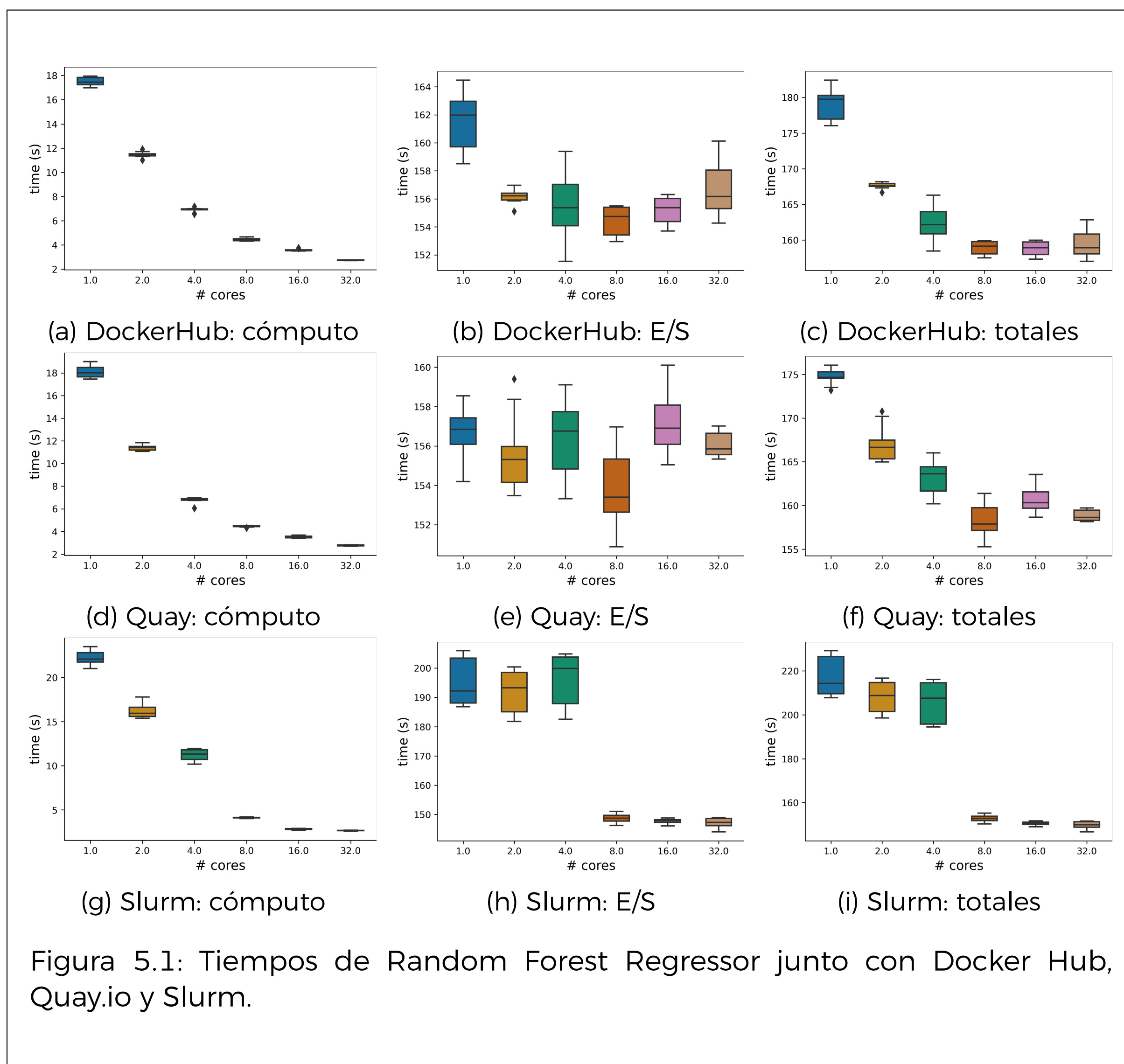
Asimismo, hemos de resaltar que el algoritmo Random Forest Regressor (RFR), se encuentra incluido dentro de la librería ”Scikit-learn” [52] (paquete que integra diferentes algoritmos de Machine Learning). En la misma instancia, hacemos hincapié en que, este algoritmo está integrado en el código Python desarrollado para este proyecto (apartado 3.3) apoyado por Pandas [51] (librería que nos permite el manejo de los datos con mayor facilidad). Este código Python lo hemos paralelizado utilizando la librería ”Joblib Parallel” [54] (paquete que nos posibilita paralelizar código en Python).

5.2.1. Análisis de rendimiento para Random Forest Regression

Nos centraremos en la observación de diferentes ejecuciones (concretamente 10 por experiencia para un número de cores y tamaño de problema fijado) del algoritmo denominado ”Random Forest Regressor”. En concreto, veremos las ejecuciones utilizando Kubernetes (k8s) con imágenes generadas a partir de imágenes de contenedores base de Docker Hub y Quay.io, y una ejecución HPC tradicional utilizando el sistema de colas (Slurm). El algoritmo está alimentado con una colección de datos que podremos ver en el cuadro 5.1 que comprende 1.126.926 registros y con un tamaño de 71.815 KB. En la figura 5.1 se muestran los tiempos de ejecución de las métricas planteadas en el análisis de rendimiento: tiempo de cómputo, tiempo de E/S y tiempos total.

Fecha	Hora	SO2	NO	PM10	O3	CO	Benceno	Tolueno	Xileno	NO2
1/1/2010	1	12	5	31	56	0.4	2	3	2	22
1/1/2010	2	12	19	55	23	0.7	2	7	3	57
1/1/2010	3	11	4	37	37	0.6	2	6	2	38
1/1/2010	4	11	3	23	48	0.5	2	4	2	26
1/1/2010	5	11	2	21	52	0.4	2	10	3	20
1/1/2010	6	11	2	19	50	0.3	2	4	2	19
1/1/2010	7	12	2	18	51	0.3	2	4	2	19
1/1/2010	8	11	2	18	43	0.4	2	4	2	25
1/1/2010	9	11	2	17	45	0.4	2	4	2	23
1/1/2010	10	13	3	18	50	0.4	2	4	2	23
1/1/2010	11	14	4	15	56	0.4	2	4	2	26
1/1/2010	12	15	1	10	78	0.4	1	2	2	10
1/1/2010	13	14	1	9	82	0.6	2	2	2	8
1/1/2010	14	13	1	9	87	0.5	1	1	2	3
1/1/2010	15	12	1	9	86	0.5	1	1	2	4
1/1/2010	16	11	1	8	90	0.7	2	1	2	2
1/1/2010	17	8	1	10	87	0.7	2	1	2	3
1/1/2010	18	7	1	13	83	0.5	1	1	2	6
1/1/2010	19	8	2	10	79	0.5	2	2	2	11
1/1/2010	20	11	1	10	81	0.4	2	2	2	9
1/1/2010	21	17	2	11	74	0.5	2	2	2	18
1/1/2010	22	18	6	17	53	0.8	2	4	2	37
1/1/2010	23	16	6	18	50	0.8	2	7	3	40
1/1/2010	24	19	3	10	63	0.6	2	4	2	24
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Cuadro 5.1: Datos de entrada (TomeCano) para el análisis de rendimiento del Random Forest Regression (RFR)



(a) DockerHub: cómputo (b) DockerHub: E/S (c) DockerHub: totales

(d) Quay: cómputo (e) Quay: E/S (f) Quay: totales

(g) Slurm: cómputo (h) Slurm: E/S (i) Slurm: totales

Figura 5.1: Tiempos de Random Forest Regressor junto con Docker Hub, Quay.io y Slurm.

En primera instancia, a través de la primera línea de gráficas de la figura 5.1, podemos ver los diferentes tiempos de ejecución del algoritmo ejecutado en k8s utilizando imágenes de Docker Hub. En relación al tiempo de cómputo (figura 5.1a), podremos observar como la distribución sigue la representación lógica: cuanto mayor es la cantidad de cores empleados menor será el tiempo que requiera el cómputo, mostrando en este caso una escalabilidad en paralelo aceptable, bajando de 18s con un core a menos de 3s con 32 cores. Si analizamos los tiempos de ejecución E/S (figura 5.1b) observamos que se mantienen más o menos constantes entre 155 y 162s con ligeras variaciones dependiendo del número de cores. Los tiempos totales (figura 5.1c) reflejan en parte el buen comportamiento del cómputo, pero que ya no es significativo a partir de 8 cores debido a que la contribución de la E/S es muy grande en este caso. La ventaja de paralelización que se obtiene con la utilización de Joblib vemos que enseguida deja de compensar debido a la poca contribución de la ganancia en cómputo cuando paralelizamos.

Realizamos un análisis similar ejecutando el problema sobre K8s e imágenes de contenedores en Quay.io (reflejado en la segunda línea de la figura 5.1). En la figura 5.1d, se puede observar que, para el tiempo de cómputo la distribución sigue una distribución similar al caso anterior (lo que quiere decir que cuanto mayor sea el número de cores que empleemos, mejor será el rendimiento). Los tiempos de E/S (figura 5.1e) se mantienen también constantes entre 154s y 158s y los tiempos totales (figura 5.1f) muestran un buen comportamiento hasta 8 cores una vez que la contribución del cómputo ya no se distingue de los tiempos de E/S que siguen siendo un orden de magnitud superiores.

En la última fila de la figura 5.1 encontramos los diferentes tiempos de ejecución con el sistema de colas (Slurm). Esta experiencia realizada con un sistema de ejecución HPC tradicional nos sirve de referencia respecto a las ejecuciones anteriores basadas en k8s. Con respecto a los tiempos de cómputo (figura 5.1g), encontramos un comportamiento habitual para un algoritmo paralelizado, mejorando los tiempos des un core hasta 32 como en los casos anteriores. Es de destacar que para los tiempo de 1, 2 y 4 cores, los tiempos de cómputo son algo peores que en los casos anteriores. Esta situación es más elocuente en los tiempos de E/S (figura 5.1h). Para las ejecuciones en 1, 2 y 4 cores se mueven en torno a los 200s, se produce una disminución a partir de 8 cores en torno a los 150s, que es congruente con los caso anteriores con k8s. Este comportamiento creemos puedes ser consecuencia del runtime del Python nativo en el nodo HPC verode21 y de cómo gestiona la memoria (puede haber un salto debido a la jerarquía de memoria en 8 cores) respecto a como lo realiza el intérprete de Python que está disponible en las imágenes de los contenedores (tanto en Docker Hub como en Quay). Esta situación se refleja también en el tiempo total (figura 5.1i), donde observamos cómo a partir de 8 cores se produce un salto en el rendimiento. Seguimos teniendo el problema de que los tiempos de E/S son un orden de magnitud superior a los de cómputo, por lo que tenemos que analizar en más detalle la implementación del problema y ver dónde se producen las pérdida de eficacia en la E/S.

5.2.2. Comparativa de los entornos de ejecución

En la figura 5.2 hemos representado los datos de rendimiento obtenidos de forma que se pueden apreciar las diferencias de los tiempos de cómputo y el tiempo total, para cada una de los entornos de ejecución utilizados. En las gráficas se han etiquetado las líneas con la etiqueta *k8s-DH* para Kubernetes con DockerHub, *k8s-Q* para Kubernetes con Quay.io y *Slurm* para la ejecución tradicional HPC con el sistema de colas Slurm.

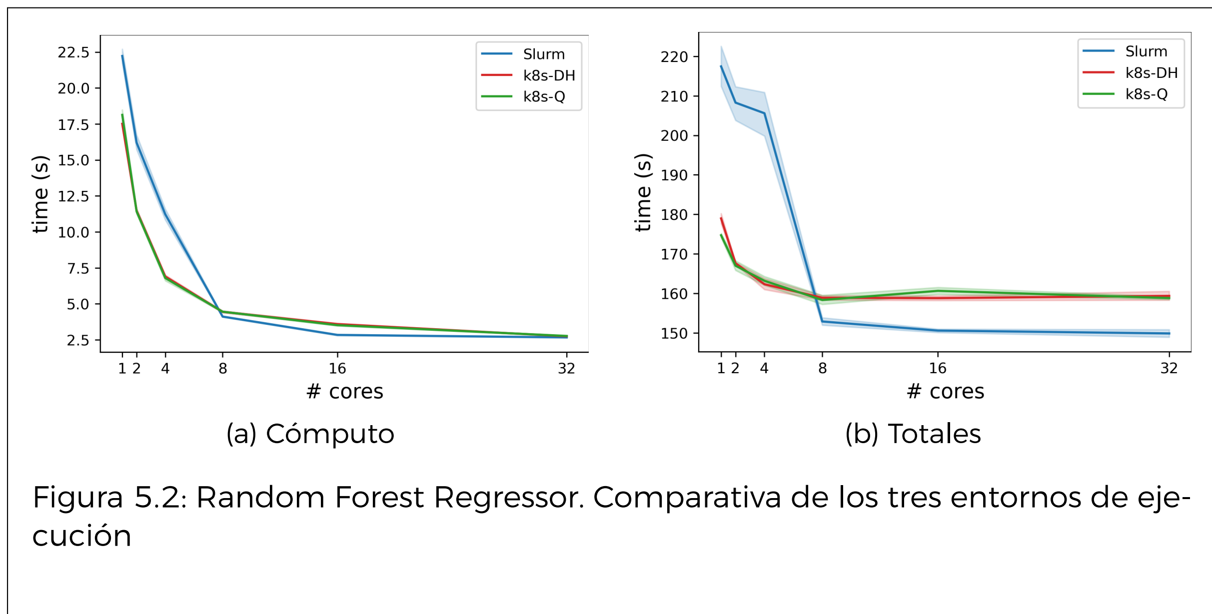


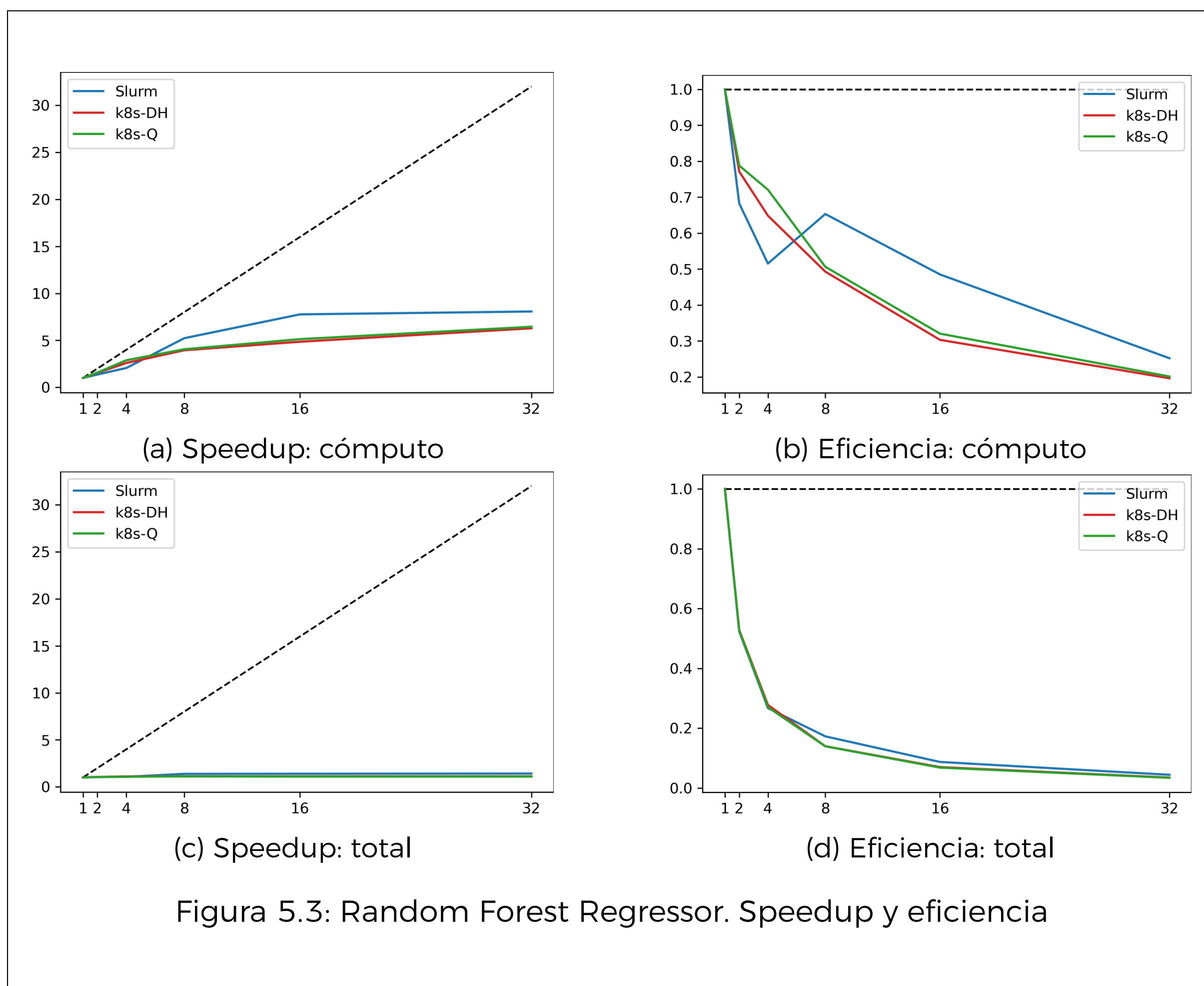
Figura 5.2: Random Forest Regressor. Comparativa de los tres entornos de ejecución

Podemos observar que el tiempo de cómputo (figura 5.2a) y el tiempo total 5.2b tanto Docker Hub como Quay.io son bastante similares y escalan con el número de elementos de proceso (*cores*). Sin embargo, en la ejecución mediante el sistema de colas (Slurm) con un número de cores inferior a 8 es peor que con los dos entornos de Kubernetes. Sin embargo, a partir de los 8 cores se revierte esta situación haciendo que la ejecución tradicional se comporte mejor que utilizando Kubernetes como cabría esperar en un principio.

Por todo lo comentado, la elección más viable para el uso de 8 cores o más, estaría en seleccionar una ejecución con sistema de colas, mientras que, por otro lado si el número de cores implementados fuera menor de 8, la mejor opción sería Kubernetes (Docker Hub y Quay.io). No obstante, la diferencia resulta poco significativa, por lo que sería mejor seleccionar la tecnología de contenerización, ya que nos aporta numerosas ventajas, como por ejemplo, tener un sistema aislado que no este relacionado con nuestro sistema, un sistema más sencillo de administrar...

Sin embargo, y como comentamos en la sección anterior, es necesario analizar en profundidad las razones del elevado coste de los tiempos de E/S en esta implementación de algoritmo Random Forest.

Como último análisis queremos mostrar los resultados obtenidos de speedup y eficiencia de esta implementación paralela basada en Joblib. En la figura figura 5.3) se muestran estas métricas de rendimiento. Si bien el análisis del tiempo de cómputo de forma aislada muestran un comportamiento razonable (el cómputo escala con el número de cores hasta 16), cuando incluimos ilustramos el proceso completo (tiempo total) no obtenemos buenos resultados y que son esperables habida cuenta el mal comportamiento de esta implementación en los procesos de E/S. Como veremos en la siguiente sección, el problema de la aplicación de un filtro Gaussiano sobre una imagen sí que tiene un mejor comportamiento desde el punto de vista de una arquitectura paralela.



(a) Speedup: cómputo

(b) Eficiencia: cómputo

(c) Speedup: total

(d) Eficiencia: total

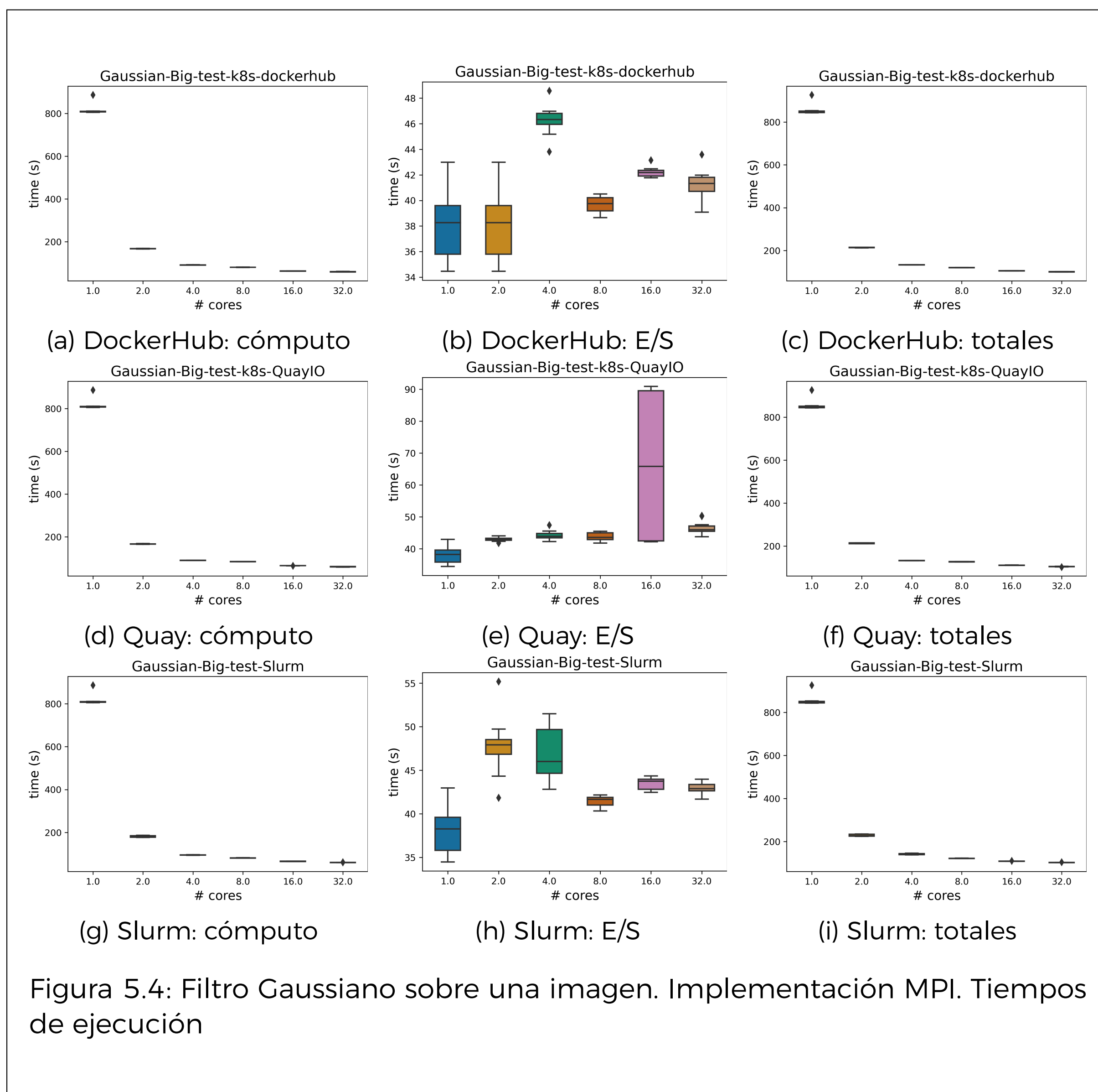
Figura 5.3: Random Forest Regressor. Speedup y eficiencia

5.3. Algoritmos de filtro Gaussiano sobre imágenes.

En esta sección se aborda el estudio de rendimiento de una implementación paralela del algoritmos de filtro Gaussiano sobre imágenes. Nuestro objetivo es medir el rendimiento de una implementación secuencial y dos implementaciones paralelas utilizando OpenMP y MPI. Realizaremos un estudio similar al efectuado en la sección anterior y calculando las mismas métricas de rendimiento.

5.3.1. Análisis de rendimiento de la implementación MPI

Presentamos a continuación los resultados sobre los códigos relacionados con el filtro Gaussiano en imágenes implementado con MPI. Utilizaremos como en el caso anterior un entorno de ejecución basado en Kubernetes (K8s) con imágenes de contenedores desplegadas en Docker Hub y Quay.io. Para comparar con la ejecución tradicional se utilizará un sistema de colas Slurm. En la figura 5.4 se muestran los tiempo de de cómputo, E/S y totales para esta implementación y con estos tres entornos de ejecución



(a) DockerHub: cómputo (b) DockerHub: E/S (c) DockerHub: totales

(d) Quay: cómputo (e) Quay: E/S (f) Quay: totales

(g) Slurm: cómputo (h) Slurm: E/S (i) Slurm: totales

Figura 5.4: Filtro Gaussiano sobre una imagen. Implementación MPI. Tiempos de ejecución

En primer lugar, cuando observamos la línea referente a la ejecución con imágenes de Docker Hub, encontramos que sus tiempos de cómputo (figura 5.4a) siguen una distribución lógica: cuanto mayor sea el número de cores que utilizamos, mayor será el rendimiento. Analizando los tiempos de E/S (figura 5.4b) observamos que se mantiene constantes en torno a 40-45s, con más variabilidad cuando utilizamos uno o dos cores y por debajo del coste asociado al cómputo. Esto explica que el tiempo total (figura 5.4c)) escale en consonancia con el tiempo de cómputo hasta casi 32 cores. Para este valor, los tiempos de cómputo y entrada salida son similares, con lo que subir el número de cores ya empieza a no compensar.

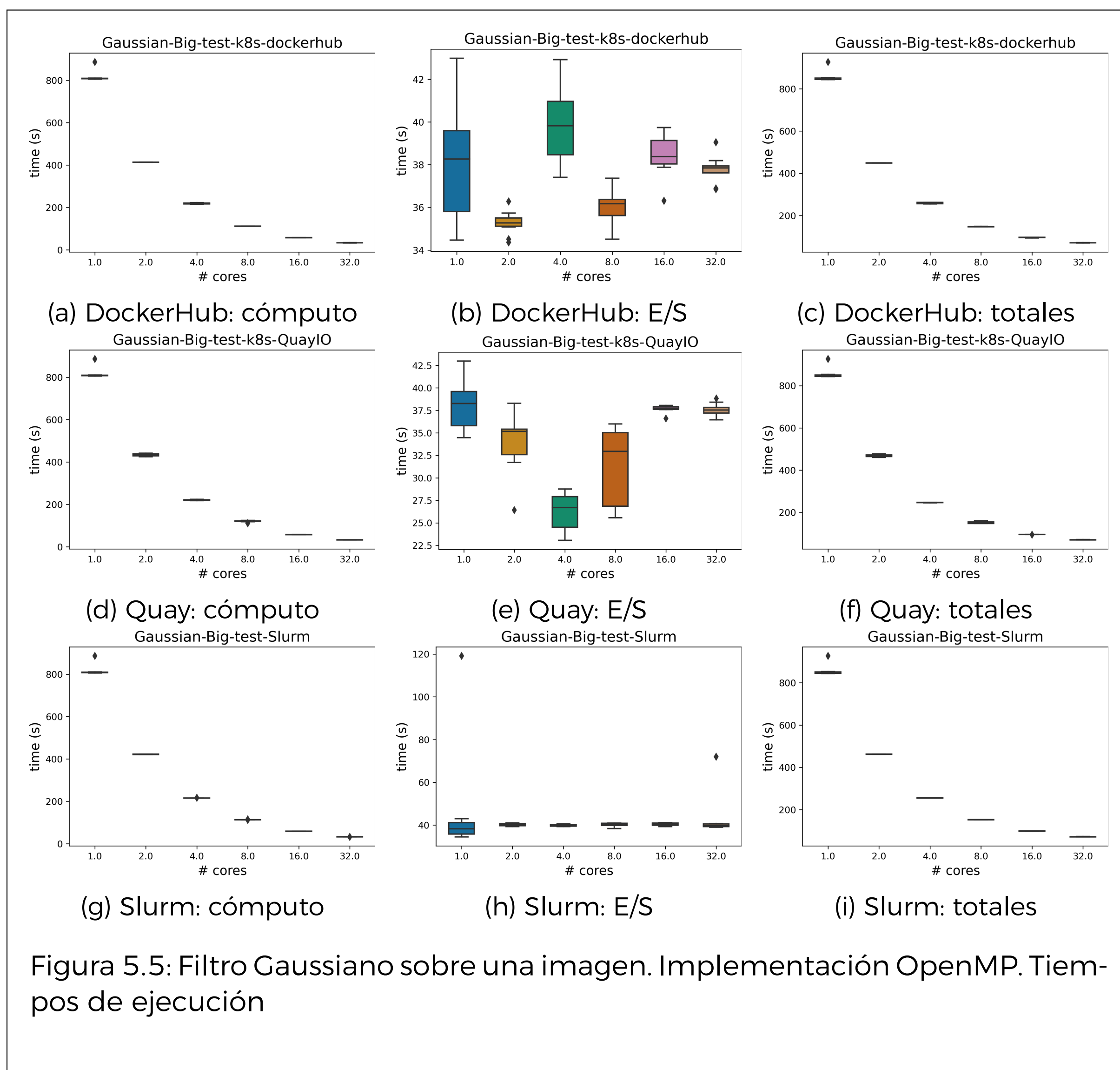
Analizando los tiempos de la ejecución en Kubernetes con imágenes de Quay.io, podremos encontrar algo muy similar al caso anterior. Nuevamente la estructura para el tiempo de cómputo (figura 5.4d), sigue una distribución lógica (menor tiempo de ejecución cuanto mayor es el número de cores). En cuanto a los tiem-

pos de E/S (figura 5.4e) se mantienen también constantes y en torno a 40-45s como en el caso anterior. Sin embargo, existe una variabilidad muy grande en el caso de 8 cores, probablemente provocado por alguna mala alineación de datos en la jerarquía de memoria. El tiempo total (figura 5.4f) registra una distribución coherente con lo hallado en el tiempo de cómputo hasta prácticamente 32 cores, por lo que concluimos que los efectos que observamos en E/S, no repercuten significativamente en el proceso total.

Por último, cuando hacemos referencia a los tiempos de ejecución en el sistema de colas (Slurm) encontramos una distribución lógica tanto para el tiempo de cómputo (figura 5.4g), como para el tiempo total (figura 5.4i), y en sintonía con los casos anteriores. Observando el comportamiento para el tiempo de E/S (figura 5.4h), encontramos variabilidad algo superior para las medidas de 1 y 4 cores. Estos efectos pueden explicarse por alguna interferencia de medida que debemos precisar, o efectos otra vez de la alineación de datos en la jerarquía de memoria. En cualquier caso, esas variaciones podrían quedar dentro del error de medida cuando consideramos los tiempos totales.

5.3.2. Análisis de rendimiento de la implementación OpenMP

Nos proponemos en esta sección estudiar el tiempo de cómputo, los tiempos de E/S y el tiempo total del proceso de ejecutar un filtro Gaussiano sobre una imagen implementado en este caso en C y anotado con directivas OpenMP que permiten su paralelización. Se han ejecutado sobre el mismo conjunto de entornos utilizados en el proyecto: Kubernetes con imágenes en Docker Hub y Quay.io y ejecución tradicional con el sistema de colas. En la figura 5.5 se muestran los resultados obtenidos.



(a) DockerHub: cómputo (b) DockerHub: E/S (c) DockerHub: totales

(d) Quay: cómputo (e) Quay: E/S (f) Quay: totales

(g) Slurm: cómputo (h) Slurm: E/S (i) Slurm: totales

Figura 5.5: Filtro Gaussiano sobre una imagen. Implementación OpenMP. Tiempos de ejecución

En primera instancia, como se puede apreciar en la primera línea de la figura 5.5, obtenemos que los tiempos de cómputo (figura 5.5a) y el tiempo total (figura 5.5c) de la ejecución en Kubernetes con imágenes de Docker Hub. Tal y como ocurre en el caso MPI, en este caso se obtienen resultados de escalabilidad razonables. No tan acentuados como en MPI cuando pasamos de uno a dos cores, pero se mantiene una progresión descendente en el tiempo de cómputo a medida que aumentamos el número de cores. No obstante, cuando hacemos alusión al tiempo de E/S (figura 5.5b) tenemos efectos de variabilidad algo superiores. Sigue manteniéndose estable entre 35-40s (algo menores que en el caso de MPI), pero en todas las ejecuciones se observan variaciones en el tiempo de E/S. Si las ponemos en contexto del tiempo total de ejecución podemos considerarlas también dentro de los errores de medida.

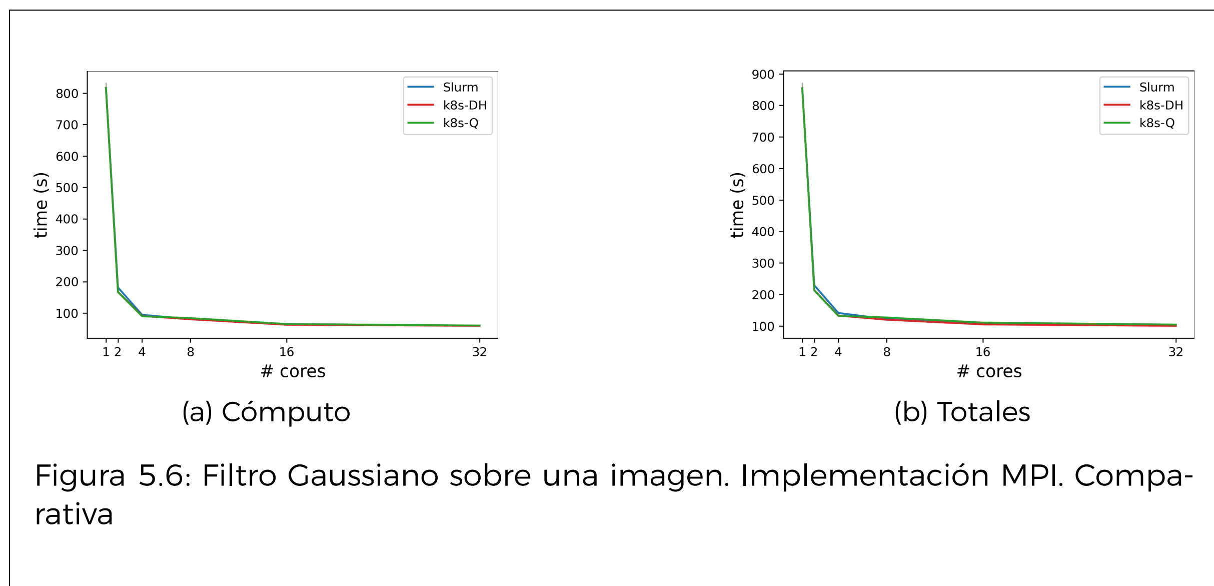
En relación a los tiempos de computo de Kubernetes con imágenes de contenedores de Quay.io que vemos en la figura 5.5d, nos encontramos con situaciones similares al caso de Docker Hub y con una escalabilidad en los tiempos de cómputo prácticamente iguales al caso anterior. Si nos centramos en la figura 5.5e) encontramos que los tiempos de E/S presentan irregularidad es en los mismo rangos que para Docker Hub, quizás algo superiores al hacer uso de 8 cores. En la figura 5.5f se ilustra el tiempo total de todo el proceso que, nuevamente, se torna en una distribución lógica y con la escalabilidad esperada al aumentar el número de cores. Esto corrobora que la anomalía surgida en los tiempos de E/S tampoco repercuten en gran medida al proceso total, si acaso influyen para 32 cores donde los tiempos de cómputo y E/S se equiparan.

Finalmente, en la figura 5.5g se muestran los tiempos de ejecución con un sistema de colas (Slurm). Volvemos a observar el mismo comportamiento que para los casos anteriores, pero en este caso, si nos fijamos en los tiempos de E/S (figura 5.5h), vemos una ejecución mucho más estable, pero con la presencia de algunos outliers. Con respecto al tiempo total, no se observa una gran influencia de los tiempos de E/S a no ser que nos aproximemos a utilizar 32 cores.

5.3.3. Comparativa de los entornos de ejecución

Igual que hicimos en la sección anterior, vamos a realizar una comparativa de los entornos de ejecución para estas dos implementaciones del algoritmo de filtro Gaussiano sobre imágenes en MPI y OpenMP. A partir de este proceso, podríamos sacar conclusiones sobre la influencia de los entornos de ejecución respecto del rendimiento de esta implementación.

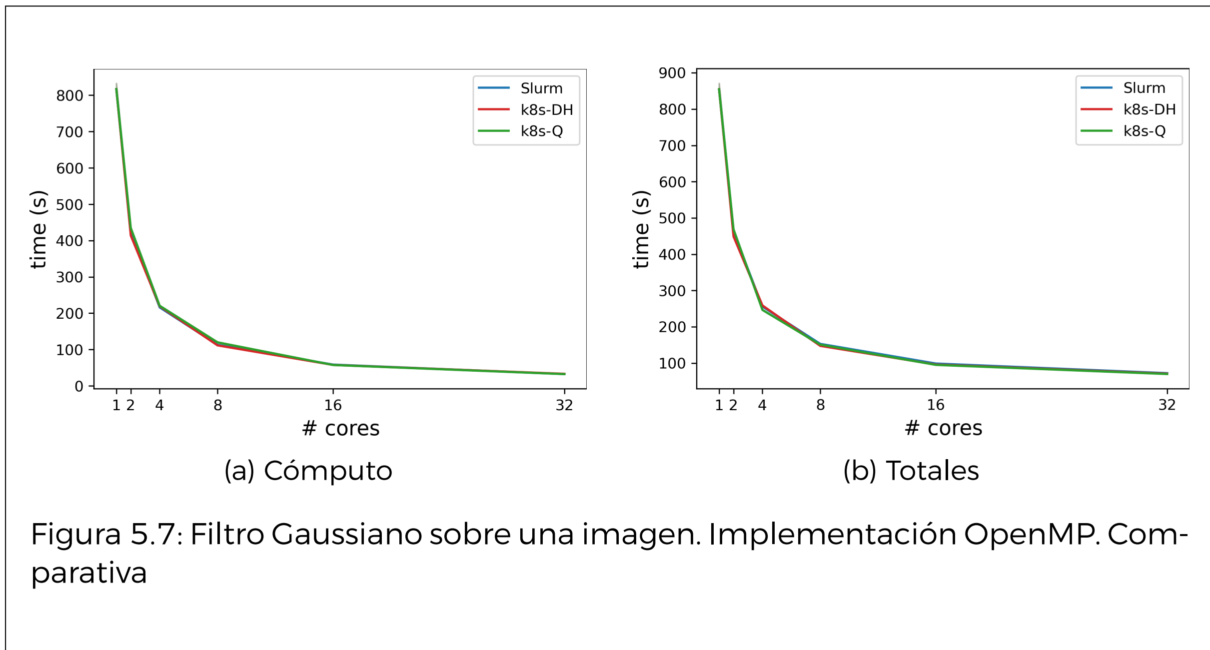
Con respecto a la implementación en MPI, observamos en la la figura 5.6 una comparativa de los tres entornos de ejecución estudiados. Nos centramos en este caso en los tiempos de cómputo y totales, que reflejan el rendimiento del algoritmo al utilizar un nodo HPC como plataforma de ejecución.



(a) Cómputo (b) Totales

Figura 5.6: Filtro Gaussiano sobre una imagen. Implementación MPI. Comparativa

Observamos en la figura 5.6 que, tanto en el tiempo de cómputo (figura 5.6a) como en el tiempo total (figura 5.6b), tenemos unos resultados prácticamente iguales. No hay ninguna ventaja en este aspecto por utilizar uno u otro entorno de ejecución. Por tanto, podríamos afirmar que la utilización de Kubernetes con imágenes tanto en Docker Hub como en Quay.io, no producen ninguna desventaja en cuanto a los tiempos de ejecución y sin embargo disponemos de diversas ventajas ajenas al rendimiento, tales como una mayor facilidad de administración, mayor seguridad (se dispone de un sistema operativo con una única función), etc.



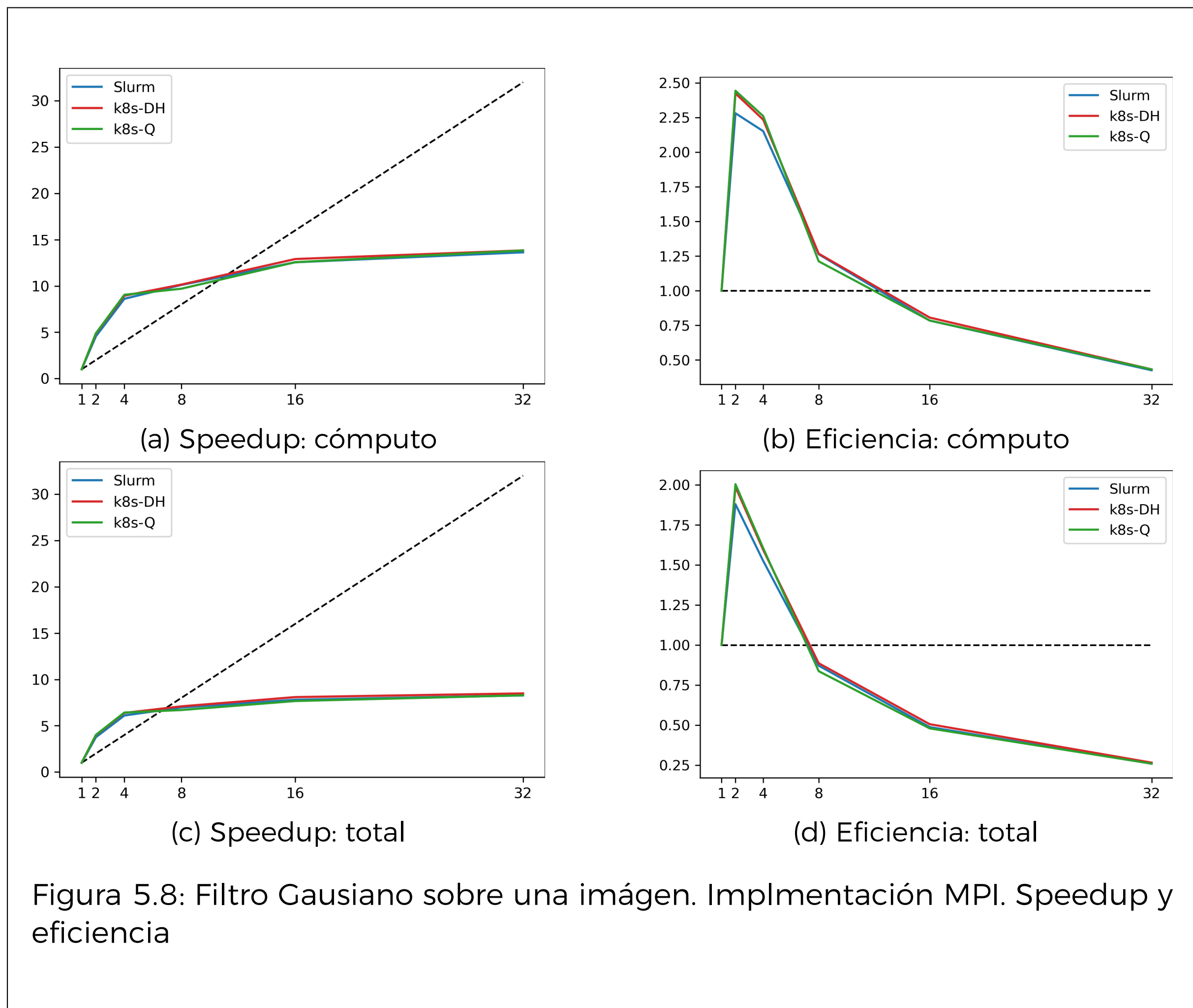
(a) Cómputo

(b) Totales

Figura 5.7: Filtro Gaussiano sobre una imagen. Implementación OpenMP. Comparativa

En en caso de la implementación OpenMP de algoritmo del filtro Gaussiano, tenemos resultados muy parecidos al caso MPI tal y como se muestran en la figura 5.7. No hay ningún entorno de ejecución que aventaje a otro y los mismos argumentos que se han mencionado en el caso de MPI son aplicables a esta implementación en OpenMP del algoritmo del filtro Gaussiano sobre imágenes.

De forma similar al estudio realizado con el algoritmo Random Forest, hemos calculado las métricas de *speedup* y *eficiencia* para las implementaciones del filtro Gaussiano en MPI y OpenMP. En la figura 5.8 se muestra una representación gráfica de estas métricas para la implementación MPI. Esta forma de visualizar el rendimiento de una aplicación paralela nos indica rápidamente cual es el número de cores óptimo para la ejecución de este código. Observamos que en torno a 8 cores la eficiencia decae rápidamente y no compensa seguir asignando recursos para la ejecución de este código. Aunque las gráficas de cómputo todavía tengamos eficiencias razonables hasta con 32 cores, la contribución del efecto de la E/S hace que el rendimiento decaiga y no compense utilizar más de 8 cores en este caso.



(a) Speedup: cómputo

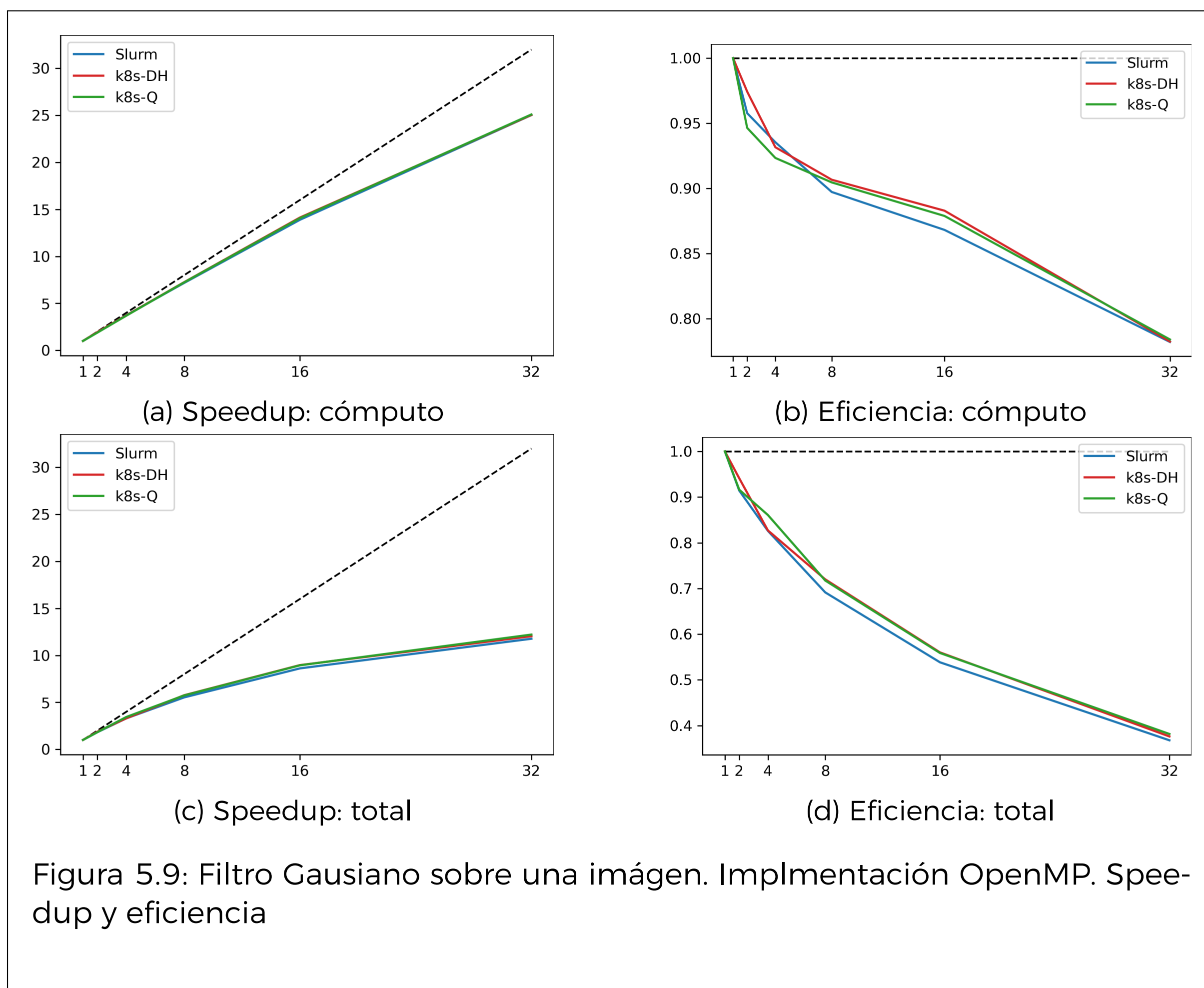
(b) Eficiencia: cómputo

(c) Speedup: total

(d) Eficiencia: total

Figura 5.8: Filtro Gausiano sobre una imágen. Implmentación MPI. Speedup y eficiencia

Cabe destacar en estas figuras el efecto de superlinealidad que observamos para la implementación MPI del algoritmo de filtro Gausiano. Los efectos de jerarquía de memoria son muy fuertes en esta implementación haciendo que por debajo de 8 cores tengamos ese efecto. Lo que justifica también utilizar ese límite como recurso a asignar a los contenedores que ejecutemos en el cluster de kubernetes para este tamaño de problema.



(a) Speedup: cómputo

(b) Eficiencia: cómputo

(c) Speedup: total

(d) Eficiencia: total

Figura 5.9: Filtro Gausiano sobre una imágen. Implmentación OpenMP. Speedup y eficiencia

En el caso de la implementación OpenMP, tal y como podremos apreciar en la figura 5.9 se pueden sacar las siguientes conclusiones. Respecto del *speedup* de cómputo (figura 5.9a) y de la eficiencia del cómputo (5.9b) podemos decir que este algoritmo escala muy bien y de forma más regular que la implementación MPI, obteniendo eficiencias entorno al 80 %. Sin embargo, este rendimiento no se ve reflejado luego cuando obtenemos las métricas para los tiempos totales de ejecución del algoritmo. Como vemos reflejado en las figuras de *speedup* total, figura 5.9c, y la figura de eficiencia total 5.9d los rendimientos caen rápidamente a medida que aumentamos el número de cores. Obtenemos eficiencias por debajo del 40 % para 32 cores que hace que esta implementación no sea razonable utilizarla con esa cantidad de recursos. Esta visualización del rendimiento ilustra muy bien el efecto de la contribución de los tiempos de E/S en el rendimiento de la aplicación.

Capítulo 6

Conclusiones y líneas futuras.

6.1. Conclusiones

En este proyecto hemos abordado el estudio de diversos entornos de ejecución basados en contenedores para computar cargas de trabajo habituales en computación de altas prestaciones. Nos planteamos como objetivo la utilización de un entorno como Kubernetes para realizar esta tarea y analizar su rendimiento frente al uso de una ejecución tradicional basada en un sistema de colas. Hemos desarrollado GraphClood, una aplicación web que permite realizar cómputo en la nube y utilizar estas tecnologías de contenerización para ejecutar diferentes algoritmos.

Según el análisis que hemos llevado a cabo, apoyado por los datos que hemos obtenido a en los capítulos anteriores, podemos decir que los entornos de ejecución con Kubernetes basados en imágenes de contenedores registradas en Quay.IO y Docker Hub no perjudican el rendimiento. De hecho, nos aportan varias ventajas como la posibilidad de abstraer las aplicaciones del sistema operativo al poder instalar paquetes y otras dependencias dentro del contenedores y que estos no afecten a nuestro sistema operativo, una mayor seguridad al no tener que instalar software específico de las aplicaciones en el sistemas base, etc. Podemos concluir, por tanto, que Kubernetes sería una opción viable frente a la ejecución tradicional basadas en sistemas de colas.

Asimismo, como hemos podido observar previamente, el número de cores a especificar en los requisitos de los pods de Kubernetes deben adaptarse al problema y al tamaño del mismo para optimizar los recursos del servidor HPC utilizado. Este es un factor que debe tenerse en cuenta en el momento de dimensionar la adquisición de equipamiento o en la planificación de tareas a ejecutar desde la aplicación GraphCloud.

6.2. Líneas futuras

Este apartado engloba toda aquella información posible sobre pasos potenciales que puedan realizarse de forma futura y que pueden abrir líneas de investigación para dotar al proyecto de una visión más completa.

En primer lugar, cabría destacar la implementación de una monitorización de la energía consumida a través de Kubernetes, mediante la que se posibilitaría el comparar la energía que se consume utilizando los dos entornos de Kubernetes utilizados y el sistema de colas. Esta comparativa permitiría comprobar cuál de las soluciones en términos de vatios consumidos sería la más rentable.

Asimismo, otro paso de cara a una investigación futura sería analizar la planificación que Kubernetes hace con los contenedores y máquinas virtuales sobre las máquinas físicas. Hay oportunidades de mejora al asignar los recursos disponibles en función de la aplicación a ejecutar.

Por otro lado, destacaríamos la posibilidad de la ejecución de otras herramientas, como podría ser el caso del NVIDIA Container Toolkit, un entorno de contenedores para aplicaciones que pueden ejecutarse sobre GPUs.

Otra posible línea de investigación estaría asociada al análisis y modelado de las aplicaciones con el fin de optimizar el número de cores que se deben utilizar para ejecutarlas.

Por último, destacamos la posibilidad de utilizar otros códigos, no solo utilizar el procesamiento de imágenes y el procesamiento de datos como se ha realizado hasta ahora, sino ampliar nuestro campo y utilizar otros tipos de funciones y, otro tipo de lenguajes de programación.

Capítulo 7

Summary and Conclusions.

7.1. Conclusions

In this project we have addressed the study of various container based runtimes to compute common workloads in High Performance Computing (HPC). Our objective is to use an environment such as Kubernetes to perform this task and analyze its performance against the use of a traditional execution based on a queuing system. We have developed GraphClood, a web application that allows computing in the cloud and uses these containerization technologies to execute different algorithms.

Based on the analysis we have carried out, supported by the data we have obtained in the previous chapters, we can say that runtime environments with Kubernetes based on container images registered in Quay.io and Docker Hub do not affect performance. In fact, they provide us with several advantages such as the possibility of abstracting the applications from the operating system by being able to install packages and other dependencies within the containers and that these do not affect our operating system, greater security by not having to install specific software from the applications in the base systems, etc. We can therefore conclude that Kubernetes would be a viable option compared to the traditional execution based on queuing systems.

Also, as we have seen previously, the number of cores to be specified in the requirements of the Kubernetes pods must be adapted to the problem and the size of the same to optimize the resources of the HPC server used. This is a factor that must be taken into account when sizing the purchase of equipment or when planning tasks to be executed from the GraphCloud application.

7.2. Future Work

This section includes all possible information on potential steps that can be carried out in the future and that can open lines of research to provide the project with a more complete vision.

In the first place, it should be noted the implementation of a monitoring of the energy consumed through Kubernetes, through which it would be possible to compare the energy consumed using the two Kubernetes environments used and the queuing system. This comparison would allow us to check which of the solutions in terms of watts consumed would be the most profitable.

Also, another step for future research would be to analyze the planning that Kubernetes does with containers and virtual machines on top of physical machines. There are opportunities for improvement by allocating available resources depending on the application to be run.

On the other hand, we would highlight the possibility of running other tools, such as the NVIDIA Container Toolkit, a container environment for applications that can run on GPUs.

Another possible line of research would be associated with the analysis and modeling of applications in order to optimize the number of cores that must be used to execute them.

Finally, we highlight the possibility of using other codes, not only using image processing and data processing as has been done so far, but also expanding our field and using other types of functions and other types of programming languages.

Capítulo 8

Presupuesto

8.1. Justificación del presupuesto

En relación al presupuesto requerido en este proyecto, debemos de hacer hincapié en la división del mismo, dando como resultado tres partes diferenciadas: a) creación de la aplicación; b) despliegue; c) comparativa entre herramientas. Asimismo, cada una de las partes constituyentes supondrá un coste distinto que, se desglosarán en el cuadro 8.1.

TIPO	CANTIDAD	COSTE UNIDAD	COSTE TOTAL
Aplicación para prueba del despliegue (GraphCloud)	**392 horas**		**5424 €**
Investigación del proyecto a desarrollar	16 horas	12 €/h	192 €
Prototipo del proyecto	16 horas	12 €/h	192 €
Frontend del proyecto	120 horas	12 €/h	1440 €
Backend del proyecto	240 horas	15 €/h	3600 €
Creando Infraestructura	**132 horas**		**5280 €**
Investigación para el desarrollo del proyecto	32 horas	20 €/h	640 €
Creando Infraestructura básica	60 horas	20 €/h	1200 €
Creando Infraestructura en IAAS	52 horas	20 €/h	1040 €
Creando Infraestructura definitiva	120 horas	20 €/h	2400 €
Comparativa de las herramientas	**30 horas**	**40 €/h**	**1200 €**
TOTAL	**554 horas**		**11904 €**

Cuadro 8.1: Presupuesto

En el cuadro 8.1, se han señalado cada uno de los precios para el proyecto, destacando cada uno de los subproyectos requeridos. En cuanto al presupuesto por cada parte, hemos podido observar que la aplicación recoge una suman de 5.424€ (392 horas), el despliegue unos 5.280€ (132 horas) y, la comparativa de herramientas un monto de 1.200€(30 horas), por lo que el presupuesto total del proyecto sería de 11.904€ (554 horas).

Bibliografía

[1] Naweiluo Zhou, Yiannis Georgiou, Marcin Pospieszny, Li Zhong, Huan Zhou, Christoph Niethammer, Branislav Pejak, Oskar Marko, and Dennis Hoppe. Container orchestration on HPC systems through kubernetes. *J. Cloud Comput.*, 10(1):16, 2021.

[2] Sergio López-Huguet, J. Damiàn Segrelles Quilis, Marek Kasztelnik, Marian Bubak, and Ignacio Blanquer. Seamlessly managing HPC workloads through kubernetes. In Heike Jagode, Hartwig Anzt, Guido Juckeland, and Hatem Ltaief, editors, *High Performance Computing - ISC High Performance 2020 International Workshops, Frankfurt, Germany, June 21-25, 2020, Revised Selected Papers*, volume 12321 of *Lecture Notes in Computer Science*, pages 310–320. Springer, 2020.

[3] Kimitoshi Takahashi, Kento Aida, Tomoya Tanjo, and Jingtao Sun. A portable load balancer for kubernetes cluster. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region, HPC Asia 2018, Chiyoda, Tokyo, Japan, January 28-31, 2018*, pages 222–231. ACM, 2018.

[4] Docker. What is a container? `https://www.docker.com/resources/what-container`. Accessed: 2020-11-17.

[5] Microsoft. ¿qué es un contenedor? `https://azure.microsoft.com/es-es/overview/what-is-a-container/`. Accessed: 2020-11-17.

[6] Red Hat. El concepto de los contenedores de linux. `https://www.redhat.com/es/topics/containers`. Accessed: 2020-11-17.

[7] Javier Gazar. ¿qué es docker? ¿para qué se utiliza? explicado de forma sencilla. `https://www.javiergarzas.com/2015/07/que-es-docker-sencillo.html`. Accessed: 2020-11-17.

[8] Docker. Developers bring their ideas to life with docker. `https://www.docker.com/why-docker`. Accessed: 2020-11-17.

[9] Red Hat. Qué es docker. `https://www.redhat.com/es/topics/containers/what-is-docker`. Accessed: 2020-11-17.

[10] Javier Garzas. ¿qué es eso de los microservicios? `https://www.javiergarzas.com/2015/06/microservicios.html`. Accessed: 2020-11-17.

[11] Kubernetes. ¿qué es kubernetes? https://kubernetes.io/es/docs/concepts/overview/what-is-kubernetes/. Accessed: 2020-11-17.

[12] BeServices. Kubernetes: qué es y qué aporta a nivel de programación. https://www.beservices.es/kubernetes-que-es-n-5364-es. Accessed: 2020-11-17.

[13] Kubernetes. Orquestación de contenedores para producción. https://kubernetes.io/es/. Accessed: 2020-11-17.

[14] ICM. Las ventajas de kubernetes. https://www.icm.es/2020/03/07/kubernetes. Accessed: 2020-11-17.

[15] Microsoft. Kubernetes o docker. https://azure.microsoft.com/es-es/topic/kubernetes-vs-docker/. Accessed: 2020-11-17.

[16] IONOS. Minikube: lo mejor de kubernetes con el mínimo esfuerzo. https://www.ionos.es/digitalguide/servidores/herramientas/kubernetes-minikube/. Accessed: 2020-11-17.

[17] Max Brenner. Minikube vs. kind vs. k3s - what should i use? https://brennerm.github.io/posts/minikube-vs-kind-vs-k3s.html. Accessed: 2020-11-17.

[18] Cloud Native MX. Iniciando kubernetes con minikube. https://cloudnative.mx/como-empezar-con-kubernetes-usando-minikube/. Accessed: 2020-11-17.

[19] Minikube. Minikube: Tip and tricks para minikube en entornos hyper-v. https://www.ionos.es/digitalguide/servidores/herramientas/kubernetes-minikube/. Accessed: 2020-11-17.

[20] Kubernetes-sigs. Kind is a tool for running local kubernetes clusters using docker container ”nodes”. https://github.com/kubernetes-sigs/kind. Accessed: 2020-11-17.

[21] Kevin Blanco. Ejecutando kind dentro de un clúster de kubernetes para integración continua. https://medium.com/kubernetes-costa-rica/ejecutando-kind-dentro-de-un-cl%C3%BAster-de-kubernetes-para-integraci%C3%B3n-continua-b2559c8d7e47. Accessed: 2020-11-17.

[22] Kubernetes. Wsl+docker: Kubernetes on the windows desktop. https://kubernetes.io/blog/2020/05/21/wsl-docker-kubernetes-on-the-windows-desktop/. Accessed: 2020-11-17.

[23] Guillermo Alvarado. 6 herramientas para desplegar un clúster de kubernetes. https://galvarado.com.mx/post/6-herramientas-para-desplegar-un-cluster-de-kubernetes/. Accessed: 2020-11-17.

[24] Elarraydejota. Probando kubespray. https://www.elarraydejota.com/probando-kubespray/. Accessed: 2020-11-17.

[25] RedHat. Drive automation across open hybrid cloud deployments. https://www.ansible.com/. Accessed: 2020-11-17.

[26] Alexellis. How do you say it? ketchup, as in tomato. https://github.com/alexellis/k3sup. Accessed: 2020-11-17.

[27] Ignacio Van Droogenbroeck. Cómo desplegar un clúster de kubernetes en 60 segundos con k3sup. https://cduser.com/como-desplegar-un-cluster-de-kubernetes-en-60-segundos-con-k3sup/. Accessed: 2020-11-17.

[28] Kubernetes. Instalar y configurar kubectl. https://kubernetes.io/es/docs/tasks/tools/install-kubectl/. Accessed: 2020-11-17.

[29] Jessica Cherr. Basic kubectl and helm commands for beginners. https://opensource.com/article/20/2/kubectl-helm-commands. Accessed: 2020-11-17.

[30] Paulbouwer. Hello kubernetes! https://github.com/paulbouwer/hello-kubernetes. Accessed: 2020-11-17.

[31] Jonatan Lobeto. Subsistema linux en windows 10: ¿qué es? ¿para qué sirve? y ¿cómo instalarlo? https://rincondelatecnologia.com/subsistema-linux-en-windows-10-que-es-para-que-sirve-y-como-instalarlo/. Accessed: 2020-11-17.

[32] Rubén Velasco. Aprende a usar wsl, el subsistema de windows 10 para linux. https://www.softzone.es/windows-10/como-se-hace/subsistema-windows-linux/. Accessed: 2020-11-17.

[33] Microsoft. Guía de instalación del subsistema de windows para linux para windows 10. https://docs.microsoft.com/es-es/windows/wsl/install-win10. Accessed: 2020-11-17.

[34] Docker. Docker desktop for windows user manual. https://docs.docker.com/docker-for-windows/. Accessed: 2020-11-17.

[35] Óscar Villacampa. Qué es docker y para qué sirve. https://www.ondho.com/que-es-docker-para-que-sirve/. Accessed: 2020-11-17.

[36] Desdelinux. Docker hub: Aprendiendo un poco más sobre la tecnología docker. https://blog.desdelinux.net/docker-hub-aprendiendo-tecnologia-docker/. Accessed: 2020-11-17.

[37] Docker. Build and ship any application anywhere. https://hub.docker.com/. Accessed: 2020-11-17.

[38] Red Hat. Quay [builds, analyzes, distributes] your container images. https://quay.io/. Accessed: 2020-11-21.

[39] Red Hat. Red hat: Quay. https://www.redhat.com/es/technologies/cloud-computing/quay. Accessed: 2020-11-21.

[40] CHRIS TOZZI. Docker hub, quay and beyond: Which container registry is right for you? https://sweetcode.io/docker-hub-quay-beyond-container-registry/. Accessed: 2020-11-21.

[41] MongoDB. The database for modern applications. https://www.mongodb.com/. Accessed: 2020-01-14.

[42] Ángel Robledano. Qué es mongodb. https://openwebinars.net/blog/que-es-mongodb/. Accessed: 2020-01-14.

[43] TutorialsPoint. Node.js - express framework. https://www.tutorialspoint.com/nodejs/nodejs_express_framework.htm. Accessed: 2020-01-14.

[44] Eneko. ¿qué es express.js? instalación y primeros pasos. https://enekodelatorre.com/expressjs-instalacion-primeros-pasos/. Accessed: 2020-01-14.

[45] Vue.js. Vue.js: The progressive javascript framework. https://vuejs.org/. Accessed: 2020-01-14.

[46] Manz. ¿qué es vue.js? https://vuejs.org/. Accessed: 2020-01-14.

[47] Node.js. Node.js® is a javascript runtime built on chrome's v8 javascript engine. https://nodejs.org/en/. Accessed: 2020-01-14.

[48] Judit Cabana. ¿qué es node.js y para qué sirve? https://www.drauta.com/que-es-nodejs-y-para-que-sirve. Accessed: 2020-01-14.

[49] Source-Making. Strategy design pattern. https://sourcemaking.com/design_patterns/strategy. Accessed: 2021-03-08.

[50] Refactoring-Guru. Open mpi: Open source high performance computing. https://refactoring.guru/design-patterns/strategy. Accessed: 2021-03-08.

[51] Pandas. Pandas is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the python programming language. https://pandas.pydata.org/. Accessed: 2020-01-14.

[52] Sckit-Learn. Scikit-learn: Machine learning in python. https://scikit-learn.org/stable/. Accessed: 2020-01-14.

[53] Matplotlib. Matplotlib: Visualization with python. https://matplotlib.org/. Accessed: 2020-01-14.

[54] Joblib. Embarrassingly parallel for loops. https://joblib.readthedocs.io/en/latest/parallel.html. Accessed: 2020-01-14.

[55] Victor Roman. Aprendizaje supervisado: Introducción a la clasificación y principales algoritmos. https://medium.com/datos-y-ciencia/aprendizaje-supervisado-introducción-a-la-clasificación-y-principales-algoritmos Accessed: 2021-03-12.

[56] Sckit-Learn. The iris dataset. https://scikit-learn.org/stable/auto_examples/datasets/plot_iris_dataset.html. Accessed: 2021-03-12.

[57] AprendeIA. Creando un clasificador para la flor iris con python. https://aprendeia.com/machine-learning-clasificador-flor-iris-python/. Accessed: 2021-03-12.

[58] Sckit-Learn. These examples illustrate the main features of the releases of scikit-learn. https://scikit-learn.org/stable/auto_examples/index.html#examples. Accessed: 2020-01-14.

[59] Sckit-Learn. Examples concerning the sklearn.cluster module. https://scikit-learn.org/stable/auto_examples/index.html#cluster-examples. Accessed: 2020-01-14.

[60] Open-MPI. Open mpi: Open source high performance computing. https://www.open-mpi.org/. Accessed: 2020-01-14.

[61] Blaise Barney. Message passing interface (mpi). https://computing.llnl.gov/tutorials/mpi/. Accessed: 2020-01-14.

[62] Inovanex. ¿qué es htop y como se usa? https://blog.inovanex.com/htop-se-usa/. Accessed: 2021-03-13.

[63] OpenMP. Openmp 5.1 released with vital usability enhancements. https://www.openmp.org/. Accessed: 2020-01-14.

[64] Arnab Chakraborty. What is openmp? https://www.tutorialspoint.com/what-is-openmp. Accessed: 2020-01-14.

[65] Kubernetes. Create a deployment. https://kubernetes.io/es/docs/concepts/workloads/controllers/deployment/. Accessed: 2021-03-13.

[66] Kubernetes. What is ingress? https://kubernetes.io/docs/concepts/services-networking/ingress/. Accessed: 2021-03-13.

[67] Kubernetes. Service resources. https://kubernetes.io/docs/concepts/services-networking/service/. Accessed: 2021-03-13.

[68] Alex Ellis. k3sup (said 'ketchup'). https://github.com/alexellis/k3sup. Accessed: 2020-01-14.

[69] Alex Ellis. Learn how to build your own kubernetes homelab with raspberry pi. https://blog.alexellis.io/raspberry-pi-homelab-with-k3sup/. Accessed: 2021-03-14.

[70] Kubernetes. Kubernetes meets high-performance computing. https://kubernetes.io/blog/2017/08/kubernetes-meets-high-performance/. Accessed: 2021-03-14.

[71] Altair. Cloud migration, automation, and spend management for hpc. https://www.altair.com/navops-launch/. Accessed: 2021-03-14.

[72] Francisco Almeida, Domingo Giménez, José Miguel Mantas, and Antonio M. Vidal. *Introducción a la programación paralela*. CENGAGE Learning Paraninfo, 1 edition, 2008.