



Universidad de Castilla-La Mancha  
Escuela Superior de Ingeniería Informática

Trabajo Fin de Grado  
Grado en Ingeniería Informática  
Tecnología Específica de  
Tecnologías de la Información

# Ejecución de aplicaciones HPC utilizando contenedores

Antonio Jacobé Galdón

Julio, 2022





Trabajo Fin de Grado  
Grado en Ingeniería Informática  
Tecnología Específica de  
Tecnologías de la Información

# Ejecución de aplicaciones HPC utilizando contenedores

**Autor:** Antonio Jacobé Galdón

**Tutor:** Diego Cazorla López

Julio, 2022



Dedicado a mi familia, mis padres Antonio y Juani y mi hermano Juan Manuel. Siempre me han transmitido su apoyo y ánimo y me han demostrado que se puede salir adelante siempre con ganas y esfuerzo. Sin ellos nada de esto hubiera sido posible.

A mi tutor Diego por guiarme durante todo el proceso y estar dispuesto a ofrecerme su ayuda en cualquier momento.

A todos los amigos reunidos a lo largo de la carrera y por todas las historias vividas.



## Declaración de Autoría

---

Yo, Antonio Jacobé Galdón con DNI 21035631S, declaro que soy el único autor del trabajo fin de grado titulado **“Ejecución de aplicaciones HPC utilizando contenedores”** y que el citado trabajo no infringe las leyes en vigor sobre propiedad intelectual y que todo el material no original contenido en dicho trabajo está apropiadamente atribuido a sus legítimos autores.

Albacete, a 08/07/2022

Fdo: Antonio Jacobé Galdón





## Resumen

---

En los últimos años la evolución de las tecnologías de contenedores ha cambiado la forma de desplegar software en cualquier servidor, llegando a ser uno de los mecanismos más comunes. No obstante, la adición de aplicaciones High Performance Computing (HPC) en contenedores todavía no está asentada pues, aunque se proporciona una portabilidad, no satisfacen los requisitos de rendimiento y seguridad.

Durante el desarrollo de este trabajo de estudio se aportan dos aplicaciones HPC y analizaremos las ventajas que pueden ofrecer los contenedores con un código HPC. No obstante, en entornos de HPC nos encontramos dificultades para preparar y configurar el ambiente de trabajo. Es por ello, por lo que en este trabajo se proponen soluciones software basadas en tecnologías como Docker, Kubernetes, Docker Swarm o Singularity para construir un clúster HPC. Estos se van a comunicar mediante Message Passing Interface (MPI) y generaremos automáticamente las máquinas mediante Vagrant.

En todo momento se ha intentado facilitar al lector la información y automatizar lo máximo posible toda la creación de tecnologías o ficheros usados de tal manera que se pueda reproducir paso a paso la ejecución del código de pi o el modelo WRF en cualquier entorno que se desee.



## Agradecimientos

---

En primer lugar, me gustaría transmitir mi más sincero agradecimiento a todas las personas que me han ayudado a llegar hasta aquí y han contribuido para ser capaz de haber logrado la realización de este trabajo de fin de grado. Me gustaría destacar el gran papel de mi tutor Diego, por su ayuda en la planificación, información y organización que ha llevado en todo momento. Además, agradecerle por estar siempre ahí y su implicación con este trabajo.

En segundo lugar a mi familia, sobre todo a mis padres y mi hermano por el apoyo incondicional que me han transmitido día tras día. Por no soltarme nunca de la mano y ayudarme a ser quién soy hoy en día.

En tercer lugar, a la gran familia que he conseguido durante esta etapa académica. Muchas gracias a cada uno de vosotros por formar parte de la mejor etapa de mi vida.

Por último, me gustaría expresar mi más sincero agradecimiento a la Universidad de Castilla La Mancha y a cada uno de los profesionales que me he cruzado durante la estancia en la universidad, por su acogimiento en sus aulas y hacerme sentir como si no me hubiera desplazado de mi tierra. He contraído una deuda impagable con cada una de estas personas, debido a sus virtudes como profesionales y bondad como personas. A todos y cada uno de ellos, mil gracias por hacerme llegar dónde estoy.



## Índice general

---

<b>Capítulo 1</b>	<b>Introducción</b>	<b>1</b>
1.1	Introducción	1
1.2	Objetivos	3
1.3	Estructura del proyecto	4
<b>Capítulo 2</b>	<b>Estado del Arte</b>	<b>7</b>
2.1	High Performance Computing	7
2.1.1	Funcionamiento HPC	8
2.1.2	Sectores de uso del High Performance Computing	8
2.2	Message Passing Interface	9
2.3	Contenedores	10
2.3.1	¿Cómo funcionan los contenedores?	10
2.3.2	Ventajas de uso de contenedores	11
2.3.3	Plataformas relacionadas con contenedores	12
2.3.4	Relación entre contenedor y máquina virtual	17
2.4	Contenedores + HPC	19
2.5	Vagrant	20
2.6	Weather Research and Forecasting	20
<b>Capítulo 3</b>	<b>Entorno de trabajo, herramientas y metodología utilizada</b>	<b>23</b>
3.1	Entorno de trabajo y herramientas utilizadas	23
3.2	Metodología	25
3.3	Vagrant	27

<b>Capítulo 4</b>	<b>Virtualización e implementación del cálculo de Pi</b>	<b>32</b>
4.1	Introducción	32
4.2	Docker y cálculo de Pi en dos máquinas	33
4.2.1	Imagen mpi_cpi	33
4.2.2	Distribución de ficheros	38
4.2.3	Proceso de instalación y configuración	39
4.2.4	Ejecución Docker	40
4.2.5	Ejecución del cálculo de Pi en dos máquinas virtuales	41
4.3	Kubernetes	41
4.3.1	Imagen mpi_cpi en Kubernetes	41
4.3.2	Distribución de ficheros	42
4.3.3	Proceso de instalación y configuración	42
4.3.4	Ejecución Kubernetes	44
4.4	Docker Swarm	48
4.4.1	Imagen mpi_cpi usada en Docker Swarm	48
4.4.2	Distribución de ficheros	48
4.4.3	Proceso de instalación y configuración	49
4.4.4	Ejecución Docker Swarm	50
4.5	Singularity	54
4.5.1	Imagen mpi_cpi en Singularity	54
4.5.2	Distribución de ficheros	56
4.5.3	Proceso de instalación y configuración	56
4.5.4	Ejecución Singularity	57
<b>Capítulo 5</b>	<b>Virtualización e implementación del modelo WRF</b>	<b>59</b>
5.1	Introducción	59
5.2	WRF	60
5.2.1	Distribución de ficheros	60
5.2.2	Proceso de instalación y configuración	61
5.2.3	Ejecución WRF	62
5.3	WRF + Singularity	64
5.3.1	Imagen WRF	64
5.3.2	Distribución de ficheros	65
5.3.3	Proceso de instalación y configuración	66
5.3.4	Ejecución WRF + Singularity	67

5.3.5	Ejecución en clúster Galgo	68
<b>Capítulo 6</b>	<b>Conclusiones y Trabajo Futuro</b>	<b>71</b>
6.1	Conclusiones	71
6.2	Trabajo futuro	74
<b>Capítulo 7</b>	<b>Bibliografía</b>	<b>75</b>
<b>Anexo I.</b>	<b>Planificación temporal</b>	<b>81</b>
I.1	Planificación	81





## Índice de figuras

---

Figura 1: Contenedores.....	10
Figura 2: Motor de contenedores.....	11
Figura 3: Arquitectura de Docker.....	13
Figura 4: Esquema de una máquina virtual .....	18
Figura 5: Esquema de contenedores .....	19
Figura 6: Topología desarrollada para el cálculo de pi .....	24
Figura 7: Topología para el modelo WRF.....	24
Figura 8: Etapas desarrolladas .....	26
Figura 9: Versión de Vagrant .....	27
Figura 10: Directorios de trabajo cálculo de PI .....	33
Figura 11: Pasos para ejecutar aplicación HPC en contenedor.....	35
Figura 12: Construcción de la imagen para los orquestadores.....	36
Figura 13: Imagen en Docker .....	37
Figura 14: Subida imagen a Docker Hub .....	37
Figura 15: Imagen en Docker Hub.....	37
Figura 16: Versión de Docker.....	39
Figura 17: Ejecución de la imagen hello-world .....	40
Figura 18: Ejecución cálculo de Pi en Docker.....	41
Figura 19: Ejecución MPI_cálculoPI en dos procesos .....	41
Figura 20: Nodos configurados en Kubernetes.....	44
Figura 21: Conceptos Kubernetes .....	45
Figura 22: Kubectl apply" .yaml" .....	45
Figura 23: Lista de deployments Kubernetes.....	46
Figura 24: Pods creados en Kubernetes.....	46
Figura 25: Descripción pods en Kubernetes .....	46
Figura 26: Ejecución cálculo de Pi en Kubernetes.....	48
Figura 27: Nodos de Docker Swarm.....	50
Figura 28: Creación servicio Docker Swarm.....	50

Figura 29: Servicio creado con .yaml .....	51
Figura 30: Ejecución cálculo de Pi en Docker Swarm .....	54
Figura 31: Construcción del contenedor en Singularity .....	55
Figura 32: Librerías dentro de contenedor de Singularity .....	56
Figura 33: Versión de Singularity .....	57
Figura 34: Ejecución cálculo de Pi en Singularity (Véase Código 11) .....	58
Figura 35: Directorios de trabajo WRF .....	60
Figura 36: Jasper-1.900.29 .....	61
Figura 37: Enlaces simbólicos WRF .....	62
Figura 38: Copiar datos de entrada WRF .....	62
Figura 39: Ejecución WRF sin Singularity .....	62
Figura 40: Output WRF.....	63
Figura 41: Tiempo WRF I .....	63
Figura 42: Ejecución WRF de dos procesos en un solo nodo .....	64
Figura 43: Imagen Singularity WRF .....	65
Figura 44: Enlaces simbólicos rotos WRF + Singularity .....	67
Figura 45: Copiar datos de entrada WRF + Singularity.....	67
Figura 46: Ejecución WRF con Singularity .....	67
Figura 47: Output WRF + Singularity .....	68
Figura 48: Tiempo WRF + Singularity .....	68
Figura 49: Diagrama de Gantt .....	83

## Índice de códigos

---

Código 1: VagrantFile cálculo pi en dos máquinas.....	29
Código 2: VagrantFile Kubernetes .....	30
Código 3: VagrantFile Docker Swarm.....	30
Código 4: Código DockerFile .....	34
Código 5: Docker-entrypoint.sh .....	35
Código 6: Creación deployment en Kubernetes .....	44
Código 7: Código ejec_mpi_k8s.sh .....	47
Código 8: Fichero Docker Compose .....	50
Código 9: Código ejec_mpi_dockerSwarm.sh.....	52
Código 10: Creación del contenedor en Singularity.....	55
Código 11: Código ejec_mpi_Singularity .....	57
Código 12: Creación de imagen WRF Singularity .....	65
Código 13: Código WRF en Galgo .....	69



# Capítulo 1

## Introducción

---

En este primer capítulo se presentará una breve introducción de este trabajo de fin de grado, posteriormente se definirán diversos objetivos a cumplimentar y la estructura que va a formar este documento.

### 1.1 Introducción

Aunque es cierto que hoy en día estamos escuchando más a menudo los términos de “Contenedores y HPC”, la mayoría de la gente no consigue comprender de una manera eficaz lo que realmente significa, qué relación guarda con la informática, qué beneficios presentan o cuál es la relación entre ambos términos. Este es el principal objeto de estudio de este trabajo. En concreto, consiste en la ejecución de aplicaciones High Performance Computing (HPC) usando contenedores software. El término HPC se refiere a la capacidad de resolver problemas muy complejos que abarcan un gran coste computacional. Para llevarlo a cabo, podemos utilizar algunas tecnologías como superordenadores, creación de clúster de varios ordenadores o las técnicas de programación paralela (IBM, s.f.). El uso de estas tecnologías queda recogido a lo largo del desarrollo de este trabajo. Respecto al término contenedor, podemos definirlo como un conjunto de elementos software dónde se empaquetan el código de la aplicación, las bibliotecas y dependencias de la aplicación. De esta manera se puede ejecutar en

cualquier lugar, independientemente del tipo de ordenador, si es en el escritorio o en el cloud. (IBM, 2021)

No obstante, instalar una aplicación HPC en un clúster es una tarea muy complicada debido a que todo el conjunto de bibliotecas, dependencias, compiladores, etc. deben estar instaladas y coincidir con los requisitos de la aplicación. Además, puede que surjan algunos conflictos a la hora de usar distintas aplicaciones ya que requieren distintas versiones. A parte de los problemas por adaptarse a los sistemas informáticos, encontramos otros problemas como la portabilidad. He aquí dónde aparecen las técnicas de virtualización. Y es que podemos almacenar en una **imagen**<sup>1</sup> todo lo necesario para ejecutar la aplicación deseada y luego desplegarla en un sistema virtual dentro de un contenedor. Por lo tanto, gracias al fomento de los contenedores software podemos acelerar la fase de despliegue de aplicaciones HPC y lo que es más importante, evitar problemas que no están en nuestra mano como es el caso de versiones o dependencias de las aplicaciones. Los contenedores permiten trabajar de una manera más efectiva e inteligente con entornos de desarrollo que son uniformes para desarrollar. Además, reducen el tiempo de algunas aplicaciones nativas de la nube. También hay que decir que los contenedores pueden aislar de una manera total las aplicaciones entre sí.

En el presente documento encontramos dos aplicaciones HPC. Por un lado, se va a ejecutar aplicación sencilla como punto de partida para abordar el problema, esta aplicación va a ser una imagen creada por el cálculo de Pi y todas las dependencias que hacen factible la ejecución dentro de contenedores. Y por otro lado, se va a llevar a aplicaciones más complejas como el modelo **WRF**<sup>2</sup>. Así que por lo tanto es pertinente mencionar el hecho de que se ha trabajado en dos entornos distintos. El trabajo ha sido depositado en **GitHub**<sup>3</sup> por si se desea ver todos los archivos empleados y la ejecución de este. Aunque es cierto que la mayoría del código empleado está en el presente documento.

---

<sup>1</sup> **Imagen:** Fichero compuesto por diversas capas y cuyo objetivo es ejecutar un código dentro de un contenedor

<sup>2</sup> **WRF:** Sistema de cálculo numérico para la simulación atmosférica

<sup>3</sup> **GitHub:** Portal para alojar código en el sistema de control de versiones de Git

- HTTP: <https://github.com/antoniojacobe/AplicacionesHPCcontenedores.git>
- SSH: git@github.com:antoniojacobe/AplicacionesHPCcontenedores.git

Por último, respecto a la motivación personal, debo de admitir la grata sorpresa que me he podido llevar en la realización de este TFG. Y es que, a pesar de ser un trabajo asignado, he conseguido ver toda la utilidad que puede existir hoy en día el uso de contenedores. En consecuencia, pienso que independientemente de las distintas ramas tratadas en la carrera, se debería enseñar en mayor o menor cantidad alguna parte relacionada con el mundo de los contenedores.

## 1.2 Objetivos

A partir de la descripción de la introducción (Véase 1.1) podemos hacernos una ligera idea de los contenidos que se van a desarrollar en este trabajo. El **objetivo principal** de este trabajo de fin de grado sería la ejecución de aplicaciones HPC a través de contenedores. Para ello, se va a llevar cabo en diferentes entornos. Pero para ser un poco más minuciosos acerca de este trabajo podemos dividir el objetivo principal en cuatro **objetivos específicos**:

- Estudio de diferentes tecnologías que se presentan en el mundo de contenedores software. Como es el caso de Docker, Singularity, Kubernetes, Docker Swarm, imagen, Docker Hub, etc. Por lo tanto, hay que obtener un gran conocimiento relacionado con todos estos conceptos y saber para qué se utiliza cada cosa.
- Estudio, uso y creación de contenedores software de aplicaciones secuenciales. Este objetivo sirve para dar el salto a los dos objetivos posteriores. Hace referencia a un aprendizaje de la creación y uso de contenedores y ejecución de una aplicación dentro de estos, ya que sin entender esto es muy complicado proceder a los siguientes objetivos.
- Estudio, uso y creación de contenedores software de aplicaciones paralelas que utilizan técnicas de paso de mensajes. Para el cumplimiento de este objetivo debemos conocer qué es una aplicación paralela y que vamos a usar OpenMPI o MPICH para la interfaz de paso de mensajes.

- Adaptación y despliegue de los contenedores software en un entorno de ejecución clúster. Para este objetivo vamos a desplegar en un clúster los contenedores software con nuestra imagen de la aplicación MPI y se ejecutará en cada uno de los nodos formados por el clúster.

### 1.3 Estructura del proyecto

Para guiar al lector durante el presente documento es conveniente realizar una primera presentación de la estructura que se ha llevado a cabo y una descripción de cada capítulo. La estructura está compuesta por un total de seis capítulos y un anexo.

- **Capítulo 1. Introducción.** Se expone una breve introducción del proyecto. Además de los objetivos que se tienen que realizar y la estructura de este.
- **Capítulo 2. Estado del Arte.** Se exponen los conceptos que son necesarios para poder efectuar los objetivos del proyecto. Además, se acompaña de unas ilustraciones y esquemas con el fin de facilitar la comprensión al lector.
- **Capítulo 3. Entorno de trabajo, herramientas y metodología utilizada.** En este capítulo se va a definir el entorno de trabajo, las herramientas y la metodología seguida.
- **Capítulo 4. Virtualización e implementación del cálculo de PI.** En este capítulo se va a describir la parte práctica de la implementación del cálculo de PI en las diferentes tecnologías. En concreto, se va a describir como podemos crear, construir y almacenar en Docker Hub la imagen que está compuesta por el código del cálculo de pi y todas sus dependencias. Además, encontraremos la distribución de ficheros, el proceso de instalación y configuración y la ejecución para cada tecnología que va a reproducir el cálculo de pi.
- **Capítulo 5. Virtualización e implementación del modelo WRF.** Dentro de este capítulo encontramos la parte práctica del modelo WRF. A diferencia del **Capítulo 4**, ejecutaremos el modelo WRF con nuestros datos de entrada. No obstante, también encontraremos la distribución de ficheros, proceso de instalación y configuración y la ejecución de este modelo con y sin Singularity.
- **Capítulo 6. Conclusiones y Trabajo Futuro.** Se va a detallar algunas conclusiones que se han obtenido a partir del desarrollo de este proyecto y los conocimientos



adquiridos para el mismo, así como el trabajo futuro para desarrollar posteriormente.

- **Anexo I. Planificación temporal.** En este anexo encontramos la división de todo el trabajo en tareas, así como un Diagrama de Gantt y unas conclusiones de si ha sido óptima la planificación llevada a cabo.



# Capítulo 2

## Estado del Arte

---

En este segundo capítulo se introducirán las diferentes tecnologías que envuelven este proyecto con el fin de construir las bases de este TFG. Se darán a conocer términos como HPC, MPI, contenedores y Vagrant. Además, comenzarán a aparecer otros como Kubernetes, Docker, Singularity, etc.

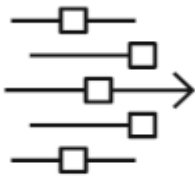
### 2.1 High Performance Computing

Es la capacidad de procesar datos y realizar cálculos complejos con ellos a una alta velocidad (NetApp, 2020). Cada día es mayor la importancia que tiene debido al crecimiento exponencial de la cantidad de datos, tamaño y tecnologías con las que se trabaja hoy en día. En consecuencia, se pueden resolver problemas más grandes y complejos en diversos campos como ingenierías, ciencias e incluso negocios. Además, es cierto que algunos computadores se están quedando obsoletos y el uso del HPC se está incrementando poco a poco debido a que este es capaz de sobrepasar incluso un millón de veces a una computadora portátil. (NetApp, 2022)

Lo último en sistemas HPC viene representado por lo que se conoce como las **supercomputadoras**. Un clúster de supercomputadores puede tener decenas de miles de procesadores y costar alrededor de 100.000.000€ en la actualidad (AMD, s.f.).

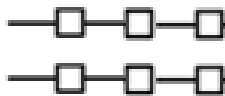
### 2.1.1 Funcionamiento HPC

Principalmente existen dos métodos para procesar la información en HPC



#### Procesamiento paralelo

Se lleva a cabo mediante varias CPU o GPU. Las GPU son diseñadas para gráficos totalmente independientes y son capaces de realizar operaciones diferentes por medio de una matriz de datos.



#### Procesamiento en serie

Las realizan las CPU, dónde cada núcleo realiza una tarea a la vez.

Estas son capaces de ejecutar funciones diferentes como SO y aplicaciones básicas.

Para que el procesamiento del primer método visto se pueda llevar a cabo, los sistemas HPC incluyen varios procesadores y módulos de memoria que se interconectan con un ancho de banda muy grande. Además, se puede llevar a cabo la combinación de varias CPU y GPU (Pérez, 2021).

### 2.1.2 Sectores de uso del High Performance Computing

El uso del HPC cada vez está siendo mayor. De hecho, antiguamente las empresas no necesitaban procesar grandes conjuntos de datos y realizar cálculos complejos a altas velocidades. Pero las cosas han cambiado, la necesidad de extraer información de Big Data ha aumentado y la demanda de un servicio casi en tiempo real. Es por ello, que HPC está llegando a ser muy relevante para los negocios más importantes en la actualidad. Además, se ha restringido el inconveniente de costes altos. ¿Pero qué sectores industriales utilizan actualmente HPC? Podemos encontrar los siguientes: (NetApp, 2022) (Oracle, s.f.) (BSC, 2019)

- **Biociencias:** descubrimiento de fármacos, comprender patrones de enfermedades y mejorar los materiales usados. Además, el HPC se implementa incluso en la lucha contra el cáncer.
- **Ingenierías:**
  - **Química:** diseño molecular

- **Geoingeniería:** para obtener nuevas iniciativas de exploración en petróleo y gas. O incluso para hacer una analítica geoespacial, mapeo de terreno, simulación de viento, procesamiento de datos sísmicos, entre otros.
- **Comercio:** ofertas de marketing, optimización de la cadena de suministro y la logística.
- **Economía:** análisis de riesgos financieros.
- **Cine, comunicación y videojuegos:** imágenes generadas por computadoras, análisis de imágenes en tiempo real, transcodificación y codificación, etc.

Estos sólo son algunos de los sectores más importantes.

## 2.2 Message Passing Interface

Es un medio para intercambiar mensajes entre distintas computadoras que van a ejecutar un programa paralelo a través de una memoria distribuida o compartida. Los datos se mueven del espacio de direcciones de un proceso al de otro proceso mediante operaciones cooperativas en cada uno de ellos. Además, suelen estar escritas en C o C++, aunque pueden estar escritas en otros como Fortran o ensamblador (Krypton Solid, 2013).

Message Passing Interface (MPI) trabaja a nivel de nodos. Los **nodos** son lo que conocemos como las computadoras o los núcleos del procesador de una computadora. El reto está en sincronizar todas las acciones en paralelo, intercambiar datos entre nodos, etc. Aunque MPI no esté respaldado por ningún estándar oficial se considera el estándar de la industria y forma la base para la computación paralela. (Bigelow, 2013)

Y es que esto no es algo nuevo, todo se remonta a la década de 1980. Al principio, MPI se diseñó para arquitecturas de memoria distribuida. Sin embargo, conforme iba pasando el tiempo, MPI tuvo que adaptar sus bibliotecas para crear sistemas de memoria compartida o híbrido. Prueba de ello son los 16 cambios de versión que ha sufrido. La versión actual sería la 4.1 que se remonta un año atrás, concretamente en noviembre de 2021. También hay indicios de que este año se va a llevar a cabo la versión 5.0. (Hpc-tutorials.llnl.gov, s.f.)

## 2.3 Contenedores

Un contenedor es un conjunto de elementos que permite ejecutar una aplicación en cualquier sistema operativo (Bejerano, 2016). Un contenedor está compuesto por un conjunto de software, como pueden ser todos los ejecutables de una aplicación, bibliotecas, códigos y archivos de configuración. Aunque a priori puede parecer que son muy pesados, no es así puesto que estos carecen de un sistema operativo propio. Todo lo que se ejecute en un contenedor se estaría ejecutando en el kernel del host (Clockworknet, 2019). El contenedor recibe los recursos del sistema operativo que considera oportunos, como CPU, RAM, disco y red. (NetApp, 2020)

Por ejemplo, podemos crear contenedores que lo formen varias imágenes como PHP, Java y PostgreSQL. O, sin embargo, podemos crear otro que tenga instalado OpenMPI y una aplicación del cálculo de Pi (Véase Figura 1).

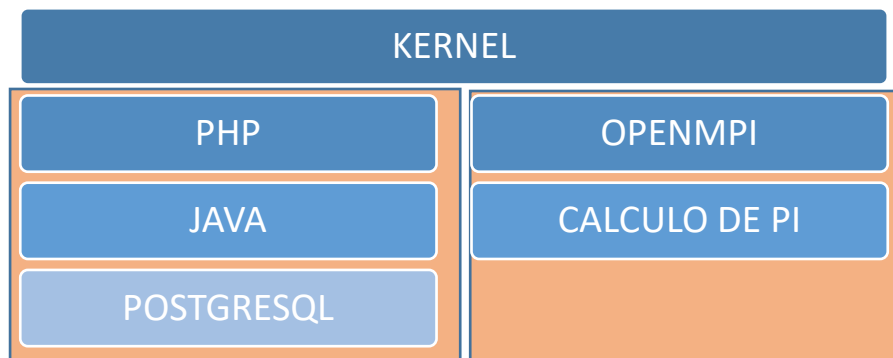


Figura 1: Contenedores

### 2.3.1 ¿Cómo funcionan los contenedores?

Los contenedores gestionan todo el empaquetado del software, dónde incluyen lo necesario para ejecutarse y hacer una adaptación en cualquier entorno. Para entender cómo funciona, es necesario distinguir el contenedor de la máquina en la que se está ejecutando, pues ambos comparten el sistema operativo instalado sobre el host y el hardware real del dispositivo. Para que un contenedor trabaje debidamente, es importante mencionar la función que realiza un motor de contenedores. Un **motor de contenedores** es una aplicación que nos permite gestionar los contenedores (Véase Figura 2). Gracias a este motor nos permite montar tantos contenedores como consideremos oportunos sin necesidad de tener que crear un hardware virtual. (Profile, 2021)

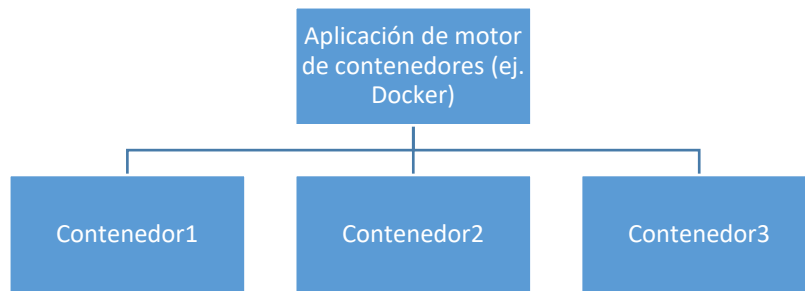


Figura 2: Motor de contenedores

### 2.3.2 Ventajas de uso de contenedores

Hay numerosas ventajas a la hora de trabajar con contenedores como es el caso de **desplegar aplicaciones**. Como sabemos podemos crear una aplicación en un contenedor y ese contenedor se puede ejecutar en cualquier sistema operativo que tenga el motor de contenedores instalado. Otra podría ser la **mayor consistencia** obtenida entre los entornos de prueba y los entornos de producción. Y es que son numerosas las empresas en el que diferencian totalmente estos dos tipos de entornos. Sin embargo, cuando se hacen pruebas de la aplicación dentro de un contenedor y la despliegas, el funcionamiento va a ser idéntico. El uso de Docker y contenedores puede mejorar la forma de colaborar entre los equipos de operaciones de TI y los desarrolladores. Por lo tanto, muchos clientes pueden seguir usando **DevOps**<sup>4</sup> y evitar una configuración más compleja de versiones y compilaciones. Además, los contenedores simplifican mucho la configuración de compilación, prueba e implementación de DevOps. (Microsoft, 2022)

También se obtiene un **mayor modularidad**. Si nuestra aplicación es bastante compleja, la podemos separar en partes más pequeñas. Por ejemplo, podemos ejecutar el servidor en un contenedor, mientras que la parte del front-end de la misma aplicación en otro distinto. (Microsoft, 2022)

Por otro lado, encontramos otras ventajas como puede ser el **ahorro de costes**. Esta es uno de los factores más importante en una empresa, y es que el uso de los contenedores minimiza los costes y aumenta los beneficios ya que se reducen costes en el hardware. También, es oportuno destacar el **sencillo mantenimiento** que tienen,

---

<sup>4</sup> **DevOps**: Conjunto de procesos que agrupan el desarrollo de software (Dev) y las operaciones TI (Ops).

puesto que se ejecuta igual independientemente del servidor de la máquina física en la que se estén llevando a cabo. Además de una configuración y despliegue más rápido, encontramos el aspecto del **aislamiento**. Al final, cada contenedor está separado de otro contenedor y tiene sus propios recursos. De hecho, cada contenedor está identificado mediante un ID, lo que conlleva a poder dirigirnos a este para saber que contiene, a quién pertenece, iniciarlo o incluso borrarlo. Al ser todos los contenedores iguales, el proceso para realizar las distintas operaciones disponibles está estandarizados. Esto hace que todo se haga completamente igual en todos ellos (Campusmvp, 2018).

Para terminar de mencionar algunas las ventajas que tienen los contenedores, encontramos que son muy **seguros**. Gracias al motor de contenedores se garantiza que las aplicaciones que se estén ejecutando en los contenedores estén aisladas y no influyen en las otras. Ningún contenedor puede ver los procesos que se están ejecutando dentro de otro contenedor. Por lo tanto, evitamos que alguien pueda manipular los productos del interior del contenedor. (Campusmvp, 2018)

### 2.3.3 Plataformas relacionadas con contenedores

Hoy en día, son numerosas las soluciones de gestión de contenedores. Algunos de ellos son de código abierto, mientras que otras son de pago. Es necesario destacar algunas plataformas y proceder a dar una explicación del uso de Docker y algunos orquestadores de contenedores empleados en este trabajo.

- Docker

Es un proyecto de código abierto que automatiza el despliegue de aplicaciones dentro de los contenedores. Además, proporciona una capa de abstracción y automatización. Fue lanzada en el año 2013 por Docker, Inc. Docker ofrece algunos servicios open-source y Enterprise. (González, 2021)

Es necesario explicar la arquitectura de Docker para saber cómo está montado. En la arquitectura de Docker podemos distinguir tres partes. El cliente que se encarga de introducir los comandos de Docker, la parte de Docker en el host, que estaría constituida por los contenedores, imágenes y el demonio de Docker y por último la parte del registro, dónde se almacenan y se distribuye las imágenes. (Véase Figura 3). (Avi, 2022)



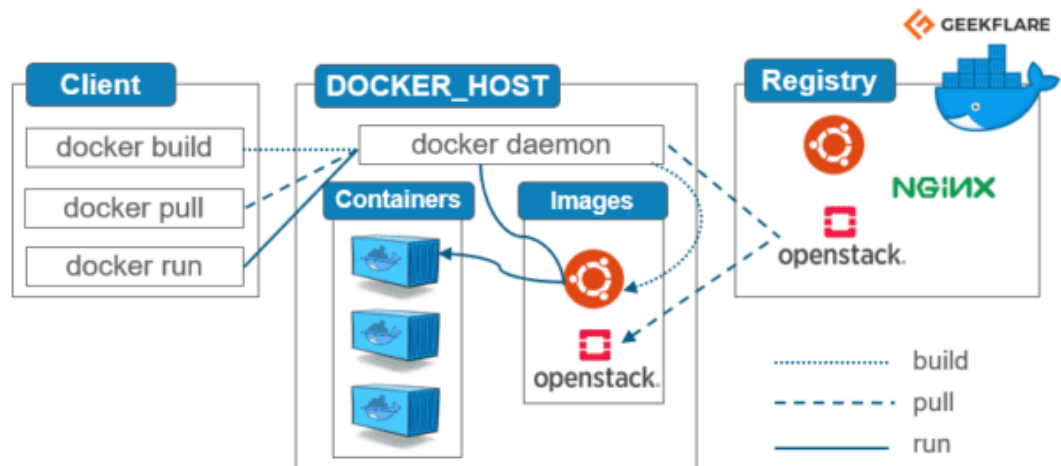


Figura 3: Arquitectura de Docker

No obstante, podemos entrar más a detalle:

- **Docker Engine:**

Es la parte central de todo el sistema Docker. En un ordenador, sería la CPU, el cerebro del ordenador. En ella podemos comunicarnos a través de una interfaz de línea de comandos. Luego también encontramos el servidor, que es el que se encarga de crear y administrar imágenes, de los contenedores, redes, etc. Por último, encontramos una API que indica al servidor qué hacer.

- **Cliente Docker:**

Es la forma en el que los usuarios interactúan con Docker. Lo que hace es enviarlo al servidor para que lo ejecute. Por lo tanto, se puede decir que utilizan la API de Docker.

- **Registros de Docker**

Corresponde a la ubicación donde se almacenan las imágenes de Docker. Hay registros privados a nivel local y en la nube pública. Por ejemplo, Docker Hub sería del segundo tipo. Docker Hub es una biblioteca con más de 100.000 imágenes de contenedores, lo que la convierte en la más grande del mundo. Cuando ejecutamos Docker run, la imagen requerida se extrae del registro configurado. O si hacemos Docker push se almacenaría. En este trabajo se ha hecho uso de nuestra propia imagen y se ha subido a Docker Hub.

- **Objetos de Docker**

Docker trabaja con una serie de objetos como pueden ser:

1. **Imágenes:** Es un archivo que se usa para ejecutar código dentro de un contenedor. Estas suelen tener relación con otras. Por ejemplo, podemos obtener una imagen de un sistema operativo y dentro de esa imagen, obtener otra imagen de un servidor o de una aplicación.

Gracias al fichero de Dockerfile, Docker es capaz de crear imágenes automáticamente. Cada instrucción que se va encontrando en este fichero genera una capa en la imagen, por lo que si se cambia solo se regenera aquellas que han sido cambiadas y las otras no. Este simple proceso hace que la tecnología Docker sea muy efectiva, pues no carga con elementos innecesarios.

2. **Contenedores:** Todas las aplicaciones se ejecutan dentro de un contenedor. Para iniciarlo, podemos usar la terminal. En términos de informática sería una instancia de una imagen.
3. **Volúmenes:** Todos los datos que genera Docker se almacenan en unos discos virtuales conocidos como volúmenes. La dirección para acceder a estos suele ser `/var/lib/docker/volumes/` en Linux. Los volúmenes se pueden compartir entre varios contenedores. Por lo tanto, su uso es totalmente recomendable, ya que de lo contrario se guardaría en la capa de escritura.
4. **Redes:** Permite que los contenedores se comuniquen entre sí. Normalmente, encontramos cinco tipos de controladores.

**4.1. Bridge:** Es el controlador predeterminado para un contenedor. Se usa cuando varios contenedores se comunican con el mismo host.

**4.2. Host:** Es el que se encarga de eliminar el aislamiento que hay entre el contenedor Docker y el host Docker.

**4.3. Overlay:** Es el que se encarga de que los servicios Swarm se comuniquen entre sí. Estos servicios permiten agrupar una serie de host de Docker en un clúster.

**4.4. None:** Es el controlador capaz de desactivar todas las redes.

**4.5. Mascvlan:** Puede asignar una dirección MAC al contenedor. De esta manera los contenedores serían dispositivos físicos.

- **Kubernetes**

Es una plataforma de código abierto que se encarga de la organización de contenedores. Todo ello lo realiza automatizando y eliminando algunos de los procesos manuales que son llevados a cabo durante la implementación y escalabilidad de las aplicaciones en los contenedores. Es conocida también con otros dos nombres como k8s o kube. (RedHAt, 2020) Su origen tiene lugar en 2014, desarrollado por ingenieros de Google. Se dice que es el predecesor de Borg<sup>5</sup>. El logotipo de este tiene referencia a su significado de timonel o capitán en griego. (Kubernetes.io, s.f.)

Es conveniente destacar el término de **orquestador de contenedores**, pues Kubernetes básicamente es eso. Es capaz de ofrecer servicios como el manejo de un clúster, es decir puede añadir o eliminar nodos, repartir la carga entre los nodos, añadir servicios de monitorización, tiene la opción del auto escalado y dispone del Service Discovery. Este servicio sirve para que un contenedor pueda encontrar rutas IP o de DNS de otro contenedor distinto. Respecto al manejo de un clúster sabemos que está formado por dos roles, máster y minion/worker. Es cierto que se suele conocer como worker antes que con minion, por lo que durante el desarrollo de este trabajo la referencia hacia el mismo va a ser con el término worker.

Un nodo máster, es el que coordina el clúster y debe existir uno mínimo. En sus características se puede destacar el hecho de que no suelen ejecutar el contenedor, sino que van decidiendo que nodo ejecuta cada contenedor. También hay que mencionar que en nuestro caso se ha configurado para que se produzca un reparto equitativo entre las máquinas considerando al máster también. El otro tipo de nodo se une al máster para recibir órdenes. Este está compuesto por un motor de contenedor instalado en el clúster correctamente, un kubelet que es el que se

---

<sup>5</sup> **Borg:** plataforma de orquestación de contenedores que usó internamente Google

encargar de poner en funcionamiento al contenedor y un kube-proxy que gestiona las IPs y red. (RedHAt, 2020)

- **Docker Swarm**

Es un orquestador de contenedores que ejecutan Docker y están configurados para unirse en un clúster. Aunque se pueden usar todos los comandos de Docker y guarda una relación muy amplia entre ambos, ahora se van a ejecutar en las máquinas del clúster creado.

Una de las ventajas claves de este funcionamiento es el alto nivel de disponibilidad que pueden llegar a ofrecer las aplicaciones. También tiene concordancia con Kubernetes, es decir hay un nodo manager o máster y otros workers que deben unirse.

Por último, es necesario saber que Docker Swarm usa un equilibrio de cargas para garantizar que haya suficientes recursos para los contenedores distribuidos. Dependiendo de la carga que haya en cada máquina, este se encargará de proporcionarle más o menos trabajo según tres estrategias: (Krypton Solid, s.f.)

- **Propagación:** Es la configuración predeterminada y equilibra la carga entre los nodos basados en la disponibilidad de CPU y RAM, que a priori todos han sido creados con la misma.
- **BinPack:** Se van llenando poco a poco los nodos, hasta que un nodo no esté completamente ocupado no pasa al siguiente.
- **Aleatorio:** Va eligiendo nodos al azar

- **Singularity**

Es una plataforma gratuita, multiplataforma y de código abierto de contenedores. Puede crear y ejecutar contenedores que se crean en forma de archivo en el host. Además, una vez que el contenedor está creado se puede llevar al mundo de la computación científica o al mundo del HPC, término que ya debe resultarnos común.

Su origen se remonta hasta el año 2015, dónde fue llevado a cabo por un grupo de investigadores liderado por Gregory Kurtzer del Laboratorio Nacional Lawrence Berkeley. (Kurtzer, s.f.) A partir de ahí, se hizo popular en otros sitios de HPC y sigue

expandiéndose. A diferencia de los demás, este se centra en otros tantos aspectos como pueden ser la reproducibilidad y seguridad que se llevan a cabo mediante firmas criptográficas, la integración sobre el aislamiento por defecto y la movilidad de computación debido a que existe un único archivo (.SIF). Además, sigue un modelo de seguridad eficaz y seguro. (Sylabs.io, 2021)

- Diferencias entre Docker, Docker Swarm, Kubernetes y Singularity

Lo primero que podemos decir es que no son tecnologías competidoras, es más son tecnologías que van de la mano. Podemos comenzar diciendo que Docker crea un entorno aislado para las aplicaciones, mientras que Kubernetes y Docker Swarm es una infraestructura para administrar esos contenedores. Además, Docker crea los contenedores, Kubernetes y Docker Swarm, sin embargo, se encargan de la programación y administración automatizada de los contenedores. Aunque es cierto que con Singularity es necesario crear nuestros contenedores también.

Kubernetes y Docker Swarm tienen muchas funcionalidades similares. No obstante, cada una tiene una forma de operar distinta. Es cierto que una está desarrollada por Docker y otra por Google. Otras diferencias que encontramos serían que Kubernetes tiene mucha más tolerancia a fallos que Docker Swarm, aunque este sea más sencillo de configurar. Respecto al balanceo de carga, Docker Swarm tiene un equilibrio de carga interno que está automatizado, Kubernetes no. Quizás Kubernetes tiene mayor cuota de mercado debido al gran respaldo de Google, pero eso no quiere decir que sea mejor o tenga mayores ventajas con respecto a Docker Swarm.

Singularity a diferencia de Docker, favorece la integración en lugar del aislamiento. Aunque tiene su formato de imagen, puede cargar imágenes de Docker. También tiene un ecosistema mucho más pequeño que Docker y es menos popular, empero si queremos procesar datos por lotes Singularity es una mejor opción

#### 2.3.4 Relación entre contenedor y máquina virtual

Una máquina virtual (MV) es un software que simula el comportamiento de otro dispositivo. Es por ello, que está compuesta por su memoria, disco duro, tarjeta gráfica, etc. Además, pueden admitir otros dispositivos que son inexistentes en la

máquina principal. Por ejemplo, puede contener una imagen ISO de un CD-ROM, pero la máquina principal no tiene un lector de CD y no serviría para leer un CD.

Estas máquinas permiten crear entornos de ejecución sin que corramos ningún riesgo en nuestra máquina principal. En consecuencia, suelen ser usadas para acceder a algunos datos infectados o algunas pruebas con cierto peligro. No obstante, puede parecer que los contenedores y las máquinas virtuales son conceptos muy parecidos. Pero cada una trabaja en niveles de capas distintas. Mientras que las MV virtualizan todas las capas hardware, los contenedores solo virtualizan las capas que están por encima del sistema operativo. Por lo tanto, los contenedores nos permiten reducir una cantidad importante la carga de nuestro ordenador físico, el espacio de almacenamiento y el tiempo que conlleva lanzar las aplicaciones. También al usar contenedores no tenemos que indicar los recursos que vamos a necesitar, como ocurre en las MV. Eso nos conlleva a que se aproveche de una forma mayor el hardware del sistema y podamos levantar muchos más contenedores que máquinas virtuales.

Aunque es cierto que cada tecnología tiene sus ventajas con respecto a la otra, el uso de contenedores es mayor que el de máquinas virtuales debido a sus ventajas en las fases de desarrollo software. Para entender mejor la composición y así la diferencia entre ambas podemos tener en cuenta los esquemas de la composición de una máquina virtual (Véase Figura 4) o de contenedores (Véase Figura 5):

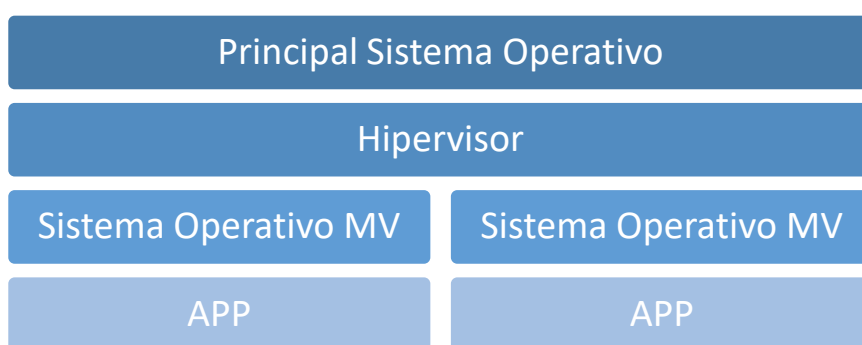


Figura 4: Esquema de una máquina virtual

Podemos ver como en las máquinas virtuales disponemos de un hipervisor, que en nuestro caso es VirtualBox. Dentro de VirtualBox obtendremos todos los sistemas operativos de las máquinas virtuales y dentro las aplicaciones.

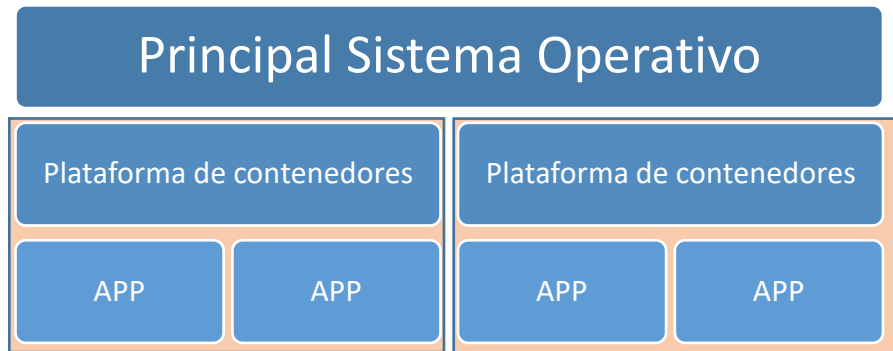


Figura 5: Esquema de contenedores

Al contrario del hipervisor, ahora contamos con una plataforma de contenedores para poder administrar las aplicaciones en contenedores. Podemos identificar varios tipos como son los siguientes: (Aquasec, s.f.)

- **Motor de contenedor:** permiten crear contenedores y administrar sus imágenes. Un ejemplo básico sería Docker.
- **Orquestador de contenedor:** permite administrar y automatizar los contenedores a escala. Encontramos Kubernetes, Docker Swarm...
- **Plataforma de contenedor administrada:** aumenta los dos tipos mencionados. Un ejemplo común sería Google Kubernetes Engine.

## 2.4 Contenedores + HPC

En el mundo empresarial son cada vez más las empresas que se esfuerzan por el uso de contenedores ya que les permiten reducir tiempo en desarrollo e implementación. Respecto al HPC ocurre algo similar. Y es que, en los últimos cinco o seis años predominaba la tecnología basada en los contenedores. Esta se podía ir desde pequeños centros de datos privados a grandes centros de datos empresariales y nubes públicas. Sin embargo, con la innovación de HPC y la IA, las empresas se están desarrollando para soportar una gran cantidad de datos.

Los contenedores pueden ser usados para las cargas de trabajo HPC ya que resuelven defectos importantes en las plataformas tradicionales. Años atrás contenedor y HPC se consideraban incompatibles. No obstante, en la actualidad hemos visto como existen proyectos de código abierto para utilizar los contenedores de carga de trabajo

HPC. Por ejemplo tenemos Podman, Shifter o incluso uno visto recientemente, Singularity. (Programmerclick, s.f.)

A modo de conclusión es importante remarcar el hecho de que se han realizado varias investigaciones y se han establecido varias soluciones de código abierto. Las empresas más grandes han usado Kubernetes y Singularity como principales opciones para manejar cargas de trabajo HPC. Aunque es cierto que con el paso del tiempo habrá más avances en este entorno. (Programmerclick, s.f.) Por ejemplo, podemos encontrar códigos de simulación como Alya que fue creado para la mecánica computacional de alto rendimiento. Resuelve problemas de multifísica usando técnicas de alto rendimiento para supercomputadores que pueden tener memoria distribuida y compartida. (BSC, s.f.)

## 2.5 Vagrant

Es un software de código abierto que permite crear y controlar múltiples máquinas virtuales. En nuestro caso, trabaja con VirtualBox. Pero puede trabajar con otras como VMware, AWS e incluso con contenedores de Docker. Fue desarrollado en 2010 por Mitchell Hashimoto y dos años después se lanzó la primera versión. Posteriormente su uso ha ido incrementándose. (Kripkit, s.f.). Vagrant destaca por la interfaz tan sencilla para crear servidores independientes del sistema operativo del dispositivo físico, ya que está creando máquinas virtuales. Sin embargo, lo hace de una manera bastante más rápida y es compatible con cualquier sistema operativo. (Campusmvp, 2016). Respecto al desarrollo de este trabajo ha supuesto un gran ahorro de tiempo. Gracias a Vagrant, hemos podido recrear de una manera sencilla un entorno clúster sobre el cual hemos realizado todo el trabajo desarrollado en el TFG.

## 2.6 Weather Research and Forecasting

Es un modelo cuya función es hacer una predicción meteorológica numérica de meso escala. En otras palabras, es un sistema de última generación que ha sido diseñado para la investigación atmosférica y para aplicaciones de predicción operativa compuesto por diversas instituciones. Este sistema ha sido diseñado por las algunas entidades como NCAR, NOAA, NCEP, ESRL, NRL, CAPS, FAA, entre otras. (MMM.ucar.edu, s.f.)



El código del modelo es libre y ha sido optimizado para que sea adaptado tanto en supercomputadoras que trabajan en paralelo como portátiles recientes de la actualidad. Además, podemos usarlos en distintas aplicaciones como simulaciones reales, modelación de la calidad del aire, etc. Prueba de esto es la gran cantidad de versiones que existen, puede haber versiones para calcular la dispersión de contaminantes en temas forestales o química atmosférica, para solucionar huracanes, etc.

Actualmente Weather Research and Forecasting (WRF) se encuentra en uso operativo en NCEP y otros centros más donde también encontramos laboratorios, universidades y empresas. La comunidad tiene un total acumulado de más de 48000 usuarios distribuidos en 160 países. También cuenta con talleres y tutoriales para su aprendizaje. (MMM.ucar.edu, s.f.)

El WRF Software Infrastructure (WFS) tiene todos los códigos que incorporan la física al modelo y los paquetes que constituyen la interfaz. El modelo cuenta con dos núcleos, el ARW y NMM. En el núcleo ARW se lleva a cabo la integración numérica del dominio principal y anidamientos. El NMM se creó con el objetivo de la predicción ambiental para centros. (Grupo-ioa.atmosfera.unam.mx, s.f.)

Durante este documento podemos ver algunos intentos de ejecutar WRF utilizando contenedores y nuestros datos de entrada (Véase Capítulo 5). Como sabemos los contenedores aportan numerosas ventajas a la hora de programar y ejecutar nuestras aplicaciones y en este modelo ocurre algo similar. Además, se ha ejecutado el modelo en secuencial para ver como el tiempo demorado es mucho mayor que en paralelo.



# Capítulo 3

## Entorno de trabajo, herramientas y metodología utilizada

---

En este capítulo se va a detallar todo el entorno de trabajo, metodología empleada y las herramientas que se han utilizado. Además, nos encontraremos con el primer software de este proyecto como es Vagrant.

### 3.1 Entorno de trabajo y herramientas utilizadas

El entorno de trabajo y las herramientas que se han usado para la ejecución del código de PI son diversas (Véase Figura 6). El ordenador local o host tiene instalado un Sistema Operativo de Windows 10. Luego gracias a VirtualBox podemos crear varias máquinas virtuales. Concretamente se ha creado una máquina virtual para instalar y llevar a cabo todo lo relacionado con Docker, cuyo sistema operativo es Kali Linux. Con la ayuda de Vagrant, se ha creado en VirtualBox dos máquinas Ubuntu 20.04 para cada prueba, ya que la idea es crear un clúster con ellas. En otras palabras, se ha creado un total de ocho máquinas. Dos para cada tecnología (Kubernetes, Docker Swarm, Singularity) y otras dos que, aunque no aparezcan en la figura anterior, es para probar que el cálculo de pi se ejecute en dos máquinas distintas, a diferencia de Docker que se

ejecutó dentro de una máquina sola. Con Vagrant podemos crear automáticamente nuestras máquinas configuradas y adaptadas a cada tipo de tecnología (Véase 3.3).

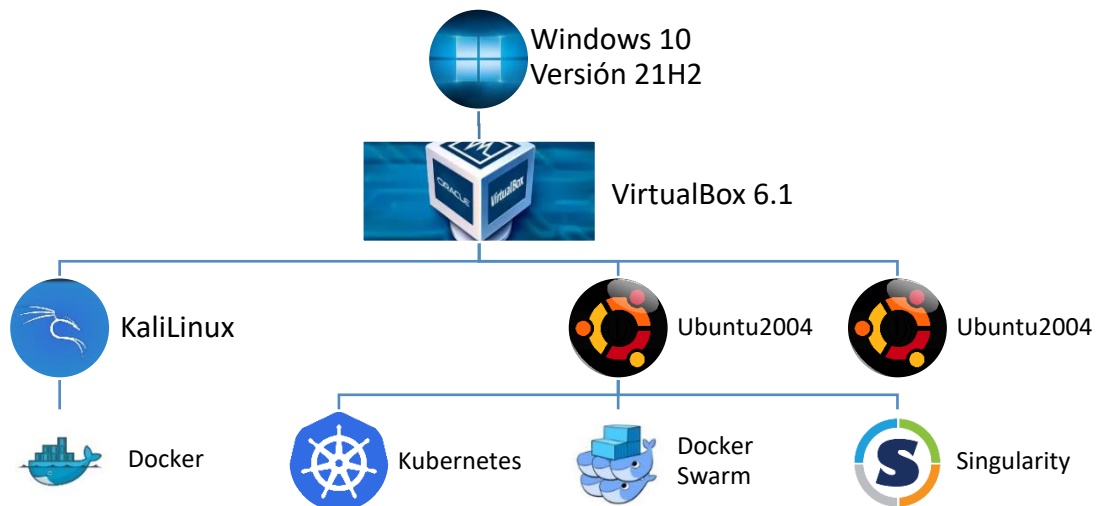


Figura 6: Topología desarrollada para el cálculo de pi

Por otra parte, podemos distinguir también la topología que se ha empleado para la parte de WRF (Véase Figura 7).

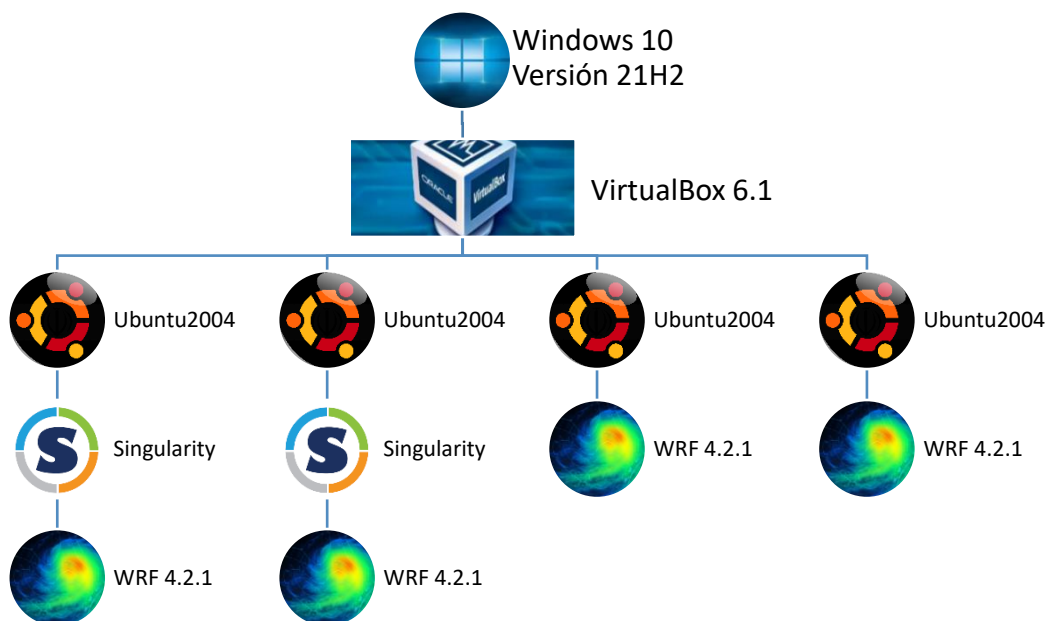


Figura 7: Topología para el modelo WRF

Como en la topología anterior con la ayuda de Vagrant vamos a crear dos máquinas Ubuntu 20.04 para comprobar la ejecución en paralelo del modelo WRF 4.2.1 utilizándolo directamente y otras dos máquinas usando Singularity. La idea de esto es comparar las prestaciones de ejecutar WRF en ambas opciones y así poder comparar el tiempo de ejecución entre ambos.

## 3.2 Metodología

Este trabajo debe concluir con la capacidad de ejecución de aplicaciones HPC en contenedores, así como la capacidad de que el lector sea capaz de ejecutar una aplicación paralela en un orquestador de contenedores y obtener una idea de todas las tecnologías que aborda este trabajo relacionadas con el mundo de los contenedores. Es por ello, que en este punto se van a indicar las etapas que se han seguido para implementar los objetivos detallados anteriormente (Véase 1.2).

En primer lugar, nos encontraríamos con la etapa de **planificación** (Véase Figura 8). Antes de comenzar a desarrollar el proyecto, tenemos que saber cuáles son los requisitos por cumplimentar del proyecto y formalizar una idea principal del mismo. Es por ello por lo que en esta primera etapa hemos redactado los objetivos para llevar a cabo el proyecto. Además, también se contempla la organización general que va a tener el proyecto, aunque luego varíe ligeramente (Véase Anexo I). Pero es importante saber los puntos que vamos a tratar y cuáles son las tecnologías con las que vamos a tratar. (Blog.tactio, s.f.).

En segundo lugar, una vez que tenemos clara la planificación de este trabajo, tendría lugar la etapa de **análisis** (Véase Figura 8). En esta etapa se descubre cuál es el software que necesitamos realmente y ver si la máquina sobre la que vamos a realizar este trabajo de fin de grado cumple los requisitos que conlleva usar todo el software para realizar este proyecto.

En tercer lugar, sería el turno de lo que se considera la fase más importante y es la fase de **ejecución** (Véase Figura 8). Esta fase consiste en la ejecución de tareas y actividades para la realización de las acciones que se han propuesto en la planificación con el fin de cumplimentar los objetivos que se han establecido. En nuestro caso, gracias a la etapa de análisis ya hemos obtenido un conocimiento previo del software a usar.

Una vez que teníamos claro esto, hemos ido instalando y configurando cada software de tal manera que hasta que no cumplíamos los objetivos con uno, no pasábamos al siguiente. De esta manera nos asegurábamos terminar con una tecnología antes de adentrarnos con otra y llevar ambas a la par. También se ha de decir que, gracias a la etapa de planificación, se ha establecido un orden lógico del tratamiento de cada software. Por ejemplo, la primera tecnología usada fue Docker y una vez que supimos ejecutar una aplicación dentro del contenedor, ya procedimos a la ejecución de esa aplicación en dos nodos. No hubiera tenido sentido haberlo hecho de manera inversa. Por otro lado, antes que nada, recogimos información de Vagrant (Véase 3.3) para facilitar la creación de los nodos del clúster. Esto permitió un gran ahorro de tiempo ya que es necesario para cumplir otros objetivos propuestos. Sin usar primeramente este software resulta imposible usar otras tecnologías que se encargan de gestionar el clúster, como Docker Swarm, Kubernetes, Singularity, etc. Por eso hay que destacar la importancia de una buena etapa de planificación.

Dentro de esta etapa, podemos encontrar la fase de **seguimiento y control** (Véase Figura 8). Esta fase se inicia a la par de la fase de ejecución y continuaría después de esta. Esto permite encontrar posibles problemas que no se detectaron en la etapa de planificación o algunos que se van desarrollando. (Blog.tactio, s.f.) Gracias a esta etapa, se ha completado el proyecto a tiempo, sin exceso de este. Por otro lado, aparte de controlar el trabajo y el cumplimiento adecuado de los objetivos descritos, en esta parte se ha implicado otros pasos como **las solicitudes de cambio**. Estas solicitudes han sido propuestas por el tutor para la mejora del proyecto. Como paso de respuesta a este, se encuentra **la reparación de defectos**. Y es que cuando se produce un resultado que no cumple con las especificaciones requeridas es necesario reparar esto. (Obsbusiness.school, 2021)

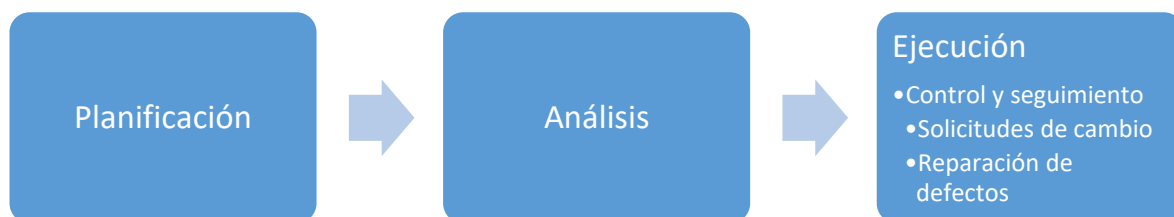
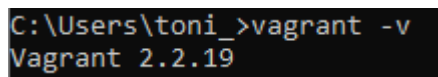


Figura 8: Etapas desarrolladas

### 3.3 Vagrant

El primer software que nos encontramos sería Vagrant debido a que se usa para desplegar todas las máquinas. Así que es necesario comentarlo como primer paso.

Instalar y configurar Vagrant es relativamente sencillo. Podemos dividirlo en varios pasos. En primer lugar, hay que descargar e instalar Vagrant (Vagrantup, s.f.). Para ello, es necesario indicar el sistema operativo desde el que vamos a trabajar. Como se ha comentado con anterioridad, necesitamos un proveedor de máquinas virtuales para ello, por defecto es **VirtualBox** y es el que hemos usado para el desarrollo de este trabajo. Así que debemos de tenerlo instalado. Una vez instaladas ambas cosas, es necesario comprobar que Vagrant se ha instalado satisfactoriamente, para ello podemos mirar la versión instalada en el host (Véase Figura 9).



```
C:\Users\toni_>vagrant -v
Vagrant 2.2.19
```

Figura 9: Versión de Vagrant

En segundo lugar, tenemos dos opciones para la creación de máquinas virtuales de VirtualBox. Podemos inicializar la máquina directamente con `vagrant init +nombreImagen` o especificarlo en un fichero denominado **VagrantFile**. Cabe destacar que con la primera opción el VagrantFile se crea automáticamente, por lo que es necesario el uso de este archivo.

En tercer lugar, una vez que tenemos el VagrantFile correspondiente, es hora de lanzar las máquinas. Se lleva a cabo mediante el comando `vagrant up`. Este comando sirve para descargar, instalar, configurar y arrancar nuevamente las máquinas virtuales pertinentes. Debido a este modo de funcionamiento hace que todo el proceso sea muy rápido y sencillo.

En cuarto lugar, necesitamos acceder a la máquina virtual. Aunque Vagrant inicie las máquinas virtuales sin interfaz gráfica, podemos acceder a ella mediante SSH. Hay varias maneras de acceder de una manera gráfica, como por ejemplo mediante VirtualBox. Una vez que hemos accedido con el comando `vagrant ssh nombreMaquina*`, desde el directorio que hay compartido entre la máquina y la máquina virtual,

obtendremos acceso a una Shell y podremos introducir todo tipo de comandos, como si estuviéramos en esa máquina. Es importante remarcar el hecho de que se comparte un directorio que es accesible desde las dos máquinas. En la máquina virtual, este directorio se encuentra en el path: /vagrant.

Una vez que hemos terminado el trabajo deseado sobre la máquina virtual, o simplemente queremos suspenderla, existen varias opciones para ello:

- **vagrant halt:** Apaga las máquinas virtuales.
- **vagrant suspend:** Pausa las máquinas virtuales y guarda el estado actual del disco duro. Por lo tanto, cuando las volvamos a arrancar con `vagrant up`, seguiremos en el estado que se encontraba la máquina.
- **vagrant destroy:** Destruye las máquinas virtuales y su contenido, sin dejar ningún rastro.

### 3.3.1.1 VagrantFile

Antes se ha mencionado el uso de un fichero para automatizar toda la configuración e instalación de las máquinas virtuales que deseamos llevar a cabo. Este fichero está escrito en formato Ruby (Véase Código 1).

---

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

NODE_01_IP   = "192.168.56.30"
NODE_02_IP   = "192.168.56.31"

Vagrant.configure("2") do |config|
  config.vm.box = "geerlingguy/ubuntu2004"
  config.vm.box_version = "1.0.3"

  boxes = [
    { :name => "node-01", :ip => NODE_01_IP, :cpus => 1, :memory => 2048 },
    { :name => "node-02", :ip => NODE_02_IP, :cpus => 1, :memory => 2048 },
  ]

  boxes.each do |opts|
    config.vm.define opts[:name] do |box|
      box.vm.hostname = opts[:name]
    end
  end
end
```



```

box.vm.network :private_network, ip: opts[:ip]

box.vm.provider "virtualbox" do |vb|
  vb.cpus = opts[:cpus]
  vb.memory = opts[:memory]
end
box.vm.provision "install", type: "shell", path: "./install.sh"
box.vm.provision "ssh", type: "shell", privileged: false, path: "./config-ssh.sh"
end
end
end

```

Código 1: VagrantFile cálculo pi en dos máquinas

Para las pertinentes modificaciones de las máquinas virtuales se configuran en el espacio de nombres `config.vm`. Este es un prefijo que antecede a varias configuraciones disponibles como `config.vm.base_mac`, `config.vm.box`, `config.vm.define`, entre otros. Para cada tipo de orquestador de contenedores se ha llevado a cabo su propio VagrantFile para su uso posterior. Sin embargo, todos comparten comandos muy parecidos al mostrado (Véase Código 1). Por ejemplo, el proveedor es VirtualBox, la memoria usada es de 2048, se ha hecho uso de una CPU por máquina o el box elegido `geerlingguy/ubuntu2004`. Como información adyacente, este box se descarga de un catálogo, **Vagrant Cloud**. (App.vagrantup, s.f.)

También es importante recalcar el hecho de que existe una opción conocida como `box.vm.provision` que permite la ejecución de comandos en la terminal durante la creación de la máquina. En consecuencia, podemos iniciar la máquina con varios programas instalados, como un servidor apache, instalar el `mpich` o incluso un editor de texto como `nano`. En nuestro caso, se ha usado para llamar a otros scripts, dónde se llama a otros archivos `.sh` para instalar y configurar las dependencias de Kubernetes/Docker, y para configurar el máster/manager y el worker correspondiente. Cada forma tratada de realizar el cálculo de pi o el modelo WRF tiene su propio VagrantFile.

El código mostrado con anterioridad sería el código correspondiente a la **ejecución del cálculo de pi en dos nodos distintos** (Véase Código 1). La única diferencia que existe es que en lugar de llamar al `install.sh` y `configure.sh`, tenemos que llamar a

otros sh para que se usen para instalar las dependencias de **Kubernetes** (Véase Código 2). Luego dependiendo de si la máquina es el nodo máster o nodo-01, se llamará al respectivo sh. Para evitar repetición de código sólo se ha introducido la parte diferencial con respecto al primer código, marcada con un **rectángulo**.

---

```
(...)  
box.vm.provision "shell", path:"./install-kubernetes-dependencies.sh"  
  if box.vm.hostname == "master" then  
    box.vm.provision "shell", path:"./configure-master-node.sh"  
  end  
  if box.vm.hostname == "node-01" then  
"shell", path:"./configure-worker-nodes.sh"  
  end  
end  
end  
end
```

---

Código 2: VagrantFile Kubernetes

Lo mismo ocurre en el caso de **Docker Swarm**. Como podemos apreciar en él debemos instalar Docker en ambas máquinas y dependiendo de si el nodo es el máster o el nodo-01 se llamará al `configure_manager` o `configure_worker` respectivamente (Véase Código 3).

---

```
(...)  
box.vm.provision "shell", path:"./install-docker.sh"  
  if box.vm.hostname == "master" then  
    box.vm.provision "shell", path:"./configure_manager.sh"  
  end  
  if box.vm.hostname == "node-01" then worker hostnames  
    box.vm.provision "shell", path:"./configure_worker.sh"  
  end  
  
end  
end  
end
```

---

Código 3: VagrantFile Docker Swarm

Para la parte de Singularity, es el mismo uso que el VagrantFile del primer código (Véase Código 1), solo que se añade la instalación de Singularity en el fichero install.sh. La explicación de los ficheros a los que hace referencia VagrantFile están descritos en los puntos:

- Docker y las dos máquinas para ejecutar cálculo de pi sin orquestadores: (Véase **4.2.2**)
- Kubernetes: (Véase **4.3.2**)
- Docker Swarm: (Véase **4.4.2**)
- Singularity: (Véase **4.5.2** para el cálculo de Pi) y (Véase **5.3.2** para el modelo WRF)

# Capítulo 4

## Virtualización e implementación del cálculo de PI

---

En este cuarto capítulo se da paso a la ejecución del cálculo de PI en todas las tecnologías desarrolladas para este proyecto. Específicamente, encontramos cuatro: Docker, Docker Swarm, Kubernetes y Singularity. El primer paso que se va a efectuar va a ser una introducción. Posteriormente encontraremos la creación de la imagen del cálculo de Pi, ya que la necesitaremos para puntos posteriores. En los puntos restantes se explicarán la distribución de ficheros que se ha seguido en cada tecnología, instalación y configuración de la tecnología presente y la ejecución del cálculo de pi en este entorno.

### 4.1 Introducción

Como primer paso podríamos independizar cada parte de este proyecto por un lado. Para ello, se ha creado un directorio para cada uno de ellos, es decir un directorio para:

- **Mpi-cpi-imagen:** dónde se encuentran los tres archivos relacionados con la creación de la imagen del cálculo de Pi y MPI. De ahí el nombre de mpi-cpi. Este directorio está dentro de la máquina virtual Kali.
- **MáquinasVagrant:** se ejecuta el cálculo de pi cooperando entre las dos máquinas.
- **Kubernetes:** dónde se realiza un clúster formado por un máster y un worker para la ejecución del cálculo de pi en varios procesos distintos.

- **Docker Swarm:** la idea es parecida a la de Kubernetes, salvo la forma en la que se hace.
- **Singularity:** contiene la misma idea que los dos descritos anteriormente.

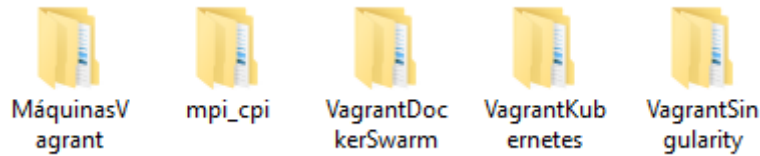


Figura 10: Directorios de trabajo cálculo de Pi

Todos estos directorios siguen una estructura similar de ficheros para la creación e instalación de lo necesario (Véase Figura 10). Aunque es cierto que no son exactamente iguales, pues cada uno tiene sus diferencias con respecto a otros.

## 4.2 Docker y cálculo de Pi en dos máquinas

### 4.2.1 Imagen mpi\_cpi

Durante el desarrollo de este proyecto se necesitan dos tipos de imágenes para la aplicación HPC del cálculo de pi. La primera de ellas se utiliza para las tecnologías de Kubernetes y Docker Swarm. La segunda solamente es para Singularity. Estas imágenes van a contener todo lo necesario para que se ejecute debidamente nuestra aplicación. Hemos usado Docker para montar la imagen que usaremos para Kubernetes y Docker Swarm. En concreto, hemos usado tres archivos:

- **DockerFile:** Está formado por todos los comandos que son necesarios para formar la imagen que deseamos.
- **Docker-entrypoint.sh:** Sirve para que la ejecución de los contenedores del clúster nunca acabe.
- **Mpi\_cpi.c:** Es el código en C del cálculo de Pi

#### 4.2.1.1 Creación imagen Docker

Para esta imagen partimos de una imagen base obtenida de una imagen común de Docker. Esta imagen base es simplemente una distribución del sistema operativo Ubuntu 20.04. Encima de esta imagen es necesario instalar OPENMPI y todas las librerías

de MPI y asignar el puerto 22 para la comunicación entre las máquinas. Además, se le da permiso de ejecución al `docker-entrypoint` y se actualizan los repositorios. Seguidamente, quedaría copiar el código de nuestra aplicación y permitir la comunicación entre las máquinas (Véase Código 4). Por último, se especifica en el código que se va a llamar a un ejecutable que va a usar el contenedor (Véase Código 5). Esta parte es fundamental, pues queremos que la ejecución en nuestro clúster nunca termine.

---

```
FROM ubuntu:20.04
EXPOSE 22
COPY docker-entrypoint.sh /docker-entrypoint.sh
RUN chmod +x /docker-entrypoint.sh
RUN apt-get update
RUN apt-get install -y net-tools
RUN apt-get install -y openssh-server
RUN apt-get install -y openmpi-bin

COPY mpi_cpi.c /root/mpi_cpi.c
RUN mpicc -o /root/mpi_cpi /root/mpi_cpi.c
RUN mkdir /root/.ssh
COPY config /root/.ssh/config
RUN chmod 664 /root/.ssh/config

# Modify `sshd_config`
RUN sed -ri 's/^#PermitRootLogin prohibit-password/PermitRootLogin yes/'
/etc/ssh/sshd_config
RUN sed -ri 's/^#PasswordAuthentication yes/PasswordAuthentication yes/'
/etc/ssh/sshd_config
RUN sed -ri 's/^#PermitEmptyPasswords no/PermitEmptyPasswords yes/'
/etc/ssh/sshd_config
RUN sed -ri 's/^UsePAM yes/UsePAM no/' /etc/ssh/sshd_config

# Delete root password (set as empty)
RUN passwd -d root

ENTRYPOINT ["sh", "/docker-entrypoint.sh"]
```

---

Código 4: Código DockerFile

Para evitar que la ejecución del clúster termine hemos creado un fichero que se encarga de crear un agujero negro que se encuentra en el archivo `Docker-entrypoint` del

DockerFile (Véase Código 4) y de esta forma el clúster estará en continua ejecución (Véase Código 5).

---

```
#!/bin/bash
service ssh restart
exec "$@"
exec tail -f /dev/null
```

---

Código 5: Docker-entrypoint.sh

Una vez definido el DockerFile es necesario construir la imagen que se va a ejecutar en un contenedor (Véase Figura 11). De esta manera ya podremos ejecutar el cálculo de Pi en nuestro contenedor.

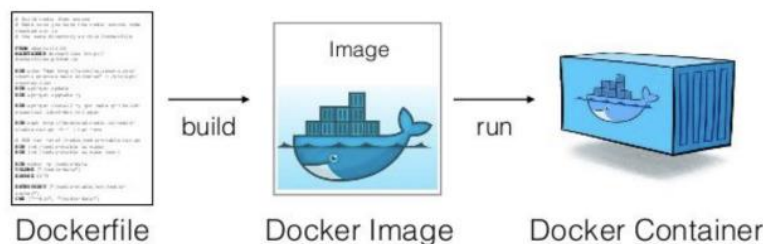


Figura 11: Pasos para ejecutar aplicación HPC en contenedor

#### 4.2.1.2 Construcción de la imagen

Para poder construir la imagen, es necesario ejecutar el comando `docker build`. Esta creación es llevada a cabo por el Daemon de Docker, el cuál va a ir ejecutando comando por comando y lanzando los resultados por pantalla (Véase Figura 12).

Es importante que nos encontremos en el mismo directorio dónde se encuentre el fichero de DockerFile (Véase Código 4) y sus dependencias, como `docker-entrypoint.sh`. En el caso de que no estemos en ese directorio, debemos cambiar el `"."`, que se refiere al directorio actual, por el directorio donde se encuentre el DockerFile descrito.

```

$ docker build -t mpi-cpi:latest .
Sending build context to Docker daemon 6.656kB
Step 1/18 : FROM ubuntu:20.04
20.04: Pulling from library/ubuntu
d5fd17ec1767: Pull complete
Digest: sha256:47f14534bda344d9fe6ffdb6effb95eefe579f4be0d508b7445cf77f61a0e5724
Status: Downloaded newer image for ubuntu:20.04
    -> 53df61775e88
Step 2/18 : EXPOSE 22
    -> Running in 0cf13e112ff4
Removing intermediate container 0cf13e112ff4
    -> 676b4bba7ad0
Step 3/18 : COPY docker-entrypoint.sh /docker-entrypoint.sh
    -> 09cdbde8c369
Step 4/18 : RUN chmod +x /docker-entrypoint.sh
    -> Running in deba5fc6e200

```

Figura 12: Construcción de la imagen para los orquestadores

El flujo típico de este tipo de fichero comienza por la selección de la imagen base. Para ello, utilizaremos la directiva **FROM**. Seguidamente, encontramos la descarga e instalación de las dependencias necesarias, donde usaremos el comando **COPY**. Esta directiva, sirve para copiar un archivo del build context y guardarlo en nuestra imagen. En definitiva, obtenemos la directiva **RUN** para ejecutar el comando necesario mientras que se crea la imagen. Además, se ha añadido una serie de comandos para que las máquinas se puedan comunicar y se facilite las comunicaciones entre las máquinas. De esto se encarga el protocolo de SSH (Véase Código 4). En última instancia una vez que hemos construido la imagen se subiría al repositorio de Docker, Docker Hub.

#### 4.2.1.3 Subir la imagen a Docker Hub

Por lo tanto, se puede definir **Docker Hub** como un repositorio mediante el que Docker obtiene todas las imágenes. Es muy similar a GitHub, solo que podemos publicar u obtener imágenes. Este almacena repositorios tanto públicos como privados. Para subir nuestra imagen a Docker Hub, lo primero que tenemos que hacer es identificarnos con nuestro usuario:

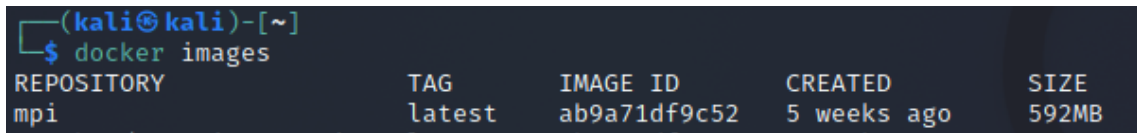
```
$sudo docker login -u toni1707 --password *****
```

El siguiente paso va a ser preparar la imagen con la siguiente nomenclatura: toni1707:nombreRepositorio:tag

Una vez que vemos tenemos construida nuestra imagen (Véase Figura 13), podemos cambiarle el tag o el nombre. Es importante saber que no se modifica la



imagen original, tan sólo se cambia el alias sobre esta imagen, pero el id sigue siendo el mismo. (Returngis, 2019)



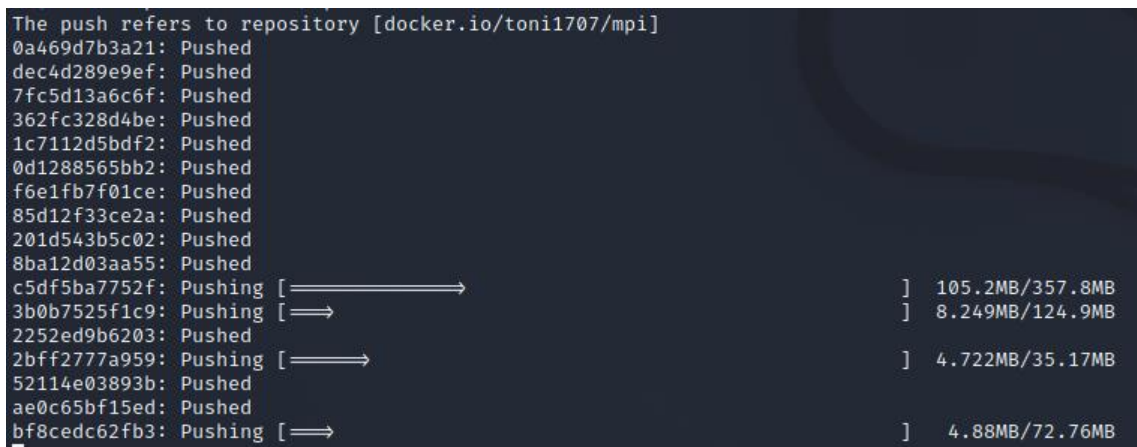
REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
mpi	latest	ab9a71df9c52	5 weeks ago	592MB

Figura 13: Imagen en Docker

Si la queremos subir a Docker Hub es importante que la prepararemos con `toni1707/nombreRepositorio:tag` para que se pueda almacenar en el repositorio.

`$ docker tag milimagen toni1707/mpi:1.0`

Para subirla introducimos el comando: `$docker push toni1707/mpi:1.0` (Véase Figura 14)



```

The push refers to repository [docker.io/toni1707/mpi]
0a469d7b3a21: Pushed
dec4d289e9ef: Pushed
7fc5d13a6c6f: Pushed
362fc328d4be: Pushed
1c7112d5bdf2: Pushed
0d1288565bb2: Pushed
f6e1fb7f01ce: Pushed
85d12f33ce2a: Pushed
201d543b5c02: Pushed
8ba12d03aa55: Pushed
c5df5ba7752f: Pushing [=====] 105.2MB/357.8MB
3b0b7525f1c9: Pushing [=====>] 8.249MB/124.9MB
2252ed9b6203: Pushed
2bff2777a959: Pushing [=====>] 4.722MB/35.17MB
52114e03893b: Pushed
ae0c65bf15ed: Pushed
bf8cedc62fb3: Pushing [=====>] 4.88MB/72.76MB
  
```

Figura 14: Subida imagen a Docker Hub

Una vez subida podemos comprobar que la imagen se ha subido correctamente en la página de [hub.docker.com](https://hub.docker.com) (Véase Figura 15).

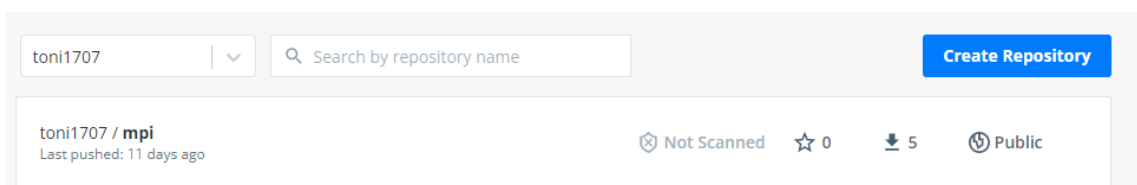


Figura 15: Imagen en Docker Hub

Una vez que disponemos de la imagen en Docker Hub ya podemos obtenerla en los orquestadores para montar nuestro clúster. Aunque con Singularity creamos el contenedor de otra manera, también podemos correr la imagen de Docker Hub como imagen de Singularity. (Sylabs.io, s.f.)

#### 4.2.2 Distribución de ficheros

Para la ejecución en Docker, tan sólo necesitamos un archivo con el código de cálculo de pi y el nombre del contenedor (Véase Figura 18). No obstante, como en este punto se va a desarrollar la ejecución del **cálculo de pi en dos máquinas**, es necesario explicar la distribución llevada a cabo (Véase Código 1):

- **Crear\_MVs.bat**

Este fichero batch consiste en ejecutar dos secuencias de comandos, estas son las siguientes:

1. `vagrant up --provision-with install`
2. `vagrant provision --provision-with ssh`

Básicamente consiste en ejecutarlo para crear las máquinas virtuales, en lugar de introducir los comandos en la Shell. Por lo tanto, es importante recordar que este sería el primer paso para crear las máquinas.

- **Install.sh**

Se ejecuta antes que configurar SSH. Este configura el fichero `/etc/hosts` con las IPs que han sido asignadas en el VagrantFile. Este fichero sirve para obtener una relación entre el nombre de la máquina y una dirección IP. Además, instala SSH y OpenMPI. En algunos casos no es necesario pues ya está incorporado en la imagen que se utiliza para lanzar el código de pi.

- **Config-ssh.sh**

Intercambia las claves SSH entre las dos máquinas creadas. Esto es fundamental para que podamos acceder sin necesidad de usar claves del usuario pues sino, no cabría la posibilidad de ejecutar programas MPI entre las distintas máquinas.

- **Ejec\_mpi**

Contiene la ejecución del cálculo de pi en paralelo en las máquinas cuya IPs quedan recogidas por el fichero `machines`.

- **Machines**

Listas de las IPs y slots de cada máquina para ejecutar el código MPI. Un **slot** define la cantidad de procesos que hay en cada nodo. Cuando hablamos de nodos, nos referimos a una máquina.

### 4.2.3 Proceso de instalación y configuración

Docker es una herramienta muy popular para implementar contenedores. Además, hay que sumarle su facilidad de compatibilidad con los distintos sistemas operativos. Dependiendo del sistema operativo, puede haber una pequeña variación a la hora de instalar y configurar esta herramienta. Podemos instalar Docker de tres formas distintas, usando los repositorios que posee, manualmente o usando scripts automatizados, ya que este ofrece múltiples repositorios y paquetes para automatizar la instalación.

Es por ello, que sólo ha sido necesario instalar el paquete Docker.io. Este paquete es mantenido por Ubuntu, a diferencia de Docker-engine que es mantenido por Docker. Pero los dos son el mismo software, sólo cambia el ente que lo gestiona. Sin embargo, es posible que la versión del paquete de instalación de esta herramienta disponible en el repositorio oficial no sea la más reciente. Por lo que, necesitaremos agregar una nueva fuente de paquetes, añadir la clave de GPG para el repositorio de Docker en nuestro sistema y por último instalar desde el repositorio de Docker en lugar del predeterminado de Ubuntu. Aunque parezca complicado a simple vista, tan sólo son una serie de comandos. Exactamente he seguido los siguientes comandos para instalar Docker:

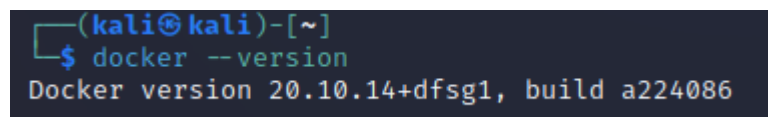
```
$sudo apt-get update
```

```
$sudo apt install -y docker.io
```

```
$sudo systemctl enable docker --now
```

```
$sudo usermod -aG docker $USER
```

A modo de comprobación de que la instalación se ha efectuado correctamente, podemos comprobar la versión (Véase Figura 16)



```
(kali㉿kali)-[~]  
$ docker --version  
Docker version 20.10.14+dfsg1, build a224086
```

Figura 16: Versión de Docker

El comando Docker sólo puede ser ejecutado por el usuario root o un usuario que tenga los mismos privilegios. Así pues, si queremos ejecutar comandos con otro tipo de usuario debemos de escalar privilegios o ejecutar comandos como súper usuario, es

decir con el comando `sudo`. Esto se debe a que el demonio Docker se une a un socket Unix en lugar de a un puerto TCP. El Socket de Unix pertenece a un usuario con privilegios de root y es lo que origina que el demonio de Docker siempre se ejecute como un usuario root. Existe una solución a este problema y principalmente, consiste en crear un grupo de Unix llamado Docker y agregarle el usuario. De esta forma, cuando se inicie el demonio Docker, va a crear el Socket mencionado con anterioridad y van a acceder los miembros que pertenezcan a ese grupo.

Para ello tenemos que crear el grupo de Docker si no está creado ya, y añadir nuestro usuario a ese grupo:

```
$sudo usermod -aG docker $USER
```

Una vez instalado podemos bajarnos una imagen y ejecutarla sin `sudo` (Véase Figura 17).

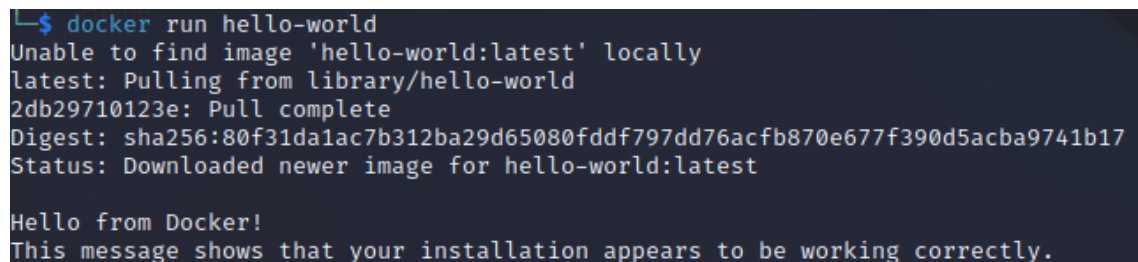
A terminal window with a dark background and light-colored text. The command `$ docker run hello-world` is entered. The output shows the process of pulling the 'hello-world:latest' image from the Docker library, including the image ID '2db29710123e', the digest 'sha256:80f31da1ac7b312ba29d65080fddf797dd76acfb870e677f390d5acba9741b17', and the status 'Downloaded newer image for hello-world:latest'. Finally, the container outputs 'Hello from Docker!' and a message stating 'This message shows that your installation appears to be working correctly.'

Figura 17: Ejecución de la imagen hello-world

#### 4.2.4 Ejecución Docker

Lo primero que debemos de hacer para poder ejecutar la aplicación sería bajarnos la imagen que está en el repositorio de Docker Hub. Para ello, `$docker push toni1707/mpi:latest`. Una vez hecho esto, si mostramos las imágenes que hay disponibles nos debería aparecer la imagen bajada. Como ya tenemos la imagen, podemos crear el contenedor con `$docker create [imagen]`. También podemos crear y ejecutarlo en un mismo paso cambiando `create` por `run`. Sin embargo, se va a quedar en ejecución siempre debido al agujero negro creado para ello y de esta manera nunca termine. Como lo que queremos es ejecutar el código de Pi, podemos crearlo y volvemos a tener dos opciones. En ambas necesitamos saber el ID del contenedor creado: `$docker ps`. Una opción sería crearnos una Shell, `$docker exec -it ID bash` y la otra sería ejecutar directamente el fichero deseado sin crearnos una Shell (Véase Figura 18).

```
(kali㉿kali)-[~]
$ docker exec -it 84c524f7a03f /root/mpi_cpi
Process 0 of 1 is on 84c524f7a03f
pi is approximately 3.1415926544231341, Error is 0.0000000008333410
wall clock time = 0.000276
```

Figura 18: Ejecución cálculo de Pi en Docker

#### 4.2.5 Ejecución del cálculo de Pi en dos máquinas virtuales

La ejecución realizada en Docker se produce desde una sola máquina (Véase Figura 18). Para llevar la ejecución con más máquinas podemos usar Vagrant, para que cree más máquinas y lanzar la aplicación con un mayor número de procesos. Podemos crear dos nodos y configurar las máquinas para que puedan acceder de una a otras por SSH sin que exista la necesidad de utilizar las claves del usuario. En este caso, sólo se ha creado dos nodos, pero podríamos tener tantos como deseemos. Toda la configuración de estas queda recogida en el VagrantFile (Véase Código 1). Para crear las máquinas ejecutamos el archivo **crear\_MVs.bat**. Ahora a diferencia de la ilustración anterior, antes sólo había un proceso y ahora hay dos que se ejecuta en un nodo cada uno (Véase Figura 19). Esta ejecución cooperativa del cálculo de Pi puede servir como antecedente a la creación de un clúster que desarrollaremos en puntos posteriores. El código del ejecutable de `ejec_mpi.sh` es el siguiente: `$mpirun -np 2 --hostfile machines`

```
vagrant@node-01:/vagrant$ ./ejec_mpi.sh
Process 0 of 2 is on node-01
Process 1 of 2 is on node-02
pi is approximately 3.1415926544231318, Error is 0.0000000008333387
wall clock time = 0.018846
```

Figura 19: Ejecución MPI\_cálculoPI en dos procesos

## 4.3 Kubernetes

### 4.3.1 Imagen mpi\_cpi en Kubernetes

La imagen que contiene al cálculo de Pi usada en Kubernetes es la creada a través de Docker (Véase 4.2.1).

#### 4.3.2 Distribución de ficheros

Se añaden nuevos ficheros. Ahora en el VagrantFile se menciona los siguientes ficheros (Véase Código 2). Además, encontramos otros distintos:

- **Configure-master-node.sh**

Este fichero va a servir para configurar el nodo máster y así permitir que los workers se unen al nodo máster. En este fichero se va a inicializar el nodo máster, vamos a crear el comando para que se unan en un archivo `join_command.sh`. Además, se configurará `kubectl`. Aunque todo suene un poco abstracto, posteriormente se explica un poco más con detalle.

- **Configure-worker-node.sh**

Una vez que tenemos el comando para unirse, tan sólo hay que abrir el archivo dónde se encuentra este, `join_command.sh`.

- **Ejec\_mpi\_k8s.sh**

Sirve para crear el fichero `machines` y ejecutar el cálculo de pi.

- **Install-kubernetes-dependencies.sh**

Se encuentra todos los paquetes, configuración e instalación de Docker para lograr instalar las dependencias de Kubernetes.

- **mpi-cpi-deployment.yml**

Sirve para crear el Deployment en Kubernetes

No hace falta instalar OpenMPI ni configurar lo de SSH pues ya queda recogida en la imagen que hemos subido a Docker Hub y lo usaremos posteriormente.

#### 4.3.3 Proceso de instalación y configuración

Para llevar a cabo la instalación y configuración de esta parte es necesario tener en cuenta que vamos a necesitar dos máquinas. Para generarlas, se generan como sabemos con el VagrantFile correspondiente (Véase Código 2). Así que hay que ejecutar el archivo **crear\_MVs.bat**.

Un clúster consiste en la conexión de dos o más computadoras para ejecutar diferentes tareas. Se consigue la independencia de trabajos, por lo que sólo tiene acceso al clúster la persona local.

Para instalar Kubernetes, hay que instalar todas las dependencias en cada una de las máquinas virtuales. Por ello se usa un script para instalar todos los paquetes necesarios, configuración de hosts, instalar Docker, etc. Una vez tenemos instalado esta parte, necesitamos configurar lo que va a ser nuestro nodo maestro y sus workers. Se inicializará Kubernetes a través de kubeadm, que es una herramienta para desplegar un clúster de Kubernetes de manera sencilla.

Principalmente, el script para instalar las dependencias se basa en cuatro sencillos pasos:

1. **Instalación de paquetes.** Es necesario instalar en todas las máquinas los paquetes necesarios para desplegar el plano de control. Exactamente, es necesario instalar cuatro paquetes; kubeadm, Docker, kubelet y kubectl.
  - a. Kubeadm: Herramienta de CLI que se encarga de instalar y configurar los componentes necesarios del clúster.
  - b. Docker: Componente que ejecuta los contenedores, que hemos visto anteriormente.
  - c. Kubelet: Servicio que se ejecuta en cada uno de los nodos y se encarga de las operaciones.
  - d. Kubectl: Herramienta de CLI que se usa para enviar comandos al clúster a través de la API.
2. **Configuración en máquinas.** Con el comando de kubeadm init, inicializaremos el nodo máster y nos devolverá un comando para ejecutar en los worker y hacer que posteriormente se unan al mismo.
3. **Configurar kubelet.** Este servicio necesita llamar al endpoint correcto en el directorio /kube. Así que es necesario crearlo y que se llame config.
4. **Instalación de la red interna.** Necesitamos que los pods tengan una red interna.

Para verificar que todo se ha configurado correctamente, accedemos a la máster principal y obtenemos los nodos configurados (Akgül, 2021). Debería de haber dos, un máster y un nodo-01 (Véase Figura 20).

```
vagrant@master:~$ kubectl get nodes
NAME        STATUS    ROLES                  AGE    VERSION
master      Ready     control-plane,master   10d    v1.20.11
node-01     Ready     <none>                 10d    v1.20.11
```

Figura 20: Nodos configurados en Kubernetes

#### 4.3.4 Ejecución Kubernetes

Se ha creado un clúster con dos roles, uno para cada rol. Ahora vamos a tener un máster y un worker01. Para llevar a cabo la implementación, se usa el archivo `mpi-cpi-deployment.yml` (Véase Código 6).

---

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mpi-cpi-deployment
  namespace: default
  labels:
    app: mpi
spec:
  replicas: 4
  selector:
    matchLabels:
      app: mpi
  template:
    metadata:
      labels:
        app: mpi
    spec:
      containers:
        - name: mpi
          image: toni1707/mpi:1.0
          ports:
            - containerPort: 22
```

---

Código 6: Creación deployment en Kubernetes

En este tipo de archivo es necesario aclarar algunos conceptos:

- **apiVersion:** define la versión del API que queremos utilizar para el recurso. En nuestro caso como queremos crear un deployment escogemos `apps/v1`



- **kind:** es el tipo de objeto que queremos crear. Si hubiéramos querido crear un pod solamente, usaríamos Pod. Un pod es un conjunto formado por uno o varios contenedores. Es la unidad más pequeña dentro del entorno de Kubernetes.
- **Metadata** que contiene tres nodos; name, namespace y labels. Aunque no sea información estándar, es útil. Como puede ser el caso del tipo de aplicación MPI.

Un deployment requiere estos tres campos, por lo que son necesarios. Por lo restante, es algo intuitivo pues lanzamos 4 réplicas y la imagen deseada.

Una vez que sabemos que un Pod es la unidad más pequeña, decir que el deployment es la unidad de más alto nivel en Kubernetes. Las réplicas nos van a asegurar que mantengamos un conjunto estable de réplicas de Pods ya que el ReplicaSet va a sustituir a los Pods que se eliminen o finalicen (Véase Figura 21).

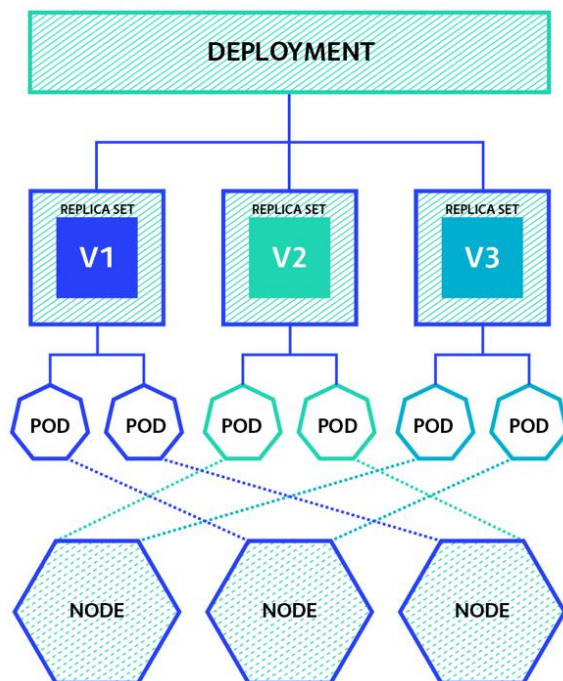


Figura 21: Conceptos Kubernetes

Los nodos son las máquinas virtuales a los que se les unen distintos Pods.

Se utiliza kubectl apply para crear el deployment (Véase Figura 22).

```
vagrant@master:/vagrant$ kubectl apply -f ./mpi-cpi-deployment.yml
deployment.apps/mpi-cpi-deployment created
```

Figura 22: Kubectl apply " .yaml"

Cuando aplicamos este fichero se crea un deployment (Véase Figura 23) y así creamos un total de 4 réplicas.

```
vagrant@master:/vagrant$ kubectl get deployments
NAME                READY   UP-TO-DATE   AVAILABLE   AGE
mpi-cpi-deployment  4/4     4            4           32s
```

Figura 23: Lista de deployments Kubernetes

Una vez aplicado el “.yaml” con nuestra imagen, puertos del contenedor, réplicas, etc. todos los contenedores deberían de estar funcionando (Véase Figura 24).

```
vagrant@master:/vagrant$ kubectl get pods
NAME                                     READY   STATUS    RESTARTS   AGE
mpi-cpi-deployment-68ff94c88d-2sp7j    1/1     Running   0          24s
mpi-cpi-deployment-68ff94c88d-f6zb4    1/1     Running   0          24s
mpi-cpi-deployment-68ff94c88d-mspv9    1/1     Running   0          24s
mpi-cpi-deployment-68ff94c88d-w9r4s    1/1     Running   0          24s
```

Figura 24: Pods creados en Kubernetes

Como podemos ver hay 4 pods que se comunican entre sí gracias al fichero machines. Luego existen dos dentro de cada máquina que hemos creado en el VagrantFile, estas se conocen a través del SSH (Véase Figura 25).

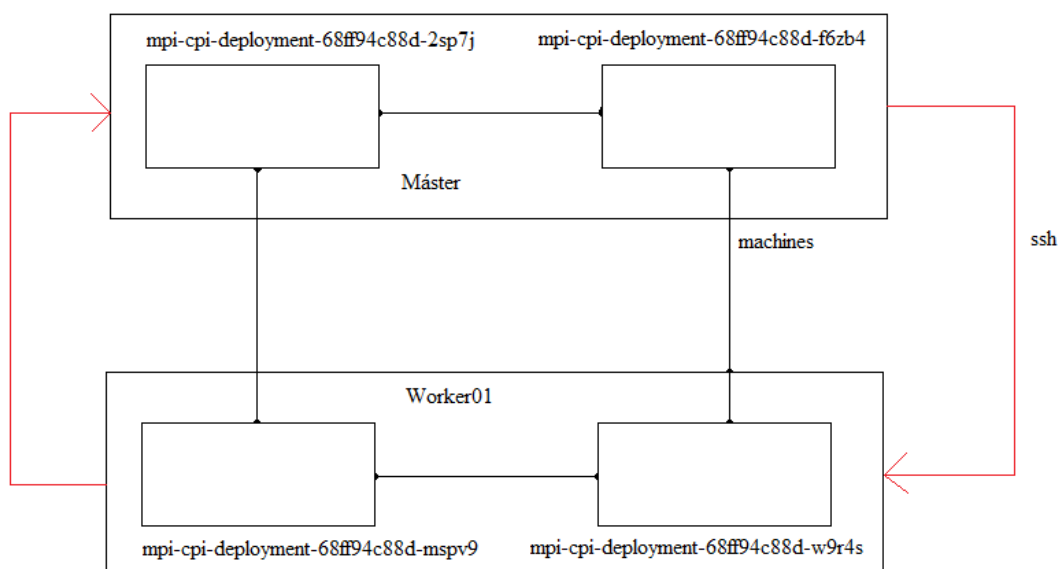


Figura 25: Descripción pods en Kubernetes

#### 4.3.4.1 Automatización de la creación fichero machines y su posterior ejecución de la aplicación

Como en los casos anteriores, se lleva a cabo mediante un archivo .sh. Pero primero, necesitamos el fichero machines con las IPs y sus slots correctamente. (Véase Código 7).

---

```
kubectrl describe pods -n default | grep IP: | awk -F: '{print $2}' | uniq | awk -F" " '{print $1, "slots=1" }' >machines
```

```
pod=$(kubectrl get pods -n default | awk 'NR==2{print $1}')
```

```
kubectrl cp machines default/${pod}:/root
```

```
kubectrl exec ${pod} -- mpirun -v -np 4 -hostfile /root/machines --allow-run-as-root /root/mpi_cpi
```

---

Código 7: Código ejec\_mpi\_k8s.sh

Para crear este fichero debemos de descubrir las IPs que están asignadas a cada contenedor. Para saberlas hay que ejecutar el comando *\$kubectrl describe pods*. Este comando va a describir todos los pods que haya, y de esta manera podremos obtener la IP de cada una de ella, filtrando por **grep** IP. Luego podemos añadir otros comandos como los explicados antes; **uniq** y **awk** para quedarnos con la IP deseada y mostrar el fichero machines de la forma deseada para poder compilarlo, añadiéndole los slots pertinentes.

Como ya tenemos el fichero machines con las IPs y slots de cada máquina, habrá que copiar ese fichero en el directorio `/root` del primer pod. Para obtener el primer pod, podremos filtrar la salida del *\$kubectrl get pods | awk 'NR==2{print \$1}'*. A diferencia de Docker Swarm, `NR==2` no está descrito. Esta parte de comando sirve para quedarnos con la segunda línea. También se puede usar otros operadores lógicos como `>` o `<`. A consecuencia de la obtención del primer pod, le copiamos el fichero generado anteriormente con **cp** y ejecutamos el **mpirun** dentro del primer pod. Para ejecutarlo se realizaría con el comando *\$kubectrl exec PRIMER-POD --mpirun ...*

Todo este proceso queda automatizado a través del fichero `ejec_mpi_k8s.sh` (Véase Código 7). Por lo que al ejecutarlo obtendríamos el cálculo de Pi en Kubernetes (Véase Figura 26).

```
vagrant@master:/vagrant$ ./ejec_mpi_k8s.sh
Warning: Permanently added '192.168.1.10' (ECDSA) to the list of known hosts.
Warning: Permanently added '192.168.1.11' (ECDSA) to the list of known hosts.
Warning: Permanently added '192.168.1.13' (ECDSA) to the list of known hosts.
Process 0 of 4 is on mpi-cpi-deployment-68ff94c88d-2sp7j
Process 2 of 4 is on mpi-cpi-deployment-68ff94c88d-mspv9
Process 1 of 4 is on mpi-cpi-deployment-68ff94c88d-f6zb4
Process 3 of 4 is on mpi-cpi-deployment-68ff94c88d-w9r4s
pi is approximately 3.1415926544231239, Error is 0.000000008333307
wall clock time = 0.180264
```

Figura 26: Ejecución cálculo de Pi en Kubernetes

## 4.4 Docker Swarm

### 4.4.1 Imagen mpi\_cpi usada en Docker Swarm

La imagen que contiene al cálculo de Pi usada en Docker Swarm es la creada a través de Docker (Véase 4.2.1).

### 4.4.2 Distribución de ficheros

La distribución que se ha seguido guarda cierta similitud con respecto al punto anterior (Véase 4.3.2). Ahora en el VagrantFile se menciona los siguientes ficheros (Véase Código 3). Aunque, podemos encontrar otros distintos:

- **Ejec\_mpi\_dockerSwarm**  
Sirve para crear el fichero machines propio y ejecutar el cálculo de pi en Docker Swarm.
- **mpi-cpi-deployment-swarm.yml**  
Sirve para crear el servicio que deseamos.
- **Configure\_manager.sh**  
Este fichero va a servir para configurar el nodo máster y así permitir que los workers se unen al nodo máster. En este fichero se va a inicializar el nodo máster, vamos a crear el comando para que se unan en un archivo worker\_token.
- **Configure\_worker.sh**

Una vez que tenemos el comando para unirse, tan sólo hay que abrir el archivo dónde se encuentra este, `worker_token`.

- **Install-docker.sh:**

En este fichero instalamos Docker y sus dependencias ya que es necesario para el uso de Docker Swarm.

Tampoco hace falta instalar OpenMPI ni configurar lo de SSH ya que está en la imagen.

#### 4.4.3 Proceso de instalación y configuración

Para llevar a cabo la instalación y configuración de esta parte es necesario tener en cuenta que vamos a necesitar dos máquinas. Para generarlas, se generan como sabemos con el VagrantFile correspondiente (Véase Código 3). Así que hay que ejecutar el archivo **crear\_MVs.bat**.

Docker Swarm es una de las herramientas más populares como marco de orquestación. Exactamente se puede dividir en tres sencillos pasos. El primer paso sería instalar Docker, junto a todas sus dependencias. También es necesario añadir el usuario Vagrant al grupo de Docker y así tener privilegios. Además, se ha creado un `mánager` y un `worker` también.

Una vez que tenemos instalado Docker e indicadas las IPs de las máquinas, es necesario crear el clúster Swarm. Primero se inicializa el modo `manager` y luego se une los `workers` al clúster. En nuestro caso, como sólo disponemos de un `worker01`, será este el que se una al `manager`. Por lo tanto, para inicializarlo en el nodo principal, sería necesario con `docker swarm init -advertise-addr IP`. Esto nos generará un `join-token` para que podamos unir los `worker` al mismo, este quedará recogido en el fichero `worker_token`. Para poder llevarlo a cabo, ejecutamos el comando obtenido en el nodo deseado.

Si todo ha ido correctamente se deberían mostrar los nodos correctamente (Véase Figura 27).

```
vagrant@manager:~$ docker node ls
```

ID	HOSTNAME	STATUS	AVAILABILITY	MANAGER STATUS	ENGINE VERSION	
ady25hxqrkga1c6m6wqv3qd1x	*	manager	Ready	Active	Leader	20.10.16
vmy8qgj3u1kbj57ygq23yvvey8		worker01	Ready	Active		20.10.16

Figura 27: Nodos de Docker Swarm

#### 4.4.4 Ejecución Docker Swarm

Dentro de este clúster existen dos roles, manager y worker. En nuestro caso hay uno que ocupa cada rol. El primero de ellos manda ordenes al worker y administra al clúster.

Hay dos formas de crear el servicio con nuestra imagen subida a Docker Hub. Una de ellas sería creando el servicio a través de comandos (Véase Figura 28).

```
$docker service create toni1707/mpi-cpi:1.0
```

```
vagrant@manager:~$ docker service create toni1707/mpi-cpi:1.0
ek3pgkt7ntaen9eoe1q4cfffht
overall progress: 1 out of 1 tasks
1/1: running [=====>]
verify: Service converged
vagrant@manager:~$ docker service ls
```

ID	NAME	MODE	REPLICAS	IMAGE	PORTS
ek3pgkt7ntae	strange_mestorf	replicated	1/1	toni1707/mpi-cpi:1.0	

Figura 28: Creación servicio Docker Swarm

Y posteriormente escalarlo hacia arriba para cambiar el número de instancias de contenedores, pasando de 1 a N. De esta forma, podemos conseguir que dos contenedores se ejecuten en el manager y otras dos en el worker01, si elegimos 4 réplicas. Otra opción para crear el servicio deseado sería a través de un fichero Docker Compose. En él sólo debemos de indicar los servicios por el que va a estar formado, con sus respectivas imágenes, los puertos, el número de réplicas, etc. (Véase Código 8)

```
version: "3.9"
services:
  mpi-cpi:
    image: toni1707/mpi:latest
    deploy:
      replicas: 4
    ports:
      - "22"
```

Código 8: Fichero Docker Compose

Docker Stack es una tecnología que está instalada en Docker. Esta se encarga de desplegar un conjunto de servicios definidos en un compose. Sin embargo, es importante que la máquina desde donde se ejecute tenga rol de manager y unida a un Swarm. Además, debe tener una versión mayor que la 3.0, como es la 3.9 en nuestro caso. Para desplegar el stack se realizaría con el siguiente comando, estando en el directorio compartido con el host que es dónde tenemos el archivo .yaml:

```
$docker stack deploy -c mpi-cpi-deployment-swarm.yaml mpi
```

Una vez ejecutado este comando, podemos ver cómo obtenemos cuatro réplicas y el servicio creado correctamente (Véase Figura 29).

```
vagrant@master:/vagrant$ docker service ls
ID                NAME          MODE           REPLICAS  IMAGE              PORTS
s8niklxxkxym     mpi_mpi-cpi   replicated     4/4       toni1707/mpi:latest *:30001->22/tcp
```

Figura 29: Servicio creado con .yaml

#### 4.4.4.1 Automatización de la creación fichero machines y su posterior ejecución de la aplicación

Todo este proceso se lleva a cabo mediante un archivo .sh, que son archivos creados y guardados en el lenguaje Bash. Así pues se han creado para interpretar los comandos para realizar lo descrito en el título.

Para crear el fichero machines, cuyo funcionamiento sirve para indicar las IPs de todos los contenedores y sus slots, que van a ser iguales a 1 debido a que buscamos que se ejecute un proceso por nodo, hay que tener en cuenta una serie de pasos:

1. Obtener la subred en la que van a estar todas las IPs de los contenedores. Para ello, primero hay que saber las subredes que hay disponibles y escoger la que estamos buscando. Para saberlas, introducimos el comando *\$docker network ls*. Una vez que hemos elegido la que necesitamos, para calcular la subred habría que hacer *\$docker network inspect subredEscogida*.
2. Obtener todos los IDs de cada contenedor, tanto los que se ejecutan en el manager como en el worker. Para ello usamos el comando *\$docker service ps servicioEscogido*

3. Una vez que tenemos todos los identificados, hay que inspeccionarlos para saber la IP de cada uno de ellos, *\$docker inspect IDContenedor*

Una vez que sabemos los pasos que hay que seguir, se ha automatizado todo este proceso (Véase Código 9).

---

```
if [ -f machines ]
then
rm machines
fi
docker service ps mpi_mpi-cpi | grep Running | awk '{print $1}' > IdContenedores
while read line
do
docker inspect "$line" | grep "10.0.*" | tr -d '"' | awk -F/ '{print $1}' | sed -r 's/\s+//g'
> IpContenedor1
tail -1 IpContenedor1 > IpContenedor
cat IpContenedor | awk -F" " '{print $a, "slots=1"}' >> machines
done < IdContenedores

rm IpContenedor1
rm IpContenedor

maquina=$(docker ps | awk 'NR==2{print $12}')
subnet=$(docker network inspect mpi_default | grep Subnet | awk '{print $2}' | uniq |
tr -d '"' | tr -d ',')
docker cp /vagrant/machines ${maquina}:/root/
docker exec ${maquina} mpirun -np 4 -hostfile /root/machines --allow-run-as-root --
mca oob_tcp_if_include ${subnet} /root/mpi_cpi
```

---

Código 9: Código ejec\_mpi\_dockerSwarm.sh

Lo primero es comprobar si existe el fichero de machines, porque se va a ir sobrescribiendo para guardar todas las IPs y necesitamos que este vacío al principio. Después, sería necesario obtener las id de los contenedores para ir obteniendo la IP de cada una, por ello hay que filtrar por los que están activos, es decir grep Running. Por cada línea del archivo obtenido, es decir cada ID vamos a ir obteniendo cada IP de cada uno de ellos.



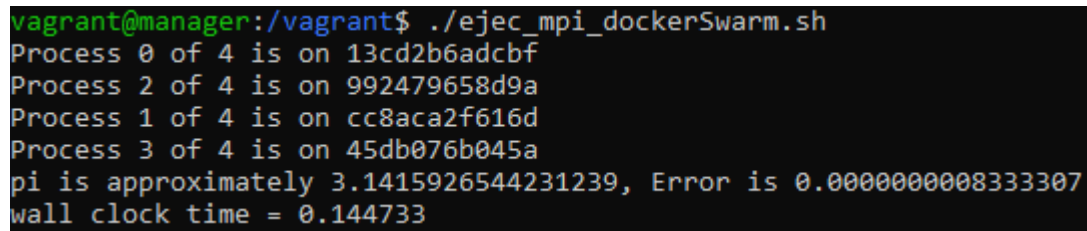
Por último, copiamos el fichero `machines`, que contiene todas las IPs para ejecutar el código deseado en la primera máquina y lo ejecutamos. Algunos comandos de `bash` usados serían:

- **grep** [opciones] patrón [archivo]. Su función principal se basa en la salida por pantalla de las líneas que contengan las coincidencias descritas para la expresión regular.
- **awk** '{action}'. Sirve para procesar texto. Por ejemplo, `awk '{print $1}'` representa el primer campo en el archivo, que es el que vamos buscando para obtener la ID de los contenedores. Además, podemos utilizarlo para eliminar caracteres no deseados. Esto ocurre cuando queremos eliminar la máscara (`/**`) y quedarnos con la IP anterior a ella. Para esto podemos usar `awk -F/ '{print $1}'`
- **tr** [opciones] caracteres1 caracteres2. Sirve para reemplazar caracteres, palabras, eliminar caracteres no deseados, etc. En nuestro caso, lo hemos usado para eliminar comillas o comas: `tr -d '"'`
- **uniq**. Su función es omitir, en el caso de que hubiera, duplicados.
- **sed**. Permite manipulaciones básicas de texto en la entrada de texto. Su principal uso es para eliminar espacios en blanco, `sed -r 's/\s+//g'`.

Una vez que tenemos el fichero con las IPs de cada parte del clúster, es hora de ejecutar el código de MPI, este proceso junto a la obtención del fichero `machines` queda recogido en el `sh`, `ejec_mpi_dockerSwarm.sh`.

La parte de código para ejecutar el código de Pi es sencilla tan sólo hay que ejecutar las cuatro últimas instrucciones del código anterior (Véase Código 9). Tan solo debemos de obtener el nombre de una máquina y la subred. Una vez esto copiamos el fichero `machines` en el directorio `:/root` en una máquina y ejecutamos el `mpirun` dentro de esa máquina obtenida. Importante el comando de `--allow-run-as-root` debido a que Open Mpi se niega a ejecutar `mpirun` como usuario `root` y debemos de ponerlo para que nos deje.

A la hora de ejecutar el cálculo de Pi (Véase Código 9), deben de producirse 4 procesos y cada uno se ejecute en un contenedor distinto (Véase Figura 30).



```
vagrant@manager:/vagrant$ ./ejec_mpi_dockerSwarm.sh
Process 0 of 4 is on 13cd2b6adcbf
Process 2 of 4 is on 992479658d9a
Process 1 of 4 is on cc8aca2f616d
Process 3 of 4 is on 45db076b045a
pi is approximately 3.1415926544231239, Error is 0.0000000008333307
wall clock time = 0.144733
```

Figura 30: Ejecución cálculo de PI en Docker Swarm

La idea de estos cuatro procesos es parecida a la descrita en el caso de Kubernetes. Disponemos de dos máquinas principales, y dentro de estas se ejecutan dos contenedores distintos (Véase Figura 25).

## 4.5 Singularity

### 4.5.1 Imagen mpi\_cpi en Singularity

Respecto a la creación de la imagen de Singularity es distinta pues así lo implementa Singularity. Aunque eso no quita que la idea sea la misma con respecto a la explicada con anterioridad.

#### 4.5.1.1 Creación imagen

Para crear la imagen de contenedor e implementarla en un clúster con una implementación de MPI hay que seguir una serie de pasos consecutivos.

En primer lugar, hay que construir una imagen a partir del archivo de definición (.def). Este tipo de archivo sirve para construir un contenedor personalizado. En él hay que incluir detalles como el SO base o el contenedor base, software que vamos a instalar, variables de entorno, archivos que queremos que comparta con nuestra máquina, etc. (Véase Código 10).

---

**Bootstrap:** docker

**From:** ubuntu:latest

**%files**

mpi\_cpi.c /opt

**%environment**

export OMPI\_DIR=/opt/mpi

(...)

**%post**

echo "Instalación paquetes requeridos"

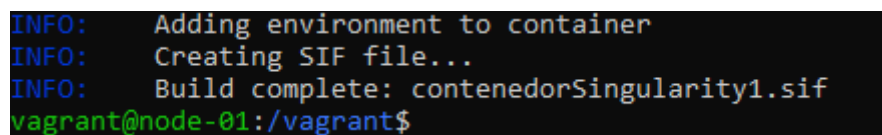
```
apt-get update && apt-get install -y wget git bash gcc gfortran g++ make file
(...)
cd /opt && mpicc -o mpi_cpi mpi_cpi.c
```

#### Código 10: Creación del contenedor en Singularity

- Definimos el encabezado que está formado por la instrucción Bootstrap que hace referencia al tipo de base Docker para poder bajarnos el contenedor de la última versión de Ubuntu. (Singularity-userdoc.readthedocs.io, 2021)
- En la sección **%files**, copiamos el cálculo de PI para ejecutarlo dentro del contenedor, concretamente en la carpeta /opt
- En la sección **%environmet**, establecemos las variables de entorno que van a estar disponibles dentro del contenedor
- En la sección **%post**, instalamos los paquetes requeridos y OpenMPI. Para instalar OpenMPI lo realizamos desde el repositorio. Por último compilamos el cálculo de pi. (Singularity-userdoc.readthedocs.io, 2021)

##### 4.5.1.2 Construcción imagen

Para construir el contenedor ejecutaríamos el siguiente comando: `$sudo singularity build contenedorSingularity1.sif contenedorSingularity.def` (Véase Figura 31).



```
INFO: Adding environment to container
INFO: Creating SIF file...
INFO: Build complete: contenedorSingularity1.sif
vagrant@node-01:~/vagrant$
```

Figura 31: Construcción del contenedor en Singularity

Sin embargo, hay que tener en cuenta que necesitamos Open MPI tanto dentro como fuera del contenedor. Las llamadas MPI realizadas dentro de nuestro contenedor son comunicadas al demonio MPI del host. En consecuencia, como hemos incluido Singularity hace que se reduzca la sobrecarga de ejecutar el código dentro del contenedor.

Por último, aunque sea sencillo ejecutar aplicaciones MPI en este entorno, las librerías MPI del contenedor y del host deben de ser compatibles. Por lo tanto, dentro del contenedor sólo necesitamos las librerías de MPI y el código de Pi (Véase Figura 32).

```
vagrant@node-01:/vagrant$ singularity shell contenedorSingularity.sif
Singularity> mpiexec --version
bash: mpiexec: command not found
Singularity> cd /opt/mpi && ls
bin etc include lib share
```

Figura 32: Librerías dentro de contenedor de Singularity

#### 4.5.2 Distribución de ficheros

Es el que más cambia con respecto a todos. Ahora en el VagrantFile se mencionan los siguientes ficheros (Véase Código 1). Además, en él encontramos otros ficheros:

- **Ejec\_mpi\_Singularity**  
Contiene las instrucciones propias para ejecutar el cálculo de PI.
- **Install.sh**  
Se ha añadido la instalación de Singularity.
- **Config-ssh.sh**  
Se usa para lo mismo que lo descrito para Docker (Véase 4.2.2).
- **contenedorSingularity.def**  
Ahora no necesitamos subir la imagen a ningún repositorio, necesitamos crear el contenedor en local ya que necesitamos el PATH de este.
- **Mpi\_cpi.c**  
Código en c del cálculo de PI.

#### 4.5.3 Proceso de instalación y configuración

Para llevar a cabo la instalación y configuración de esta parte es necesario tener en cuenta que vamos a necesitar dos máquinas. Para generarlas, se generan como sabemos con el VagrantFile correspondiente (Véase Código 1). Así que hay que ejecutar el archivo **crear\_MVs.bat**.

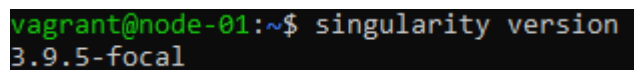
La instalación de Singularity puede llegar a ser la más sencilla que se ha llevado a cabo durante la realización de este trabajo. Para instalarlo, tan sólo hay que modificar el archivo de install.sh. En este archivo, incluiremos un método que lleve a cabo la instalación de este.

Lo primero que hay que hacer sería, instalar los paquetes necesarios.

- **Apt-get update:** Actualiza las listas de paquetes para los que necesitan ser actualizados, así como los paquetes nuevos que han llegado a los repositorios. Se obtiene en la ubicación `/etc/apt/source.list`.
- **Apt-get install**
  - `build-essential`, `libseccomp-dev`, `pkg-config`, `squashfs-tools`, `cryptsetup`

Una vez instalados, necesitamos obtener Singularity. Para ello, obtendremos mediante la herramienta Wget una descarga del paquete para Ubuntu de este a través de HTTPS. Una vez obtenido, tan sólo necesitamos descomprimir el deb que hemos obtenido y ya estaría.

Para corroborar que la instalación es correcta, podemos comprobar la versión (Véase Figura 33).



```
vagrant@node-01:~$ singularity version
3.9.5-focal
```

Figura 33: Versión de Singularity

#### 4.5.4 Ejecución Singularity

Se han creado un clúster con dos nodos, `node01` y `node02`, especificados en el `VagrantFile`. A diferencia de los tratados con anterioridad, este no es necesario que obtengamos nuestra imagen de ningún lado pues vamos a crear nuestro propio contenedor. Este contenedor va a ser un fichero que va a compartir la máquina local con las creadas a través de Vagrant.

Para realizar la ejecución paralela del cálculo de pi es necesario indicar el fichero `machines`, que contiene la IP de las dos máquinas. Ahora ya no es necesario que se genere automáticamente, pues conocemos las IPs. También tenemos que indicar el path del contenedor (`.sif`) que se genera a través del `.def` y el archivo generado al usar `mpicc` al cálculo de pi (Véase Código 11), que queda recogido dentro del contenedor ya que está en el archivo de creación del contenedor.

---

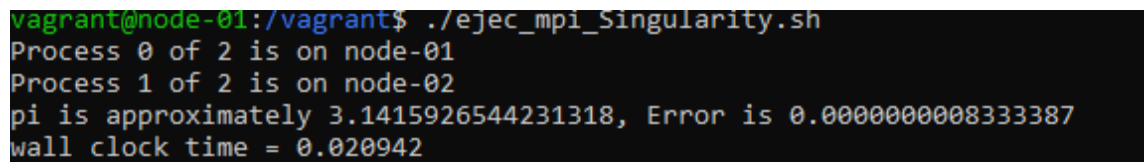
```
mpirun -np 2 --hostfile machines singularity exec contenedorSingularity.sif
/opt/mpi_cpi
```

---

Código 11: Código ejec\_mpi\_Singularity

Para obtener el fichero machines, simplemente lo podemos crear a mano pues sabemos que las IPs son las mismas que a la hora de creación en el VagrantFile.

Por último ejecutamos el cálculo de Pi (Véase Código 11), sólo que ahora hay que indicarle el PATH del contenedor creado anteriormente y el de la salida del programa que está dentro del contenedor (Véase Figura 34).



```
vagrant@node-01:/vagrant$ ./ejec_mpi_Singularity.sh
Process 0 of 2 is on node-01
Process 1 of 2 is on node-02
pi is approximately 3.1415926544231318, Error is 0.000000008333387
wall clock time = 0.020942
```

Figura 34: Ejecución cálculo de Pi en Singularity (Véase Código 11)

# Capítulo 5

## Virtualización e implementación del modelo

### WRF

---

En este quinto capítulo se procede a la ejecución del modelo WRF. Primeramente, se va a explicar la creación de la imagen WRF para la tecnología de contenedor software Singularity. El objetivo principal de este capítulo es la comparación del uso o no de Singularity. Como en el capítulo anterior, se procede a explicar la distribución de ficheros, configuración y ejecución de este modelo. Idealmente se podría haber ejecutado el modelo WRF para Docker Swarm y Kubernetes pero por motivos de tiempo y la no disposición de un clúster real de ninguno de los dos tipos se ha considerado abordar tan sólo el caso de Singularity debido a que en principio sí que se podría ejecutar sobre un clúster real, el clúster Galgo del instituto de Investigación en Informática de Albacete (I3A).

### 5.1 Introducción

Como primer paso podríamos independizar cada parte de este proyecto por un lado. Para ello, se ha creado un directorio para cada uno de ellos, es decir un directorio para:

- **WRF-1:** en este directorio vamos a encontrar el modelo WRF y Singularity, ya que la idea es ejecutarlo a través de los contenedores.
- **WRF-2:** dónde se encuentran todos los archivos para ejecutar el modelo WRF con nuestros datos de entrada.

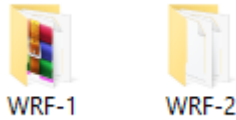


Figura 35: Directorios de trabajo WRF

Estos directorios (Véase Figura 35), van a compartir archivos, la única diferencia va a ser la intervención de Singularity.

## 5.2 WRF

La idea es comparar el tiempo de ejecución entre la reproducción de un modelo mediante el uso de contenedor. Una de ellas directamente con el WRF instalado en las máquinas virtuales y la otra va a ser con el WRF instalado dentro del contenedor de Singularity de ambas máquinas también. La única diferencia que va a haber es la presencia de Singularity, como se ha indicado antes.

### 5.2.1 Distribución de ficheros

La distribución de ficheros que se ha seguido guarda cierta similitud con la ejecución del cálculo de Pi en dos nodos, sin ningún orquestador. Se encuentra en el directorio WRF\_2.

En los tres primeros archivos que se van a mencionar, el contenido es el mismo (Véase 4.2.2)

- **Crear\_MV.bat**
- **Config-ssh.sh**
- **Install.sh**
- **Datos de entrada:** namelis.input, wrfbdy\_d01, wrfinput\_d01, wrfinput\_d02.

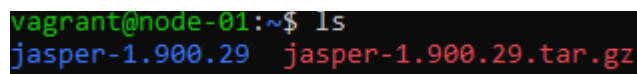
Además, como en todos los casos debemos obtener un fichero machines, que aunque en este no se cree automáticamente, es necesario para que las máquinas puedan cooperar entre sí.



### 5.2.2 Proceso de instalación y configuración

Para llevar a cabo la instalación y configuración de esta parte es necesario tener en cuenta que vamos a necesitar dos máquinas. Para generarlas, se generan como sabemos con el VagrantFile correspondiente (Véase Código 1).

En primer lugar, vamos a instalar el software WRF versión 4.2.1 en nuestras máquinas virtuales. (Pratiman, 2020) Como la parte del WPS no la vamos a utilizar no es necesario instalarla. Debemos tener en la ruta `/home/vagrant` un directorio cuyo nombre sea `Jasper-1.900.29` (Véase Figura 36)



```
vagrant@node-01:~$ ls
jasper-1.900.29  jasper-1.900.29.tar.gz
```

Figura 36: Jasper-1.900.29

Como obtenemos datos de entrada, es necesario eliminar el fichero de `namelist.input`. Este fichero se usa para ejecutar el modelo WRF. Este tiene varias secciones para controlar el tiempo de duración del modelo (días, horas, minutos, segundos). Cuando ejecutamos `wrf.exe` que es parte del ejecutable de este modelo, las fechas/horas de inicio/finalización se sobrescriben. Además tiene otras secciones como el control de los dominios y algunas otras más.

En las dos máquinas virtuales necesitamos crear un directorio que va a contener un enlace simbólico<sup>6</sup> a cada uno de los ficheros del directorio `run` de WRF. En nuestro caso, este directorio se va a encontrar dentro del directorio `Jasper-1.900.29`, ya que lo instalamos ahí. (Véase Figura 37). Así que es necesario copiar los datos de entrada y el fichero `machines` (Véase Figura 38) para poder llevar a cabo la ejecución paralela en los dos nodos y ejecutar el `wrf.exe` con `mpirun` (Véase Figura 39). Es importante remarcar el hecho de que se realice en los mismos directorios en las máquinas.

Por lo tanto, para copiar todos los ficheros en el directorio creado, podemos usar el siguiente comando, para no ir uno a uno: `$find /home/vagrant/jasper-1.900.29/WRF-4.2.1/run | xargs -t -I {} ln -s {} . 2>/dev/null`

---

<sup>6</sup> **Enlace simbólico:** Acceso a un directorio o fichero que se encuentra en otro lugar

```
vagrant@node-01:~/directorío$ find /home/vagrant/jasper-1.900.29/WRF-4.2.1/run | xargs -t -I {} ln -s {} . 2>/dev/null
vagrant@node-01:~/directorío$ ls
aerosol.formatted          CLM_ALB_ICE_DFS_DATA      ETAMPNEW_DATA.expanded_rain ozone.formatted            SOILPARM.TBL
aerosol_lat.formatted      CLM_ALB_ICE_DRC_DATA      ETAMPNEW_DATA.expanded_rain_DBL ozone_lat.formatted        SOILPARM.TBL_Kishne_2017
aerosol_lon.formatted      CLM_ASM_ICE_DFS_DATA      GENPARM.TBL                  ozone_plev.formatted      tc.exe
aerosol_plev.formatted     CLM_ASM_ICE_DRC_DATA      grib2map.tbl                p3_lookup_table_1.dat-v4.1 termvels.asc
BROADBAND_CLOUD_GODDARD.bin CLM_DRDSDT0_DATA          gribmap.txt                  p3_lookup_table_2.dat-v4.1 tr49t67
bulkdens.asc_s_0_03_0_9    CLM_EXT_ICE_DFS_DATA      HLC.TBL                     README.namelist           tr49t85
bulkradii.asc_s_0_03_0_9   CLM_EXT_ICE_DRC_DATA      ishmael-gamma-tab.bin       README.rasm_diag          tr67t85
CAM_ABS_DATA               CLM_KAPPA_DATA            ishmael-qi-qc.bin           README.tslist             URBPARM.TBL
CAM_AEROPT_DATA            CLM_TAU_DATA              ishmael-qi-qr.bin           real.exe                  URBPARM_UZE.TBL
CAMtr_volume_mixing_ratio.A18 co2_trans                  kernels.asc_s_0_03_0_9      RRTM_DATA                 VEGPARM.TBL
CAMtr_volume_mixing_ratio.A2 coeff_p.asc                kernels_z.asc               RRTM_DATA_DBL             wind-turbine-1.tbl
CAMtr_volume_mixing_ratio.RCP4.5 coeff_q.asc                LANDUSE.TBL                RRTMG_LW_DATA             wrf.exe
CAMtr_volume_mixing_ratio.RCP6 constants.asc               MPTABLE.TBL               RRTMG_LW_DATA_DBL
CAMtr_volume_mixing_ratio.RCP8.5 create_p3_lookupTable_1.f90 MPTABLE.TBL               RRTMG_SW_DATA
capacity.asc               ETAMPNEW_DATA             namelist.input              RRTMG_SW_DATA_DBL
CCN_ACTIVATE.BIN          ETAMPNEW_DATA_DBL         ndown.exe                   run
```

Figura 37: Enlaces simbólicos WRF

De esta forma ya tenemos creados los enlaces simbólicos, copiamos todos nuestros ficheros necesarios de entrada (wrfbdy\_d01, wrfininput\_d01, wrfininput\_d02, namelist.input, machines) que están en el directorio compartido:

```
$cp /vagrant/wrf* /vagrant/namelist.input /vagrant/machines
/home/vagrant/directorío
```

```
vagrant@node-01:~/directorío$ ls
aerosol.formatted          capacity.asc               constants.asc              kernels_z.asc             README.tslist             tr49t85
aerosol_lat.formatted      CCN_ACTIVATE.BIN          create_p3_lookupTable_1.f90 LANDUSE.TBL              real.exe                  tr67t85
aerosol_lon.formatted      CLM_ALB_ICE_DFS_DATA      ETAMPNEW_DATA             machines                 RRTM_DATA                URBPARM.TBL
aerosol_plev.formatted     CLM_ALB_ICE_DRC_DATA      ETAMPNEW_DATA.expanded_rain masses.asc               RRTM_DATA_DBL            URBPARM_UZE.TBL
BROADBAND_CLOUD_GODDARD.bin CLM_ASM_ICE_DFS_DATA      ETAMPNEW_DATA.expanded_rain_DBL MPTABLE.TBL             RRTMG_LW_DATA            VEGPARM.TBL
bulkdens.asc_s_0_03_0_9    CLM_DRDSDT0_DATA          GENPARM.TBL              namelist.input           RRTMG_LW_DATA_DBL       wind-turbine-1.tbl
bulkradii.asc_s_0_03_0_9   CLM_EXT_ICE_DFS_DATA      grib2map.tbl            ndown.exe                RRTMG_SW_DATA            wrfbdy_d01
CAM_ABS_DATA               CLM_EXT_ICE_DRC_DATA      gribmap.txt             ozone.formatted           RRTMG_SW_DATA_DBL       wrf.exe
CAM_AEROPT_DATA            CLM_KAPPA_DATA            HLC.TBL                 ozone_lat.formatted       run                       wrfininput_d01
CAMtr_volume_mixing_ratio.A18 CLM_TAU_DATA              ishmael-gamma-tab.bin   ozone_plev.formatted      SOILPARM.TBL            wrfininput_d02
CAMtr_volume_mixing_ratio.A2 CLM_EXT_ICE_DFS_DATA      ishmael-qi-qc.bin       p3_lookup_table_1.dat-v4.1 SOILPARM.TBL_Kishne_2017
CAMtr_volume_mixing_ratio.RCP4.5 co2_trans                  ishmael-qi-qr.bin       p3_lookup_table_2.dat-v4.1 termvels.asc
CAMtr_volume_mixing_ratio.RCP6 coeff_p.asc                kernels.asc_s_0_03_0_9  README.namelist          tr49t67
CAMtr_volume_mixing_ratio.RCP8.5 coeff_q.asc                kernels_z.asc            README.rasm_diag          tr49t85
```

Figura 38: Copiar datos de entrada WRF

Una vez instalado y configurado todo como es debido, ya estamos listos para la ejecución.

### 5.2.3 Ejecución WRF

Una vez que tenemos todo instalado y configurado, vamos a lanzar MPI para los dos casos, ejecutando el exe de WRF. Para ello vamos a usar el comando: `$mpirun -np 2 -hostfile machines --mca btl_tcp_if_include 192.168.56.0/24 ./wrf.exe` (Véase Figura 39).

```
vagrant@node-01:~/directorío$ mpirun -np 2 -hostfile machines --mca btl_tcp_if_include 192.168.56.0/24 ./wrf.exe
starting wrf task      0 of      2
starting wrf task      1 of      2
```

Figura 39: Ejecución WRF sin Singularity

Es importante ver que se crean dos procesos y cada uno se ejecute en un nodo distinto.

Si todo se ha creado correctamente, en el directorio que creamos antes deberíamos obtener algunas salidas más aparte de las anteriores como estas: (Véase Figura 40).

```

RRTMG_LW_DATA      wrf.exe
RRTMG_LW_DATA_DBL  wrfinput_d01
RRTMG_SW_DATA      wrfinput_d02
RRTMG_SW_DATA_DBL  wrfout_d01_2022-02-17_00:00:00
rsl.error.0000      wrfout_d02_2022-02-17_00:00:00
rsl.out.0000        wrfout_d02_2022-02-17_00:30:00
run                 wrfout_d02_2022-02-17_01:00:00
SOILPARM.TBL        wrfout_d02_2022-02-17_01:30:00
SOILPARM.TBL_Kishne_2017 wrfout_d02_2022-02-17_02:00:00
tc.exe              wrfout_d02_2022-02-17_02:30:00
termvels.asc        wrfout_d02_2022-02-17_03:00:00
tr49t67

```

Figura 40: Output WRF

Es decir, se deben de haber generado dos ficheros rsl.error.0000 y rsl.out.0000. Esto en el primer y segundo nodo. Podemos diferenciarlos mediante los últimos números. En el caso del primer nodo sería 0000 y en el segundo 0001. El fichero rsl.out tiene unas líneas de cabecera y según el paso del tiempo va añadiendo líneas según se ejecuta WRF. Estas líneas tienen la forma de Timing for main: time 2022-02-17\_00:00:33 on domain 2: 7.87377 elapsed seconds. La penúltima de ellos va a corresponder a las 03:00:00 del 17 de febrero. La última nos va a indicar que se ha completado correctamente WRF.

Además se generan otros archivos del tipo wrfout\*, uno para cada 30 minutos de predicción. Estos son los que se usan para crear los mapas de la temperatura, humedad, precipitación, viento, etc. Aunque estos no nos interesan, pues sólo queremos calcular la diferencia de tiempos.

El primer tiempo que hemos obtenido ha sido sobre unos 14 minutos para la ejecución de WRF sin Singularity (Véase Figura 41).

```

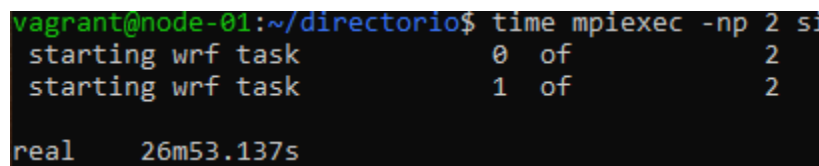
vagrant@node-01:~/directorio$ time mpiexec -np 2 -f
starting wrf task      1 of      2
starting wrf task      0 of      2

real    13m54.518s

```

Figura 41: Tiempo WRF I

A modo de anexo, si ejecutásemos este modelo en un solo nodo en lugar de dos, debería salir el doble de tiempo aproximadamente. Pues si nos damos cuenta no se repartiría la carga y se ejecutaría en un solo nodo (Véase Figura 42).



```
vagrant@node-01:~/directorio$ time mpiexec -np 2 si
starting wrf task      0  of      2
starting wrf task      1  of      2
real    26m53.137s
```

Figura 42: Ejecución WRF de dos procesos en un solo nodo

## 5.3 WRF + Singularity

Una vez realizada la ejecución de este modelo, se va a realizar usando Singularity. De esta manera vamos a ejecutar el modelo nuevamente pero dentro de los contenedores. Una vez realizado compararemos los resultados y veremos cuál es mejor.

### 5.3.1 Imagen WRF

#### 5.3.1.1 Creación de la imagen

Respecto a la parte de WRF, es pertinente crear nuevamente otra imagen. Por ello, es conveniente definirnó otro archivo de definición distinto (Véase Código 12).

---

**Bootstrap:** Docker

**From:** ubuntu:latest

**%files**

WRF-4.2.1.tgz /opt

**%post**

echo "Instalación requisitos previos"

apt-get update && apt-get install -y wget && apt install -y csh gfortran m4 openmpi-bin libhdf5-openmpi-dev libpng-dev libnetcdf-dev netcdf-bin ncl-ncarg build-essential  
wget https://www.ece.uvic.ca/~frodo/jasper/software/jasper-1.900.29.tar.gz

tar xvf jasper-1.900.29.tar.gz

cd jasper-1.900.29/

./configure --prefix=/opt/jasper-1.900.29

make

make install

echo "Descompresión WRF.tgz"

cd /opt

---

```
tar xzvf WRF-4.2.1.tgz
```

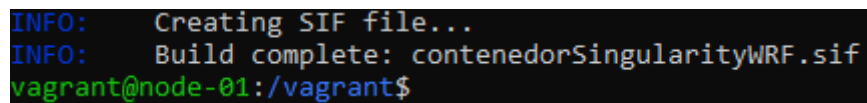
---

#### Código 12: Creación de imagen WRF Singularity

A la hora de generar el contenedor, existe un concepto clave y es el empaquetado en un fichero .tgz del directorio WRF-4.2.1 para evitar tiempos de compilación. Por lo tanto, instalamos toda la parte de WRF, borramos el fichero namelist.input del directorio run y lo comprimimos. Esto se puede hacer ya que tenemos el mismo sistema operativo en la máquina virtual. Copiamos el archivo comprimido en la dirección /opt del contenedor. Luego a la hora de compilarlo, tan solo debemos descomprimirlo. También es necesario instalar los prerequisites y Jasper, tal como aparece en la página dónde instalamos WRF (Véase Código 12). (Pratiman-91.github, s.f.)

##### 5.3.1.2 Construcción imagen

Una vez que tenemos el archivo de definición, es hora de construir la imagen con la misma instrucción de antes (*\$sudo singularity build contenedorSingularityWRF.sif contenedorSingularityWRF.def*) (Véase Figura 43)



```
INFO: Creating SIF file...  
INFO: Build complete: contenedorSingularityWRF.sif  
vagrant@node-01:/vagrant$
```

Figura 43: Imagen Singularity WRF

##### 5.3.2 Distribución de ficheros

La distribución de ficheros es la misma que antes, sólo que encontramos algunas variaciones. Queda reflejado en el directorio WRF-1. El contenido de los tres primeros es exactamente igual que a los explicados en la sección de la distribución de ficheros de Singularity (Véase 4.5.2)

- **Install.sh**
- **Config-ssh**
- **Crear\_MVs.bat**
- **ContenedorSingularity.def**

Este archivo es la definición del contenedor (Véase Código 12).

- **WRF-4.2.1.tgz**

Fichero comprimido del WRF para ahorrar tiempo en la creación de la imagen.

En lugar de compilar nuevamente, lo descomprimos solamente.

- **Datos de entrada:** namelis.input, wrfbdy\_d01, wrfinput\_d01, wrfinput\_d02.

Además como en todos los casos debemos obtener un fichero machines, que aunque en este no se cree automáticamente, es necesario para que las máquinas puedan cooperar entre sí.

### 5.3.3 Proceso de instalación y configuración

Antes de comenzar la instalación y configuración y posteriormente la ejecución del modelo WRF con el uso de Singularity, es importante saber que para esta parte vamos a necesitar la creación de una imagen para Singularity (Véase Figura 43).

Para llevar a cabo la instalación y configuración de esta parte es necesario tener en cuenta que vamos a necesitar dos máquinas. Para generarlas, se generan como sabemos con el VagrantFile correspondiente (Véase Código 1).

En primer lugar, vamos a instalar el software WRF versión 4.2.1 en nuestras máquinas virtuales. (Pratiman, 2020) Como la parte del WPS no la vamos a utilizar no es necesario instalarla.

Para esta parte es necesario tener instalado el software Singularity. Pero de esto ya se ocupa el fichero install.sh que llama el VagrantFile, por lo que no tenemos que preocuparnos de instalarlo (Véase 4.5.2).

Los pasos son similares a lo de la sección anterior. En primer lugar, necesitamos crear los enlaces simbólicos en el directorio de trabajo que nos hemos creado /home/vagrant/directorio (Véase Figura 44). Es importante recordar que se debe de realizar en las dos máquinas y no tan sólo en una.

Para crear los enlaces simbólicos ejecutamos la orden: `$singularity exec /vagrant/contenedorSingularityWRF.sif find /opt/WRF-4.2.1/run | xargs -t -I {} ln -s {} . 2>/dev/null`

```

vagrant@node-01:~/directorio$ singularity exec /vagrant/contenedorSingularityWRF.sif find /opt/WRF-4.2.1/run | xargs -t -I {} ln -s {} . >>/dev/null
vagrant@node-01:~/directorio$ ls
aerosol.formatted          CCN_ACTIVATE.BIN          ETAMPNEW_DATA              MPTABLE.TBL               RRTMG_SW_DATA
aerosol_lat.formatted      CLM_ALB_ICE_DFS_DATA      ETAMPNEW_DATA_DBL         ndown.exe                 RRTMG_SW_DATA_DBL
aerosol_lon.formatted      CLM_ALB_ICE_DRC_DATA      ETAMPNEW_DATA.expanded_rain ozone.formatted            run
aerosol_plev.formatted     CLM_ASM_ICE_DFS_DATA      ETAMPNEW_DATA.expanded_rain_DBL ozone_lat.formatted        SOILPARM.TBL
BROADBAND_CLOUD_GOODARD.bin CLM_ASM_ICE_DRC_DATA      GENPARM.TBL               ozone_plev.formatted       SOILPARM.TBL_Kishne_2017
bulkradnii.asc_s_0_03_0_9 CLM_DRDSDT0_DATA         gribmap.tbl               p3_lookup_table_1.dat-v4.1 tc.exe
bulkradnii.asc_s_0_03_0_9 CLM_EXT_ICE_DFS_DATA      hlc.tbl                   p3_lookup_table_2.dat-v4.1 termvels.asc
CAM_ABS_DATA               CLM_EXT_ICE_DRC_DATA      ishrael-gamma-tab.bin     README.namelist            tr49t67
CAM_AEROPRT_DATA          CLM_KAPPA_DATA            ishrael-qi-qr.bin         README.rasm_diag           tr49t85
CAMtr_volume_mixing_ratio.A18 CLM_TAU_DATA              ishrael-qi-qr.bin         README.tslist              tr67t85
CAMtr_volume_mixing_ratio.A2 co2_trans                  kernels.asc_s_0_03_0_9    RRTM_DATA                  URBPARM.TBL
CAMtr_volume_mixing_ratio.RCP4.5 coeff_p.asc                kernels_2.asc              RRTM_DATA_DBL              URBPARM_UZE.TBL
CAMtr_volume_mixing_ratio.RCP6 coeff_q.asc                 LANDUSE.TBL                RRTMG_LW_DATA              VEGPARM.TBL
CAMtr_volume_mixing_ratio.RCP8.5 constants.asc              LANDUSE.TBL                RRTMG_LW_DATA              wind-turbine-1.tbl
capacity.asc                create_p3_lookupTable_1.f90 masses.asc                  RRTMG_LW_DATA_DBL         wrf.exe

```

Figura 44: Enlaces simbólicos rotos WRF + Singularity

A diferencia de los enlaces de WRF (Véase Figura 37), ahora aparecen en color rojo debido a que son enlaces rotos porque se encuentran dentro del contenedor de Singularity. Todos los comandos que estamos ejecutando se realizan dentro del contenedor que hemos creado (Véase Figura 43 ).

Una vez que tenemos los enlaces simbólicos, copiaremos los datos de entrada y el fichero machines (Véase Figura 45).

```

vagrant@node-01:~/directorio$ cp /vagrant/wrf* /vagrant/namelist.input /vagrant/machines /home/vagrant/directorio
vagrant@node-01:~/directorio$ ls
aerosol.formatted          capacity.asc                constants.asc                kernels_2.asc              README.tslist              tr49t85
aerosol_lat.formatted      CCN_ACTIVATE.BIN          create_p3_lookupTable_1.f90 LANDUSE.TBL                real.exe                   tr67t85
aerosol_lon.formatted      CLM_ALB_ICE_DFS_DATA      ETAMPNEW_DATA              machines                    RRTM_DATA                 URBPARM.TBL
aerosol_plev.formatted     CLM_ALB_ICE_DRC_DATA      ETAMPNEW_DATA_DBL         masses.asc                 RRTM_DATA_DBL             URBPARM_UZE.TBL
BROADBAND_CLOUD_GOODARD.bin CLM_ASM_ICE_DFS_DATA      ETAMPNEW_DATA.expanded_rain namelist.input             RRTMG_LW_DATA              VEGPARM.TBL
bulkradnii.asc_s_0_03_0_9 CLM_ASM_ICE_DRC_DATA      ETAMPNEW_DATA.expanded_rain_DBL ndown.exe                 RRTMG_LW_DATA_DBL         wind-turbine-1.tbl
bulkradnii.asc_s_0_03_0_9 CLM_DRDSDT0_DATA         GENPARM.TBL               ozone.formatted            RRTMG_SW_DATA             wrf.exe
CAM_ABS_DATA               CLM_EXT_ICE_DFS_DATA      gribmap.tbl               ozone_lat.formatted        RRTMG_SW_DATA_DBL         wrfinput_d01
CAM_AEROPRT_DATA          CLM_EXT_ICE_DRC_DATA      hlc.tbl                   ozone_plev.formatted       run                         wrfinput_d01
CAMtr_volume_mixing_ratio.A18 CLM_KAPPA_DATA            ishrael-gamma-tab.bin     p3_lookup_table_1.dat-v4.1 SOILPARM.TBL              wrfinput_d02
CAMtr_volume_mixing_ratio.A2 CLM_TAU_DATA              ishrael-qi-qr.bin         p3_lookup_table_2.dat-v4.1 SOILPARM.TBL_Kishne_2017 tc.exe
CAMtr_volume_mixing_ratio.RCP4.5 co2_trans                  kernels.asc_s_0_03_0_9    README.namelist            termvels.asc
CAMtr_volume_mixing_ratio.RCP6 coeff_p.asc                kernels_2.asc              README.rasm_diag           tr49t67
CAMtr_volume_mixing_ratio.RCP8.5 coeff_q.asc                 LANDUSE.TBL                RRTMG_LW_DATA              wind-turbine-1.tbl

```

Figura 45: Copiar datos de entrada WRF + Singularity

Ya estaría lista para ejecutar la combinación de este modelo a través de contenedores.

### 5.3.4 Ejecución WRF + Singularity

Una vez que tenemos todo instalado y configurado, vamos a lanzar MPI para los dos casos, ejecutando el exe de WRF. Para ello vamos a usar el comando: `$mpirun -np 2 -hostfile machines --mca btl_tcp_if_include 192.168.56.0/24 singularity exec /vagrant/contenedorSingularityWRF.sif /opt/WRF-4.2.1/run/wrf.exe` (Véase Figura 46).

```

vagrant@node-01:~/directorio$ mpirun -np 2 -hostfile machines --mca btl_tcp_if_include 192.168.56.0/24 singularity exec /vagrant/contenedorSingularityWRF.sif /opt/WRF-4.2.1/run/wrf.exe
starting wrf task      0 of      2
starting wrf task      1 of      2

```

Figura 46: Ejecución WRF con Singularity

Es importante ver que se crean dos procesos y cada uno se ejecute en un nodo distinto. Si todo se ha creado correctamente, en el directorio que creamos antes

deberíamos obtener algunas salidas más aparte de las anteriores (Véase Figura 47). Además, debemos de obtener la `namelist.output`.

```

RRTMG_LW_DATA      wrf.exe
RRTMG_LW_DATA_DBL  wrfinput_d01
RRTMG_SW_DATA      wrfinput_d02
RRTMG_SW_DATA_DBL  wrfout_d01_2022-02-17_00:00:00
rsl.error.0000      wrfout_d02_2022-02-17_00:00:00
rsl.out.0000        wrfout_d02_2022-02-17_00:30:00
run                 wrfout_d02_2022-02-17_01:00:00
SOILPARM.TBL        wrfout_d02_2022-02-17_01:30:00
SOILPARM.TBL_Kishne_2017 wrfout_d02_2022-02-17_02:00:00
tc.exe              wrfout_d02_2022-02-17_02:30:00
termvels.asc        wrfout_d02_2022-02-17_03:00:00
tr49t67

```

Figura 47: Output WRF + Singularity

Como comparación con el tiempo obtenido en el apartado anterior, podemos usar la instrucción `time` antes del nuevo comando (Véase Figura 48). Además, podemos ver como la sobrecarga que introduce Singularity es mínima y no tiene impacto alguno.

```

vagrant@node-01:~/directorio$ time mpiexec -np 2 -hostf
/WRF-4.2.1/run/wrf.exe
  starting wrf task      0  of      2
  starting wrf task      1  of      2
real    13m59.086s

```

Figura 48: Tiempo WRF + Singularity

### 5.3.5 Ejecución en clúster Galgo

En primer momento, se podría haber llevado la ejecución a un clúster real. Sin embargo, nos hemos encontrado con un problema y es que al tener un kernel muy antiguo no se podía instalar Singularity en el clúster Galgo.

Para sustituir la ejecución en las máquinas virtuales por el entorno real de Galgo habría que realizar una serie de pasos consecutivos:

- Creamos un directorio con el caso que vamos a ejecutar, tal como se ha realizado con las máquinas virtuales. En este clúster disponemos de un sistema de ficheros **NFS**<sup>7</sup> compartido por todas las máquinas. Esto va a hacer que sea accesible por todas las máquinas del clúster en tiempo de ejecución.

---

<sup>7</sup> **Network File System:** Tipo de mecanismo de sistema de archivos que permite la recuperación y el almacenamiento de datos y directorio en una red compartida



- Copiamos el fichero .sif generado con Singularity (Véase Figura 43) al directorio de trabajo situado en Galgo.
- El clúster utiliza un sistema de colas **PBS PRO**. Este sistema es un administrador de cargas de trabajo cuya función es mejorar la productividad, optimizar la utilización y simplificar la administración de clústeres, nubes y supercomputadoras (Altair, s.f.). Por ello, necesitamos crear un script indicando los recursos que necesitamos y qué es lo que vamos a ejecutar (Véase Código 13).

---

```
#PBS -l select=2:ncpus=16:mpiprocs=16:cluster=galgo2:mem=62GB
#PBS -l walltime=600:00:00
#PBS -V

export DIR=$PBS_O_WORKDIR
cd $DIR

# Cargar MPI (funciona con v3 en adelante)
./home/toni_/software/cargar_openmpi-3.0.4.sh

mpirun singularity exec $DIR/wrf.sif $DIR/wrf.exe
```

---

#### Código 13: Código WRF en Galgo

En este código a través de la instrucción de **select** vamos a ejecutar en dos nodos o máquinas físicas que van a estar formadas por 16 cpus cada una y con 16 números de procesos mpi por nodo. Los nodos que estamos cogiendo son del galgo2, ya que el galgo 1 sólo contiene máquinas con 8 cpus, y la memoria por cada nodo es de 62 GB. Con la segunda línea establecemos un tiempo máximo de ejecución, por lo que si se superase ese tiempo establecido, el proceso se abortaría. En la tercera línea con la opción de -V todas las variables de entorno definidas se van a heredar. Luego es pertinente exportar en la variable DIR el directorio desde donde hemos lanzado el trabajo y cambiar a ese directorio.

En penúltimo lugar, cargamos la librería de OpenMpi (Azab, 2018). En nuestro contenedor usamos exactamente la versión 4. Después ejecutamos con `mpiexec` el ejecutable de WRF dentro del contenedor, al igual que antes (Véase Figura 46).

Por último lanzamos a la cola mpi, `$qsub -q mpi nombreScriptCódigoAnterior`.

Al terminar la ejecución obtendremos resultados idénticos a los obtenidos sin usar el clúster real (Véase Figura 47). Aunque tardaría mucho menos tiempo al utilizar 32 procesos en lugar de 2.

# Capítulo 6

## Conclusiones y Trabajo Futuro

---

En este último capítulo se van a describir las conclusiones que se han obtenido a lo largo del proceso de elaboración de este proyecto, además de mostrar el trabajo futuro que se podría llevar a cabo.

### 6.1 Conclusiones

El objetivo principal de este TFG, ya mencionado con anterioridad (Véase 1.2) ha sido cumplido de manera satisfactoria. Como punto de partida, se ha comenzado con una aplicación sencilla como es el cálculo de  $\pi$ . Hemos creado una imagen que contiene el código del cálculo de  $\pi$  y todas sus dependencias y en primer lugar la hemos compilado dentro de un contenedor de Docker. Posteriormente, hemos conseguido realizar la aplicación en paralelo. Una vez hecho, se ha compilado en algunos orquestadores como Kubernetes, Docker Swarm e incluso en Singularity. Seguidamente, se ha llevado a una aplicación mucho más compleja como es el caso del modelo WRF. En ella hemos conseguido comparar el tiempo de ejecución en dos entornos, uno sin usar Singularity y otro usándolo.

En todo momento, durante el desarrollo de este trabajo se ha buscado facilitar la sencillez al lector, pues el tema puede parecer un poco abstracto y complejo. Además de que se pueda reproducir paso a paso el desarrollo de cualquier parte del proyecto. El uso de todas las tecnologías que hemos tratado es cada vez mayor y hemos visto las

numerosas ventajas que presentan. Hay que destacar la importancia que pueden llegar a tener los contenedores o los clústeres, pues sin ellos sería imposible alcanzar algunos objetivos de manera individual. Tal y como hemos podido comprobar, nos encontramos con algunas conclusiones bastante claras a simple vista.

En primer lugar, podemos deducir que es mucho más cómodo obtener las imágenes desde **plataformas** como Docker Hub y de esta manera no ocupar un espacio determinado en el host principal debido al contenedor. Aunque con Singularity podemos recogerlo de Docker Hub o incluso de Singularity Hub.

En segundo lugar, tal y como se observa, la **instalación y configuración** de toda la tecnología usada no parece muy compleja a simple vista. Aunque es cierto que hay notables diferencias en cuanto a la instalación y configuración nos referimos. Por ejemplo, instalar todas las dependencias de Kubernetes puede suponer una mayor dificultad que a la hora de instalar Singularity o Docker, que solo basta con un par de comandos ya que tienen pocas dependencias con otro software. Esto hace que su mantenimiento sea más fácil. Eso sí, Docker necesita permisos **root** en comparación con Singularity, por lo que habría que tener cuidado respecto a la seguridad. También se ha de destacar que a la hora de **crear y configurar un clúster** va a tener una mayor complejidad que lanzar la aplicación en dos máquinas que están relacionadas con SSH solamente. No obstante, en cuanto a diferencias acerca de Kubernetes y Docker Swarm son escasas pues las dos siguen considerablemente el mismo patrón. Lo mismo ocurre con la creación del fichero machines, ya que a la hora de obtener las IPs es mucho más complejo en los dos últimos orquestadores descritos que en Singularity, donde escogemos las IPs que hemos generado a la hora de crear las máquinas en nuestro VagrantFile.

En tercer lugar, se ha de destacar el gran impacto que está teniendo Kubernetes en la orquestación de contenedores, pero no creo que Docker Swarm se quede mucho más atrás. Aunque es cierto que Kubernetes está siendo respaldada por comunidades muy conocidas como Google, Red Hat, AWS, etc y las mejoras son continuas. Pero gracias a estas tecnologías que se han descrito estamos viviendo una revolución en la forma de desarrollar y desplegar aplicaciones HPC.

En cuarto lugar, es conveniente destacar el hecho del **tiempo** empleado en el modelo WRF. Y es que en ambos obtenemos el mismo tiempo aproximadamente, eso

quiere decir que la **sobrecarga** que introduce Singularity es mínima. Respecto a la instalación, es cierto que es más tedioso la de ejecutar WRF mediante Singularity pues es un paso más intermedio que no supone tiempo de ejecución alguno. Eso sí, usando contenedores encontramos las ventajas descritas anteriormente (Véase 2.3.2), como la aceleración de la fase de despliegue de aplicaciones HPC, se evitan problemas de versiones y dependencias y por su puesto permite aislar las aplicaciones. En el caso en el que se desee probar este modelo con otros datos de entrada personales, como ya tenemos la creación del contenedor, solo habría que cambiar estos datos de entrada.

En relación con el **ámbito académico y profesional**, la finalización de este trabajo ha supuesto poner en práctica una parte de conocimientos adquiridos a lo largo de la carrera. Aunque es cierto que ninguna de las materias de la rama de Tecnologías de la información tiene relación directa con el trabajo desarrollado. No obstante, cabe destacar algunas asignaturas como Sistemas Operativos I debido a que todo el desarrollo del TFG se ha realizado en entornos **Linux**. Por lo que, gracias a la experiencia obtenida sobre este sistema operativo, la realización de este TFG ha sido un poco más sencilla. También considero oportuno nombrar a la asignatura de Gestión de Sistemas de Información. Esta asignatura ha conseguido una mejora en la organización, recopilación y procesado de la información. Así como un mejor análisis y tomas de decisiones.

Respecto a las **competencias** adquiridas son diversas. Se han logrado obtener la mayoría de ellas, tanto en las competencias de formación básica como algunas de la rama específica, Tecnologías de la información. Todas estas quedan recogidas en la página de la Escuela Superior de Ingeniería Informática. A continuación se enumeran algunas de las competencias logradas con respecto a la rama específica:

- **[TI1]** Capacidad para comprender el entorno de una organización y sus necesidades en el ámbito de las tecnologías de la información y las comunicaciones.
- **[TI2]** Capacidad para seleccionar, diseñar, desplegar, integrar, evaluar, construir, gestionar, explotar y mantener las tecnologías de hardware, software y redes, dentro de los parámetros de coste y calidad adecuados.

Por último, es importante mencionar el hecho de que a pesar de no comenzar con algo de conocimientos sobre el tema de este TFG, con esfuerzo y ganas de trabajar se puede conseguir completarlo de una manera satisfactoria.

## 6.2 Trabajo futuro

Como continuación de este trabajo existen diversas líneas que quedan abiertas y es posible continuar trabajando en ellas.

- Realizar la parte de WRF para los orquestadores de Kubernetes y Docker Swarm.
- Ejecución de WRF + Singularity en Galgo una vez que se actualice el sistema operativo de Galgo.

# Capítulo 7

## Bibliografía

---

- Akgül, U. (2021). *Creating a local kubernetes cluster with Vagrant*. Recuperado el 07 de Julio de 2022, de Ugurakgul.medium: <https://ugurakgul.medium.com/creating-a-local-kubernetes-cluster-with-vagrant-ba591ab70ee2>
- Altair. (s.f.). *Industry-leading Workload Manager and Job Scheduler for HPC and High-throughput Computing*. Recuperado el 07 de Julio de 2022, de <http://altair.com.es/pbs-professional/>
- AMD. (s.f.). *Explicación del Concepto de Computación de Alto Rendimiento*. Recuperado el 07 de Julio de 2022, de <https://www.amd.com/es/technologies/hpc-explained>
- App.vagrantup. (s.f.). *Discover Vagrant Boxes*. Recuperado el 07 de Julio de 2022, de <https://app.vagrantup.com/boxes/search>
- Aquasec. (s.f.). *What are Container Platforms?* Recuperado el 07 de Julio de 2022, de <https://www.aquasec.com/cloud-native-academy/container-platforms/container-platforms-6-best-practices-and-15-top-solutions/#:~:text=What%20are%20Container%20Platforms%3F,enterprise%20support%20for%20container%20architectures>.
- Avi. (6 de Mayo de 2022). *Docker Architecture y sus componentes para principiantes*. Recuperado el 07 de Julio de 2022, de Geekflare: <https://geekflare.com/es/docker-architecture/>

- Azab, A. (Noviembre de 13 de 2018). *The deployment of containers and full virtualised tools into HPC infrastructures*. Recuperado el 07 de Julio de 2022, de Prace-ri.eu: <https://prace-ri.eu/wp-content/uploads/PRACE-at-SC18-Containers-for-HPC-in-PRACE.pdf>
- Bejerano, P. G. (2016). *Para qué sirven los contenedores en software*. Recuperado el 07 de Julio de 2022, de Blogthinkbig: <https://blogthinkbig.com/para-que-sirven-los-contenedores-en-software#:~:text=Los%20contenedores%20de%20software%20son,de%20los%20contenedores%20de%20software.>
- Bigelow, S. J. (10 de Julio de 2013). *Message passing interface (MPI)*. Recuperado el 07 de Julio de 2022, de Techtarget: [https://www.techtarget.com/searchenterprisedesktop/definition/message-passing-interface-MPI#:~:text=The%20message%20passing%20interface%20\(MPI,computer%20%2D%2D%20are%20called%20nodes.](https://www.techtarget.com/searchenterprisedesktop/definition/message-passing-interface-MPI#:~:text=The%20message%20passing%20interface%20(MPI,computer%20%2D%2D%20are%20called%20nodes.)
- Blog.tactio. (s.f.). *Las tres fases de un proyecto exitoso: Planificar, implementar y controlar*. Recuperado el 07 de Julio de 2022, de <https://blog.tactio.es/consultoria/control-gestion/planificar-implementar-controlar-tres-fases-proyecto/#:~:text=La%20fase%20de%20planificaci%C3%B3n%20consta,meta%20que%20se%20quiere%20alcanzar.>
- BSC. (10 de Julio de 2019). *Descubre la supercomputación (HPC)*. Recuperado el 07 de Julio de 2022, de <https://www.bsc.es/es/tech-transfer/descubre-la-supercomputacion>
- BSC. (s.f.). *Alya - High Performance Computational Mechanics*. Recuperado el 07 de Julio de 2022, de <https://www.bsc.es/research-development/research-areas/engineering-simulations/alya-high-performance-computational>
- Campusmvp. (14 de Abril de 2016). *Docker vs Vagrant: diferencias y similitudes y cuándo usar cada uno*. Recuperado el 07 de Julio de 2022, de <https://www.campusmvp.es/recursos/post/Docker-vs-Vagrant-diferencias-y-similitudes-y-cuando-usar-cada-uno.aspx>



- Campusmvp. (09 de Octubre de 2018). *Los beneficios de utilizar Docker y contenedores a la hora de programar*. Recuperado el 07 de Julio de 2022, de <https://www.campusmvp.es/recursos/post/los-beneficios-de-utilizar-docker-y-contenedores-a-la-hora-de-programar.aspx>
- Clockworknet. (1 de Febrero de 2019). *¿Los contenedores tienen un sistema operativo?* Recuperado el 07 de Julio de 2022, de Enmimaquinafunciona: <https://www.enmimaquinafunciona.com/pregunta/165008/los-contenedores-tienen-un-sistema-operativo>
- González, D. B. (02 de Agosto de 2021). *¿Qué es Docker y para qué sirve?* Recuperado el 07 de Julio de 2022, de Profile: [https://profile.es/blog/que\\_es\\_docker/](https://profile.es/blog/que_es_docker/)
- Grupo-ioa.atmosfera.unam.mx. (s.f.). *WRF*. Recuperado el 07 de Julio de 2022, de <http://grupo-ioa.atmosfera.unam.mx/pronosticos/index.php/meteorologia/inf-wrf>
- Hpc-tutorials.llnl.gov. (s.f.). *What is MPI?* Recuperado el 07 de Julio de 2022, de [https://hpc-tutorials.llnl.gov/mpi/what\\_is\\_mpi/](https://hpc-tutorials.llnl.gov/mpi/what_is_mpi/)
- IBM. (s.f.). *¿Qué es la computación de alto rendimiento (HPC)?* Recuperado el 07 de Julio de 2022, de IBM Web Site: <https://www.ibm.com/es-es/topics/hpc>
- IBM. (23 de Junio de 2021). *¿Qué son los contenedores?* Recuperado el 07 de Julio de 2022, de <https://www.ibm.com/es-es/cloud/learn/containers>
- Krikit. (s.f.). *Vagrant (software)*. Recuperado el 07 de Julio de 2022, de <https://krikit.com/vagrant-software/>
- Krypton Solid. (s.f.). *¿Qué es Docker Swarm?* Recuperado el 07 de Julio de 2022, de <https://kryptonsolid.com/que-es-docker-swarm-definicion-de-krypton-solid/>
- Krypton Solid. (Julio de 2013). *¿Qué es la interfaz de paso de mensajes (MPI)?* Recuperado el 07 de Julio de 2022, de <https://kryptonsolid.com/que-es-la-interfaz-de-paso-de-mensajes-mpi/>
- Kubernetes.io. (s.f.). *What is Kubernetes?* Recuperado el 07 de Julio de 2022, de <https://kubernetes.io/es/docs/concepts/overview/what-is-kubernetes/>
- Kurtzer, G. M. (s.f.). *Gregory M. Kurtzer*. Recuperado el 07 de Julio de 2022, de Github.io: <https://gmekurtzer.github.io/>
- Microsoft. (23 de Junio de 2022). *Contenedores como base para la colaboración de DevOps*. Recuperado el 07 de Julio de 2022, de <https://docs.microsoft.com/es->

- es/dotnet/architecture/containerized-lifecycle/docker-application-lifecycle/containers-foundation-for-devops-collaboration
- MMM.ucar.edu. (s.f.). *Weather research and forecasting model*. Recuperado el 07 de Julio de 2022, de <https://www.mmm.ucar.edu/weather-research-and-forecasting-model>
- NetApp. (30 de Octubre de 2020). *What are containers?* Recuperado el 07 de Julio de 2022, de <https://www.netapp.com/es/devops-solutions/what-are-containers/>
- NetApp. (04 de Marzo de 2022). *¿Qué es la computación de alto rendimiento?* Recuperado el 07 de Julio de 2022, de <https://www.netapp.com/es/data-storage/high-performance-computing/what-is-hpc/>
- Obsbusiness.school. (21 de Marzo de 2021). *Cómo afrontar la etapa de seguimiento y control de mi proyecto*. Recuperado el 07 de Julio de 2022, de <https://www.obsbusiness.school/blog/como-afrontar-la-etapa-de-seguimiento-y-control-de-mi-proyecto>
- Oracle. (s.f.). *¿Qué es la computación de alto rendimiento (HPC)?* Recuperado el 07 de Julio de 2022, de <https://www.oracle.com/es/cloud/hpc/what-is-hpc/>
- Pérez, L. (4 de Octubre de 2021). *HPC: Computación de alto rendimiento*. Recuperado el 07 de Julio de 2022, de Azken: <https://www.azken.com/blog/hpc-computacion-de-alto-rendimiento-2/#:~:text=Para%20procesar%20la%20informaci%C3%B3n%20en,y%20el%20procesamiento%20en%20paralelo>.
- Pratiman. (2020). *Installing WRF 4.2.1 on Ubuntu LTS 20.04*. Recuperado el 07 de Julio de 2022, de Github.io: <https://pratiman-91.github.io/2020/07/28/Installing-WRF-4.2.1-on-Ubuntu-LTS-20.04.html>
- Pratiman-91.github. (s.f.). *Installing WRF 4.2.1 on Ubuntu LTS 20.04*. Recuperado el 07 de Julio de 2022, de <https://pratiman-91.github.io/2020/07/28/Installing-WRF-4.2.1-on-Ubuntu-LTS-20.04.html>
- Profile. (19 de Julio de 2021). *¿Qué son los contenedores de software y por qué utilizarlos?* Recuperado el 07 de Julio de 2022, de <https://profile.es/blog/contenedores-de-software/>
- Programmerclick. (s.f.). *Beneficio mutuo de contenedores y HPC*. Recuperado el 07 de Julio de 2022, de <https://programmerclick.com/article/94472512544/>

- RedHAt. (2020). *What is Kubernetes?* Recuperado el 07 de Julio de 2022, de <https://www.redhat.com/es/topics/containers/what-is-kubernetes>
- Returngis. (12 de Febrero de 2019). *Publicar tu imagen en Docker Hub*. Recuperado el 07 de Julio de 2022, de <https://www.returngis.net/2019/02/publicar-tu-imagen-en-docker-hub/>
- Singularity-userdoc.readthedocs.io. (29 de Enero de 2021). *Container Recipes*. Recuperado el 07 de Julio de 2022, de [https://singularity-userdoc.readthedocs.io/en/latest/container\\_recipes.html](https://singularity-userdoc.readthedocs.io/en/latest/container_recipes.html)
- Sylabs.io. (04 de Noviembre de 2021). *Introduction to Singularity*. Recuperado el 07 de Julio de 2022, de <https://sylabs.io/guides/3.5/user-guide/introduction.html>
- Sylabs.io. (s.f.). *Singularity and Docker*. Recuperado el 07 de Julio de 2022, de [https://docs.sylabs.io/guides/2.6/user-guide/singularity\\_and\\_docker.html](https://docs.sylabs.io/guides/2.6/user-guide/singularity_and_docker.html)
- Teamleader. (18 de Agosto de 2021). *¿Qué es y para qué sirve un diagrama de Gantt?* Recuperado el 07 de Julio de 2022, de <https://www.teamleader.es/blog/diagrama-de-gantt>
- Vagrantup. (s.f.). *Development Environments Made Easy*. Recuperado el 07 de Julio de 2022, de <https://www.vagrantup.com/>



# Anexo I. Planificación temporal

---

## I.1 Planificación

Durante la realización de este TFG lo primero que se ha llevado a cabo ha sido una planificación (Véase 3.2). Con una buena planificación podemos obtener numerosos beneficios como puede ser la autodisciplina y la motivación. No obstante, podemos encontrarnos otros como el aprovechamiento del tiempo e incluso podemos alcanzar de una manera más rápida y eficaz el objetivo final.

Es por ello, que en horas estimadas sobrepasa de manera ligera las horas dedicadas a la correspondencia de este Trabajo de Fin de Grado, aproximadamente 300 horas. Para planificar el proyecto se ha creado un **Diagrama de Gantt**<sup>8</sup> (Véase Figura 49). En este diagrama podemos ver las fechas de inicio y finalización del proyecto, las tareas que la componen, las fechas programadas de inicio y finalización de cada tarea, una estimación del tiempo de cada tarea y ver si varias tareas se superponen o existe una relación entre ellas. (Teamleader, 2021)

En nuestro caso, hemos dividido este trabajo en **ocho tareas** para cumplir los objetivos que tenemos (Véase 1.2).

- **Adquisición de conocimientos previos al TFG:** En esta tarea el objetivo principal consiste en adquirir conocimientos acerca del proyecto. Se ha previsto una duración de unas dos semanas aproximadamente.

---

<sup>8</sup> **Diagrama de Gantt:** Herramienta gráfica para ver la composición de un proyecto y el tiempo en sus respectivas tareas

- **Docker y cálculo de Pi en dos máquinas:** Nos encontramos con la primera tarea práctica. En primer lugar, debemos de ejecutar el cálculo de Pi dentro de un contenedor de Docker y luego generamos con Vagrant dos máquinas y lo ejecutamos paralelamente. El tiempo que se ha estimado es un mes. Se comenzará a posteriori de la tarea anterior y se acabará antes de la segunda quincena de abril.
- **Estado del arte:** El desarrollo de esta tarea la podemos realizar a la misma vez que la tarea descrita con anterioridad ya que tan sólo debemos de ir escribiendo la información recopilada. Una vez que hemos recopilado información hacer de Docker y Vagrant que usamos en el capítulo anterior, podemos ir redactando en el documento presente el estado del arte. La duración sería un mes aproximadamente.
- **Kubernetes cpi:** Una vez que hemos realizado la parte de Docker y la ejecución del cálculo de Pi, debemos de subir la imagen a Docker Hub. Posteriormente, ejecutaremos el cálculo de Pi pero ahora en un clúster formado por cuatro contenedores y dos máquinas. Se ha estimado una duración de un mes aproximadamente. Aunque es cierto que dura un poco más.
- **Docker Swarm cpi:** Esta tarea consiste en rehacer lo hecho con Kubernetes. No obstante, se realiza en otro entorno. En comparación con Kubernetes, se estimó un mes, pero la planificación fue errónea. Esta tarea duró un poco más de una semana.
- **Singularity cpi:** Una vez que hemos terminado las tareas de Docker Swarm y Kubernetes, vamos a ejecutar el cálculo de Pi en un contenedor de Singularity. En comparación con Kubernetes, se estimó un mes, pero la planificación fue errónea. Esta tarea duró un poco más de una semana.
- **WRF y WRF + Singularity:** Esta tarea se va a comenzar una vez que hemos escrito y terminado en la memoria todas las tareas anteriores. En primer lugar, comenzaremos con la ejecución del modelo WRF sin Singularity y posteriormente, con Singularity para poder realizar una comparación. El tiempo estimado fueron dos semanas y fue idóneo.

- **Corrección y mejora de la memoria:** Como última tarea, se corregirán algunos fallos y una posterior mejora de la memoria. Se llevo a cabo mediante la tarea de WRF y WRF + Singularity y se estimó un mes. Aunque en el Diagrama de Gantt no aparezca Julio, esta tarea ha llevado la primera decena de este mes.

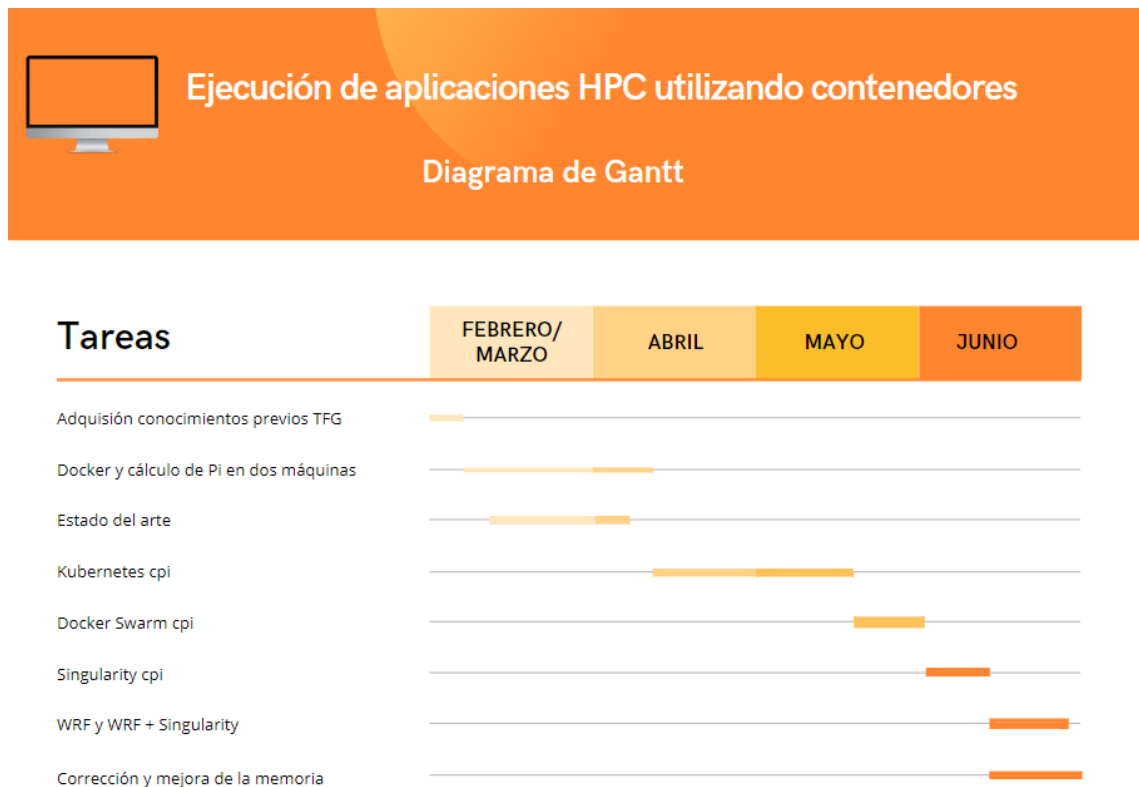


Figura 49: Diagrama de Gantt

El **tiempo total** ha sido realizado en cinco meses, aunque también se ha dedicado tiempo en Julio para mejoras del Proyecto, corrección y mejora de la memoria. Se ha unificado los meses de febrero y marzo debido a la menor inactividad en el proyecto en comparación con otros meses. A lo largo de los meses, conforme más nos aproximamos a la fecha de entrega, la cantidad de actividad crecía de manera lineal. Esto hace que las horas realizadas a las tareas de Docker Swarm y Singularity para el cálculo de pi no sean tan diferentes a la de Kubernetes. Pero como está basado en días puede parecer que hay una diferencia de horas abismal, pero no es así. Tan sólo se dedicaron más horas en los días.

Por último, podemos concluir este anexo destacando la buena planificación inicial que se ha llevado a cabo. Aunque es cierto que quizás se debería de haber ido echando un cómputo de horas más o menos equilibrado por días. Pero esto no ha sido posible por causas externas, como prácticas curriculares o asignaturas de este último cuatrimestre.



