Tecnológico de Costa Rica

Escuela de Computación

Programa de *Magíster Scientiae* en Computación, con énfasis en Ciencias de la Computación

Método para el administración de cargas de trabajo de computación paralela en un clúster de Kubernetes

Tesis para optar por el título de Magister en Computación con énfasis en Ciencias de la Computación

Cristian Arias Chaves

Tutor: Dr. Antonio Gonzalez Torres

San José, Junio del 2024

Escuela de Ingeniería en Computación
Unidad de Posgrado

TEC | Tecnológico de Costa Rica

ACTA DE APROBACION DE TESIS

Método para la administración de cargas de trabajo de computación paralela en un clúster de Kubernetes

Por: ARIAS CHAVES CRISTIAN

TRIBUNAL EXAMINADOR

ANTONIO GONZALEZ TORRES (FIRMA)
Digitally signed by ANTONIO GONZALEZ TORRES (FIRMA)
DN: cn=ANTONIO GONZALEZ TORRES (FIRMA), c=CR, o=PERSONA FISICA, ou=CIUDADANO
Date: 2024.08.21 21:46:29 -05'00'

Dr. Antonio González Torres
Profesor Asesor

ESTEBAN MENESES ROJAS (FIRMA)
Digitally signed by ESTEBAN MENESES ROJAS (FIRMA)
Date: 2024.08.16 15:09:59 -06'00'

Dr. Esteban Meneses Rojas
Profesor Lector

Documento firmado digitalmente a través de Central Directo del BCCR
Se puede validar en el sitio https://www.centraldirecto.fi.cr/
FIRMA DIGITAL

Máster Luis Loría Chavarría
Lector Externo

LILIANA SANCHO CHAVARRIA (FIRMA)
PERSONA FISICA, CPF-03-0257-0983.
Fecha declarada: 22/08/2024 08:37:37 AM

Dra.-Ing. Lilliana Sancho Chavarría
Presidente, Tribunal Evaluador Tesis
Programa Maestría en Computación

SINAES POST-GRADO ACREDITADO COSTA RICA

13 de agosto, 2024

Índice general

Capítulo 1

Introducción

La computación de alto rendimiento (HPC)[1] permite solucionar problemas complejos mediante el uso de supercomputadores [67]. La computación paralela es una de las áreas más prominentes de HPC que contribuye en la solución problemas que pueden requerir días, meses o años con otros métodos. Razón por la cual los científicos e investigadores se han beneficiado del uso de supercomputadores para ejecutar tareas complejas con computación paralela, aprovechando el aumentado de su rendimiento de manera acelerada desde el año de 1980 [20].

Los supercomputadores y la computación paralela permiten utilizar un gran número de procesadores[2] que en conjunto permiten la solución de problemas de forma más eficiente al aprovechar los recursos disponibles. Sin embargo, las características particulares del hardware de los supercomputadores y su alto costo restringe el acceso a estos a unos pocos. Por lo que se requiere de sistemas accesibles a una mayor base de usuarios, pero además, a su utilización de forma más extensa en la implementación de grandes sistemas de uso empresarial y comercial.

Los métodos de despliegue de aplicaciones utilizando computación paralela se ven influidos por diferentes factores, entre estos que se requiere de expertos en el área de HPC y ciencias de la computación, y muchas veces, es necesario cambiar el paradigma de desarrollo. Sin embargo, entre las nuevas tecnologías que se pueden considerar para la modernización de los métodos de despliegue de las aplicaciones de computación paralela desarrolladas para HPC se encuentran los contenedores y computación distribuida en la nube.

Los contenedores se basan en imágenes de sistemas operativos que cuentan con los elementos indispensables para la ejecución de los sistemas y herramienta particulares que se desarrollan. Las imágenes incluyen las librerías necesarias para la ejecución del sistema, permiten empaquetarlo y distribuirlo. Las imágenes pueden ser utilizadas para su ejecución como contenedores de forma independiente al sistema operativo en un ambiente

[1] La computación de alto rendimiento se conoce en inglés como *High Performance Computing* (HPC).

[2] El supercomputador Frontier disponía en el 2023 de 8776 CPU con 64 núcleos y totalizaba 561664 núcleos [3]. Este ocupó el puesto número 1 del *Top* 500 en el 20234, el cual es elaborado por el proyecto del Top500, que clasifica los 500 supercomputadores más poderosos en el mundo [10].

de contenedores.

Kubernetes es el orquestador de contenedores[3] más usado hoy en día. Este se encarga de administrar los contenedores como cajas negras y permite crear grandes clúster con n nodos de forma sencilla y más económica en comparación con la creación y configuración de un supercomputador con ese mismo número de n nodos. Lo que hace que la implementación de computación paralela en contenedores mediante un clúster de Kubernetes pueda ser considerada como una solución factible para ampliar el acceso al HPC.

Además, los clúster de Kubernetes se pueden utilizar como servicios en la nube o de forma local mediante su creación con alguna de las distribuciones existentes. Lo que incrementa las posibilidades de que Kubernetes sea un recurso de procesamiento masivo más accesible que los supercomputadores. En el contexto de esta investigación se considera que los operadores de Kubernetes pueden facilitar la definición de un método que explote las ventajas de Kubernetes para ejecutar sistemas de gran escala que trabajan en sintonía [31] con computación paralela, tomando como referencia el estándar MPI y el uso de operadores de Kubernetes.

1.1 Antecedentes

La ejecución de cargas de trabajo de computación paralela en contenedores ha sido explorada por varias investigaciones. En el 2015 un grupo de investigadores del Centro Nacional de HPC de China propusieron la creación de un clúster de HPC virtual con contenedores Docker. El clúster virtual estaba conformado por varios nodos y construyeron puentes personalizados para la comunicación entre estos, pero sólo contaba con un contenedor por máquina física. Las imágenes se construían con lo necesario para ejecutar MPI, SSH, y el software de comunicación. El descubrimiento de nuevos nodos se realizó con una librería de red en base a identidades Consul [1]. Esto facilitó la configuración del archivo de hosts, agregar nuevos nodos y el correcto funcionamiento de MPI. Esta fue una de las primeras investigaciones que demostró la factibilidad de ejecutar cargas de trabajo de MPI en contenedores. Los resultados y conclusiones obtenidas sentaron un precedente para investigaciones futuras [68].

En el año 2017 otro grupo de investigadores en IBM efectuaron la ejecución de programas para MPI con Docker Swarm mediante una extensión de la implementación MPICH de MPI que utilizó la flexibilidad de Hydra, el *framework* de gestión de procesos de MPICH. Para lograrlo crearon un *wrapper* cuya funcionalidad era inicializar una red superpuesta y los contenedores necesarios para ejecutar el programa en MPI dentro de los contenedores. Una vez que el programa terminaba se eliminaban los contenedores y la red superpuesta. Los resultados demostraron que es posible ejecutar este tipo de programas de HPC en contenedores preservando los beneficios que estos brindan. El artículo no detalla números

[3] Un orquestador de contenedores permite el manejo automatizado de cientos o miles de contenedores de una manera abstracta a la infraestructura.

ni métricas en cuanto a rendimiento. Sin embargo, los investigadores resaltaron el mayor uso de disco, y recomendaron el uso de redes de ultra alta velocidad como InfinitaBand [30].

En el año 2018, investigadores utilizaron COMPS Superscalar, un modelo de programación el cual permitía convertir aplicaciones secuenciales a paralelas y luego usaron su runtime para ejecutarlas, su enfoque consistió en un *framework* el que convertía el programa creando una imagen de contenedor y almacenándola en el repositorio DockerHub, luego creaba los contenedores necesarios para ejecutar la aplicación paralela a partir de la imagen creada. El *framework* poseía la capacidad de ejecutar los contenedores en tres plataformas de contenedores distintas, Docker, Singularity y Mesos. Se evaluó el tiempo para el despliegue y el rendimiento de dos benchmarks comparando la integración de COMPS en ambientes para contenedores ,bare metal, ambientes en la nube y en un clúster de HPC. Los resultados demostraron que el despliegue usando COMPS y contenedores se reducía sustancialmente comparado con los otros métodos, y el rendimiento de ejecución era similar con set de datos pequeños, sin embargo en escenarios con set de datos grandes si había una reducción de rendimiento especialmente por la red sobrepuesta de docker [57].

En el año 2019, durante la CANOPIE-HPC, se evaluó el rendimiento y la factibilidad de ejecutar aplicaciones de OpenMPI en tres ambientes en la nube. Contenedores usando Docker Swarm, contenedores usando Kubernetes, y bare metal, los resultados de los experimentos de latencia y uso de ancho de banda de red indicaron, que estos fueron los aspectos donde se perdió más rendimiento de los contenedores en comparación con bare metal, con diferencias de 42.14% y 59.34% en ancho de banda y una diferencia del 400% de latencia entre Kubernetes y Docker Swarm al utilizar el protocolo TCP/IP. Kubernetes fue el más afectado debido al fuerte uso de virtualización en su red sobrepuesta. Sin embargo al utilizar dispositivos InfinitiBand en aplicaciones de baja complejidad las diferencias se redujeron al 1%. En cuando a la evaluación con HPCG un benchmark mas centrado en usos reales, las diferencias entre Docker Swarm y bare metal no fueron significativas, Kubernetes si tuvo una diferencia del 13.15% esto usando TCP/IP, en cuando al uso de InfinitiBand Docker Swarm presentó una degradación de rendimiento del 4.19% y Kubernetes redujo la diferencia con bare metal del 13.15% al 10.31% [21, 43].

En el año 2019, otra investigación fue realizada por Lizhen Shi y Zhong Wang, se exploraron varias estrategias de escalar análisis de genomas usando tecnologías de computación distribuida. Las estrategias fueron sistemas multihilo de memoria compartida, clusters de HPC de varios nodos, frameworks de *big data* en la nube y contenedores. Para su evaluación se usó una métrica de 1 a 3 siendo tres la mejor puntuación en las áreas de facilidad de uso y desarrollo, robustez, escalabilidad y eficiencia. Los contenedores obtuvieron la máxima nota en facilidad de uso y desarrollo, robustez y escalabilidad siendo la única tecnología en alcanzarlo en esas áreas sin embargo en eficiencia obtuvieron la nota más baja junto con los *frameworks* de *big data* en la nube [62].

En el año 2020, Joshua Hursey expuso ciertas consideraciones necesarias a la hora de diseñar y ejecutar cargas de trabajo de HPC en contenedores. En concreto se centró en

siete áreas. La primera era como construir las imágenes, indicó que estas deben llevar todo lo necesario para ejecutar las aplicaciones MPI incluido. Además, las librerías se deben ejecutar en el espacio de usuario y no de *Kernel* debido a que esto es una consideración de seguridad de los contenedores. Se mencionaron dos posibles rutas a seguir para construir imágenes, especializadas al sistema *Host* donde van a ejecutar o mas genéricas que puedan ser utilizadas más libremente, siendo la primera opción más difícil de mantener pero otorgando mejor rendimiento.

Como segundo punto se habló de la ejecución de las aplicaciones de MPI y como unir los procesos, se pueden ejecutar procesos de forma directa o indirecta, otra consideración importante es la forma de ejecutar los procesos de MPI lo que impacta la forma en que estos se ven unos a los otros. Una forma es un contenedor por proceso, existiendo múltiples contenedores en un nodo, esta estrategia puede facilitar la implementación pero afectar el rendimiento. La otra forma es un contenedor por nodo, todos los procesos en el nodo se corren en el mismo contenedor, la ventaja es que la comunicación dentro del nodo es más directa al no existir barreras entre los procesos. Para unir los procesos después de iniciada la ejecución se habló de la adopción de la popular librería PMIx [24].

La tercera y cuarta área a en consideración fue la comunicación intra nodo y entre nodos, cuando los procesos se comunican dentro del mismo nodo sigue siendo recomendable usar una estrategia de memoria compartida, para la comunicación entre nodos existen varias soluciones como redes sobrepuestas o exponer las interfaces de red del *host* dentro de los contenedores. La quinta área que se explora es la decisión entre usar el sistema de MPI del *host* donde se puede obtener el mejor rendimiento pero pierde compatibilidad con otras versiones y librerías, otra manera es incluir el sistema de MPI dentro del contenedor de esta forma se puede perder rendimiento pero no existe restricción en cuanto a que librerías se pueden utilizar.

La sexta área estudiada fue la longevidad de las imágenes, especialmente cuando estas dependan del uso de librerías instaladas en el *host*, cuando esto ocurre es recomendable actualizar las imágenes en conjunto con el *host*. Como última área se exponen consideraciones a tomar en cuenta a la hora de ejecutar aplicaciones MPI en contenedores manejados por orquestadores tales como Kubernetes, en este tipo de ambientes las opciones expuestas por el investigador se reducen, por ejemplo ya no es posible usar el sistema de MPI del *host*, por lo que las imágenes deben tener todo los necesario para ejecutar las aplicaciones, lo común para ejecutar los procesos es la estrategia de un contenedor por nodo, en el caso de Kubernetes esto se traduce a un contenedor por *pod*. Las consideraciones para la comunicación intra nodo y entre nodos no varían mucho sin embargo Kubernetes nos fuerza a utilizar una red sobrepuesta lo que ocasiona una degradación del rendimiento, otro aspecto importante es que para el correcto funcionamiento de las aplicaciones de MPI en Kubernetes son necesarias modificaciones a su scheduler [45].

En el año 2021, Zhou et al. presentó una arquitectura híbrida donde coexistían un clúster de HPC con un clúster de Kubernetes en la nube, pudiendo utilizar una interfase en común para ejecutar aplicaciones en ambos. Permitía ejecutar las aplicaciones en contenedores o

fuera de estos, sin embargo, dependiendo de la aplicación la mejor opción sería ejecutar la aplicación en el clúster de HPC o en el clúster de Kubertenes. Para esto crearon una herramienta basada en un proyecto anterior llamado WLM-Operator [8], esta herramienta fue nombrada torque-operator y su función fue hacer de puente entre TORQUE [11], un sistema de colas para trabajos de computación paralela, y Kubernetes. La arquitectura consistió en un nodo central el cual tenía acceso tanto al clúster Kubernetes como al clúster de HPC, los trabajos de computación paralela se modelaron por medio de un *Custom Resource Definition* (CRD)[16], el nodo central podía decidir si ejecutar el trabajo en el clúster de HPC o en el de Kubernetes, para la ejecución de contenedores en el clúster de HPC, los investigadores utilizaron Singularity [6]. Para la evaluación de la propuesta se estudiaron dos casos de uso y se ejecuto un *benchmark* de MPI, las conclusiones fueron que efectivamente la arquitectura fue funcional y se logró unificar ambos mundos HPC y la nube, sin embargo el rendimiento siempre fue superior cuando se compararon los entornos virtualizados contra el clúster de HPC [69].

1.2 Planteamiento del problema

Las aplicaciones de computación paralela normalmente se despliegan y ejecutan en supercomputadores de HPC. Los supercomputadores son máquinas no distribuidas donde uno o varios usuarios pueden calendarizar sus trabajos por medio de herramientas como Slurm [7], que permiten obtener resultados de rendimiento satisfactorios. Sin embargo, el acceso a los supercomputadores es limitado y son usados mayoritariamente en centros de investigación académicos o gubernamentales. Esto queda en evidencia al revisar la lista del Top500 de los supercomputadores más potentes, por lo que el uso de computación paralela fuera de esos círculos es limitado.

Otro aspecto con los supercomputadores es que generalmente están configurados para apoyar en la resolución de problemas específicos y ejecutan aplicaciones relacionadas con una o pocas áreas de investigación. La configuración de los sistemas de HPC, específicamente su sistema operativo, librerías y drivers, son gestionadas por los administradores del sistema. Por lo que los usuarios del sistema no tienen la posibilidad de modificar la configuración del ambiente, y si requieren el uso de otros sistema operativo, librerías o *drivers* no pueden efectuarlo. En contraste con la ejecución utilizando contenedores, donde la configuración, del sistema operativo, librerías, drivers puede ser cambiada sin riesgos y con una mayor facilidad de uso.

Esto sucede para evitar problemas de configuración y compatibilidad que puedan afectar el desempeño del sistema y los trabajos de otras personas que lo estén utilizando para ejecutar sus aplicaciones. A lo anterior se debe agregar que los sistemas de HPC con frecuencia son propietarios y poco flexibles, que el costo de reconfiguración es muy alto, y generalmente se enfocan a un dominio específico.

La computación en la nube permite el acceso instantáneo a recursos computacionales, por

medio de proveedores públicos que cobran una tarifa por uso. La mayoría de recursos disponibles están enfocados en otras áreas como el desarrollo de aplicaciones con micro servicios y no de tareas de HPC y computación paralela. En los últimos años varios proveedores de estos servicios han comenzado a ofrecer soluciones de HPC en la nube que pueden competir con los mejores supercomputadores del Top500. Según un estudio realizado en 2020 alrededor del 20% de los trabajos de HPC se ejecutan con ayuda de la nube o completamente en ella [54]. La mayoría de estos trabajos usan los recursos de la nube para aumentar la capacidad de sus sistemas en-sitio. De manera similar otro estudio encontró que el 39% de pequeñas y medianas empresas estaban planeando ejecutar sus trabajos de HPC e IA en la nube [54].

Las soluciones de HPC en la nube son plataformas de pago y propietarias que requieren efectuar pagos elevados para su uso y vincularse por largo tiempo con un determinado proveedor para evitar efectuar procesos complicados de migración en caso de querer cambiar a otro proveedor. Estas son dos características que restringen el acceso a algunas instituciones e individuos con presupuestos reducidos. Algunos ejemplos de plataformas en la nube orientadas a HPC son Azure High Performance Computing, Google Cloud HPC solutions y AWS HPC. Estas pertenecen a las tres principales soluciones de servicios en la nube Azure, GCP y AWS, respectivamente[4].

Por lo que se requiere contar con métodos para la ejecución de tareas de computación paralela que sean accesibles, en términos económicos, y configurables para que el usuario pueda contar los recursos que requiere de acuerdo con sus necesidades particulares. Las propuestas anteriores son avances interesantes en esa dirección, como se describe en la sección 1.1. Sin embargo, este campo de investigación presenta oportunidades para realizar propuestas que ofrezcan métodos más adecuados en esa dirección.

Con base en lo anterior, este trabajo de investigación propone un método abierto para la creación de ambientes que permitan la ejecución de aplicaciones de computación paralela mediante la orquestación de contenedores y procesos que se ejecuten de forma distribuida en equipos de bajo costo utilizando operadores de Kubernetes. El propósito es facilitar el acceso a recursos de computación paralela utilizando sistemas distribuidos orquestados por un método basado en operadores de Kubernetes. Lo anterior, utilizando sistemas de bajo costo en el sitio[5] o en la nube, sin requerir un supercomputador o pagar servicios especializados de HPC en la nube. Por lo que la pregunta de investigación que se busca responder con esta investigación es la siguiente:

> ¿Cómo definir un método para la creación automatizada de clústers que permitan la ejecución de aplicaciones de computación paralela basadas en MPI mediante el uso de computación distribuida en un clúster de Kubernetes?

[4] Microsoft con Azure, Amazon con Amazon Web Services (AWS) y Google con Google Cloud (GCP) son las compañías con la mayor cuota del mercado y las que ofrecen una gran variedad de servicios de computación en la nube [53, 14, 13, 12]

[5] El término en el sitio se refiere a lo que se conoce en inglés como sistemas *on-premise*.

1.3 Justificación del problema

El uso de la computación distribuida y recursos computacionales, para el despliegue de aplicaciones de computación paralela en la nube tendrá un incremento sostenido en los próximos años. Los sistemas tradicionales para HPC son los supercomputadores, pero tienen un alto costo de construcción y su vida útil se limita a unos pocos años. Por lo que es necesario encontrar alternativas viables que brinden rendimientos similares a los supercomputadores [54].

El surgimiento de nuevas tecnologías no sustituye a los supercomputadores, porque estos seguirán siendo de suma importancia para la comunidad científica, las instituciones educativas y los centros de investigación. Sin embargo, el uso de las nuevas tecnologías y métodos de despliegue de aplicaciones podrían permitir que las instituciones más pequeñas aprovechen las ventajas del uso de la computación paralela en proyectos de pequeña y mediana escala. Además, estas también permitirán la mayor utilización de la computación paralela en la industria y el comercio.

La virtualización basada en contenedores ofrece menores rendimientos para la ejecución de aplicaciones de HPC y computación paralela, como lo evidencian diferentes estudios que detallan resultados desfavorables en cuanto a rendimiento [30, 57, 21, 43]. Sin embargo, además del rendimiento existen otros factores que pueden ser tomados en cuenta a la hora de decidir si se debe ejecutar una aplicación de computación paralela en contenedores o en supercomputadores. Al respecto, algunos estudios evidencian que el uso de contenedores benefician la facilidad de uso y desarrollo, la robustez, la escalabilidad y la eficiencia de las aplicaciones [62]. Lo que hace importante continuar realizando investigación para buscar mejores soluciones que potencien los beneficios y permitan mejores rendimientos.

La inteligencia artificial generativa[6] es un tema que con rapidez pasó de ser un tema de investigación a una área que se aplica con frecuencia en entornos empresariales reales. En la actualidad existen herramientas que se encuentran disponibles al público general, que puede utilizarlas para resolver problemas complejas sin contar con conocimiento previo ni especialización. Estas herramientas utilizan modelos con cientos o miles de millones de parámetros que hacen necesario utilizar poder de cómputo a gran escala para su entrenamiento, por lo que con frecuencia se requiere usar computación paralela. Esto hace imperativo contar con suficientes alternativas para tener acceso al poder de computo necesario para el entrenamiento de los modelos actuales, pero también de los que se utilicen en el futuro.

[6] La inteligencia artificial generativa, es un área de la inteligencia artificial que permite generar contenido nuevo a partir de una entrada en lenguaje general. El contenido generado puede variar e incluye texto, imágenes, vídeos y código funcional escrito en diferentes lenguajes de programación. El aspecto clave de los modelos de inteligencia artificial generativa, es que su arquitectura y volumen de datos de entrenamiento son enormes.[37]

1.4 Hipótesis

Esta sección presenta las hipótesis de este trabajo de investigación, las cuales serán probadas por medio de un conjunto de experimentos durante el desarrollo de la tesis:

Hipótesis 1: Un método para la creación automatizada de un ambiente de ejecución de aplicaciones de computación paralela basadas en MPI en un clúster de Kubernetes en una infraestructura computacional basada en procesadores x86 puede crear el ambiente en un tiempo inferior a 1 * N minutos donde N es el número de nodos del ambiente.

Hipótesis 2: Un método para la ejecución de aplicaciones de computación paralela basadas en MPI en un clúster de Kubernetes puede proporcionar niveles de rendimiento de un 70% en relación con el rendimiento de un clúster físico de computación paralela que utiliza una infraestructura computacional similar.

1.5 Objetivos

Esta sección presenta el objetivo general y los objetivos específicos de este trabajo de investigación.

1.5.1 Objetivo general

En concordancia con lo mencionado en las secciones anteriores, el objetivo general de esta investigación es el siguiente:

> Definir un método para la creación automatizada de ambientes que permitan ejecutar aplicaciones de computación paralela basadas en MPI mediante el uso de computación distribuida en un clúster de Kubernetes cuyo tiempo de preparación para la ejecución de tareas requiere menos 1 * N minutos donde N es el número de nodos del ambiente.

El método que se propondrá debe permitir obtener rendimientos de ejecución iguales o superiores a un 70% en relación con el rendimiento de un clúster físico de computación paralela que utiliza una infraestructura computacional similar en equipos con procesadores x86. Además el método debe permitir que el clúster de Kubernetes esté preparado para ejecutar tareas de computación paralela en menos de 1 * N minutos donde N es el número de nodos del ambiente.

1.5.2 Objetivos específicos

1. Determinar la estrategia para diseñar un método que permita la creación automatizada de ambientes para la ejecución de tareas de computación paralela basadas en MPI en un clúster de Kubernetes.

2. Diseñar un método para la creación de clústers para la ejecución tareas de computación paralela basadas en MPI en un clúster de Kubernetes.

3. Validar el método mediante la implementación de una prueba de concepto de acuerdo con el diseño propuesto utilizando el lenguaje de programación Go.

4. Documentar de forma detallada el diseño y la implementación de la prueba de concepto con el fin de que pueda ser replicado y extendido por otros investigadores.

5. Validar que el método propuesto puede crear el ambiente en un clúster de Kubernetes con arquitectura x86 en menos de 1 * N minutos donde N es el número de nodos del ambiente.

6. Probar que el método propuesto obtiene rendimientos de ejecución iguales o superiores a un 70% en un clúster de Kubernetes en relación con el rendimiento de un clúster físico utilizando benchmarks de HPC.

7. Efectuar la comparación del rendimiento entre un clúster de Kubernetes y un clúster de computación paralela utilizando MPI y equipos físicos con arquitectura de procesadores x86.

1.6 Alcance de la investigación

El alcance del presente trabajo de investigación incluye el diseño de un método que permitirá crear clústers de trabajo[7] en un clúster de Kubernetes con un determinado número de nodos que permite que los usuarios ejecuten tareas de computación paralela basadas en MPI.

El diseño del método, se llevará a cabo a partir del análisis que se efectuará en el marco teórico y los trabajos de investigación expuestos en los antecedentes. Estos permitirán contar con la base teórica para diseñar el método de computación paralela y la implementación de una prueba de concepto orientada a validar las hipótesis planteadas.

La implementación de la prueba de concepto del método, se llevará a cabo utilizando un clúster de Kubernetes en un equipo local y al menos 3 nodos. Una vez que se hayan realizado las pruebas correspondientes para determinar la factibilidad del método en este

[7]Los clúster de trabajo se conocen en inglés como *workload clusters* y se utilizan para ejecutar tareas de aplicaciones. Los clúster de trabajo son creados dentro de un clúster de Kubernetes el cuál gestiona y controla los clústers de trabajo. Es posible utilizar una combinación de recursos del clúster de Kubernetes como *namespaces*, *quotas* y *policies* para la creación de estos.

ambiente, se procederá a implementar la prueba de concepto en un clúster de Kubernetes distribuido con al menos tres nodos.

Los experimentos para validar los objetivos 5, 6 y 7 serán efectuados utilizando herramientas disponibles de uso gratuito de la comunidad de HPC. Las herramientas seleccionadas deberán haber sido utilizadas en otros trabajos de investigación relacionados.

Este documento en conjunto con un repositorio de código, proporcionarán toda la documentación, referencias y código necesarios para replicar la implementación y validaciones realizadas en la presente investigación. El alcance de la investigación se limita a ejecutar aplicaciones existentes de computación paralela basadas en MPI y no contempla desarrollarlas. La implementación de la prueba de concepto proporcionará el ambiente correcto para la ejecución de las aplicaciones en el clúster de trabajo creado utilizando el método propuesto.

Capítulo 2

Marco teórico

Esta sección muestra el marco teórico de este trabajo de investigación, la teoría presentada abarca la definición, uso y medición de HPC y Computación Paralela, la arquitectura e importancia de la virtualización basada en contenedores, la arquitectura y principales componentes que conforman Kubernetes y por último se cubren los Operadores y el SDK de Operadores utilizados. Todos los temas presentados en el marco teórico tienen especial relevancia para la implementación y corroboración de la hipótesis de esta investigación.

2.1 HPC y Computación Paralela

La computación paralela es el uso de un grupo de entidades computacionales trabajando en conjunto para resolver un problema, el cual es dividido en partes más pequeñas que son enviadas a cada entidad para su resolución. Su meta es aumentar el rendimiento y resolver el problema en una menor cantidad de tiempo aprovechando los recursos disponibles. Estas entidades están altamente entrelazadas, lo que significa que si una de ellas falla, todo el computo puede fallar.

HPC es un acrónimo de la frase anglosajona *"High Performance Computing"* que se puede traducir al español como Computación de Alto Desempeño, se refiere al uso de súper computadoras para la resolución de problemas computacionales donde el uso de computadoras convencionales para resolverlos no es factible debido a su gran tamaño y complejidad.

En un sistema de HPC convergen cinco áreas de estudio de las ciencias de la computación, redes computacionales, rendimiento computacional, computación distribuida, computación concurrente y por supuesto computación paralela [52].

Los sistemas de computación paralela se puede categorizar en dos:

- SIMD: Un programa o instrucción, y múltiples datos.
- MIMD: Múltiples programas o instrucciones, y múltiples datos. A su vez MIMD se

subdivide en dos grupos:

- Sistemas de memoria compartida donde todos los procesadores comparten un espacio de memoria y se comunican por medio de variables compartidas, es un modelo simple pero poco escalable.
- Sistemas de memoria distribuida donde cada procesador posee su propia memoria privada, la comunicación se hace a través de mensajes, es escalable.

2.1.1 *Messaging Passing Interface* (MPI)

El pase de mensajes es un paradigma de programación para el intercambio de datos de manera cooperativa, los datos son enviados y recibidos explícitamente de un procesador a otro. MPI[1] es una especificación de librería para el pase de mensajes, se desarrollo en 1992 debido a la necesidad de estandarizar el pase de mensajes en sistemas de memoria compartida, su primer borrador se completo en 1994 y la última versión aprobada es MPI 4.0 [39].

Los mensajes son enviados por medio de SSH[2], un protocolo de comunicación que permite comunicaciones de red seguras sobre una red insegura, una vez se lleva a cabo la autenticación entre las partes, todos los intercambios de datos son encriptados [44]. La autenticación es llevada acabo por medio de un par de llaves, una privada y una pública, el emisor es dueño de la llave pública, y el receptor lo autoriza por medio de la llave pública.

Debido a que MPI es un estándar agnóstico de un lenguaje de programación o arquitectura, existen varias librerías que lo implementan, las más populares son OpenMPI y MPICH. MPICH es un proyecto de código abierto listo para producción, su meta es combinar la portabilidad con el alto rendimiento, siendo la referencia desde los primeros hasta los últimos estándares publicados y la base para trabajos derivados [40].

OpenMPI, es una implementación del estándar a partir de su versión 2, es un proyecto de código abierto listo para producción, su meta es ofrecer un rendimiento listo para producción que soporte un amplio rango de máquinas paralelas [38].

El uso que las personas alrededor del mundo le dan a los sistemas de HPC y la computación paralela puede variar, sin embargo las formas de hacerlo son muy parecidas. Generalmente una aplicación de computación paralela es creada utilizando estándares y lenguajes ya conocidos como MPI y C o C++. Una vez creada la aplicación esta es compilada lista para ser ejecutada desde una ventana de línea de comandos pasando los argumentos necesarios para el correcto funcionamiento de la aplicación.

Los recursos de *hardware* de HPC son limitados debido a su alto costo de compra y operación, debido a esto es necesario limitar su uso para no sobrecargar el sistema, para cumplir este requisito las instituciones han instaurado sistemas de colas, donde las personas

[1] Siglas de la frase en inglés *Message Passing Interface*
[2] Siglas de la frase en inglés *Secure Shell Protocol*

interesadas en usar los recursos de HPC para ejecutar sus aplicaciones de computación paralela pueden enviar trabajos a la cola.

Hay distintos sistemas para realizar la implementación de este modelo siendo el más popular SLURM [7]. Cuando el usuario tiene su turno el trabajo es ejecutado y el resultado guardado en la forma en que el usuario especifique, generalmente un archivo de texto. Bajo este modo los usuarios no tienen acceso directo al *hardware* de HPC sino que interactúan con el atreves de un nodo intermedio conocido como nodo de sesión, para acceder al nodo de sesión el usuario debe estar previamente autorizado.

2.1.2 Medición del rendimiento de sistemas de HPC

Par cumplir el objetivo de que un sistema de HPC pueda proporcionar los recursos necesarios para la ejecución de aplicaciones y resolución de problemas en un tiempo determinado es necesario que se cumpla que el poder de computo del sistema se incremente a medida que la cantidad de recursos de este incremente, es decir que sea escalable. El diseño y construcción de sistemas de HPC requieren de la ejecución de pruebas de rendimiento utilizando *Benchmarks*[3] para el posterior análisis del rendimiento del sistema. Esto con el fin de realizar las modificaciones necesarias al sistema para el funcionamiento idóneo de acuerdo al tipo de problemas que se quieran solucionar.

Existen gran cantidad de benchmarks de acceso libre utilizados para evaluar los sistemas de HPC en distintos dominios. a continuación se mencionan cinco *Benchmarks* comúnmente utilizados en la industria para evaluar el rendimiento entre súper computadoras y para la evaluación de trabajos de investigación.

Los *NAS Parallel Benchmarks* (NPB), un conjunto de programas pequeños diseñados para evaluar el rendimiento de computación paralela en supercomputadores desarrollados por la NASA. Sus primeros *benchmarks* tenían como objetivo simular el computo y movimiento de datos de aplicaciones de dinámica de fluidos, luego fueron agregados otros *benchmarks* para probar más escenarios. Como por ejemplo, Computación no estructurada, entrada/salida paralela, movimientos de datos y el rendimiento de redes computacionales [18, 66, 35, 64, 36, 63].

Linpack es una colección de subrutinas de Fortran para la resolución de varios sistemas de ecuaciones lineales que describen el rendimiento a diferentes tamaños de problema [32]. Hoy día el *Linpack* es utilizado por científicos en todo el mundo para evaluar el rendimiento computacional de computadores de alto rendimiento [32]. La lista del Top500 utilizan *Linpack* como el *Benchmark* referencia que permite a los usuarios ajustar el tamaño del problema y el software para lograr los mejores resultados para un sistema específico.[9].

[3]Los *Benchmarks* de HPC son un conjunto de herramientas en forma de aplicaciones o *kernels* que ayudan a evaluar el rendimiento y las capacidades de un sistema de HPC. Miden diferentes aspectos del sistema tales como utilización de recursos (CPU, Memoria, Disco), velocidad y latencias en la comunicación, tiempos parciales y totales para finalizar ciertas tareas y cantidad de procesos y threads.

Los *Coral 2 Benchmarks* son un conjunto de comprensivo de *Benchmarks* científicos relacionados a diferentes dominios de investigación y generalmente de código abierto [59]. Forman parte del proyecto Coral, una colaboración entre los laboratorios de Oak Ridge, Argonne y Livermore del Departamento de Energía de EEUU con el objetivo de construir sistemas de computación de alto rendimiento de última generación que ayuden al descubrimiento científico y a la seguridad Nacional [61]. Actualmente son utilizados por por diferentes fabricantes para evaluar y construir sistemas de HPC [59].

El proyecto Mantevo esta conformado por varias *miniapps*, estas son aplicaciones de *Benchmarks* que extraen las partes de mayor requerimiento computacional de aplicaciones más grandes, las cuales generalmente son una pequeña porción del código de la aplicación [29]. Las *miniapps* permiten explorar rápidamente el rendimiento de diferentes áreas de un sistema computacional como *hardware*, ambiente de ejecución, lenguajes y compiladores, algoritmos e implementaciones con el fin de ayudar a la toma de decisiones para la optimización del sistema [29].

Los *MLPerf* es un estándar hecho por la comunidad para medir el rendimiento de trabajos de *machine learning* enfocado en métricas de rendimiento alcanzado durante el entrenamiento de modelos y también durante la inferencia´utilizando modelos entrenados.*MLPerf HPC* es un *benchmark* de aplicaciones científicas de aprendizaje de *machine learning* de gran escala [34]. Su meta es acelerar la innovación de los sistemas de HPC y el software utilizado para trabajos de *machine learning*[34].

MLPerf Training es un *benchmark* que mide que tan rápido un sistema puede entrenar modelos de inteligencia artificial con una métrica de calidad objetivo [51]. Uno de los modelos entrenados es el ResNet-50 un modelo RNN de clasificación de imágenes de gran relevancia debido a su amplio uso en la comunidad científica [60]. El *benchmark* de ResNet-50 del *MLPerf Training* utiliza el conjunto de datos de *ImageNet* [58].

AI Benchmark Alpha es un paquete de *python*[4] que contiene 42 pruebas distribuidas en 19 etapas basadas en modelos de diferentes artículos científicos las cuales abarcan la mayoría de tareas y arquitecturas de *deep learning* [46]. Su enfoque es determinar el rendimiento que puede proporcionar un equipo al entrenar modelos de inteligencia artificial o realizar inferencias utilizando estos modelos. Su limitante es que no es posible ejecutar los benchmarks en un ambiente distribuido. Sin embargo, es un excelente punto de partida para determinar el rendimiento que un equipo puede brindar para tareas de inteligencia artificial.

[4] *Python* es un lenguaje de programación el cuál es utilizado ampliamente para el desarrollo de aplicaciones y modelos de inteligencia artificial.

2.2 Contenedores

Hasta hace unos años cuando se hablaba de virtualización, esta se relacionaba con la típica arquitectura de un *hypervisor*[5]. Al ser una capa de emulación tiene sus ventajas como permitir virtualizar diferentes arquitecturas y sistemas operativos, sin embargo esto viene con un alto costo de rendimiento, generalmente para emular una arquitectura y obtener resultados similares se ocupa un *hardware* muchas veces más potente [33]. La figura 2.1 ejemplifica gráficamente la virtualización por medio de *hypervisors* donde las instrucciones de las aplicaciones de cada guest deben pasar por varias capas de emulación y el *hypervisor* para acceder al *hardware* físico.

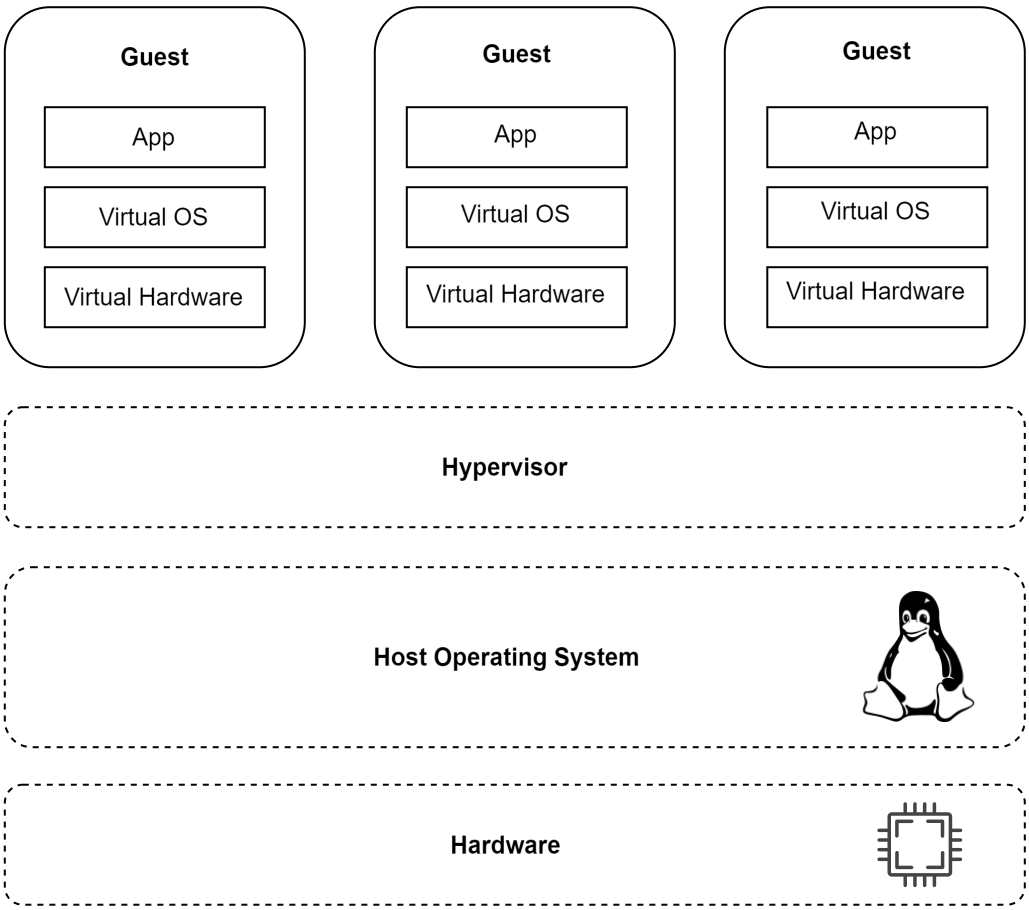

Figura 2.1: Virtualización con *Hipervysor*

Hoy día existen otros tipos de virtualización, siendo los contenedores una de los más populares. A diferencia de un *hypervisor* que debe traducir las instrucciones del sistema virtualizado a instrucciones entendibles para el *hardware* físico, los contenedores se ejecutan sobre el sistema operativo. Dentro del contenedor los procesos se ejecutan en un ambiente aislado de otros para proporcionar seguridad, sin embargo estos se comunican directamente

[5]Un *hypervisor* es una capa de emulación entre el *hardware* físico o sistema operativo y el *hardware* virtualizado de la máquina virtual

con el kernel del sistema operativo sin una capa de emulación de por medio [33].

Debido a que no existe emulación los contenedores son una opción mucho más eficiente y liviana, un solo sistema operativo ya sea en *hardware* físico o virtualizado puede ejecutar cientos de contenedores al mismo tiempo. La figura 2.2 ejemplifica gráficamente la virtualización por medio de contenedores donde estos por medio del *engine* que no es más que un software instalado en el sistema operativo pueden utilizar las funciones del mismo sistema operativo y kernel para acceder al *hardware*.

Con el nacimiento de los contenedores, la forma de distribución del software cambio radicalmente, ya sea un ambiente de desarrollo, o una aplicación lista para producción, un equipo de desarrollo puede generar una imagen, que contiene las instrucciones para construir el contenedor, y ser utilizada cuantas veces y donde requiera. Esto tiene grandes ventajas, el equipo de desarrollo pueden tener un enfoque centrado en aplicación donde la aplicación es lo más importante, y el equipo de infraestructura tiene la posibilidad de actualizar el *hardware* con un impacto mínimo o nulo en las aplicaciones [33, 23].

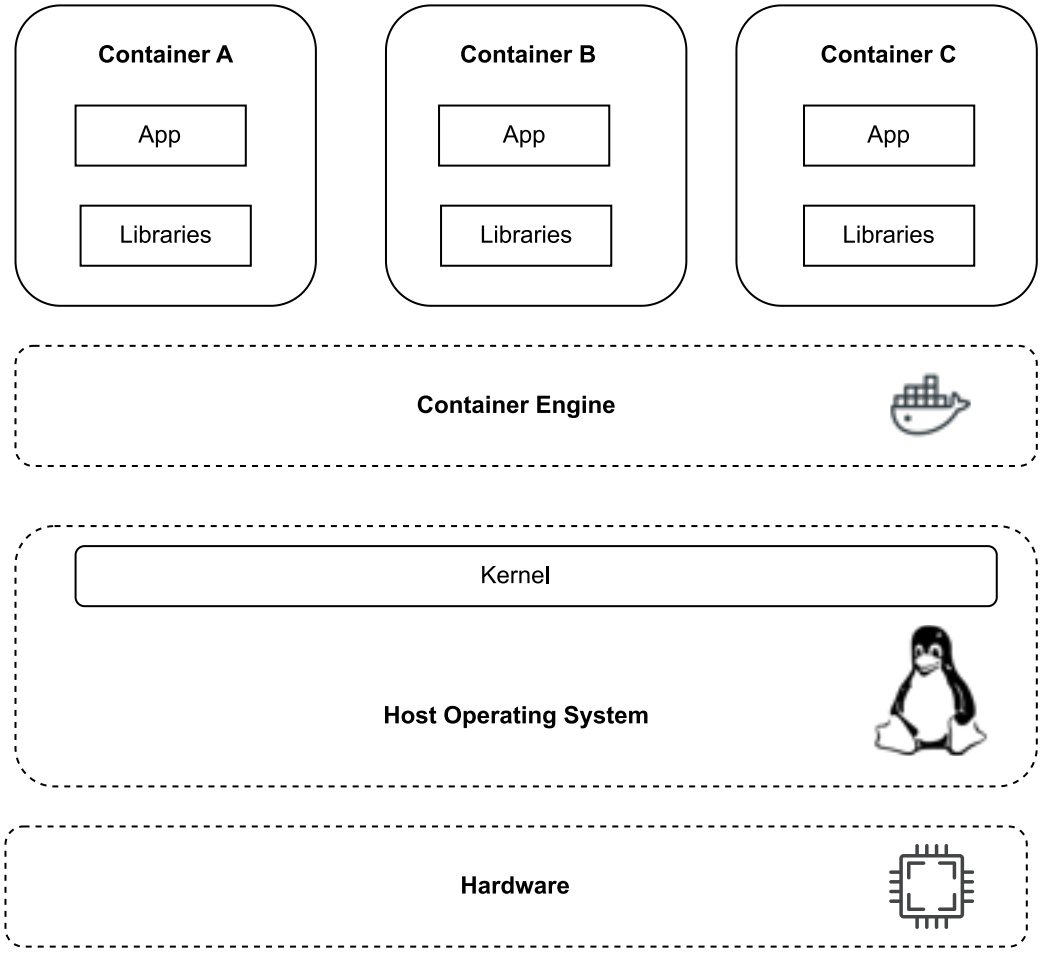

Figura 2.2: Virtualización con Contenedores

2.3 Kubernetes

Kubernetes es un orquestador de contenedores, un orquestador es un sistema que despliega y maneja aplicaciones en contenedores, este se encarga de todo el manejo de manera automática sin intervención del usuario. Fue creado inicialmente por Google como un proyecto de código abierto, con base en experiencias previas de sus dos sistemas internos para el manejo de contenedores Borg y Omega.[23].

Desde su creación, Kubernetes a adoptado una filosofía donde el sistema se extiende en base a *plugins*. Por ejemplo, Kubernetes como tal no se encarga de ejecutar los contenedores, este es el trabajo del *"Container Runtime Interface"* (CRI), el CRI es la capa encargada de ejecutar los contenedores, por defecto el *plugin* para la CRI depende de la distribución de Kubernetes sin embargo la mayoría usa ContainerD[2].

Es posible configurar uno o más CRI en Kubernetes, esto permite hacer el despliegue de imágenes de aplicaciones creadas con diferentes runtimes de contenedores. Así como el runtime para los contenedores, la interfase para el manejo de red y almacenamiento siguen una filosofía parecida, esto permite a la comunidad extender fácilmente el proyecto.

Un clúster de Kubernetes se compone de uno o más nodos de control llamados *control plane nodes*, y uno o más nodos de trabajo llamados *worker nodes*, una representación gráfica puede verse en el diagrama de la figura 2.3. Los nodos pueden ser servidores *bare metal*, máquinas virtuales, instancias de una nube privada o instancias de una nube pública. Se pueden ver los *control plane nodes* como el cerebro de Kubernetes y los *worker nodes* como las partes que controla el cerebro, pero que hacen el trabajo [56].

El *control plane* es el encargado del manejo del clúster, en el se encuentran los servicios que permiten su correcto funcionamiento, toda interacción con el *control plane* ocurre a través del API de Kubernetes, el API es uno de los componentes del *control plane*. Kubernetes expone varios *endpoints* los cuales pueden ser utilizados por medio de verbos REST, como *GET*, *POST*, *PUT* y *DELETE*.

Etcd es una base de datos distribuida clave-valor con alta disponibilidad, también forma parte del *control plane* y es donde el clúster de Kubernetes almacena todo su estado, incluido las especificaciones de los objetos creados en el clúster. Todo lo que creamos con el API, tiene una especificación la cuál es guardada en etcd.

El *scheduler* es el componente dentro del control plane que se encarga de elegir en que nodos puede ejecutarse una tarea. Por ejemplo, cuando se inicializa una acción por medio del API que involucra la creación de nuevos contenedores, el *scheduler* selecciona los nodos capacitados para ejecutar el contenedor en base a la especificación proporcionada, y los recursos disponibles en cada nodo [16] [56].

En Kubernetes el estado del sistema es supervisado y manejado por diferentes controladores, un controlador se encarga de monitorear y mantener el estado deseado de un recurso determinado. Siempre que el estado actual sea diferente al estado deseado el controlador se encarga de realizar las tareas pertinentes para volver al estado deseado, este proceso se le

llama reconciliación [16] [56]. El *control plane* es el encargado de ejecutar los controladores por medio del *controller manager* (c-m).

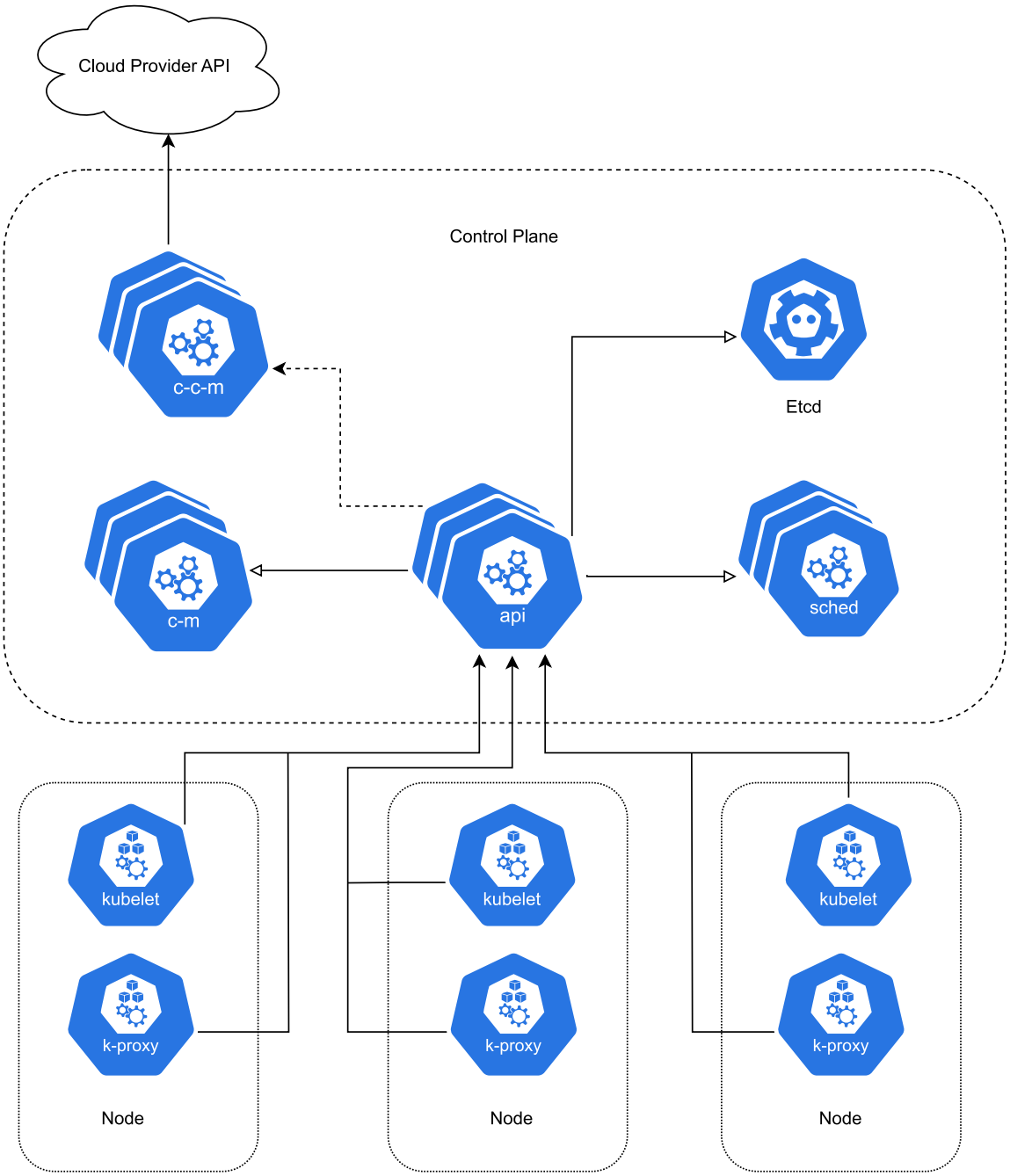

Figura 2.3: Clúster de Kubernetes [16]

Un *cloud controller manager* (c-c-m) ejecuta controladores que manejan recursos específicos de un proveedor de servicios en la nube, de esta manera Kubernetes divide la responsabilidad en controladores que manejan recursos de nuestro clúster gestionados por el c-m, y controladores que manejar recursos del proveedor gestionados por el c-c-m [16].

La unidad de despliegue más pequeña en Kubernetes son los *pods*. Un *pod* es un grupo de uno o más contenedores, estos comparten acceso a una red y almacenamiento común. Un *pod* puede tener uno o más contenedores ejecutándose en su interior, y a su vez un *pod* puede tener una o más replicas repartidas en el mismo o diferentes nodos, todas las réplicas del *pod* son similares y contienen los mismos contenedores generados a partir de la misma imagen [16].

Los *worker nodes* poseen un agente llamado kubelet, el kubelet recibe las solicitudes para la creación de *pods* por medio del *scheduler*. Después de recibir una solicitud, el kubelet basado en las especificaciones crea los *pods*, y asegura que esos permanezcan en el estado deseado. Para ejecutar los *pods* Kubernetes necesita de un *runtime* que le permita ejecutar los contenedores dentro del *pod*, cada *worker node* posee un CRI el cual se encarga de esta tarea.

La comunicación de red entre contenedores de un mismo pod, se lleva a acabo dentro del mismo *pod* usando la dirección local. Sin embargo, otro tipo de comunicación requiere del uso del k-proxy este componente instalado en cada nodo, permite la comunicación de un *pod* y sus contenedores hacia otros *pods* dentro o fuera del clúster de Kubernetes [56].

Kubernetes es un ambiente altamente dinámico, creando y ejecutando contenedores en *pods*, distribuidos en diferentes nodos. Además, se asegura de tener siempre el estado deseado según las especificaciones dadas, debido a esto la dirección IP interna de un *pod* es no determinista. Por ejemplo, si tenemos un *pod* y este es recreado por Kubernetes, no hay forma de asegurar que la IP va a ser la misma, el *pod* puede ser recreado en el mismo u otro nodo completamente distinto, cambiando su ubicación física.

Debido a lo anterior, es importante que exista una forma determinística en que los *pods* dentro de Kubernetes puedan descubrirse mutuamente, esto es posible gracias al servicio de descubrimiento de Kubernetes[6]. El servicio de descubrimiento en Kubernetes se lleva a cabo por medio del objeto de servicio. Un servicio en Kubernetes es un tipo de recurso, así como los *pods*. Un servicio mantiene una relación uno a uno, o uno a muchos con otros recursos, esta relación se lleva a cabo por medio de selectores[7].

Los servicios, exponen aplicaciones o APIs que se ejecuten en un puerto determinado dentro de los contenedores. Además, a cada servicio le es asignada una IP virtual llamada *"ClusterIP"*, esta IP es estable y permite que un pod sea descubierto por otros en el clúster. Al tener una IP estable cada servicio también recibe un DNS[8], que es proporcionado por el servicio de DNS de Kubernetes también conocido como clúster DNS, todos los *pods* de un clúster tienen acceso al clúster DNS [22].

En el ejemplo de la figura 2.4 se puede apreciar gráficamente la interrelación entre los *pods*

[6]El *service discovery* como es conocido en inglés, es el servicio que ayuda a buscar que procesos están siendo escuchados en determinadas direcciones para determinados servicios. [22]

[7]En Kubernetes los objetos pueden ser identificados por medio de etiquetas, estas después pueden ser utilizadas en selectores para crear una relación entre dos tipos diferentes de recursos [22]

[8]Las siglas DNS provienen de la palabra anglosajona *Domain Name System*, se refiere a un nombre que es utilizado para identificar una dirección IP especifica en la red.

y los componentes del servicio de descubrimiento. Los IPs asignados a los pods pueden cambiar en el tiempo sin embargo el clusterIP seguirá siendo el mismo aún cuando los pods se eliminen y vuelvan a crear.

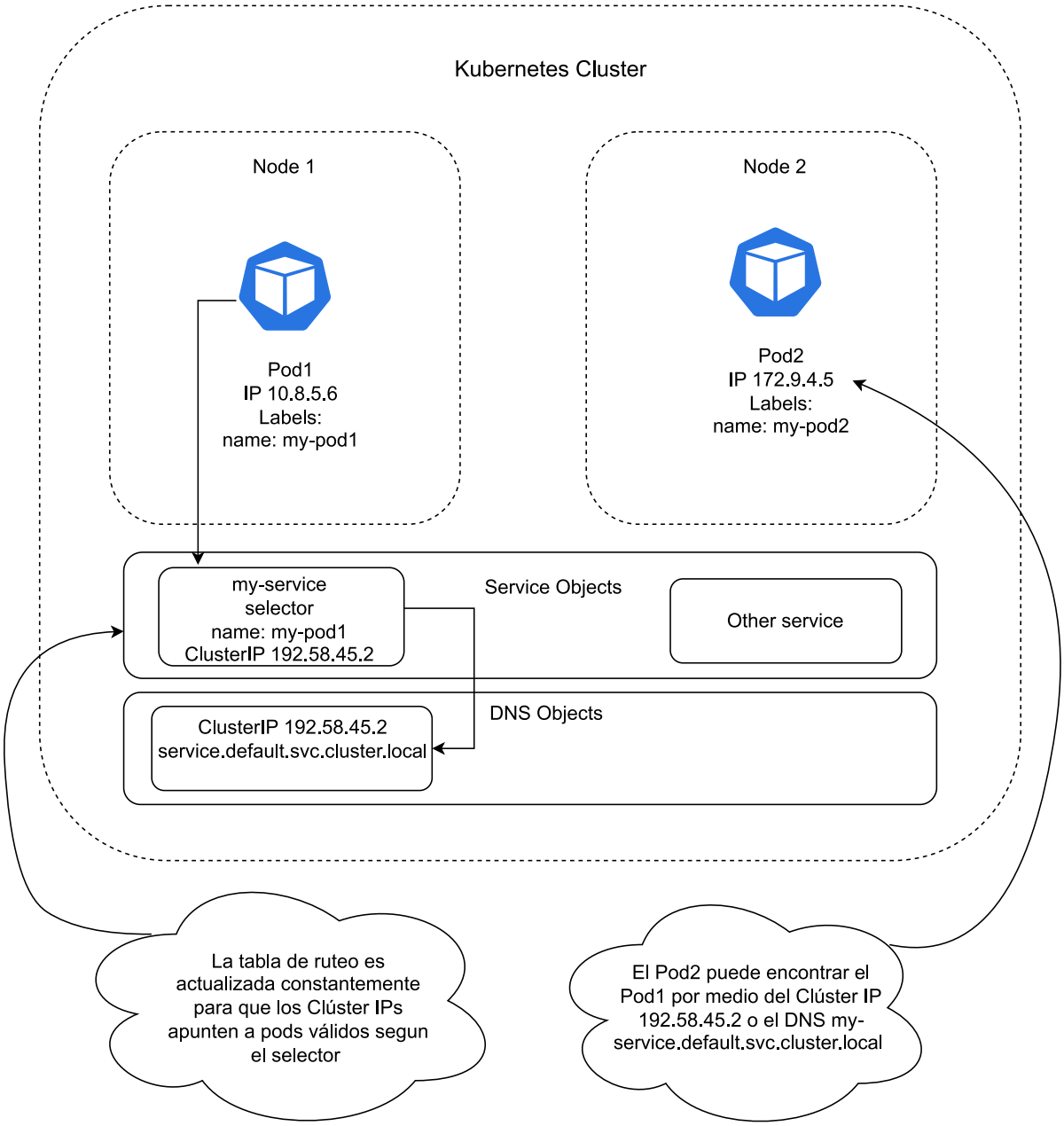

Figura 2.4: Ejemplo de un *pod* expuesto mediante un objeto de servicio.

Para el manejo de aplicaciones sin estado, Kubernetes provee un recurso llamado *deployments*. Estos permiten generar cuantos *pods* sean necesarios a partir de una imagen de contenedor, lo componen dos partes esenciales la especificación[9] y el controlador de

[9]La especificación de un *deployment* esta conformada por un archivo de texto escrito en lenguaje YAML, contiene meta-datos donde el estado deseado de la aplicación es definido. Esta incluye número de

deployments.

Cada deployment esta limitado a una aplicación, es decir el alcance de los *pods* que se generen a partir del *deployment* se limita a la aplicación del deployment. El controlador de deployment se encarga de mantener el estado deseado, en caso de que un *pod* deje de funcionar el controlador se encargará de crear uno nuevo. [56].

Algunos sistemas no necesitan de un estado permanente para funcionar, sin embargo muchísimos sistemas necesitan persistir su estado en un lugar seguro. Archivos de configuración, bases de datos, reportes, entre muchos otros necesitan de un lugar donde estos puedan ser almacenados permanentemente. Los pods lamentablemente no son un lugar seguro debido a su naturaleza temporal.

En Kubernetes, el subsistema de volúmenes persistentes[10], permite establecer una relación entre un espacio de almacenamiento y uno o más *pods*. Actualmente existen una gran variedad de tipos de almacenamientos para Kubernetes, son soportados desde carpetas compartidas en la red local hasta almacenamientos elásticos en la nube, todo esto gracias a la Interfaz de almacenamiento de contenedores (CSI), un estándar abierto que provee una interfaz de almacenamiento común para todos los orquestadores de contenedores. El subsistema de volúmenes persistentes está conformado por tres tipos de objetos que hacen posible el mapeo y uso del almacenamiento en los pods [56].

- Volumen persistente (PV): Mapean una directorio dentro del contenedor con un espacio de almacenamiento fuera de este, cualquier cambio realizado en fuera o dentro del *pod* será reflejado en ambas partes.
- *Claims* de volumen persistente (PVC): Permiten dar accesos a los *pods* para escribir, leer o ambos, en un PV.
- Clases de almacenamiento (SC): Un objeto compuesto que unifica los PV con los PVC.

La figura 2.5 presenta la interacción necesaria entre componentes, para que un *pod* pueda acceder a un almacenamiento, en este caso un servidor de archivos compartidos NFS. Sin embargo, el tipo de almacenamiento podría ser cualquier siempre y cuando se disponga del plugin necesario para el CSI. Los *pods* siempre verán el almacenamiento como un volumen mapeado a una ruta visible dentro del contenedor. Un volumen no esta ligado a un único *pod*, varios *pods* pueden utilizar el mismo volumen, leyendo y escribiendo datos al mismo tiempo.

Una característica importante de los *deployment* es que los *pods* generados a partir de estos no son determinísticos, su nombre y localidad pueden variar cada vez que se inicie o recree un *pod*. Esto tiene ciertas implicaciones. La primera es que el DNS del *pod* es

réplicas, comandos de inicialización, variables de entorno, puertos expuestos, y características propias de los contenedores entre otros.

[10]Conocido en inglés como *Persistent Volume Subsystem*

variable, la segunda es que su IP interna es variable, y la tercera es que si se tiene un volumen asociado al nombre del *pod* este va a variar.

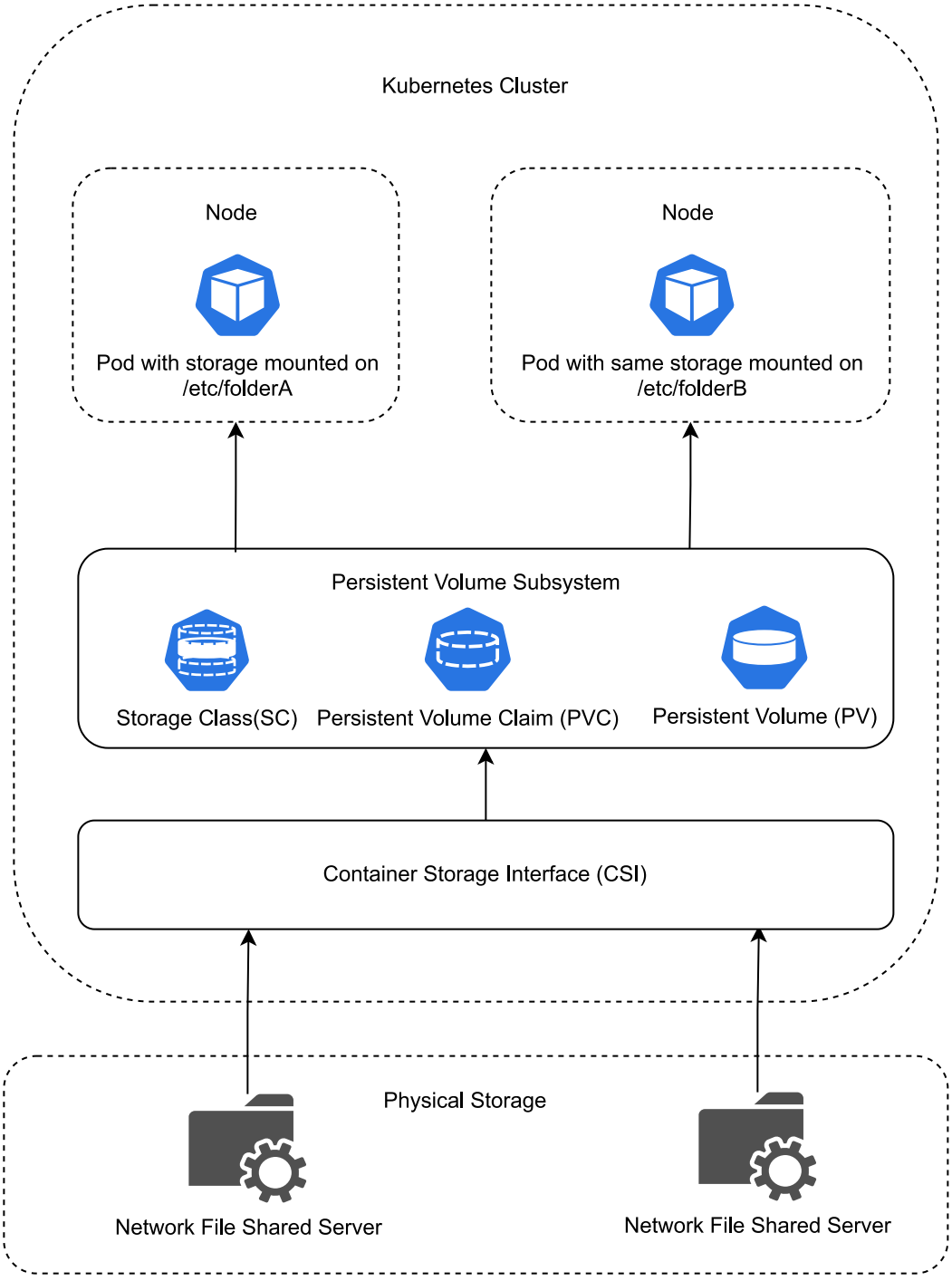

Figura 2.5: Subsistema de volúmenes permanentes

Kubernetes posee un tipo de recurso llamado *statefulset*, estos son muy parecidos a los *deployments*, ambos se encargan de mantener el estado deseado de un pod o varios durante su creación, fallos o operaciones de escalado. La diferencia radica en que los

pods generados a partir de una especificación de *statefulset* tendrán un nombre, DNS y volúmenes persistentes estables, si un *pod* falla y es recreado en otro nodo, será recreado con el mismo nombre, DNS y mapeo de volúmenes anteriores.

Para lograr que los *pods* posean un DNS estable, es utilizado un servicio *headless*, la diferencia del servicio *headless* con un servicio normal es que estos no poseen un clusterIP, el servicio headless luego es asignado al *statefulset* en su especificación y por cada *pod* creado se crear una entrada nueva en el servicio *headless* creando un DNS que apunte a ese *pod* el cual se mantendrá estable en el tiempo [56].

Todo sistema contiene variables de configuración, unas con información sensible que no pueden ser mostradas generalmente conocidas como secretos, y otras que pueden ser mostradas sin ningún tipo de encriptación. La solución de Kubernetes para esto son los recursos de tipo *configMap* y *secrets*. Los *configMap* y *secrets* son colecciones de objetos clave-valor, con la diferencia que los *secrets* son convertidos a su valor en base64. Kubernetes, puede mapear estos objetos clave-valor a variables de entorno o incluso a archivos de configuración creados en una ruta dentro del contenedor [56].

Toda interacción con los recursos de Kubernetes pasa por su API, por lo que es necesario un sistema de seguridad para protegerlo. El API permite el uso de diferentes acciones CRUD (Crear, Leer, Actualizar, Remover) para interactuar con los recursos. Todos recurso dentro del clúster de Kubernetes utiliza el API para interactuar con otros recursos. Ya sea el mismo Kubernetes, un script de automatización o un usuario usando la línea de comandos de Kubectl, todos ellos necesitan pasar por el API.

La seguridad en Kubernetes la conforman varias capas, la autenticación, la autorización, y el control de admisión. Todo pedido hacia el API debe incluir credenciales que autentiquen el origen en la capa de autenticación, el medio para autenticar el origen puede variar según la necesidad del sistema, certificados, *webhooks* o un proveedor de identidad en la nube pueden ser utilizados para este fin.

Luego de que la identidad del origen es validada, el pedido es enviado a la capa de autorización, al igual que la capa de autenticación, diferentes módulos pueden ser utilizados, sin embargo el más común es el Control de accesos basado en roles (RBAC), con el cual es posible especificar que usuarios pueden realizar que acciones sobre que recursos. Una vez autorizado el pedido pasa al control de admisión, este se encarga de validar políticas para determinar si el pedido en cuestión puede llevarse a cabo, llevarse a cabo con modificaciones, o debe ser rechazado. [56]

2.3.1 Operadores de Kubernetes y CRDs

Los servicios, *deployments*, *statefulsets*, entre otros son recursos predeterminados de Kubernetes cada uno con su propio controlador que permiten cubrir la mayoría de los casos de uso. Sin embargo, es posible extender la funcionalidad de Kubernetes creando recursos y controladores a la medida para cubrir determinada lógica y funcionalidades, los

cuales no puedan ser cubiertos con los recursos predeterminados.

Kubernetes permite crear recursos personalizados utilizando el objeto *custom resource definition* (CRD), un CRD actúa similar a un recurso predeterminado, puede ser creado, actualizado y borrado por medio del API, la diferencia radica en que este tiene sus propias propiedades. Por ejemplo, es posible crear un CRD "Automóvil" que contenga propiedades como marca, modelo, cilindrada y color, de esta manera es posible crear 1 o más recursos nuevos de tipo Automóvil cada uno con diferentes propiedades [56, 31].

Los operadores son la herramienta que Kubernetes provee a los ingenieros de infraestructura y desarrolladores para extender sistemas desplegados y ejecutados en Kubernetes, que cuenten con funcionalidades específicas que no puedan ser cubiertas por los recursos predeterminados del sistema.

Un CRD por si solo no es de mucha utilidad, es necesario un controlador que monitorice su estado y se encargue de realizar las acciones correctivas en caso de que el estado actual difiera del deseado. Un operador consiste en la unión de un CRD y un controlador que se encarga de mantener el estado deseado de todos los recursos creados con la especificación de su CRD. Debido a que es un controlador desarrollado a la medida es posible ejecutar eventos que respondan a cambios de estado de los recursos según se requiera. Por ejemplo, en el CRD del automóvil, cada vez que el color cambie, es posible enviar una notificación a un sistema externo para que este actualice su valor en una base de datos [31].

Operator SDK es un *toolkit* [11] que forma parte del proyecto *Operator Framework*, y permite que los desarrolladores se concentren en la implementación del núcleo de los operadores de Kubernetes. Este fue creado como un proyecto de código abierto para administrar aplicaciones nativas de Kubernetes. Entre las ventajas que proporciona se encuentran las siguientes:

- Cuenta con un conjunto de plantillas que permiten acelerar el proceso de desarrollo y elimina la necesidad de programar código repetitivo.
- Proporciona herramientas para desplegar y gestionar los operadores en clústers de Kubernetes.
- Proporciona flujos para desarrollar operadores con Go, Ansible y Helm, aunque el nivel de customización con estos últimos es mucho menor que con Go.

Entre las alternativas que ofrece *Operator SDK* para la implementación de los operadores, el uso del lenguaje de programación Go es la más poderosa, porque permite personalizar y configurar el operador de acuerdo con los requerimientos que se necesitan satisfacer [31, 5].

[11] Un *toolkit* o SDK por sus siglas en Ingles (*Software Development Kit*), es un conjunto de herramientas que permiten que los desarrolladores de software creen aplicaciones para determinada plataforma o lenguaje de programación, para el que fue hecho el SDK.

Instalación y administración de Operadores de Kubernetes

El *Operator Lifecycle Manager*(OLM) es un componente esencial del *Operator SDK* nos permite publicar un operador como un conjunto de imágenes de contenedor en un repositorio público o privado, una imagen funciona como el catalogo del operador la cuál contiene las diferentes versiones de este y punteros a las imágenes que contienen la implementación del mismo [31, 5].

Para utilizar los operadores publicados es necesario instalar el OLM en un clúster de Kubernetes e instalar el catalogo del operador deseado. Una vez instalado el catálogo es posible suscribirse a este lo cual generará un plan de instalación para crear todos los CRDs y controladores necesarios el correcto funcionamiento del operador [31, 5].

Una suscripción también permite actualizar el operador cada vez que una nueva versión es liberada. De esa forma para corregir errores o introducir una nueva funcionalidad al operador de Kubernetes, el proceso a seguir es el de crear un nuevo conjunto de imágenes de contenedor y publicarlas en el mismo repositorio utilizado la primera vez pero con una versión mayor, al actualizar la subscripción con la nueva versión OLM se encargará de actualizar todos los recursos asociados en el clúster de Kubernetes [31].

Capítulo 3

Metodología

En este capítulo se detalla la metodología de esta investigación, la cual se basa en la definición de la estrategia para diseñar un método, el diseño del método y los experimentos necesarios para probar las hipótesis descritas en el Capítulo 1. A continuación se describe la estrategia a seguir, las tareas necesarias para el diseño del método junto a la implementación de una prueba de concepto, y los experimentos a realizar.

3.1 Estrategia para el diseño del método

De acuerdo a la teoría estudiada, para lograr la correcta ejecución de una aplicación de computación paralela basada en MPI en un sistema distribuido se deben cumplir ciertas condiciones:

1. Los nodos deben conectarse entre si haciendo uso del protocolo SSH.

2. El intercambio de mensajes de MPI, requiere que los nodos mantengan una IP estable, ya que cada nodo debe mantener un registro de los otros para llevar a cabo la transmisión mediante SSH. Si un nodo cambia su IP, los demás no podrán conectarse a el por ende la trasmisión de mensajes se vería interrumpida.

3. Es conveniente que los nodos, puedan acceder a un espacio de memoria de almacenamiento compartido, esto para tener acceso a diferentes archivos necesarios para la ejecución de la aplicación, tales como archivos de configuración, ejecutables y bases de datos. Además permite la escritura de archivos de resultados visibles para todos los nodos.

4. Los nodos deben poseer el ambiente necesario para la ejecución de las aplicaciones. Este ambiente debería contar como mínimo con un sistema operativo basado en UNIX, servidor y cliente de SSH debidamente configurados, y todas las librerías necesarias para la ejecución de la aplicación en específico.

5. Cada nodo debe asegurar que dispone de cierta cantidad de recursos disponibles para la ejecución de las tareas. De otra manera, los resultado de diferentes iteraciones de un mismo experimento no serían comparables.

Para lograr estas condiciones es necesario seleccionar cuidadosamente los recursos adecuados que provee Kubernetes. Por ejemplo, los deployments nos permiten crear varios pods cada uno con su propia IP, que podrían conectarse mediante SSH. Sin embargo, los *pods* de un deployment no poseen una IP ni DNS estables, lo que provocaría que la configuración realizada deje de ser valida por ende una desconexión entre los *pods*.

Los *statefulsets* en conjunto con un servicio *headless*, son recursos de un clúster de Kubernetes, que por su naturaleza nos proporcionan una IP y DNS estables en comparación con los *deployments*. Se puede afirmar que un conjunto de *pods* creados a partir de un *statefulset*, pueden llegar a configurarse para su correcta conexión mediante SSH por medio de sus DNS, además la configuración de SSH relacionada a sus IPs puede configurarse y ser reescrita si fuera necesario a través del tiempo con la ayuda de sus DNS.

En base a lo anterior, el conjunto de *pods* creados a partir de un statefulset, puede ser visto como un clúster de trabajo. Donde cada *pod* representa un nodo del clúster de trabajo.

Para lograr un espacio de almacenamiento disponible en todos los nodos, es recomendable utilizar un volumen persistente montando en una ruta común en cada nodo del clúster de trabajo. Una clase de almacenamiento, un PV y un PVC son recursos del clúster de Kubernetes, que permiten automatizar la creación de un volumen de almacenamiento compartido para el *statefulset*. Este almacenamiento compartido debe ser lo suficientemente rápido de otra forma se convertiría en un cuello de botella para el resto del sistema.

En primera instancia se podría considerar NFS[1], sin embargo sufre de grandes problemas de escalabilidad y rendimiento en ambientes distribuidos y de HPC ya que no fue diseñado para esto. Una mejor opción es utilizar BeeGFS[2], un sistema de archivos paralelo diseñado para ambientes de HPC que combina la capacidad y rendimiento de varios servidores con el fin de proporcionar un almacenamiento compartido de alto rendimiento accesible desde cualquier cliente con acceso a la red. [50].

Una de las condiciones principales que el diseño debe cumplir, es proporcionar el ambiente necesario para la ejecución de las tareas de aplicación paralela en cada nodo. Existen ciertas librerías en común para las aplicaciones de computación paralela basadas en MPI, sin embargo existirán grupos de aplicación con uno o más requisitos específicos. Por ejemplo, habrán aplicaciones que necesiten cierta versión de Python o C para funcionar

[1] *Network File System* (NFS) es un protocolo de UNIX que permite acceder sistemas de archivos remotos a través de la red. El kernel de Linux posee un componente llamado el *NFS Server* que permite configurar los directorios que estarán disponibles a través de NFS y que pueden ser accedidos por otros equipos que cuenten con el cliente de NFS [44].

[2] BeeGFS es un sistema de archivos en la red, y a su vez es un sistema de archivos paralelo, es altamente escalable y fue creado para aquellos sistemas que requieren de almacenamientos compartidos de gran volumen y velocidad [42]. Soporta comunicación cliente-servidor por medio de interconexiones TCP/IP o RDMA como InfiniBand [42].

correctamente.

Una forma eficiente de configurar cada nodo es utilizando un ambiente auto-contenido en una imagen de contenedor. Esta imagen debe ser capaz de crear contenedores que proporcionen las librerías y configuraciones necesarias de acuerdo al tipo de aplicaciones que se deseen ejecutar en el clúster de trabajo. La imagen puede ser publicada en un repositorio de imágenes de contenedores, y ser utilizada luego por el *statefulset*.

Para asegurar los recursos asignados a cada nodo, y al clúster de trabajo que estos conforman. Una estrategia a seguir es aislar todos los recursos de Kubernetes creados para la creación del clúster de trabajo, bajo un mismo espacio de nombres y asignar cuotas de recursos a ese espacio de nombre en específico. De esta forma, es posible asignar recursos como RAM, Procesador y espacio de almacenamiento disponible.

Con el fin de cumplir el objetivo específico cinco de tiempo de despliegue, es necesario automatizar la creación y configuración de los recursos de Kubernetes. Lo primero, es identificar todos los pasos a seguir para la creación de un clúster de trabajo, esta validación puede ser realizada por medio de la aplicación de línea de comandos kubectl. Una vez identificados los pasos, el proceso se puede parametrizar y automatizar utilizando operadores de Kubernetes.

3.2 Validación del método y recursos definidos para su implementación en un ambiente de desarrollo

Este proceso de validación consiste en crear un clúster de trabajo de al menos 3 nodos en un clúster local de Kubernetes, para posteriormente ejecutar la implementación de Hola Mundo de MPI en todos los nodos. Un adecuado funcionamiento de los resultados obtenidos en este experimento, aporta la evidencia necesaria para afirmar que el método es válido y realizar la implementación de una prueba de concepto del mismo utilizando lenguaje Go y operadores de Kubernetes.

Objetivo: Probar que es posible la ejecución de aplicaciones de computación paralela basadas en MPI, en un clúster de Kubernetes. Utilizando la imagen base y los recursos de Kubernetes seleccionados para la implementación del método.

Descripción: Para este experimento es necesario configurar un clúster de Kubernetes en un ambiente local con un nodo de *control plane* y tres nodos de trabajo, existen varias opciones de distribuciones que se pueden utilizar para este fin, siendo una de las más utilizadas kind [56].

Una vez creado el clúster, se debe configurar siguiendo el diseño propuesto, la diferencia es que para el presente experimento se utilizaran archivos YAML que contengan los parámetros necesarios en lugar de un CRD y el operador. Para aplicar los archivos YAML en el clúster se utilizará la herramienta de línea de comandos

kubectl, la referencia de los comandos se encuentra en la documentación oficial de Kubernetes [16].

Una vez creado el clúster de trabajo se deberán ejecutar las siguientes pruebas dentro de este:

1. Probar en cada nodo perteneciente al clúster que puede conectarse a los otros nodos por medio de SSH, el resultado es positivo si todos los nodos pudieron conectarse a los otros sin problemas.
2. Ejecutar la implementación de Hola Mundo para MPI en todos los nodos del clúster, el resultado es positivo si la aplicación imprime la ejecución de Hola Mundo en todos los nodos.

3.3 Medición del tiempo de creación de un clúster de trabajo utilizando el método propuesto

La Medición del tiempo de creación de un clúster de trabajo consiste en medir la cantidad de tiempo en minutos, en que el operador tarda en crear un clúster de trabajo y este esta listo para empezar la ejecución de una aplicación de computación paralela utilizando la implementación del método propuesto.

Objetivo: Probar que es posible la creación de un clúster de trabajo y empezar la ejecución de una aplicación de computación paralela donde el tiempo de creación e inicialización de cada nodo del clúster de trabajo no supere 1 minuto. Y el tiempo para iniciar la ejecución de aplicaciones de computación paralela sea igual o menor a 1*N minutos donde N es el número de nodos del clúster de trabajo.

Probar que es posible la creación de un clúster de trabajo utilizando diferentes imágenes de contenedor de ambientes auto-contenidos.

Descripción: Para este experimento se requiere tener acceso de administrador a un clúster de Kubernetes dónde el operador de Kubernetes que implementa el método haya sido instalado exitosamente. El acceso necesario es de shell, no es necesario el estar físicamente en la misma localización del clúster de Kubernetes. Una vez dentro del clúster de Kubernetes la medición consiste en realizar las siguientes tareas:

1. Clonar el repositorio de código del operador alojado en Github.
2. Crear un nuevo CRD de clúster tomando como referencia el archivo de ejemplo del repositorio.
3. Consultar el tiempo de creación para cada nodo del clúster de trabajo utilizando el comando *watch* de kubectl.
4. Esperar a que el operador cree el clúster de trabajo.

5. Crear un nuevo CRD de ejecución utilizando el archivo de ejemplo del repositorio.
6. Esperar a que el operador inicie la ejecución de la aplicación de computación paralela.
7. Validar que el resultado de la aplicación sea el esperado.

El proceso debe repetirse al menos tres veces y promediar el resultado del tiempo total de creación de los nodos del clúster de trabajo así como la desviación estándar, el resultado del experimento es positivo si el promedio obtenido es menor o igual al número de nodos del clúster de trabajo que quiere decir que el promedio de creación y configuración de cada nodo del clúster de trabajo es igual o menor a 1 minuto.

Para esta medición se deben utilizar al menos 3 implementaciones diferentes de imágenes de ambientes auto-contenidos.

3.4 Medición del rendimiento de entrada/salida de red y almacenamiento utilizando el método propuesto

Esta medición contempla la medición del rendimiento de entrada/salida de red y disco y la recolección de los resultados. Los resultados obtenidos en este experimento, aportan la evidencia necesaria para evaluar el desempeño del método en relación al rendimiento de entrada y salida de datos.

Objetivo: Evaluar el rendimiento de disco y red en el clúster de trabajo, es decir dentro de Kubernetes en relación con el rendimiento de un clúster fuera de Kubernetes que use la misma infraestructura computacional. A este último se le llamará clúster físico.

Descripción: Esta medición requiere medir el rendimiento de escritura y lectura desde y hacia el almacenamiento compartido en cada uno de los nodos del clúster de trabajo, utilizando la herramienta de línea de comandos de Ubuntu dd[3]. La misma medición se deberá efectuar dentro de cada *worker node* hacia el almacenamiento compartido.

Para efectuar la medición del rendimiento de entrada y salida de datos utilizando la red se deberá utilizar el paquete de Ubuntu iperf[4] el cual puede ser instalado desde el administrador de paquetes de Ubuntu APT.

[3]DD permite evaluar el rendimiento de lectura y escritura en un volumen de almacenamiento que se encuentre en un sistema UNIX, permite crear archivos de diferentes tamaños para luego leerlos y proporciona la velocidad alcanzada en *megabytes* por segundo.

[4]Iperf permite enviar y recibir paquetes a travez de la red desde un servidor hacia otro, este proporciona la velocidad alcanzada en *Gigabits* por segundo.

La prueba consiste en transferir datos hacia el primer nodo del clúster de trabajo donde se obtendrá la velocidad entrante desde los demás nodos del clúster de trabajo desde donde se obtendrá la velocidad saliente. La misma medición será efectuada utilizando los *worker nodes* directamente.

Cada medición será ejecutada múltiples veces y el resultado de cada una será promediado a partir de todas sus ejecuciones, además será calculada la desviación estándar para evaluar la consistencia del experimento. Todos los resultados serán tabulados para una mejor lectura e interpretación.

3.5 Medición del rendimiento de la ejecución de aplicaciones utilizando el método propuesto

Esta medición contempla ejecutar un conjunto de *benchmarks* de MPI y la recolección de los resultados. Los resultados obtenidos en este experimento, aportan la evidencia necesaria para evaluar el desempeño del método en diferentes escenarios de ejecución de aplicaciones de computación paralela.

Objetivo: Evaluar el rendimiento de la ejecución de aplicaciones de computación paralela basadas en MPI en el clúster de trabajo, es decir dentro de Kubernetes en relación con el rendimiento de un clúster fuera de Kubernetes que use la misma infraestructura computacional a este último se le llamará clúster físico.

Descripción: Esta medición requiere ingresar a cualquier nodo del clúster de trabajo y un clúster físico de similares características y recolectar los resultados de un conjunto *benchmarks*. Cada *benchmark* será ejecutado múltiples veces, el resultado de cada *benchmarks* será promediado a partir de todas sus ejecuciones, además será calculada la desviación estándar para evaluar la consistencia del experimento.

Una vez obtenidos los resultados de las diferentes pruebas y los promedios, se obtendrá la diferencia porcentual entre el desempeño obtenido en el clúster de trabajo utilizando el método propuesto y el desempeño obtenido en el clúster físico.

Para cada problema serán evaluados su escalamiento fuerte y escalamiento débil en el clúster físico y dentro del clúster de trabajo. Escalamiento fuerte es cuando incrementa el número de procesadores mientras el tamaño del problema permanece constante, la meta es la reducción del tiempo de ejecución [41]. Escalamiento débil es cuando el número de procesadores aumenta conforme aumenta el tamaño del problema, la idea es que para problemas pequeños se utilicen pocos recursos pero para grandes problemas se utilicen muchos más, esto permite una escalabilidad más lineal en comparación al escalamiento fuerte [41].

Las métricas a evaluar serán tiempo de ejecución que es el tiempo total que dura la aplicación en terminar exitosamente. *SpeedUp* que es la relación entre el mejor

tiempo secuencial sobre el tiempo paralelo utilizando N procesadores [52]. eficiencia que es el porcentaje de utilización de cada procesador en el sistema paralelo [52].

La fórmula para el cálculo del *speedUp* en escalamiento fuerte esta dada por la ley de Amdahl, que dice que el *speedUp* está limitado por la fracción serial del programa a ejecutar la cuál no puede ser paralelizable [41]. La fórmula esta dada por

$$SpeedUp = t(1)/t(N)$$

donde t(1) es igual al tiempo requerido para completar el programa con un procesador, es decir de forma serial y t(N) es igual al tiempo requerido para completar el programa utilizando N procesadores. La eficiencia en escalamiento fuerte esta dada por

$$Efficiency = t(1)/t(N) * N$$

donde t(1) es igual al tiempo requerido para completar el programa con un procesador, es decir de forma serial y t(N) es igual al tiempo requerido para completar el programa utilizando N procesadores y N es el numero de procesadores utilizados.

En escalamiento débil la métrica que nos concierne es la de eficiencia conforme aumenta el número de procesadores y tamaño del problema. La fórmula esta dada por

$$Efficiency = t(1)/t(N)$$

donde t(1) es igual al tiempo requerido para completar el programa con un procesador con un tamaño de problema p, es decir de forma serial y t(N) es igual al tiempo requerido para completar el programa utilizando N procesadores con un tamaño de problema p.

Para realizar de la medición serán utilizados diferentes *benchmarks* de Mantevo. Las *miniapps* de Mantevo permiten medir el rendimiento de sistemas de computación paralela en diferentes dominios aplicables a usos de la vida real [29]. Los *benchmarks* de Matevo que serán evaluados en esta investigación son los siguientes:

MiniFE: Permite aproximar el rendimiento de aplicaciones de ingeniería que requirieren la solución implícita de un sistema no lineal de ecuaciones, donde la mayor parte del tiempo computacional se utiliza en alguna variación del método del gradiente conjugado [29]. MiniFE es una *miniapp* que imita la generación de elementos finitos, su dominio físico es una caja 3D con dimensiones configurables [29].

MiniMD: Es una *miniapp* de la aplicación de dinámica molecular LAMMPS [29]. MiniMD usa el algoritmo de descomposición espacial, donde cada procesador de un clúster procesa un subconjunto de la simulación y permite especificar diferentes parámetros como el tamaño del problema, la densidad de átomos, temperatura, tamaño de los saltos de tiempo, número de saltos de tiempo y la distancia del corte de la interacción de partículas [29]. Debido a que es una *miniapp* solo un tipo de interacción esta disponible la de Lennard-Jones [29].

MiniXyce: Es una *miniapp* de la aplicación de simulación de circuitos Xyce, la simulación de circuitos es la parte más importante en la industria de la automatización de diseños eléctricos [29]. MiniXyce esta basada en una formulación de análisis nodal modificado, que resulta en que las leyes de corrientes de Kirchoff se apliquen a través de una red potencialmente arbitraria lo que resulta en un sistema de ecuaciones algebraicas diferenciales que se resuelve implícitamente utilizando métodos basados en Newton [29].

CloverLeaf: Es una *miniapp* que resuelve las ecuaciones de Euler comprimibles en una cuadrícula cartesiana, utilizan un método explícito y preciso de segundo orden, donde cada celda guarda tres valores: energía, densidad y presión. [17]. CloverLeaf resuelve las ecuaciones en dos dimensiones, y también existe una implementación para resolverlas en tres dimensiones [17]. La implementación utilizada en esta investigación es la que resuelve las ecuaciones en dos dimensiones.

El resultado de esta medición es positivo si los resultados promediados en el clúster de trabajo son de al menos un 70% con respecto a los resultados promediados en el clúster físico. Todos los resultados serán tabulados y se elaboraran gráficos para las métricas de *speedUp* y eficiencia para una mejor lectura e interpretación.

3.6 Medición del rendimiento de entrenamiento de modelos de I.A utilizando el método propuesto

Esta medición contempla ejecutar un conjunto de *benchmarks* de entrenamiento de modelos de A.I utilizando aceleradores gráficos y la recolección de los resultados. Los resultados obtenidos en este experimento, aportan la evidencia necesaria para evaluar el desempeño del método para el entrenamiento de modelos de A.I.

Objetivo: Evaluar el rendimiento al entrenar modelos de inteligencia artificial utilizando un nodo o más de un nodo por medio de MPI en el clúster de trabajo, es decir dentro de Kubernetes en relación con el rendimiento de un clúster fuera de Kubernetes que use la misma infraestructura computacional a este último se le llamará clúster físico.

Descripción: Esta medición requiere ingresar a cualquier nodo del clúster de trabajo y un clúster físico de similares características y recolectar los resultados de un conjunto *benchmarks*. Cada *benchmark* será ejecutado múltiples veces, el resultado de cada *benchmarks* será promediado a partir de todas sus ejecuciones, además será calculada la desviación estándar para evaluar la consistencia del experimento. Los nodos del clúster físico deben contar con aceleradores gráficos para su uso en el entrenamiento de los modelos. Así mismo, el clúster de trabajo deberá utilizar los mismos aceleradores gráficos para el entrenamiento de los modelos.

Una vez obtenidos los resultados de las diferentes pruebas y los promedios, se obtendrá la diferencia porcentual entre el desempeño obtenido en el clúster de trabajo utilizando el método propuesto y el desempeño obtenido en el clúster físico. Para realizar la medición serán utilizados los *benchmarks* de *AI Benchmark Alpha* y las métricas a recolectar serán el tiempo total de ejecución y la puntuación final de entrenamiento para el dispositivo. Esta medición será ejecutada en un único nodo del clúster físico y el clúster de trabajo.

Para realizar la medición del rendimiento de entrenamiento de los modelos de A.I utilizando más de un nodo se utilizará el modelo ResNet-50 utilizando el conjunto de datos de Cifar-10[5] y MPI para la comunicación entre nodos. Las métricas a estudiar en este caso serán número de muestras por segundo procesadas y tiempo total de ejecución para un *accuracy*[6] objetivo de al menos el 70%.

El resultado de esta medición es positivo si los resultados promediados en el clúster de trabajo son de al menos un 70% con respecto a los resultados promediados en el clúster físico. Todos los resultados serán tabulados para una mejor lectura e interpretación.

[5]El conjunto de datos de Cifar-10 consiste en 60000 imágenes de color de 32x32 divididas en 10 clases cada uno con 6000 imágenes, fue publicado en 2009 por Alex Krizhevsky de la Universidad de Toronto. [47]

[6]La métrica de textitaccuracy en I.A se refiere a la proporción de predicciones correctas realizadas por un modelo de I.A con respecto al número total de predicciones. Se utiliza generalmente para evaluar el proceso de entrenamiento de los modelos.

Capítulo 4

Diseño del Método

La presente sección describe en detalle el diseño del método. Se describen los tres actores que participan en las diferentes etapas del método. Luego la figura 4.1 presenta un diagrama de alto nivel con las etapas e interacción entre los diferentes componentes. Se explica cada etapa y sus respectivos flujos y componentes en forma detallada.

Administrador del Clúster: Es la persona que se encarga de administrar el clúster de Kubernetes. Tiene conocimientos técnicos suficientes para instalar nuevos componentes y resolver problemas del clúster en caso de ser necesario.

Creador ambiente auto contenido: Es el encargado de crear la imagen de contenedor con el ambiente auto-contenido, tiene conocimientos en contenedores y el dominio para el cual va a ser utilizado el ambiente auto contenido.

Usuario del Clúster: Es quien desea utilizar los recursos de un clúster de trabajo para la ejecución de una aplicación de computación paralela basada en MPI. Es el encargado de ejecutar la aplicación de computación paralela en el clúster de trabajo. Además verifica los resultados una vez la ejecución finaliza.

Para el manejo de todo lo pertinente a la implementación del software, el cual incluye documentación técnica, código, y distribuciones del *software*, se utilizará un repositorio de código creado en la plataforma Github[1].

[1]El repositorio de Github puede ser clonado y accedido en la siguiente dirección: https://github.com/Crisarias/kubernetes-mpi-operator .

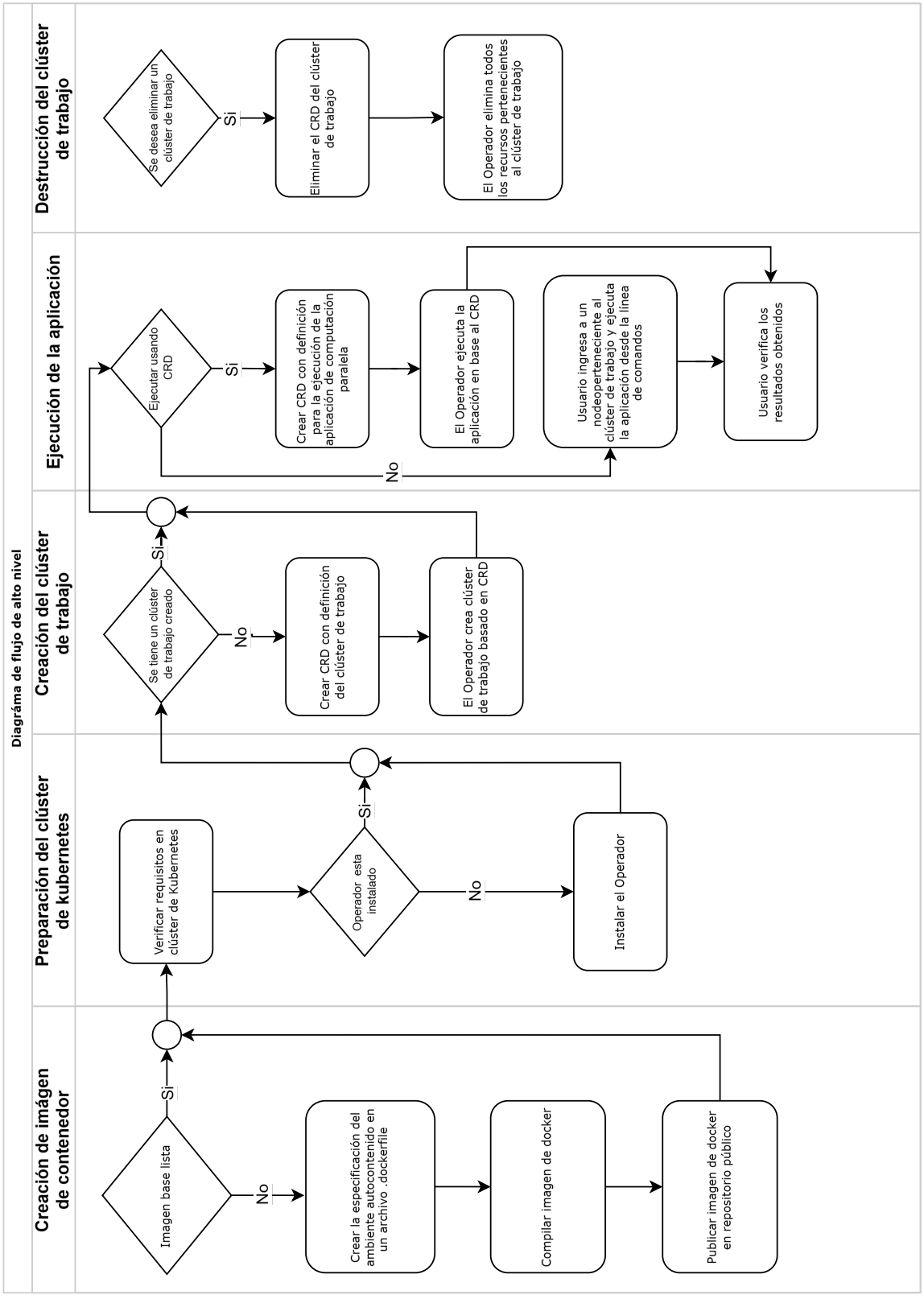

Figura 4.1: Diagrama de flujo de alto nivel

4.1 Creación de imagen de contenedor

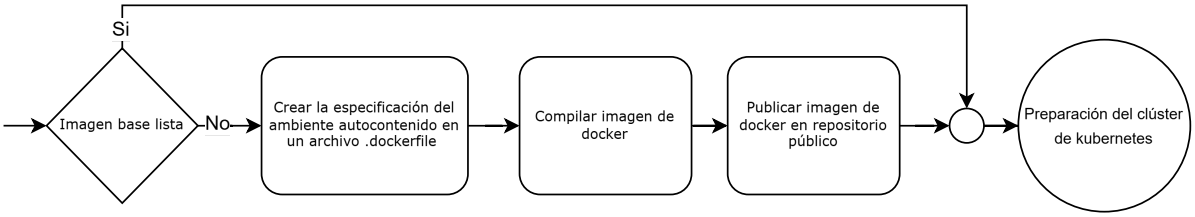

Figura 4.2: Etapa 1 del método

Consiste en diseñar la imagen de contenedor que sirva para crear contenedores con un ambiente auto-contenido y listo para la ejecución de aplicaciones de computación paralela basadas en MPI.

Esta imagen se utilizará como base para crear los nodos del clúster de trabajo y es una de las partes esenciales para que el método funcione. La imagen tendrá como base una distribución de Linux, para esta investigación se utilizará Debian con la imagen oficial de Ubuntu 22.04.

La sintaxis a utilizar será la de imágenes de Docker, en un archivo con extensión dockerfile. Durante la creación de la imagen se deben instalar todas las herramientas y librerías necesarias para la ejecución de las aplicaciones de computación paralela basadas en MPI. Además se deben instalar las herramientas y librerías para configurar y probar que todos los nodos puedan conectarse por medio de SSH. Es esperado que se generen múltiples imágenes de contenedor para diferentes casos de uso. Sin embargo, todas estas imágenes tendrán en común una serie de pasos necesarios para la correcta creación del ambiente auto-contenido los cuales son explicados a continuación.

4.1.1 Instalación de librerías

Este paso consiste en instalar las librerías necesarias para el correcto funcionamiento del ambiente auto-contenido y la comunicación entre nodos una vez creado el clúster. Se instalarán librerías que faciliten la configuración de los nodos ya sea al momento de la creación del clúster o al momento en que un nodo deba reiniciarse en caso de fallo. Además se instalarán librerías para validar que todos los nodos puedan comunicarse entre sí una vez finalizada la configuración del clúster de trabajo. Las librerías se instalarán por medio del manejador de paquetes de Ubuntu apt[2] cuando la imagen es compilada.

Las librerías y herramientas a instalar en todas la imágenes de ambientes auto-contenidos serán:

[2]El manejador de paquetes de Ubuntu apt es un sistema fácil de usar para instalar, actualizar,configurar y remover *software*. Además provee acceso a una base de datos con más de 60000 paquetes para Ubuntu [49].

Git: Necesaria para trabajar con repositorios de código alojados en un servidor de Git. Permitirá al usuario del clúster de trabajo clonar dentro de cualquier nodo del clúster de trabajo repositorios de código que contengan posibles aplicaciones a ejecutar.

Curl: Necesaria para la descarga de archivos. Curl puede ser utilizado en la descarga de archivos que sirven para la instalación de paquetes, para agregar nuevos repositorios al manejador de paquetes apt o cualquier archivo necesario para la ejecución de las aplicaciones dentro del clúster de trabajo.

Vim: Será utilizada para la edición de archivos de texto ya sea de configuración o código dentro de cada Nodo.

Dnsutils: Esta librería instala dig[3]. Dig puede ser utilizado por los *scripts* de configuración para el descubrimiento de las Ips de los diferentes nodos y posterior configuración del archivo de *hosts*.

Opessh-server: El servidor de SSH, permite realizar conexiones desde un cliente SSH hasta la máquina donde este instalado. La configuración debe ser ajustada para permitir autenticación sin contraseña desde clientes autorizados.

Opessh-client: El cliente de SSH, permite realizar conexiones hacia un servidor SSH desde la máquina donde este instalado. La configuración debe ser ajustada para permitir autenticación sin contraseña hacia servidores autorizados.

Además, de las librerías citadas que son necesarias en todas la imágenes de ambientes auto-contenidos. Cada imagen de contenedor de manera individual requerirá la instalación de librerías y herramientas necesarias para la compilación y ejecución de los programas de computación paralela que se deseen ejecutar dependiendo del dominio específico de estas aplicaciones. Por ejemplo para la ejecución de aplicaciones basadas en MPI, es necesario como mínimo la instalación de los compiladores de C o Fortran, más alguna de las distribuciones que implementan MPI como MPICH o OpenMPI.

4.1.2 Configuración para la comunicación entre nodos

Cada imagen de contenedor debe poseer los medios necesarios para configurar la comunicación entre nodos de manera autónoma una vez estos hayan sido creados. Para esto se crearán una serie de *scripts* que se ejecutan cada vez que inician los contenedores, es decir cada que inicia un nodo. Cada script será copiado en tiempo de compilación de la imagen a la ruta /bin/ y se asignarán permisos para que puedan ser ejecutados.

El diseño y el orden de ejecución de los *scripts* se basa en el hecho de que los nodos pertenecientes al *statefulset* son creados uno a la vez. Y una vez creado el último nodo

[3]Dig también conocido como *Domain Information Groper* es una herramienta línea de comandos para la búsqueda en servidores DNS, proporciona información relacionada a un servidor DNS y la salida puede ser customizada a necesidades específicas.

quiere decir que todos los demás nodos del clúster de trabajo han sido creados e inicializados satisfactoriamente.

Para el correcto funcionamiento de los *scripts* de configuración a nivel del clúster de trabajo, es esperado que todos los nodos tengan acceso a un almacenamiento compartido que servirá para la escritura y lectura de archivos desde y hacia cualquier nodo. A continuación se listan los *scripts* de configuración en el orden en que serán ejecutados y su objetivo:

***Script* para la creación del archivo de *hosts* para MPI:** Crea un archivo en el almacenamiento compartido que contendrá la lista de todos los nodos que pueden ser utilizados para la ejecución de aplicaciones basadas en MPI. Además del nombre de cada nodo, el archivo también especifica la cantidad de procesadores o *Ranks* que pueden ser utilizados para MPI en cada nodo.

***Script* para la creación de la llave pública y privada para SSH:** Crea un par de llaves pública y privada por medio de línea de comandos utilizando encriptación rsa y sin contraseña. Una vez creadas las llaves estas son guardadas en el almacenamiento compartido en una carpeta llamada `keys`, este paso de la creación del par de llaves se ejecuta una única vez por clúster de trabajo.

Luego las llaves son copiadas en cada nodo en la ruta por defecto generalmente `~/.ssh`. La llave pública debe agregarse al archivo de llaves autorizadas *authorized_keys* en la ruta `~/.ssh`, si este no existe será creado. Tanto el archivo de llaves autorizadas, y el par de llaves se les proporcionarán los permisos necesarios para ser utilizados luego en cada conexión SSH requerida.

Script* para actualizar el archivo de *known_hosts : Para que dos nodos se conecten de forma exitosa por medio de SSH. Se tienen que cumplir dos condiciones. La primera la llave pública del nodo cliente esta autorizada en el nodo servidor, el script de creación de llave pública y privada se encarga de esto. La segunda el nodo cliente debe autorizar la conexión hacia el nodo servidor.

Para esto es necesario agregar la firma de la llave junto al servidor autorizado en el archivo *known_host* ubicado en la ruta por defecto `~/.ssh`. Al todos los nodos dentro de un mismo clúster de trabajo poseer la misma copia de llave pública y privada es posible que un nodo se conecte hacia si mismo para obtener la firma de la llave que puede ser utilizada para todos los nodos sin de un mismo clúster de trabajo. Con la firma disponible al ser los nodos parte de un *statefulset* es posible escribir en el archivo *known_host* todas las líneas requeridas de firma-servidor en cada uno de los nodos.

***Script* para actualizar el archivo de *hosts* del sistema:** El *statefulset* nos asegura que cada uno de los nodos tendrá un nombre o DNS estable. Sin embargo, lo mismo no puede asegurarse para las direcciones IP de cada nodo. Estas pueden cambiar si un nodo es recreado al ser reiniciado o destruido.

Para que todos los nodos puedan conocer las IPs de otros nodos es necesario que el archivo de *hosts* del sistema se encuentre debidamente actualizado. Este *script* se ejecuta en el último nodo del statefulset una vez todos los demás nodos han sido configurados. Utilizando la herramienta de línea de comandos *Domain Information Groper* descubre cuales son las IPs de los demás nodos en ese momento y crea un archivos de *hosts* el cual es copiado a todos los nodos por medio de SSH.

En caso de que un nodo se destruya y vuelva a recrearse, este *script* consulta por medio de la herramienta de línea de comandos *Domain Information Groper* la IP del último nodo del clúster de trabajo. Si logra obtener la IP quiere decir que en efecto el nodo fue destruido y recreado. Por medio de SSH se ejecuta el *script* en el último nodo el cual recrea el archivo de *hosts* con la nueva IP del nodo recreado y copia el archivo actualizado a todos los nodos del clúster de trabajo.

***Script* para iniciar el servidor de SSH:** Este script inicializará el servidor de SSH. Debe ser ejecutado una vez la configuración de los nodos para una comunicación exitosa haya sido finalizada.

***Script* para realizar la prueba de conexión:** Es utilizado por Kubernetes para realizar la prueba de inicio de cada contenedor. El pod donde se encuentra el nodo no cambiará su estado a listo hasta que el *script* haya sido ejecutado y no retorne un código de error. El *script* valida que el nodo pueda conectarse exitosamente por medio de SSH hacia el mismo de manera desatendida o hacia otros nodos en caso de ser el último nodo del clúster de trabajo.

En el capítulo de implementación se describe la implementación de tres imágenes de contenedores para la creación de diferentes ambientes auto-contenidos.

4.2 Preparación del clúster de Kubernetes

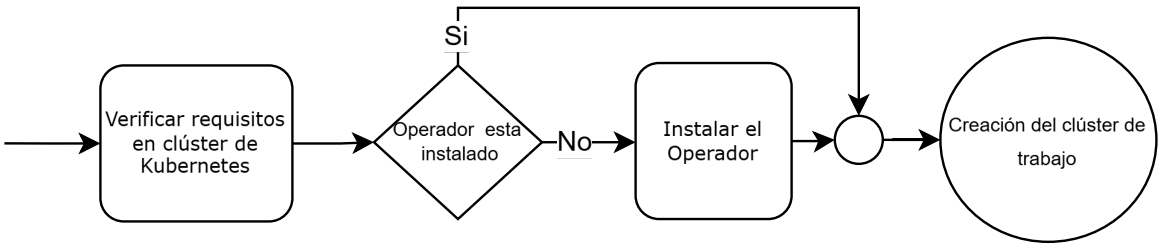

Figura 4.3: Etapa 2 del método

Consiste en verificar que los requisitos para la implementación del método se cumplan. Como mínimo se debe tener un clúster de Kubernetes debidamente configurado con al menos dos *worker nodes* y un *controlplane node*. El método esta diseñando de manera que los nodos del clúster de trabajo sean creados en los *worker nodes*. Lo ideal es que cada

nodo del clúster de trabajo consuma en su totalidad los recursos computacionales de un *worker node*. Es decir, la relación entre *worker nodes* y nodos del clúster de trabajo es uno a uno, pueden existir más *worker nodes* que nodos del clúster de trabajo pero no al contrario.

De acuerdo a lo mencionado anteriormente, los *worker nodes* que van a hacer utilizados para la creación de ambientes para la ejecución de tareas de computación paralela deben estar libres. Ninguna otra carga de trabajo más que las esenciales para su funcionamiento deben ser creadas en estos antes de la creación del clúster de trabajo. Además se requiere que los *worker nodes* sean máquinas similares en cuando a arquitectura, procesador, GPU, memoria RAM y disco.

Cada *worker node* deberá tener acceso a un volumen compartido de almacenamiento, mismo que va a ser utilizado por los diferentes nodos del clúster de trabajo. De acuerdo a la estrategia presentada en el capítulo de Metodología este almacenamiento compartido va a ser creado utilizando BeeGFS. BeeGFS es un almacenamiento paralelo que combina las capacidades de varios nodos para formar un solo espacio común que puede ser utilizado por diferentes clientes.

Cada *worker node* será un cliente de BeeGFS. Por su parte un servidor dedicado será el servidor de administración, meta datos y almacenamiento de BeeGFS. Por defecto el cliente de BeeGFS monta el almacenamiento en la ruta `/mnt/beegfs`.

Esta ruta será utilizada para la creación de un volumen de Kubernetes disponible dentro de cada nodo del clúster de trabajo en la ruta `/nfs`. Una vez creado el volumen de Kubernetes este creará una nueva carpeta en la ruta `/mnt/beegfs`. Por lo que cada vez que se escriban o lean datos en la ruta `/nfs` en cualquiera de los nodos del clúster de trabajo, en realidad estos se están escribiendo o leyendo en la ruta `/mnt/beegfs/nombre_de_volumen` del *worker node* donde se aloja el pod que contiene ese nodo del clúster de trabajo.

4.2.1 CRDs del Operador de Kubernetes

El operador de Kubernetes será el encargado de crear todos los recursos necesarios dentro del clúster de Kubernetes para la creación de uno o más clúster de trabajo para la ejecución de aplicaciones de computación paralela. El operador de Kubernetes esta conformado por uno o más CRDs y sus controladores. Este método contendrá dos CRDs. El primero que representa un clúster de trabajo y el segundo que representa una solicitud para ejecutar una aplicación de computación paralela utilizando los recursos del clúster de trabajo.

Las propiedades que contendrá el CRD del clúster de trabajo son las siguientes:

- *ClusterName*: El nombre del clúster de trabajo. Es único a través del clúster de Kubernetes. Es decir si otra persona intenta crear un clúster de trabajo con el mismo nombre, el operador no lo creará y imprimirá en consola que el clúster ya a sido creado.

- *ClusterNodeImage*: La imagen base para la creación de los nodos. Debe ser el nombre completo de la imagen que se va a utilizar como la base para el ambiente auto-contenido. El nombre completo lo forman la dirección del repositorio donde se guardo la imagen, el nombre de la imagen y la versión deseada. Por ejemplo si la imagen fue guardada en el repositorio de imágenes *Docker Hub* el nombre completo será `docker.io/username/base-image-name:version`. Si la versión no se especifica la última versión de la imagen generalmente *latest* será utilizada.

- *ClusterNodesCount*: Número entero que representa la cantidad de nodos que conforman el clúster de trabajo. No puede exceder el número de *worker nodes* disponibles. Es posible definir un mínimo y un máximo que aplique a todos las instancias de CRDs de este tipo.

- *CoresPerNode*: Número entero que representa la cantidad de procesadores disponibles para cada nodo del clúster de trabajo. Se recomienda que esta cantidad sea igual al número de procesadores lógicos disponibles en un *worker node* menos 1. El razonamiento detrás de esta recomendación es dejar cierta cantidad de recursos disponibles para el funcionamiento del sistema operativo y sus servicios.

- *MemoryPerNode*: Número entero que representa la cantidad de memoria volátil RAM disponible para cada nodo del clúster de trabajo. Se recomienda que esta cantidad sea igual al total de la memoria RAM disponibles en un *worker node* menos 2 *gigabytes*. El razonamiento detrás de esta recomendación es dejar cierta cantidad de recursos disponibles para el funcionamiento del sistema operativo y sus servicios.

- *SharedMemoryPerNode*: Número entero que representa la cantidad de memoria volátil RAM disponible para el segmento de memoria compartida *shm* para cada nodo del clúster de trabajo. Esta memoria es utilizada para el intercambio de información entre procesos por medio del API de memoria compartida de POSIX. Es opcional ya que por defecto su valor es 64.

- *SshPort*: El puerto interno para la comunicación utilizando SSH. Es el puerto utilizado por el servidor de SSH dentro de los nodos del clúster de trabajo. Es opcional ya que por defecto este puerto es el 22.

- *LocalVolumePath*: La ruta de la carpeta de almacenamiento compartida en los *worker nodes*. Es la ruta donde se creará el volumen de Kubernetes a utilizar por los nodos del clúster de trabajo.

- *LocalVolumeCapacity*: Número entero que representa la capacidad de almacenamiento del volumen de Kubernetes creado para el clúster de trabajo. La unidad de medida es en *gigabytes*. Es posible definir un mínimo y un máximo que aplique a todas las instancias de CRDs de este tipo.

- *EnableNvidiaGPU*: Valor booleano. Si el *worker node* posee una GPU este parámetro habilita la posibilidad de utilizar esa tarjeta gráfica dentro del clúster de trabajo. Es opcional ya que por defecto esta deshabilitado.

- *NumGPUs*: Número entero que representa la cantidad de GPUs que tendrá cada nodo del clúster de trabajo. No puede exceder el número disponible de GPUs en los *worker nodes*. Para ser utilizado es necesario que el parámetro *EnableNvidiaGPU* tenga un valor verdadero. Es opcional ya que por defecto su valor es 1.

- *NameGPUs*: El nombre de la GPU en el *worker node*. Para ser utilizado es necesario que el parámetro *EnableNvidiaGPU* tenga un valor verdadero.

El anexo 1(8.1) muestra la definición de un archivo YAML para la creación de una instancia del CRD de tipo clúster que contiene las propiedades detalladas en esta sección.

Las propiedades que contendrá el CRD para la ejecución de una aplicación de computación paralela son las siguientes:

- *JobName*: Un identificador único para la solicitud de ejecución.

- *ClusterName*: El nombre del clúster de trabajo donde se desea ejecutar la aplicación de computación paralela.

- *Command*: El comando a ejecutar con los argumentos necesarios. Pueden ser varios comandos mientras sean en una solo línea. Este comando será ejecutado en una terminal de *bash* de forma desatendida en el primer nodo del clúster de trabajo.

El anexo 2(8.2) muestra la definición de un archivo YAML para la creación de una instancia del CRD de tipo *job* que contiene las propiedades detalladas en esta sección.

Ambos CRDs contendrán un estado. El estado de un CRD se puede ver como propiedades las cuales pueden ser actualizadas por el operador y luego ser consultadas por medio del API de Kubernetes. Estas propiedades indican el estado actual de la instancia del CRD.

Las propiedades de estado que contendrá el CRD del clúster de trabajo son las siguientes:

- *Ready*: Es un booleano que indicará si el clúster de trabajo a sido creado exitosamente. Si hubo algún fallo la propiedad tendrá un valor falso.

- *ClusterStoragePath*: La ruta dentro de la carpeta compartida del *worker node* donde se almacenan los datos del clúster de trabajo. De esta manera es posible identificar la ruta de almacenamiento para cada clúster de trabajo después de su creación.

Las propiedades de estado que contendrá el CRD para la ejecución de una aplicación de computación paralela son las siguientes:

- *Result*: El resultado de la ejecución. Contendrá el valor *Success* en caso de que la ejecución del programa de computación paralela haya sido exitoso o *Fail* en caso de que la ejecución no se haya podido realizar satisfactoriamente.

- *Stderr*: La salida de error de la consola al ejecutar la aplicación. Este valor se mantendrá vacío en caso de que no hayan errores o el usuario especifique un archivo de salida en el comando de ejecución.
- *Stdout*: La salida de la consola al ejecutar la aplicación. Este valor se mantendrá vacío en caso de que el usuario especifique un archivo de salida en el comando de ejecución.
- *ExecutionFolder*: La ruta dentro de la carpeta compartida del *worker node* donde se almacenan los datos de la ejecución. De esta manera es posible identificar la ruta de almacenamiento para cada ejecución y consultar los resultados una vez finalizada.

4.2.2 Instalación del Operador de Kubernetes

El operador de Kubernetes será implementado utilizando *Operator SDK* y el lenguaje de programación GO. La manera de distribuir, instalar y administrar operadores creados con este *framework* es por medio del *Operator Lifecycle Manager* el cual fue introducido en la sección 2.3.1 del marco teórico.

4.3 Creación del clúster de trabajo

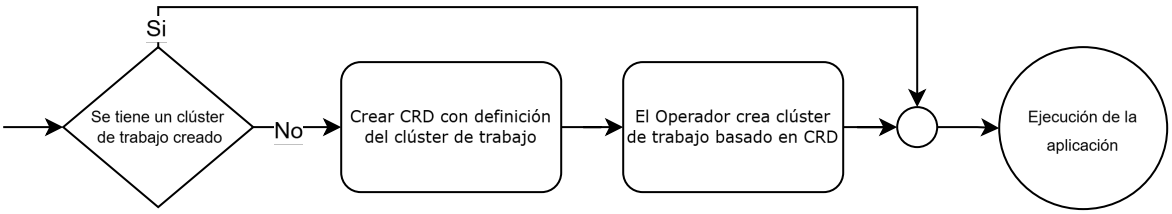

Figura 4.4: Etapa 3 del método

Consiste en la creación del clúster de trabajo dentro del clúster de Kubernetes, el cual será utilizado para la ejecución de las aplicaciones de computación paralela.

Se creará un archivo con extensión .YAML con la especificación del clúster de trabajo. Este archivo sera utilizado para crear una nueva instancia de un CRD de tipo clúster de trabajo. Una vez el operador identifique que existe un nuevo CRD de tipo clúster de trabajo, su controlador ejecutará las siguientes tareas en el clúster de Kubernetes por medio del ciclo de reconciliación:

4.3.1 Creación del espacio de nombres

Se debe crear un espacio de nombres único que contendrá todos los recursos asociados al clúster de trabajo. Esto permitirá que los recursos asignados al espacio de nombres sean

utilizados únicamente por el clúster de trabajo. El nombre del espacio de nombres será a1-ns donde a1 es el nombre del clúster de trabajo y ns es un sufijo indicando que es un recurso de tipo *namespace*. Si el espacio de nombres ya existe, el operador escribirá un mensaje en el *log* del controlador y el ciclo de reconciliación terminará.

4.3.2 Creación del servicio *headless*

Una vez creado el espacio de nombres con éxito. Se debe crear un servicio *headless* para el descubrimiento de los nodos a través de la red interna del clúster de trabajo. Este servicio permite que se le asigne una dirección IP a cada nodo del clúster de trabajo. El nombre del servicio será a1-headless-svc donde a1 es el nombre del clúster de trabajo y headless-svc es un sufijo indicando que es un recurso de tipo *service*. Si el servicio *headless* ya existe, el operador escribirá un mensaje en el *log* del controlador y el ciclo de reconciliación terminará.

4.3.3 Creación del volumen de almacenamiento compartido

Una vez creado el servicio *headless* con éxito. Se debe crear un volumen de almacenamiento que pueda ser accedido desde cualquier nodo perteneciente al clúster de trabajo y utilice el espacio de almacenamiento compartido creado anteriormente en los *worker nodes*. En la imagen 4.5 se presenta gráficamente la arquitectura propuesta para el almacenamiento del clúster de trabajo.

Es esperado que los *worker nodes* sean los que contengan todos los servicios y mecanismos necesarios para la correcta replicación y uso del almacenamiento. Esto facilita la creación del volumen para el operador y reduce la complejidad del método. Para esto el operador creará tres recursos que se detallan a continuación.

Una clase de almacenamiento, esta debe tener un *VolumeBindingMode=Inmmediate* y un *Provisioner=kubernetes.io/no-provisioner* lo que quiere decir que el volumen estará listo para utilizarse apenas es creado y no es necesaria ninguna interfaz de almacenamiento de contenedores (CSI) para su creación. El nombre de la clase de almacenamiento será a1-sc donde a1 es el nombre del clúster de trabajo y sc es un sufijo indicando que es un recurso de tipo *storageClass*. Si la clase de almacenamiento ya existe, el operador escribirá un mensaje en el *log* del controlador y el ciclo de reconciliación terminará.

Una vez creada la clase de almacenamiento exitosamente. Se creará un volumen persistente que hace referencia a la clase de almacenamiento creada anteriormente. El modo de acceso debe ser *ReadWriteMany* lo que permite que varios nodos del clúster de trabajo puedan escribir y leer del volumen al mismo tiempo. También debe especificarse la capacidad en *gigabytes*, este valor se extrae de los metadatos del CRD de tipo clúster.

El volumen persistente debe contener una fuente de tipo *hostPath* donde se especifica la ruta destino en el *worker node* la cual se extrae de los metadatos del CRD de tipo

clúster. El nombre del volumen persistente será a1-pv donde a1 es el nombre del clúster de trabajo y pv es un sufijo indicando que es un recurso de tipo *pesistentVolume*. Si el volumen persistente ya existe, el operador escribirá un mensaje en el *log* del controlador y el ciclo de reconciliación terminará.

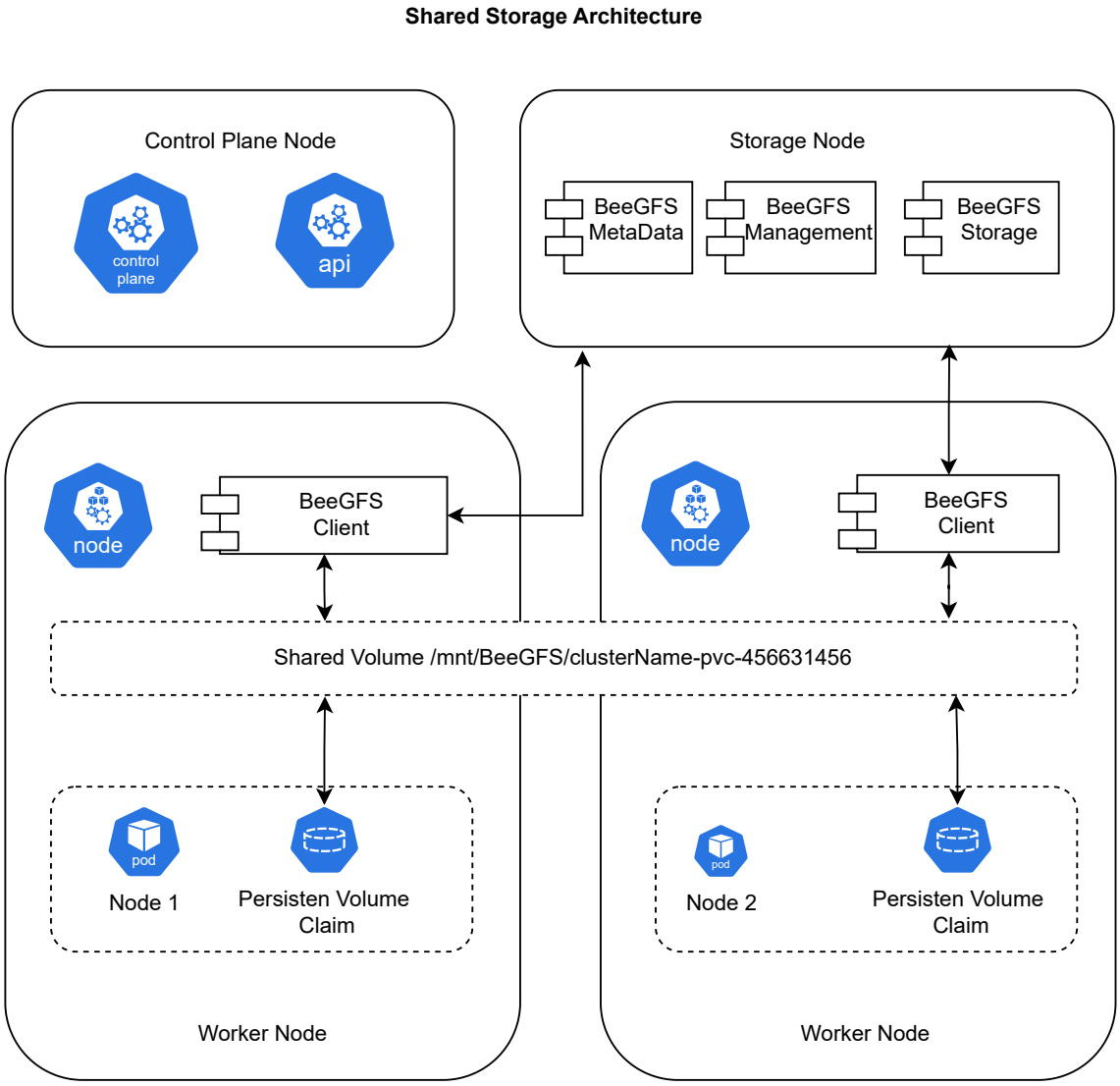

Figura 4.5: Almacenamiento compartido para el clúster de trabajo

Al crearse el volumen persistente se creará una carpeta en el almacenamiento compartido del *worker node* con el nombre a1-pv-a2 donde a1 es el nombre del clúster de trabajo y a2 es el tiempo en formato UNIX al momento de creación. Esta ruta será guardada en la propiedad *ClusterStoragePath* de la instancia del CRD de tipo clúster.

Después de la creación exitosa del volumen persistente. Se debe crear un *claim* de volumen persistente el cual hace referencia a la clase de almacenamiento y al volumen persistente creados con anterioridad. Al igual que el volumen persistente el *claim* de volumen persistente debe tener un modo de acceso debe ser *ReadWriteMany*. Y la cantidad

de almacenamiento requerido debe ser igual a la cantidad especificada en los metadatos del CRD de tipo clúster.

El nombre del *claim* de volumen persistente será a1-pvc donde a1 es el nombre del clúster de trabajo y pvc es un sufijo indicando que es un recurso de tipo *pesistentVolumeClaim*. Si el *claim* de volumen persistente ya existe, el operador escribirá un mensaje en el *log* del controlador y el ciclo de reconciliación terminará.

4.3.4 Creación del *statefulset*

Una vez el *claim* de volumen persistente es creado exitosamente. Se debe crear un *statefulset* que haga referencia al servicio *headless* y al *claim* de volumen persistente creados previamente. El *statefulset* es el recurso que une todos los demás para crear el clúster de trabajo. El nombre del *statefulset* será a1-sts donde a1 es el nombre del clúster de trabajo y sts es un sufijo indicando que es un recurso de tipo *statefulset*. Si el *statefulset* ya existe, el operador escribirá un mensaje en el *log* del controlador y el ciclo de reconciliación terminará.

El número de replicas del *statefulset* es igual al número de nodos que tendrá el clúster de trabajo. Su valor es tomado de la propiedad *ClusterNodesCount* existente en los metadatos de la instancia del CRD de clúster.

La especificación del contenedor del *statefulset* deberá contener las siguientes variables de entorno las cuales serán utilizadas como parámetros de entrada en los *scripts* de configuración que contiene cada imagen de ambiente auto-contenido.

- *CLUSTERNODESCOUNT*: Especifica la cantidad de nodos que tendrá el clúster de trabajo. Su valor es tomado de la propiedad *ClusterNodesCount* existente en los metadatos de la instancia del CRD de clúster.
- *CORESPERNODE*: Especifica la cantidad de procesadores lógicos que estarán disponibles en cada nodo del clúster de trabajo. Su valor es tomado de la propiedad *CoresPerNode* existente en los metadatos de la instancia del CRD de clúster.
- *CLUSTERNAME*: Especifica el nombre del clúster de trabajo. Su valor es tomado de la propiedad *ClusterName* existente en los metadatos de la instancia del CRD de clúster.

El valor de la imagen del *statefulset* es tomado de la propiedad *ClusterNodeImage* existente en los metadatos de la instancia del CRD de clúster. Todos los pods que son los nodos del clúster de trabajo serán creados a partir de esta imagen de contenedor.

Cada pod será creado en un *worker node* diferente, para cumplir esta premisa se especificarán limites de cpu y memoria utilizando los valores especificados en las propiedades *CoresPerNode* y *MemoryPerNode* existentes en los metadatos de la instancia del CRD de clúster.

La especificación del contenedor también tendrá la definición del *StartupProbe* esta propiedad permite especificar un prueba mediante la cual el clúster de Kubernetes determina si un pod está listo para ser utilizado o debe ser recreado. La prueba a ejecutar será la ejecución del *script* copiado dentro de la imagen de contenedor en tiempo de compilación el cual valida que el nodo puede conectase por medio de SSH sin problemas.

4.4 Ejecución de la aplicación

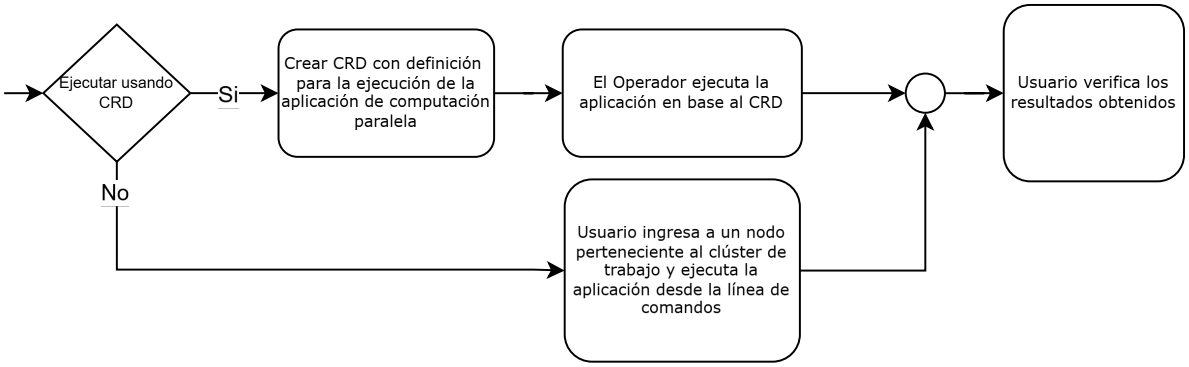

Figura 4.6: Etapa 4 del método

Consiste en utilizar los recursos del clúster de trabajo creado anteriormente, para la ejecución de aplicaciones de computación paralela basadas en MPI. La ejecución se llevará a cabo por medio del operador o mediante una sesión interactiva.

La ejecución por medio del operador, requiere la creación de un archivo con extensión .YAML con la especificación de la aplicación de computación paralela. La especificación contendrá el comando de terminal que se desea ejecutar. Este archivo sera utilizado para crear una nueva instancia de un CRD de tipo aplicación.

Una vez el operador identifique que existe un nuevo CRD de tipo *job*, ejecutará la aplicación en uno, varios o todos los nodos pertenecientes al clúster de acuerdo al comando de ejecución especificado anteriormente en el CRD.

Los resultados serán pueden guardarse en uno o más archivos dentro de la carpeta de ejecución. Esto requiere que le programa y el comando de ejecución lo soporten. De esta forma el usuario podrá disponer de los resultados como requiera.

La ejecución por medio de una sesión interactiva, requiere proporcionar un medio, por el cual un usuario pueda acceder a una terminal en cualquier nodo perteneciente al clúster de trabajo.

Una vez dentro de la terminal el usuario dispondrá de los permisos necesarios para descargar los binarios de su aplicación y ejecutar su aplicación de computación paralela en uno, varios o todos los nodos pertenecientes al clúster de trabajo.

Los resultados serán desplegados en la misma terminal de la sesión interactiva. De esta forma el usuario podrá disponer de los resultados como requiera.

4.5 Destrucción del clúster de trabajo

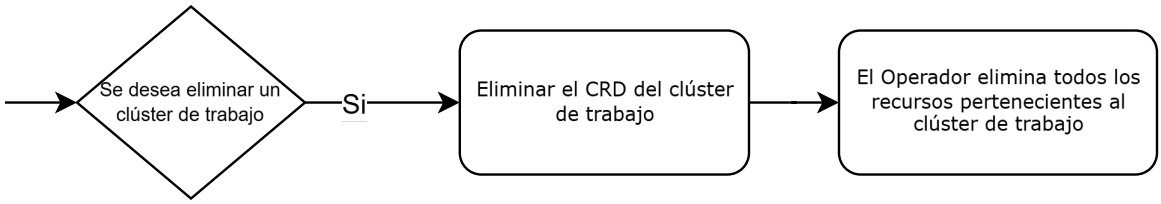

Figura 4.7: Etapa 5 del método

Consiste en la destrucción del clúster de trabajo para liberar los recursos usados por este, una vez este ya no sea necesario. Para esta tarea solo será necesario eliminar el CRD usado para la creación del clúster. Una vez eliminado el CRD el operador deberá identificar todos los recursos asociados al clúster de trabajo y eliminarlos del clúster de Kubernetes liberado los recursos utilizados. Las carpeta creada en el almacenamiento compartido no será borrada y deberá ser eliminada manualmente en caso de requerirlo.

Capítulo 5

Implementación del Método

La presente sección describe las tareas necesarias para la implementación de una prueba de concepto del método utilizando el diseño propuesto en el capítulo anterior. Se detallan aspectos propios de la implementación de las imágenes de contenedor que sirven como ambientes auto-contenidos. Además se describe el desarrollo del operador utilizando el *Operator SDK*, su distribución e implementación en un ambiente distribuido en la nube.

5.1 Implementación de imágenes de ambientes auto-contenidos

Se crearán tres ambientes auto-contenidos sobre la imagen oficial de Ubuntu en su versión 22.04 basadas en el diseño propuesto en la sección 4.1 del capítulo de diseño del método. La especificación de cada imagen puede ser consultada en el repositorio de código del operador del método en la carpeta `.docker-images` y en los anexos 8.17, 8.18 y 8.19 [26]. Los diferentes *scripts* utilizados por las imágenes pueden ser consultados en el repositorio de código del operador del método en la carpeta `.misc` [26].

Todas las imágenes de ambientes auto-contenidos tienen en común una serie de pasos como la instalación de librerías necesarias para la clonación de proyectos, pruebas de conexión, compilación de código, transferencia de datos por medio del protocolo seguro SSH. Además contienen todos los *scripts* necesarios para la correcta configuración e inicialización del contenedor. Estos *scripts* se explican en la sección 4.1 del capítulo de diseño del método y son copiados a la carpeta `/bin` donde se les concede permiso de ejecución.

La primera imagen de ambiente auto-contenido esta enfocada a ejecutar aplicaciones de computación paralela basadas en MPI utilizando la distribución MPICH en su versión 4.2.0. La instalación de MPICH se realiza por medio del administrador de paquetes de Ubuntu APT, este se encarga de instalar las librerías de MPICH y agregar los ejecutables a la ruta `/usr/bin` lo que permite que se puedan invocar desde la línea de comandos.

Hydra es el administrador de procesos predeterminado de MPICH [19]. *Hydra* puede leer

la información de los nodos disponibles para la ejecución de aplicaciones de MPI de un archivo de *hosts*. La ruta a este archivo puede ser especificada mediante el argumento -f al utilizar el ejecutable de línea de comandos *mpirun* o mediante la variable de entorno llamada *HYDRA_HOST_FILE*. La imagen de contenedor contiene un *script* llamado *generateHostsFileScriptMpich.sh* que se encarga de crear el archivo de *hosts* para MPICH en la ruta de almacenamiento compartido `/nfs/hosts.mpich`. Esto permite que todos los nodos pertenecientes al clúster de trabajo puedan leer este archivo.

A si mismo dentro de la especificación de la imagen se genera un *script* llamado *generateBashInit.sh* que se encarga de exportar la variable de entorno *HYDRA_HOST_FILE* a cada sesión de *bash*. Esto es posible al agregar la instrucción al archivo `~/.bashrc`.

La segunda imagen de ambiente auto-contenido esta enfocada a ejecutar aplicaciones de computación paralela basadas en MPI utilizando la distribución OpenMPI en su versión 4.1.6. Al ser una versión mayor anterior a la versión actual 5.x las instrucciones de instalación de OpenMPI por medio de su código fuente se encuentran en la sección de preguntan frecuentes para las versiones 4.1 y anteriores [4]. La especificación de la imagen instala OpenMPI en la ruta `/opt/.openmpi`.

El método propuesto en esta investigación esta diseñado para que el usuario que ejecute las aplicaciones de computación paralela sea el usuario *root*. OpenMPI bloquea por defecto la ejecución de programas de MPI por medio del usuario *root*. Para evitar el bloqueo es necesario exportar las variables de entorno *OMPI_ALLOW_RUN_AS_ROOT* y *OMPI_ALLOW_RUN_AS_ROOT_CONFIRM* con valores de 1. Además es necesario exportar la ruta de instalación de las librerías y ejecutables de OpenMPI dentro de la variable de *LD_LIBRARY_PATH* para que estos pueden ser llamados desde la línea de comandos.

A si mismo dentro de la especificación de la imagen se genera un *script* llamado *generateBashInit.sh* que se encarga de exportar las variables de entorno mencionadas anteriormente a cada sesión de *bash*. Esto es posible al agregar la instrucciones al archivo `~/.bashrc`.

OpenMPI puede leer la información de los nodos disponibles para la ejecución de aplicaciones de MPI de un archivo de *hosts*. La ruta a este archivo puede ser especificada mediante el argumento -hostfile al utilizar el ejecutable de línea de comandos *mpirun*. La imagen de contenedor contiene un *script* llamado *generateHostsFileScriptOpenmpi.sh* que se encarga de crear el archivo de *hosts* para OpenMPI en la ruta de almacenamiento compartido `/nfs/hosts.openmpi`. Esto permite que todos los nodos pertenecientes al clúster de trabajo puedan leer este archivo.

La tercera imagen de ambiente auto-contenido esta enfocada a ejecutar aplicaciones de computación paralela basadas en MPI utilizando la distribución MPICH en su versión 4.2.0 por medio del administrador de colas de trabajo SLURM [7]. La instalación y configuración de SLURM se realiza durante la fase de compilación de la imagen y al iniciar el contenedor, esto por medio del *script* llamado *installSlurm.sh*.

SLURM necesita de un nodo maestro el cual es el encargado de calendarizar los trabajos

en la cola de ejecución, para que estos sean ejecutados en los demás nodos del clúster de trabajo. Para esto se asignará el primer contenedor del *statefulset* como el nodo maestro y los demás como los nodos de ejecución. Es decir los nodos disponibles para la ejecución de aplicaciones de computación paralela basadas en MPI utilizando este ambiente auto-contenido van a ser igual al total de nodos menos uno.

Durante la fase de compilación de la imagen se crean los usuarios y grupos necesarios para la autenticación por medio de Munge. Munge es el servicio de autenticación utilizado por SLURM permite al nodo maestro ejecutar de forma desatendida aplicaciones en los demás nodos del clúster de trabajo. Además la imagen copia el archivo para la configuración de SLURM `slurm.conf` en la ruta `/opt/operator/misc/slurm.conf`.

El *script installSlurm.sh* se ejecuta en todos los nodos del clúster de trabajo y es el encargado de inicializar el servicio de munge y las librerías de SLURM utilizando el administrador de paquetes de Ubuntu APT. Además, actualiza el archivo de configuración `slurm.conf` utilizando las variables de entorno proporcionadas por el operador e inicializa el servicio de SLURM.

Después de instalado y configurado SLURM, MPICH es instalado siguiendo el mismo método utilizado para la creación del primer ambiente auto-contenido. La distribución de MPICH en su versión 4.1.6 es compatible con el administrador de colas de trabajo de SLURM lo que permitirá la ejecución de aplicaciones de computación paralela en el nodo maestro por medio de las aplicaciones de línea de comandos *srun* y *sbatch*.

5.2 Desarrollo del operador de Kubernetes

El operador de Kubernetes fue desarrollado en una máquina con sistema operativo Windows 11 en un ambiente de desarrollo de Ubuntu 22.04 por medio del subsistema de Linux de Windows. Se utilizó el *framework* de desarrollo *Operator SDK* en su versión 1.34.1. El lenguaje utilizado fue golang en su versión 1.22.2. El primer paso para la implementación fue crear un nuevo proyecto llamado *kubernetes-mpi-operator*. El código fuente del proyecto puede ser consultado en el repositorio de código del operador [26].

Una vez creado el proyecto se procedió a crear dos recursos customizados o CRDs. El recurso llamado MpiCluster el cual podrá ser utilizado para la automatización de la creación de un clúster de trabajo enfocado a la ejecución de aplicaciones de computación paralela. El recurso llamado MpiJob el cual podrá ser utilizado para la automatización de la ejecución de aplicaciones de computación paralela en cualquier clúster de trabajo creado anteriormente mediante el operador.

Al crearse el CRD de MpiCluster el *operator SDK* crea el archivo de tipos llamado `mpicluster_types.go` y el archivo de controlador llamado `mpicluster_controller.go`. Se modificó el archivo `mpicluster_types.go` de acuerdo a la especificación propuesta la sección 4.2.1 del capítulo de diseño del método incluyendo todas las propiedades necesarias para su correcta implementación. Cada propiedad cuenta con un tipo y decoradores de

kubebuilder que permiten al operador efectuar validaciones sobre las instancias del CRD de tipo MpiCluster [25].

A continuación se detalla el tipo de dato de cada propiedad del CRD de tipo MpiCluster así como las validaciones incluidas:

- *ClusterName*: Tipo de dato *string*. Campo obligatorio.
- *ClusterNodeImage*: Tipo de dato *string*. Campo obligatorio.
- *ClusterNodesCount*: Tipo de dato *int32*. Campo obligatorio. El valor mínimo permitido es de 1. El valor máximo permitido es de 100.
- *CoresPerNode*: Tipo de dato *int32*. Campo obligatorio. El valor mínimo permitido es de 1.
- *MemoryPerNode*: Tipo de dato *int32*. Campo obligatorio. El valor mínimo permitido es de 1.
- *SharedMemoryPerNode*: Tipo de dato *int32*. Campo obligatorio. El valor mínimo permitido es de 1 y su valor por defecto es 64.
- *SshPort*: Tipo de dato *int*. Campo obligatorio. El valor por defecto es de 22.
- *LocalVolumePath*: Tipo de dato *string*. Campo obligatorio.
- *LocalVolumeCapacity*: Tipo de dato *int*. Campo obligatorio. El valor mínimo permitido es de 1. El valor máximo permitido es de 1000.
- *EnableNvidiaGPU*: Tipo de dato *booleano*. Campo opcional. El valor por defecto es falso.
- *NumGPUs*: Tipo de dato *int32*. Campo opcional. El valor mínimo permitido es de 1.
- *NameGPUs*: Tipo de dato *string*. Campo opcional.

Estas validaciones permiten que solo especificaciones de CRD de tipo MpiCluster que cumplan con lo estipulado sean aceptadas. El archivo `mpicluster_controller.go` contiene el método para la implementación del ciclo de reconciliación y finalización. Este método se actualizó según la propuesta definida en la sección 4.3 del capítulo de diseño del método.

Para que el operador sea capaz de crear las instancias de CRD de tipo MpiCluster además de todos los recursos necesarios para el clúster de trabajo dentro del clúster de Kubernetes, es necesario otorgarle los permisos para que este pueda leer, crear, actualizar y remover estos tipos de recursos. Los permisos son concedidos al operador por medio de decoradores de kubebuilder [25].

En diagrama 5.1 muestra el flujo de operaciones implementadas en el método de reconciliación del controlador del CRD de tipo MpiCluster.

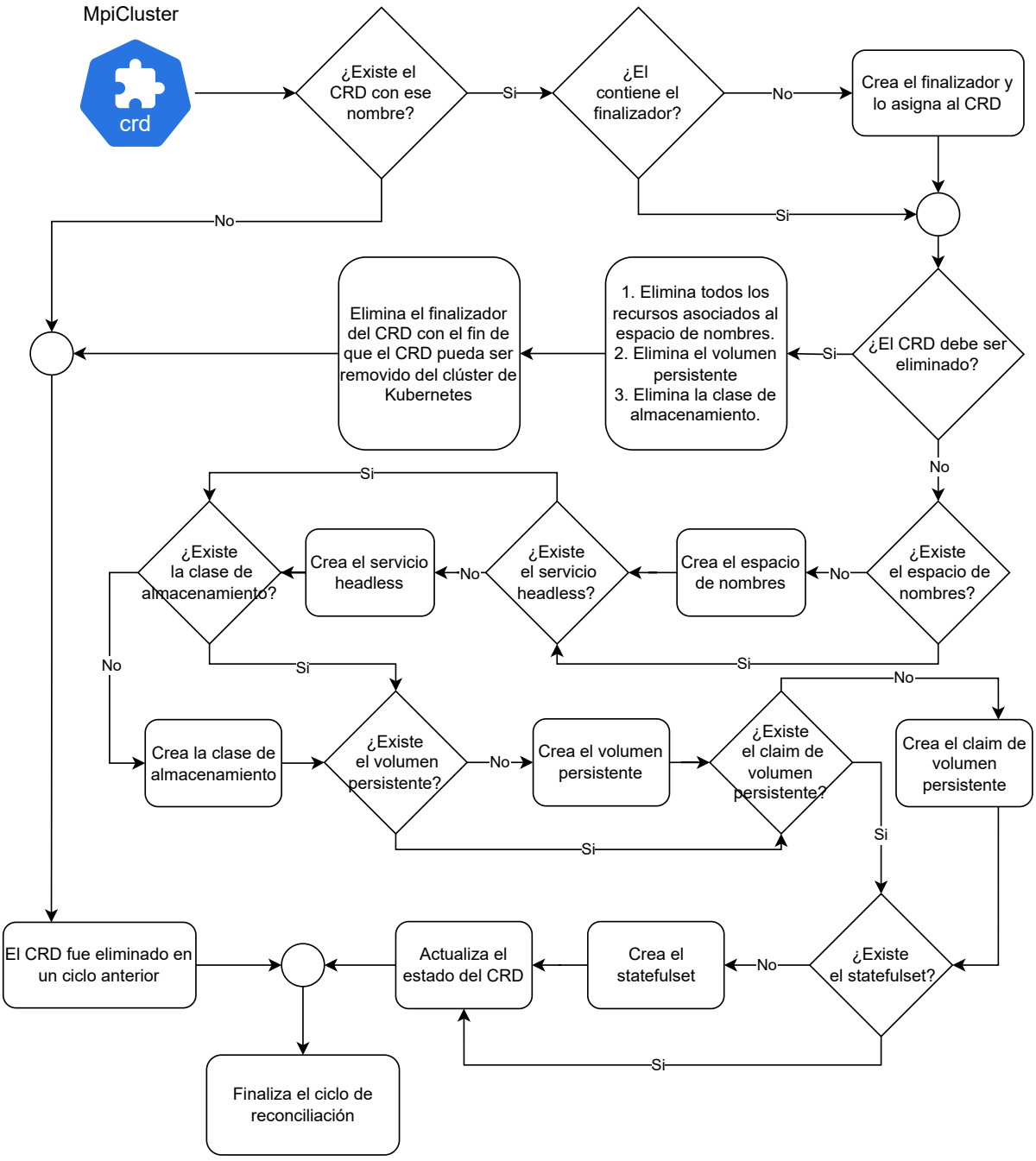

Figura 5.1: Ciclo reconciliación del operador para el CRD MpiCluster

Al crearse el CRD de MpiJob el *operator SDK* crea el archivo de tipos llamado `mpijob_types.go` y el archivo de controlador llamado `mpijob_controller.go`. Se modificó el archivo `mpijob_types.go` de acuerdo a la especificación propuesta la sección 4.2.1 del capítulo de diseño del método incluyendo todas las propiedades necesarias para su correcta implementación. Cada propiedad cuenta con un tipo y decoradores de kubebuilder que

permiten al operador efectuar validaciones sobre las instancias del CRD de tipo MpiJob [25].

A continuación se detalla el tipo de dato de cada propiedad del CRD de tipo MpiJob así como las validaciones incluidas:

- *JobName*: Tipo de dato *string*. Campo obligatorio.
- *ClusterName*: Tipo de dato *string*. Campo obligatorio.
- *Command*: Tipo de dato *string*. Campo obligatorio.

Estas validaciones permiten que solo especificaciones de CRD de tipo MpiJob que cumplan con lo estipulado sean aceptadas. El archivo `mpijob_controller.go` contiene el método para la implementación del ciclo de reconciliación y finalización. Este método se actualizó según la propuesta definida en la sección 4.4 del capítulo de diseño del método.

Para que el operador sea capaz de crear las instancias de CRD de tipo MpiJob además de todos los recursos necesarios para la ejecución de los comandos definidos en el clúster de trabajo dentro del clúster de Kubernetes, es necesario otorgarle los permisos para que este pueda leer, crear, actualizar y remover estos tipos de recursos. Los permisos son concedidos al operador por medio de decoradores de kubebuilder [25].

Para la ejecución desatendida de comandos de forma remota fue desarrollado un ejecutor. Este ejecutor hace uso del cliente de Kubernetes de Go para ejecutar los comandos por medio de la creación de un recurso *exec*. La solicitud de ejecución utiliza *bash* como consola de línea de comandos y los comandos son únicamente ejecutados en el primer nodo del clúster de trabajo es decir en el a1-sts-0 donde a1 es el nombre del clúster de trabajo especificado en el archivo YAML del CRD.

Los comandos son ejecutados únicamente en el primer nodo del clúster ya que se espera que el usuario del clúster de trabajo utilice el almacenamiento compartido y los ejecutables de MPI para la ejecución de su aplicación de computación paralela. Los ejecutables de MPI se encargarán de distribuir la carga a travez de todos los nodos del clúster de trabajo.

Al utilizar el cliente de Kubernetes y un recurso de tipo *exec* es posible obtener la salida final de la consola *stdout* y la salida de error de la consola *stderr* las cuales serán almacenadas en el estado del CRD al finalizar la ejecución de los comandos especificados.

Si por alguna razón la ejecución no finaliza y consume los recursos computacionales del clúster de trabajo de manera indefinida. El administrador del clúster de Kubernetes deberá eliminar el CRD del clúster de Kubernetes. Es posible que este deba finalizar los procesos de manera manual mediante una sesión interactiva si el CRD no es eliminado por el finalizador.

El diagrama 5.2 muestra el flujo de operaciones implementadas en el método de reconciliación del controlador del CRD de tipo MpiJob.

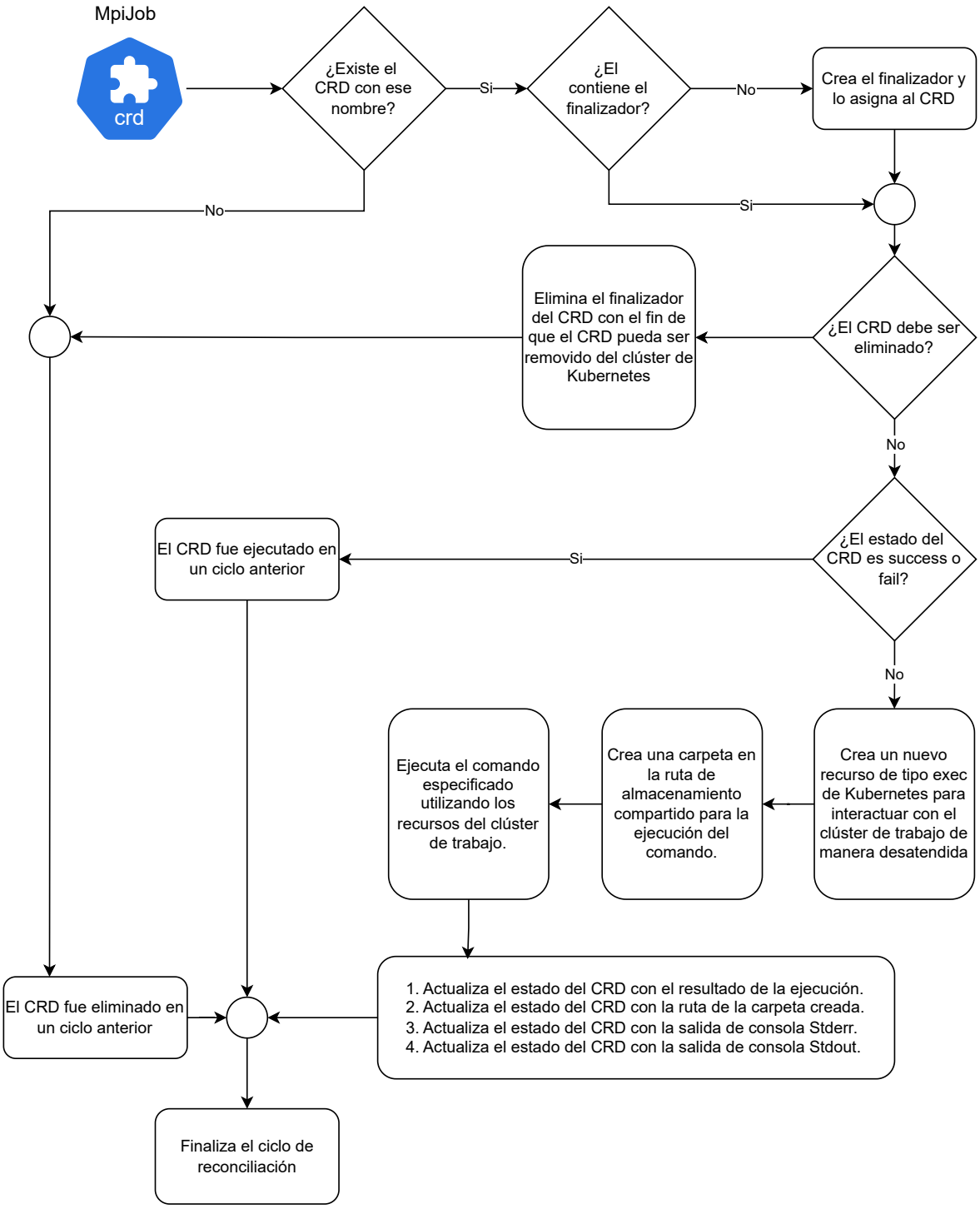

Figura 5.2: Ciclo reconciliación del operador para el CRD MpiJob

Al eliminar el CRD del clúster de Kubernetes el operador no eliminará la carpeta y los archivos creados en el almacenamiento compartido durante la ejecución de los comandos especificados. Esto es por diseño y su fin es que el usuario pueda consultar los resultados aún después de eliminado su CRD.

5.3 Implementación en un ambiente distribuido

El método será implementado en un ambiente distribuido en la nube. El proveedor público de recursos computacionales en la nube seleccionado para la implementación es Vultr. Vultr fue fundado en 2014 y su misión es ayudar a desarrolladores y negocios simplificando el despliegue de infraestructura mediante su plataforma en la nube [65].

Actualmente Vultr cuenta con 32 centros de datos localizados alrededor del mundo y provee el servicio de Infraestructura de GPUs, servidores bare metal, CPUs virtuales, entre otros [65]. Es relativamente fácil de utilizar y permite rentar recursos computacionales mediante una tarifa de uso por hora.

Debido a la disponibilidad de servidores de bare metal al momento de realizar esta investigación. Se optó por una infraestructura basada en CPUs virtuales dedicados. De este modo se creará un clúster de 10 nodos utilizando CPUs virtuales dedicados en el cual se instalará Kubernetes. Un nodo será utilizado para el *control plane*. Un nodo será utilizado como servidor de almacenamiento y los restantes 8 nodos serán utilizados como *worker nodes*. Los nodos estarán conectados por medio de una red privada virtual ipv4.

Además para las mediciones del tiempo de entrenamiento de modelos de I.A se creará un clúster de 4 nodos utilizando CPUs y GPUs virtuales dedicados en el cual se instalará Kubernetes. Un nodo será utilizado para el *control plane*. Un nodo será utilizado como servidor de almacenamiento y los restantes 2 nodos serán utilizados como *worker nodes*. Los nodos estarán conectados por medio de una red privada virtual ipv4.

Para la configuración del almacenamiento compartido. Un nodo dedicado será utilizado para los servicios de administración, meta-datos y almacemaniento de BeeGFS. Mientras que los *worker nodes* serán utilizados como clientes de BeeGFS.

Las especificaciones de CPU, memoria, y disco del nodo de servidor de almacenamiento son las siguientes:

- Tipo: Máquina virtual
- Procesadores lógicos: 4
- Modelo CPU: AMD EPYC-Milan
- RAM: 32G
- Almacenamiento: 640G
- SO: Ubuntu 22.04

La imagen 5.3 presenta la infraestructura propuesta para la implementación, los tres puntos en la sección de worker nodes y pods significan que existen más nodos del mismo tipo sin embargo estos no se encuentran representados dentro del gráfico.

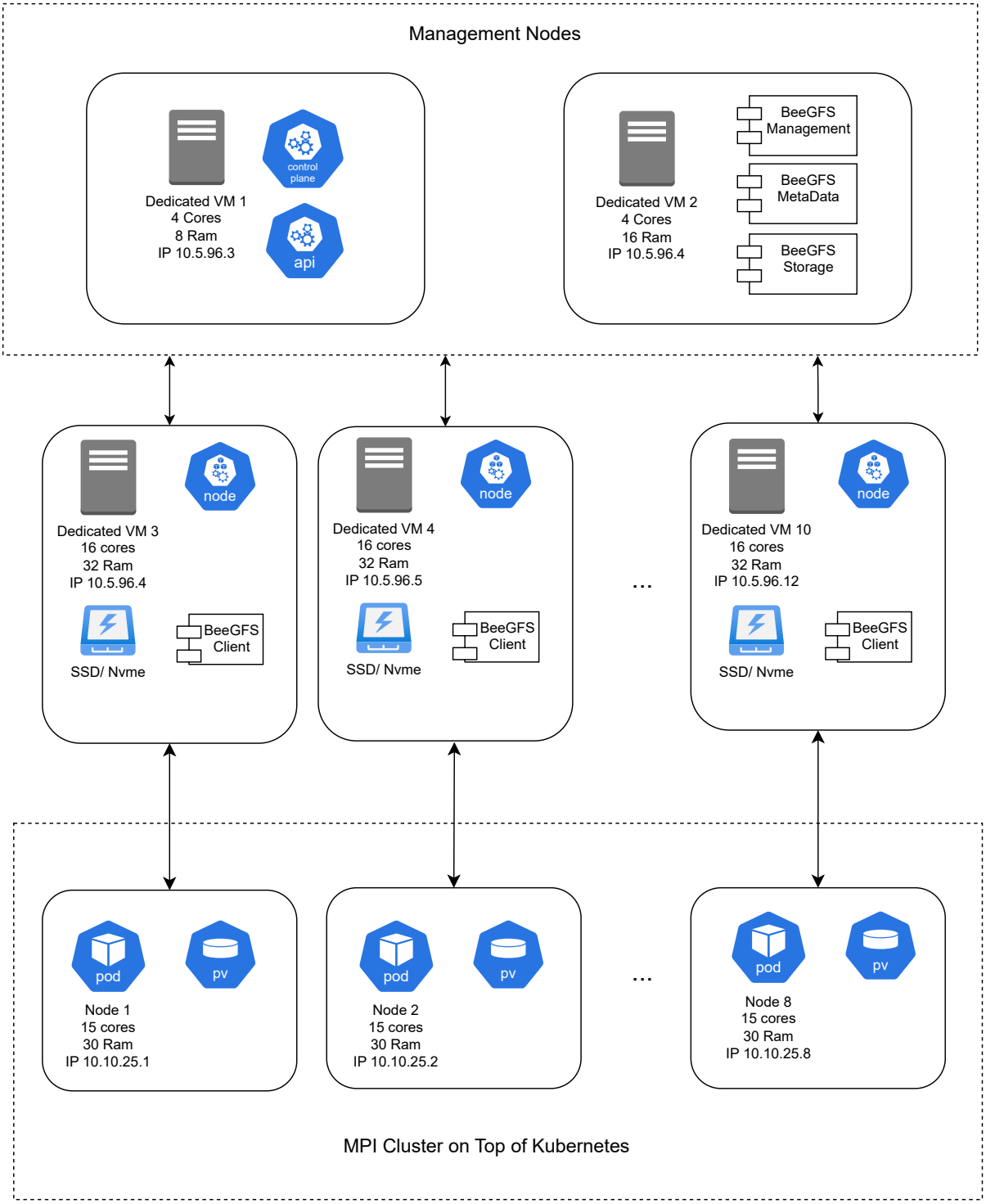

Figura 5.3: Infraestructura para la implementación del método

Las especificaciones de CPU, memoria, y disco del nodo de *control plane* son las siguientes:

- Tipo: Máquina virtual
- Procesadores lógicos: 4

- Modelo CPU: AMD EPYC-Milan
- RAM: 8G
- Almacenamiento: 75G
- SO: Ubuntu 22.04

Las especificaciones de CPU, memoria, y disco de los *worker nodes* son las siguientes:

- Tipo: Máquina virtual
- Procesadores lógicos: 16
- Modelo CPU: AMD EPYC-Rome
- RAM: 32G
- Almacenamiento: 300G
- SO: Ubuntu 22.04

Las especificaciones de CPU, GPU, memoria, y disco de los *worker nodes* habilitados para I.A son las siguientes:

- Tipo: Máquina virtual
- Procesadores lógicos: 6
- Modelo CPU: Intel Xeon (Cascadelake)
- Modelo GPU: Nvidia A100 MIG 1/2 40G
- RAM: 64G
- Almacenamiento: 700G
- SO: Ubuntu 22.04

Una vez creados todos los nodos estos contarán con una IP interna y una IP pública. La IP pública puede ser utilizada para conectarse a los nodos a través de SSH por medio de la llave configurada previamente en Vultr.

Para esta investigación fueron creados una serie de *scripts* de inicialización para la configuración idónea del clúster. Estos *scripts* se encuentran en la carpeta `.infrastructure` del repositorio de código del operador [26]. Existen dos *script* que se necesitan ejecutar para la configuración del clúster. Estos a su vez utilizan otros *scripts* para la ejecución de diferentes tareas.

Para el correcto funcionamiento de los *scripts* de configuración es necesario especificar los datos de cada uno de los nodos del clúster. Para esto se creo un archivo llamado *machines.txt* el cual se encuentra en la carpeta `.infrastructure` del repositorio de código del operador [26]. Este archivo contiene la lista de las 10 máquinas que formarán parte de la infraestructura del clúster de Vultr. Cada línea del archivo corresponde a una máquina y contiene el nombre de la máquina, su IP interna, la contraseña para conectarse por medio de SSH y la IP pública. A continuación se detalla el funcionamiento de cada *script* con el fin de que otros investigadores puedan recrear el ambiente utilizado.

***SetupVultrCluster.sh*:** Es el primer *script* que debe ser ejecutado, su función es instalar y configurar las librerías necesarias para que los nodos puedan comunicarse entre ellos y acceder al almacenamiento compartido. Primero el *script* copia las llaves pública y privada a cada uno de los nodos especificados en el archivo *machines.txt* y las agrega al archivo de llaves autorizadas.

Segundo el *script* configura la comunicación por medio de SSH en todas las máquinas actualizando los archivos `/etc/hosts` y `/etc/cloud/templates/hosts.debian.tmpl` con las direcciones IP internadas de las demás máquinas. El archivo path/etc/-cloud/templates/hosts.debian.tmpl es de suma importancia ya que si las direcciones IP no son agregadas el archivo `/etc/hosts` será sobrescrito al reiniciarse el nodo. El *script* también configura la conexión sin contraseña entre todos los nodos del clúster de Vultr utilizando las llaves pública y privada copiadas previamente.

Tercero el *script* instala y configura los servicios de administración, meta-datos y almacenamiento de BeeGFS en la máquina designada para ser el servidor de almacenamiento. Cuarto el *script* instala y configura el cliente de BeeGFS en las máquinas destinadas a ser *worker nodes* del clúster de Kubernetes.

Por último el *script* instala MPI utilizando el administrador de paquetes de Ubuntu APT en las máquinas destinadas a ser *worker nodes* del clúster de Kubernetes. La distribución de MPI a instalar dependerá del primer argumento del *script* siendo el valor 1 MPICH y el valor 2 OpenMPI.

Si MPICH fue seleccionado se configura el archivo de *hosts* para MPI y crea la variable de entorno *HYDRA_HOST_FILE* con su ubicación. Esto permitirá utilizar todos los procesadores de todas las máquinas al momento de realizar las diferentes mediciones.

Si OpenMPI fue seleccionado se configura el archivo de *hosts* para MPI y se crean las variables de entorno *LD_LIBRARY_PATH* con la ubicación de las librerías de OpenM-PI, *OMPI_ALLOW_RUN_AS_ROOT* y *OMPI_ALLOW_RUN_AS_ROOT_CONFIRM* que permiten ejecutar OpenMPI como usuario *root*. Esto permitirá utilizar todos los procesadores de todas las máquinas al momento de realizar las diferentes mediciones.

Por último si el segundo argumento del *script* contiene un valor de 1. Se instalará miniConda un ambiente de desarrollo y ejecución de Python que permite crear ambientes virtuales con diferentes requerimientos. Una vez instalado miniConda se

creará un ambiente capaz de entrenar modelos de I.A utilizando GPUs de Nvidia que esten disponibles en los *worker nodes*.

***SetupVultrK8sCluster.sh*:** Es el segundo *script* que debe ser ejecutado, su función es instalar y configurar las librerías necesarias para crear el clúster de Kubernetes donde se pueda desplegar el método para la ejecución de tareas de computación paralela. Este *script* recibe un argumento el cual especifica la tarea que se desea ejecutar.

Al recibir como argumento de entrada el número 1 el *script* configura el ambiente para la instalación. Instala y configura *containerd* el cual será utilizado como el *runtime* de contenedores para el clúster de Kubernetes [2]. Luego de instalar *containerd* el *script* se encarga de instalar el servicio kubelet para la comunicación entre los *worker nodes* y el nodo de *control plane*, la herramienta de línea de comandos kubectl y kubeadm[1].

Al recibir como argumento de entrada el número 2 el *script* inicializa el nodo de *control plane* utilizando kubeadm en la máquina designada para este propósito. Al finalizar el proceso kubeadm muestra un token y un *hash* que deben ser utilizados para agregar los *worker nodes* al clúster de Kubernetes recién creado. Estos valores deben ser actualizados en el *script* llamado *setupK8sWorkerNode.sh*.

Al recibir como argumento de entrada el número 3 el *script* agrega las máquinas designadas como *worker nodes* al clúster de trabajo utilizando los valores proporcionados en el *script setupK8sWorkerNode.sh*.

Al recibir como argumento de entrada el número 4 el *script* instala y configura el CNI en el clúster de Kubernetes utilizando Calico. Calico permite a los contenedores dentro del clúster de Kubernetes comunicarse con otros contenedores en el mismo clúster o direcciones de red externas al clúster de Kubernetes.

La configuración aplicada permitirá a Calico evitar la encapsulación del tráfico y enviar y recibir los paquetes utilizando la infraestructura de red de los *worker nodes* directamente siempre y cuando las máquinas se encuentren el la misma red interna. Este modo es requerido para obtener el rendimiento deseado al aplicar el método propuesto. A este punto existe un clúster de Kubernetes totalmente funcional desplegado en el clúster de Vultr.

Por último cuando el argumento de entrada es el número 5 el *script* instala y configura el Operador de Nvidia. El operador de Nvidia permite a los contenedores dentro del clúster de Kubernetes tener acceso a GPUs de la marca Nvidia que se encuentren dentro de los *worker nodes*.

El siguiente paso en la implementación es instalar el *Operator Lifecycle Manager* o OLM en el clúster de Kubernetes desde su repositorio de Github. Luego de instalado

[1]Kubeadm es una herramienta de comandos de línea que perimite crear y configurar un clúster de Kubernetes en conformidad con las mejores prácticas de la industria.

OLM se procede a crear el catálogo del operador utilizando el archivo YAML en la ruta `operator/olm-catalog.yaml` del repositorio de código de ejemplos para el operador [28]. Una vez instalado el catálogo se procede a crear una subscripción al catálogo utilizando el archivo YAML en la ruta `operator/olm-subscription.yaml` del repositorio de código de ejemplos para el operador [28].

Cuando la subscripción es creada. Un plan de instalación para el operador es creado automáticamente lo que ocasiona que el operador sea instalado en el clúster de Kubernetes. Una vez instalado el operador el ambiente a sido totalmente creado y configurado. A partir de ese momento es posible crear un clúster de trabajo para la ejecución de aplicaciones de computación paralela utilizando los recursos computacionales proporcionados por Vultr.

Capítulo 6

Experimentos y Resultados

En este capítulo se presentan los experimentos y resultados de la investigación. Cada experimento detalla las condiciones en la que fue efectuado para que su replicación por otros investigadores sea posible.

6.1 Validación en un ambiente de desarrollo

Se validará el método y los recursos definidos con la implementación de una prueba de concepto. El ambiente utilizado para esta validación consta de una máquina con sistema Operativo Windows 11, procesador AMD Ryzen 5600X, 32GB RAM 3200MHZ CL14 y un disco duro Nvme M2 PICE3 de 512GB. La implementación se hará sobre el subsistema de Linux en Windows WSL2 utilizando Ubuntu 22.04.

Para el almacenamiento compartido se creo una carpeta compartida en la ruta `/nfsshare` la cual es utilizada en todos los *worker nodes* del clúster de Kubernetes local. Se creó un clúster de Kubernetes local con 1 nodo de *control plane* y 3 *worker nodes* utilizando Kind en su versión v0.20.0 linux/amd64. El anexo 3(8.3) muestra la especificación del archivo YAML utilizado para la creación del clúster.

La imagen de contenedor para el ambiente auto-contenido a utilizar es la detallada en la sección del capítulo de Implementación del Método. Esta imagen de contenedor esta publicada en el repositorio Docker y puede ser accedida desde la dirección `docker.io/crisarias/mpicluster-mpich-operator-base:latest`.

Para crear el clúster de trabajo en el clúster de Kubernetes local se siguieron los siguientes pasos.

1. Se definió un archivo YAML con la especificación del espacio de nombres de acuerdo al diseño propuesto el cual puede consultarse en el anexo 4(8.4). El espacio de nombres fue creado en el clúster local utilizando el comando:

   kubectl apply −f namespace−template.yaml

2. Se definió un archivo YAML con la especificación del servicio *headless* de acuerdo al diseño propuesto el cual puede consultarse en el anexo 5(8.5). El servicio *headless* fue creado en el clúster local utilizando el comando:

   kubectl apply −f headless−svc−template.yaml

3. Se definió un archivo YAML con la especificación de la clase de almacenamiento de acuerdo al diseño propuesto el cual puede consultarse en el anexo 6(8.6). La clase de almacenamiento fue creada en el clúster local utilizando el comando:

   kubectl apply −f storageclass−template.yaml

4. Se definió un archivo YAML con la especificación del volumen persistente de acuerdo al diseño propuesto el cual puede consultarse en el anexo 7(8.7). El volumen persistente fue creado en el clúster local utilizando el comando:

   kubectl apply −f pv−template.yaml

5. Se definió un archivo YAML con la especificación del *claim* de volumen persistente de acuerdo al diseño propuesto el cual puede consultarse en el anexo 8(8.8). El *claim* de volumen persistente fue creado en el clúster local utilizando el comando:

   kubectl apply −f pvc−template.yaml

6. Se definió un archivo YAML con la especificación del *statefulset* de acuerdo al diseño propuesto el cual puede consultarse en el anexo 9(8.9). El *statefulset* fue creado en el clúster local utilizando el comando:

   kubectl apply −f statefulset−template.yaml

Una vez creados todos los recursos necesarios en el clúster de trabajo se procedió a verificar que todos los nodos del mismo estuvieran en estado *Running*. Todos los comandos ejecutados para la creación del clúster de trabajo y la salida se muestran a continuación:

```
root@DESKTOP−E198SET:/home/sources/kubernetes−mpi−operator/.yaml−templates#
    ↪ kubectl apply −f namespace−template.yaml
namespace/irazu−ns created
root@DESKTOP−E198SET:/home/sources/kubernetes−mpi−operator/.yaml−templates#
    ↪ kubectl apply −f headless−svc−template.yaml
service/irazu−headless−svc created
root@DESKTOP−E198SET:/home/sources/kubernetes−mpi−operator/.yaml−templates#
    ↪ kubectl apply −f storageclass−template.yaml
storageclass.storage.k8s.io/irazu−sc created
root@DESKTOP−E198SET:/home/sources/kubernetes−mpi−operator/.yaml−templates#
    ↪ kubectl apply −f pv−template.yaml
persistentvolume/irazu−pv created
root@DESKTOP−E198SET:/home/sources/kubernetes−mpi−operator/.yaml−templates#
    ↪ kubectl apply −f pvc−template.yaml
```

persistentvolumeclaim/irazu−pvc created
root@DESKTOP−E198SET:/home/sources/kubernetes−mpi−operator/.yaml−templates#
↪ *kubectl apply −f statefulset−template.yaml*
statefulset.apps/irazu−sts created
root@DESKTOP−E198SET:/home/sources/kubernetes−mpi−operator/.yaml−templates#
↪ *kubectl get pods −n irazu−ns*
NAME READY STATUS RESTARTS AGE
irazu−sts−0 1/1 Running 0 8m28s
irazu−sts−1 1/1 Running 0 8m7s
irazu−sts−2 1/1 Running 0 7m47s

6.1.1 Validación de la conexión SSH

Para validar que todos los nodos del clúster de trabajo pudieran conectarse por medio de SSH, se utilizó una sesión interactiva en cada nodo del clúster de trabajo y se ejecutó el comando:

ssh a1−sts−N−a1−headless−svc.a1−ns.svc.cluster.**local** 'echo-$HOSTNAME-&&-exit-0'

Donde a1 es el nombre del clúster y N es el número de nodo siendo empezando desde 0. Lo esperado es que se imprima el mensaje mostrando el nombre del nodo al que se esta conectando y se desconecte volviendo al nodo que solicito la conexión. El resultado de la validación determinó que todos los nodos pudieron conectarse de manera exitosa entre ellos. El *log* de todas las conexiones realizadas y los comandos utilizados puede verse a continuación.

root@DESKTOP−E198SET:/home# *kubectl exec −it irazu−sts−0 −n irazu−ns −− bash*
root@irazu−sts−0:/# *ssh irazu−sts−0.irazu−headless−svc.irazu−ns.svc.cluster.local 'echo*
↪ *$HOSTNAME && exit 0'*
irazu−sts−0
root@irazu−sts−0:/# *ssh irazu−sts−1.irazu−headless−svc.irazu−ns.svc.cluster.local 'echo*
↪ *$HOSTNAME && exit 0'*
irazu−sts−1
root@irazu−sts−0:/# *ssh irazu−sts−2.irazu−headless−svc.irazu−ns.svc.cluster.local 'echo*
↪ *$HOSTNAME && exit 0'*
irazu−sts−2
root@irazu−sts−0:/# *exit*
exit
root@DESKTOP−E198SET:/home# *kubectl exec −it irazu−sts−1 −n irazu−ns −− bash*
root@irazu−sts−1:/# *ssh irazu−sts−0.irazu−headless−svc.irazu−ns.svc.cluster.local 'echo*
↪ *$HOSTNAME && exit 0'*
irazu−sts−0
root@irazu−sts−1:/# *ssh irazu−sts−1.irazu−headless−svc.irazu−ns.svc.cluster.local 'echo*
↪ *$HOSTNAME && exit 0'*
irazu−sts−1

```
root@irazu-sts-1:/# ssh irazu-sts-2.irazu-headless-svc.irazu-ns.svc.cluster.local 'echo
    ↪ $HOSTNAME && exit 0'
irazu-sts-2
root@irazu-sts-1:/# exit
exit
root@DESKTOP-E198SET:/home# kubectl exec -it irazu-sts-2 -n irazu-ns -- bash
root@irazu-sts-2:/# ssh irazu-sts-0.irazu-headless-svc.irazu-ns.svc.cluster.local 'echo
    ↪ $HOSTNAME && exit 0'
irazu-sts-0
root@irazu-sts-2:/# ssh irazu-sts-1.irazu-headless-svc.irazu-ns.svc.cluster.local 'echo
    ↪ $HOSTNAME && exit 0'
irazu-sts-1
root@irazu-sts-2:/# ssh irazu-sts-2.irazu-headless-svc.irazu-ns.svc.cluster.local 'echo
    ↪ $HOSTNAME && exit 0'
irazu-sts-2
root@irazu-sts-2:/# exit
exit
root@DESKTOP-E198SET:/home#
```

6.1.2 Ejecución de aplicación de MPI en el clúster de trabajo

Esta prueba consistió en la ejecución de la implementación del programa hola mundo en MPI utilizando todos los nodos del clúster de trabajo. Para esto se utilizó una sesión interactiva y el repositorio público de GitHub de Tutoriales para MPI el cual tiene diferentes implementaciones de programas para MPI incluido hola mundo [55]. El repositorio se clonó en la ruta `/nfs` la cual es el almacenamiento compartido del clúster de trabajo. De otra manera el programa no estaría disponible en los otros nodos del clúster.

Como la especificación utilizada para crear el clúster de trabajo indicaba que tendría 3 nodos con 4 procesadores cada uno, el programa se ejecutó con un parámetro de np=12 para utilizar todos los recursos del clúster de trabajo. A continuación se muestran los comandos y resultados utilizados para la ejecución los cuales son la prueba de que el programa logró ejecutase en todos los *ranks* del clúster de trabajo satisfactoriamente.

```
root@DESKTOP-E198SET:/home# kubectl exec -it irazu-sts-0 -n irazu-ns -- bash
root@irazu-sts-0:/# cd /nfs
root@irazu-sts-0:/nfs# git clone https://github.com/mpitutorial/mpitutorial
Cloning into 'mpitutorial'...
remote: Enumerating objects: 1449, done.
remote: Counting objects: 100% (134/134), done.
remote: Compressing objects: 100% (87/87), done.
remote: Total 1449 (delta 63), reused 108 (delta 46), pack-reused 1315
Receiving objects: 100% (1449/1449), 4.17 MiB | 6.79 MiB/s, done.
Resolving deltas: 100% (768/768), done.
root@irazu-sts-0:/nfs# cd mpitutorial/tutorials/mpi-hello-world/code
```

```
root@irazu-sts-0:/nfs/mpitutorial/tutorials/mpi-hello-world/code# make
mpicc -o mpi_hello_world mpi_hello_world.c
root@irazu-sts-0:/nfs/mpitutorial/tutorials/mpi-hello-world/code# mpirun -np 12 ./
    ↪ mpi_hello_world
Hello world from processor irazu-sts-0, rank 1 out of 12 processors
Hello world from processor irazu-sts-1, rank 6 out of 12 processors
Hello world from processor irazu-sts-0, rank 0 out of 12 processors
Hello world from processor irazu-sts-2, rank 8 out of 12 processors
Hello world from processor irazu-sts-1, rank 7 out of 12 processors
Hello world from processor irazu-sts-2, rank 9 out of 12 processors
Hello world from processor irazu-sts-1, rank 4 out of 12 processors
Hello world from processor irazu-sts-2, rank 10 out of 12 processors
Hello world from processor irazu-sts-2, rank 11 out of 12 processors
Hello world from processor irazu-sts-1, rank 5 out of 12 processors
Hello world from processor irazu-sts-0, rank 2 out of 12 processors
Hello world from processor irazu-sts-0, rank 3 out of 12 processors
```

6.2 Tiempo de creación de un clúster de trabajo

Para esta medición será utilizado el ambiente creado en la sección 5.3 del capítulo de implementación. Todos los resultados de los experimentos incluyendo la salida de consola de cada comando serán guardados en la carpeta `Operator-Experiments` del repositorio de código de resultados de experimentos para el operador[27].

Tanto la creación del clúster de trabajo como la ejecución de una aplicación de computación paralela serán realizados por medio del operador de Kubernetes sin utilizar una sesión interactiva a ninguno de los nodos del clúster de trabajo.

La medición consiste en crear un clúster de trabajo utilizando los YAML de ejemplo del repositorio de código de ejemplos para el operador [28]. Existen tres archivos YAML para la creación de diferentes clúster de trabajo utilizando las imágenes base para el ambiente auto-cuya implementación es descrita en la sección 5.1 del capítulo de implementación del método.

Las imágenes poseen ambientes para un clúster de trabajo para MPI utilizando MPICH incluido en el anexo 10(8.10). Un clúster de trabajo para MPI utilizando OpenMPI incluido en el anexo 11(8.11). Un clúster de trabajo utilizando MPICH y Slurm incluido en el anexo 12(8.12).

Cada que un clúster es creado se verificará el tiempo que toman los nodos en llegar al estado de *Running*. La suma del tiempo en minutos debe ser menor al número total de nodos del clúster de trabajo. Si esto se cumple, el experimento será exitoso ya que quiere decir que el promedio de creación y configuración de los nodos del clúster de trabajo es igual o menor a 1 minuto.

Para que este resultado sea válido es necesario verificar que el clúster de trabajo es capaz de ejecutar aplicaciones de computación paralela basadas en MPI de manera satisfactoria. Para esto se utilizarán los archivos YAML para la creación de un CRD que ejecute el programa hola mundo utilizando MPI y todos los *Ranks* disponibles en el clúster de trabajo. Estos archivos YAML están disponibles en el repositorio de código de ejemplos para el operador [28]. La especificación de los archivos puede ser consultada en los anexos 13(8.13), 14(8.14) y 15(8.15).

Cada clúster de trabajo asociado a una de las tres imágenes de contenedor disponibles, será creado en tres ocaciones. Después de cada creación el clúster será eliminado eliminando el CRD creado a partir del archivo YAML.

Las tablas de resultados contienen las columnas T1, T2, T3 que corresponden a los tiempos de creación en segundos para cada nodo en diferentes repeticiones de creación del clúster de trabajo. Las columnas T-De y T-Avg muestran la desviación estándar de los tiempos obtenidos entre experimentos por nodo y la media del tiempo de creación por nodo respectivamente.

La tabla 6.1 muestra los resultados del tiempo tomado para crear los nodos del clúster de trabajo utilizando la imagen de ambiente auto-contenido de MPICH. El total promediado para la creación de un clúster de trabajo de 8 nodos con 15 procesadores cada uno utilizando la imagen de ambiente auto-contenido de MPICH fue de 21.583 segundos por lo que la medición en este caso fue exitosa.

Tabla 6.1: Tabla de resultados creación de clúster MPICH

Nodo	**T 1**	**T2**	**T3**	**T-De**	**T-Avg**
1	33	35	20	6,649979114	29,33333333
2	20	21	21	0,471404521	20,66666667
3	21	20	20	0,471404521	20,33333333
4	20	21	20	0,471404521	20,33333333
5	21	20	20	0,471404521	20,33333333
6	20	21	21	0,471404521	20,66666667
7	21	20	20	0,471404521	20,33333333
8	20	21	21	0,471404521	20,66666667

La tabla 6.2 muestra los resultados del tiempo tomado para crear los nodos del clúster de trabajo utilizando la imagen de ambiente auto-contenido de OpenMPI. El total promediado para la creación de un clúster de trabajo de 8 nodos con 15 procesadores cada uno utilizando la imagen de ambiente auto-contenido de OpenMPI fue de 21.125 segundos por lo que la medición en este caso fue exitosa.

Tabla 6.2: Tabla de resultados creación de clúster Open MPI

Nodo	**T 1**	**T2**	**T3**	**T-De**	**T-Avg**
1	21	35	22	6,3770	26
2	20	20	20	0	20
3	21	21	21	0	21
4	20	20	20	0	20
5	21	21	21	0	21
6	20	20	20	0	20
7	21	21	21	0	21
8	20	20	20	0	20

La tabla 6.3 muestra los resultados del tiempo tomado para crear los nodos del clúster de trabajo utilizando la imagen de ambiente auto-contenido de MPICH y SLURM. El total promediado para la creación de un clúster de trabajo de 7 nodos con 15 procesadores cada uno utilizando la imagen de ambiente auto-contenido de MPICH y SLURM fue de 21.125 segundos por lo que la medición en este caso fue exitosa.

Tabla 6.3: Tabla de resultados creacion de clúster Slurm

Nodo	**T 1**	**T2**	**T3**	**T-De**	**T-Avg**
1	20	34	22	6,1824	25,3333
2	21	21	21	0	21
3	20	20	20	0	20
4	21	21	21	0	21
5	20	20	20	0	20
6	20	21	21	0,4714	20,6666
7	21	20	20	0,4714	20,3333
8	20	21	21	0,4714	20,6666

En todas las mediciones anteriores el tiempo de creación del primer nodo fue mayor debido a que este es el encargado de crear la llave pública y privada que será utilizada en todos los nodos del clúster de trabajo. El tiempo de creación de todos los nodos también incluye el tiempo que dura el *worker node* en descargar la imagen de contenedor del repositorio de imágenes. Todas las mediciones en este caso descargaron la imagen ya que no se tenía en cache en ninguna de las repeticiones. El utilizar un repositorio de imágenes dentro de la red local o descargar las imágenes con anterioridad mejoraría el tiempo de creación de los nodos, especialmente en casos donde las imágenes de contenedor pesen varios cientos de *megabytes*.

6.3 Rendimiento de E/S de red y almacenamiento

Para esta medición será utilizado el ambiente creado en la sección 5.3 del capítulo de implementación y un clúster de trabajo basado en la imagen de ambiente auto-contenido de MPICH utilizando el archivo YAML disponible en el repositorio de código de ejemplos para el operador [28]. La especificación del archivo YAML puede ser consultada en el anexo 10(8.10). Todos los resultados de los experimentos incluyendo la salida de consola de cada comando serán guardados en la carpeta `ES-Experiments` del repositorio de código de resultados de experimentos para el operador[27].

La medición del almacenamiento será realizada utilizando la herramienta de línea de comandos DD. La prueba consiste en crear un archivo de prueba de 500 *megabytes* en el almacenamiento compartido. El mismo archivo será leído y la velocidad en *megabytes* por segundo de escritura y lectura debe ser registrada. Al finalizar la prueba el archivo será borrado del almacenamiento compartido.

Se realizarán tres repeticiones del experimento desde cada nodo del clúster de Vultr en la carpeta compartida ubicada en la ruta `/mnt/BeeGFS`. Y tres repeticiones del experimento desde cada nodo del clúster de trabajo de MPICH en la carpeta compartida ubicada en la ruta `/nfs`. Los resultados de escritura y lectura para disco son presentados en megabytes por segundo.

El operador de Kubernetes crea un clúster de la cantidad de nodos especificados, un nodo en cada *worker node*. Sin embargo, el primer nodo del clúster de trabajo es decir el nodo a-sts-0 puede no corresponder al primer *worker node* de *Vultr*. Para mostrar consistencia en este experimento se identificaron los *worker nodes* en que cada nodo del clúster de trabajo residía para comparar cada nodo del clúster de trabajo con su nodo el *worker node* donde estaba alojado.

Los resultados de las repeticiones para el clúster de Vultr y el clúster de trabajo se presentan en las tablas 6.4 y 6.5 respectivamente. En estas tablas la columna N contiene el nodo utilizado para la prueba, las columnas E1,E2,E3 contienen las velocidades de escritura alcanzadas en cada uno de las repeticiones por los diferentes nodos y las columnas L1,L2 y L3 contienen las velocidades de lectura alcanzadas en cada uno de las repeticiones por los diferentes nodos. Las columnas E-De y L-De contienen la desviación estándar de las repeticiones. Las columnas E-Avg y L-Avg contienen la media de velocidad alcanzada para escritura y lectura en cada nodo respectivamente.

La tabla 6.6 muestra el rendimiento de escritura y lectura en disco de los nodos del clúster de trabajo con respecto al rendimiento alcanzado en los nodos del clúster de Vultr. Las columnas E-Avg-WN y L-Avg-WN muestran la velocidad media de escritura y lectura alcanzada en cada uno de los nodos en el clúster de Vultr. La columnas E-Avg-CT y L-Avg-CT muestran las velocidad media de escritura y lectura alcanzada en cada uno de los nodos en el clúster de trabajo. Las columnas R-E-CT y R-L-CT muestran en rendimiento de velocidad de escritura y lectura obtenido en los nodos del clúster de trabajo con respecto al rendimiento obtenido en el clúster de Vultr.

Tabla 6.4: Tabla de resultados E/S disco en *worker nodes*

N	E1	E2	E3	E-De	E-Avg	L1	L2	L3	L-De	L-Avg
1	389	382	387	2,9439	386,0000	325	352	273	32,7855	316,6667
2	458	473	446	11,0454	459,0000	484	488	464	10,4987	478,6667
3	416	357	409	26,3186	394,0000	323	340	330	6,9761	331,0000
4	406	391	404	6,6500	400,3333	297	304	311	5,7155	304,0000
5	416	430	407	9,4634	417,6667	338	328	363	14,7196	343,0000
6	370	392	386	9,2856	382,6667	316	314	301	6,6500	310,3333
7	371	410	405	17,3269	395,3333	331	325	291	17,6131	315,6667
8	420	433	423	5,5578	425,3333	365	344	367	10,4030	358,6667

Tabla 6.5: Tabla de resultados E/S disco en clúster de trabajo

N	E1	E2	E3	E-De	E-Avg	L1	L2	L3	L-De	L-Avg
1	387	365	385	9,9331	379	322	313	302	8,1786	312,3333
2	453	449	476	11,8977	459,3333	515	514	487	12,9701	505,3333
3	393	397	379	7,7172	389,6667	318	302	308	6,5997	309,3333
4	366	404	387	15,5421	385,6667	286	306	296	8,1650	296,0000
5	373	365	375	4,3205	371,0000	329	331	309	9,9331	323,0000
6	384	344	341	19,6016	356,3333	297	296	317	9,6724	303,3333
7	395	396	400	2,1602	397,0000	335	322	314	8,6538	323,6667
8	350	407	380	23,2809	379,0000	315	316	300	7,3182	310,3333

Tabla 6.6: Tabla comparativa del rendimiento de E/S en disco.

N	E-Avg-WN	E-Avg-CT	L-Avg-WN	L-Avg-CT	R-E-CT	R-L-CT
1	386,0000	379,0000	316,6667	312,3333	98,1865	98,6316
2	459,0000	459,3333	478,6667	505,3333	100,0726	105,5710
3	394,0000	389,6667	331,0000	309,3333	98,9002	93,4542
4	400,3333	385,6667	304,0000	296,0000	96,3364	97,3684
5	417,6667	371,0000	343,0000	323,0000	88,8268	94,1691
6	382,6667	356,3333	310,3333	303,3333	93,1185	97,7444
7	395,3333	397,0000	315,6667	323,6667	100,4216	102,5343
8	425,3333	379,0000	358,6667	310,3333	89,1066	86,5242

Si se analiza la desviación estándar en términos del coeficiente de variación se tiene que el valor máximo para la velocidad de lectura fue del 6.67% en los *worker nodes* y del 6.14% en el clúster de trabajo. El coeficiente de variación máximo para la velocidad de escritura fue del 10.35% en los *worker nodes* y del 3.18% en el clúster de trabajo. Estos valores indican que la media es representativa del conjunto de datos es decir es un conjunto de datos homogéneo.

Al analizar el rendimiento alcanzado por los nodos del clúster de trabajo con respecto a los *worker nodes* tanto en escritura y lectura es menor en al menos el 75% de los casos. Sin embargo, el rendimiento alcanzado siempre fue superior al 85%, siendo capaces la mayoría de nodos de alcanzar un rendimiento superior al 90%.

El resultado del experimento de la medición del rendimiento de la escritura y lectura del almacenamiento fue exitoso, ya que el rendimiento en todos los nodos del clúster de trabajo superó el 70% en todos los nodos con respecto al rendimiento alcanzado en el clúster de Vultr.

La medición del rendimiento de entrada y salida de red será realizada utilizando la herramienta de línea de comandos *iperf*. La prueba consiste en medir la velocidad de transferencia de datos alcanzada en *gigabits* por segundo al enviar datos de un nodo a otro durante 5 segundos. El tamaño de datos transferidos variara dependiendo de la velocidad alcanzada. Se realizarán tres repeticiones del experimento desde cada nodo del clúster de Vultr. Y tres repeticiones del experimento desde cada nodo del clúster de trabajo de MPICH.

Los resultados de las repeticiones para el clúster de Vultr y el clúster de trabajo se presentan en las tablas 6.7 y 6.5 respectivamente. En estas tablas las columnas T1,T2 y T3 contienen las velocidades de transferencia alcanzadas en cada uno de las repeticiones por los diferentes nodos. Las columna T-De contiene la desviación estándar de las repeticiones. La columna T-Avg contiene la media de velocidad de transferencia alcanzada en cada nodo respectivamente.

La tabla 6.9 muestra el rendimiento de velocidad de transferencia de red de todos los nodos del clúster de trabajo con respecto al rendimiento alcanzado en los nodos del clúster de Vultr. Las columna T-Avg-WN muestra la velocidad media de transferencia alcanzada en cada uno de los nodos en el clúster de Vultr. La columnas T-Avg-CT muestra la velocidad media de transerencia alcanzada en cada uno de los nodos en el clúster de trabajo. La columna Rendimiento-CT muestra el rendimiento de velocidad de transferencia obtenido en los nodos del clúster de trabajo con respecto al rendimiento obtenido en el clúster de Vultr.

Tabla 6.7: Tabla de resultados E/S red en *worker nodes*

Nodo	T1	T2	T3	T-De	T-Avg
1	5,37	4,19	4,87	0,4836	4,8100
2	4,54	4,7	4,32	0,1558	4,5200
3	4,14	3,91	4,65	0,3092	4,2333
4	3,77	4,58	4,28	0,3344	4,2100
5	4,18	3,62	3,85	0,2298	3,8833
6	3,57	3,08	3,9	0,3369	3,5167
7	4,02	3,79	3,54	0,1960	3,7833
8	4,2	3,6	3,6	0,2828	3,8000

Tabla 6.8: Tabla de resultados E/S red en clúster de trabajo

Nodo	**T1**	**T2**	**T3**	**T-De**	**T-Avg**
1	4,97	4,94	5,43	0,2243	5,1133
2	4,62	4,58	4,71	0,0544	4,6367
3	4,61	4,51	4,41	0,0816	4,5100
4	4,69	5,25	4,8	0,2423	4,9133
5	4,96	4,85	5	0,0634	4,9367
6	4,96	4,73	4,56	0,1639	4,7500
7	4,43	5,02	4,91	0,2562	4,7867
8	4,73	4,42	4,37	0,1592	4,5067

Tabla 6.9: Tabla comparativa del rendimiento de E/S en red

Nodo	**T-Avg-WN**	**T-Avg-CT**	**Rendimiento-CT**
1	4,8100	5,1133	106,3063
2	4,5200	4,6367	102,5811
3	4,2333	4,5100	106,5354
4	4,2100	4,9133	116,7063
5	3,8833	4,9367	127,1245
6	3,5167	4,7500	135,0711
7	3,7833	4,7867	126,5198
8	3,8000	4,5067	118,5965

Si se analiza la desviación estándar en términos del coeficiente de variación se tiene que el valor máximo para la velocidad de transferencia fue del 10.05% en los *worker nodes* y del 5.35% en el clúster de trabajo. Estos valores indican que la media es representativa del conjunto de datos es decir es un conjunto de datos homogéneo.

El resultado del rendimiento de transferencia de datos por la red utilizando los nodos del clúster de trabajo es sumamente bueno. Según las pruebas realizadas el rendimiento supero el rendimiento obtenido en los *worker nodes* en todos los casos. Una explicación para este fenómeno podría ser que las pruebas se realizaron con unas horas de diferencia y que la red interna del proveedor de servicios en la nube utilizado se encontrara saturada en el momento de las pruebas en los *worker nodes*. Otra explicación posible es que el CNI de Kubernetes, es este caso Calico en modo *Cross-subnet* transmita los paquetes de red de una forma más eficiente.

El resultado del experimento de la medición del rendimiento de la velocidad de transferencia de red fue exitoso ya que el rendimiento en todos los nodos del clúster de trabajo supero el 70% en todos los nodos con respecto al rendimiento alcanzado en el clúster de Vultr.

6.4 Rendimiento de la ejecución de aplicaciones

Para la medición del rendimiento de la ejecución de aplicaciones utilizando el método propuesto y los *benchmarks* de Mantevo será utilizado el ambiente creado en la sección 5.3 del capítulo de implementación. Además será utilizado un clúster de trabajo basado en la imagen de ambiente auto-contenido de MPICH utilizando el archivo YAML disponible en el repositorio de código de ejemplos para el operador [28]. La especificación del archivo YAML puede ser consultada en el anexo 10(8.10).

Para cada *benchmark* se evaluara escalamiento fuerte donde se medirá tiempo de ejecución, *speedUp* y eficiencia. Y escalamiento débil dónde se medirá tiempo de ejecución y eficiencia. Los resultados serán mostrados de manera tabular y gráfica. En cada benchmarks serán realizadas 3 repeticiones con diferente número de *ranks*, y el *speedUp* y eficiencia utilizando N *ranks* se calculará utilizando la media del tiempo de ejecución obtenido en las 3 repeticiones.

El anexo 16(8.16) muestra los detalles de la instalación de MPICH en el clúster de Vultr y en el clúster de trabajo. Para cada *benchmark* se realizaran las mediciones primero en el clúster de Vultr. Luego se reiniciarán todos los nodos. Y se realizaran las mediciones en el clúster de trabajo. Esto con el fin de que ambas ejecuciones se ejecuten en iniciales condiciones similares.

Para el escalamiento fuerte se presentarán dos tablas por *benchmark*. La primera contendrá las columnas T1,T2 y T3 con los tiempos de ejecución de las repeticiones por número de *ranks*. La columna T-De contendrá la desviación estándar del tiempo de ejecución por número de *ranks* y la columna T-Avg que muestra la media del tiempo de ejecución obtenido por número de *ranks*. La segunda tabla contendrá la columna T-Avg que muestra la media del tiempo de ejecución obtenido por número de *ranks* y las columnas que muestran el *SpeedUp* y eficiencia alcanzada por número de *ranks*.

Para el escalamiento débil se presentarán dos tablas por *benchmark*. La primera contendrá las columnas T1,T2 y T3 con los tiempos de ejecución de las repeticiones por número de *ranks*, la columna T-De con la desviación estándar del tiempo de ejecución por número de *ranks*, la columna T-Avg que muestra la media del tiempo de ejecución obtenido por número de *ranks* y la columna T-Size que muestra el tamaño del problema utilizado por número de *ranks*. La segunda tabla contendrá la columna T-Avg que muestra la media del tiempo de ejecución obtenido por número de *ranks*, la columna de eficiencia alcanzada por número de *ranks* y la columna T-Size que muestra el tamaño del problema utilizado por número de *ranks*.

La tabla comparativa muestra el rendimiento obtenido en el clúster de trabajo con respecto al clúster de Vultr. La columna T-Avg-WN muestra la media del tiempo de ejecución obtenido en el clúster de Vultr por número de *ranks*. La columna T-Avg-CT muestra la media del tiempo de ejecución obtenido en el clúster de Trabajo por número de *ranks*. La columna R-CT muestra el rendimiento porcentual del clúster de trabajo con respecto al rendimiento obtenido en el clúster de Vultr por número de *ranks*.

Todos los resultados de los experimentos incluyendo la salida de consola de cada comando serán guardados en las carpetas `HPC-Experiments-Vultr` y `HPC-Experiments-K8s` del repositorio de código de resultados de experimentos para el operador[27].

El código fuente utilizado para compilar los diferentes *benchmarks* se encuentra en un repositorio de Github llamado mantevo-benchmarks [48]. Este repositorio contiene una copia de los *benchmarks* utilizados en esta investigación descargados del sitio oficial del proyecto Mantevo [29].

6.4.1 MiniFE

Para este *benchmark* fue utilizada la versión 2.2.0 de referencia sin modificación y el código fue compilado utilizando el archivo de *Makefile* predeterminado. Una vez compilado el código se genera un ejecutable llamado `miniFE.x`. Este ejecutable recibe como argumentos las dimensiones nx, ny y nz las cuales determinan el tamaño del problema.

Durante el experimento el clúster de trabajo presentó un problema al ejecutar el *benchmark* con quince *ranks*. Este problema no se presentó en el clúster de Vultr. Se encontraron dos soluciones la primera era reducir el tamaño del problema y la segunda era ejecutar el *benchmark* con catorce *ranks* en vez de quince. Se optó por la segunda con el fin de no reducir el tamaño del problema.

MiniFE prueba escalamiento fuerte

La prueba de escalamiento fuerte fue realizada utilizando un tamaño de problema de 50 millones mediante las dimensiones nx=1000, ny=1000 y nz=50. Las tablas 6.10 y 6.11 muestran los resultados de tiempo de ejecución obtenidos en las diferentes repeticiones por número de *ranks* en el clúster de Vultr y el clúster de trabajo respectivamente.

Tabla 6.10: Tiempos de ejecución MiniFE en clúster de Vultr

Ranks Vultr	T1	T2	T3	T-De	T-Avg
1	249,6800	245,3630	246,1660	1,8747	247,0697
14	32,6879	33,3092	30,7608	1,0850	32,2526
30	14,9999	16,2430	16,9231	0,7963	16,0553
45	13,4663	12,8001	13,2855	0,2813	13,1840
60	8,7532	9,4699	9,7432	0,4175	9,3221
75	7,9391	8,2799	8,0799	0,1398	8,0996
90	8,2126	8,2517	8,9698	0,3481	8,4780
105	7,3116	6,9324	7,2388	0,1643	7,1609
120	5,4263	5,2723	5,6581	0,1586	5,4522

Tabla 6.11: Tiempos de ejecución MiniFE en clúster de Kubernetes

Ranks K8S	T1	T2	T3	T-De	T-Avg
1	248,6180	245,9500	245,5600	1,3590	246,7093
14	26,9905	27,3912	26,0876	0,5452	26,8231
30	16,3065	16,7862	15,9196	0,3545	16,3374
45	14,2526	14,2982	15,5650	0,6082	14,7053
60	8,7001	9,8262	9,5971	0,4859	9,3745
75	9,8458	8,8999	8,7388	0,4883	9,1615
90	10,3498	8,8160	8,2416	0,8899	9,1358
105	7,6462	7,8536	6,9800	0,3727	7,4933
120	5,2330	5,7056	5,4514	0,1931	5,4633

Si se analiza la desviación estándar en términos del coeficiente de variación se tiene que el valor máximo para el tiempo de ejecución fue del 4.95% en los *worker nodes* y del 9.74% en el clúster de trabajo. Estos valores indican que la media es representativa del conjunto de datos es decir es un conjunto de datos homogéneo. Además la variación fue menor en los *worker nodes*.

La tabla comparativa 6.12 muestra el rendimiento alcanzado del clúster de trabajo con respecto al clúster de Vultr con respecto al tiempo medio de ejecución por número de *ranks*.

Tabla 6.12: Escalamiento fuerte comparativa de rendimiento MiniFE

Ranks	T-Avg-WN	T-Avg-CT	R-CT
1	247,0697	246,7093	100,1458
14	32,2526	26,8231	116,8344
30	16,0553	16,3374	98,2430
45	13,1840	14,7053	88,4610
60	9,3221	9,3745	99,4382
75	8,0996	9,1615	86,8895
90	8,4780	9,1358	92,2417
105	7,1609	7,4933	95,3591
120	5,4522	5,4633	99,7971

Analizando el rendimiento de los nodos del clúster de trabajo con respecto al rendimiento obtenido en los *worker nodes*. La media alcanzada de rendimiento fue del 97.48% siendo el valor mínimo 86%. El rendimiento en el caso de este *benchmark* no se ve afectado por el número de nodos utilizados. Esto se puede corroborar si se compara el rendimiento del 99.79% utilizando 120 ranks(8 nodos) contra otros resultados con menor número de ranks donde el rendimiento es menor.

Las tablas 6.13 y 6.14 muestran los resultados obtenidos de *speedup* y eficiencia utilizando

el tiempo medio de ejecución por número de *ranks* en el clúster de Vultr y el clúster de trabajo respectivamente. Las métricas de *speedup* y eficiencia son valores proporcionales de acuerdo al tiempo de ejecución utilizando un solo nodo, el cual fue menor en el clúster de trabajo por lo que estas se ven influenciadas por esta variación.

Tabla 6.13: Escalamiento fuerte MiniFE en clúster de Vultr

Ranks Vultr	T-Avg	SpeedUp	Eficiencia
1	247,0697	1,0000	1,0000
14	32,2526	7,6604	0,5472
30	16,0553	15,3886	0,5130
45	13,1840	18,7402	0,4164
60	9,3221	26,5036	0,4417
75	8,0996	30,5039	0,4067
90	8,4780	29,1424	0,3238
105	7,1609	34,5025	0,3286
120	5,4522	45,3152	0,3776

Tabla 6.14: Escalamiento fuerte MiniFE en clúster de Kubernetes

Ranks K8S	T-Avg	SpeedUp	Eficiencia
1	246,7093	1,0000	1,0000
14	26,8231	9,1976	0,6570
30	16,3374	15,1009	0,5034
45	14,7053	16,7769	0,3728
60	9,3745	26,3171	0,4386
75	9,1615	26,9289	0,3591
90	9,1358	27,0047	0,3001
105	7,4933	32,9242	0,3136
120	5,4633	45,1575	0,3763

El gráfico 6.1 compara visualmente el *speedup* obtenido en los nodos del clúster de trabajo contra el *speedup* obtenido en los *worker nodes*. El eje x muestra los ranks utilizados mientras que el eje y muestra el *speedup* alcanzado. La curva de escalamiento con ambos ambientes es ascendente sin declinarse o aplanarse en ningún momento. Esto es debido a que MiniFE es un *benchmark* que escala eficientemente con gran cantidad de nodos.

Otra característica de la prueba de escalamiento fuerte es que el rendimiento alcanzado por los nodos del clúster de trabajo con 2(30 *ranks*) ,4(60 *ranks*) y 8(120 *ranks*) fue virtualmente idéntico al rendimiento alcanzado en los *worker nodes* con la misma cantidad de *ranks*.

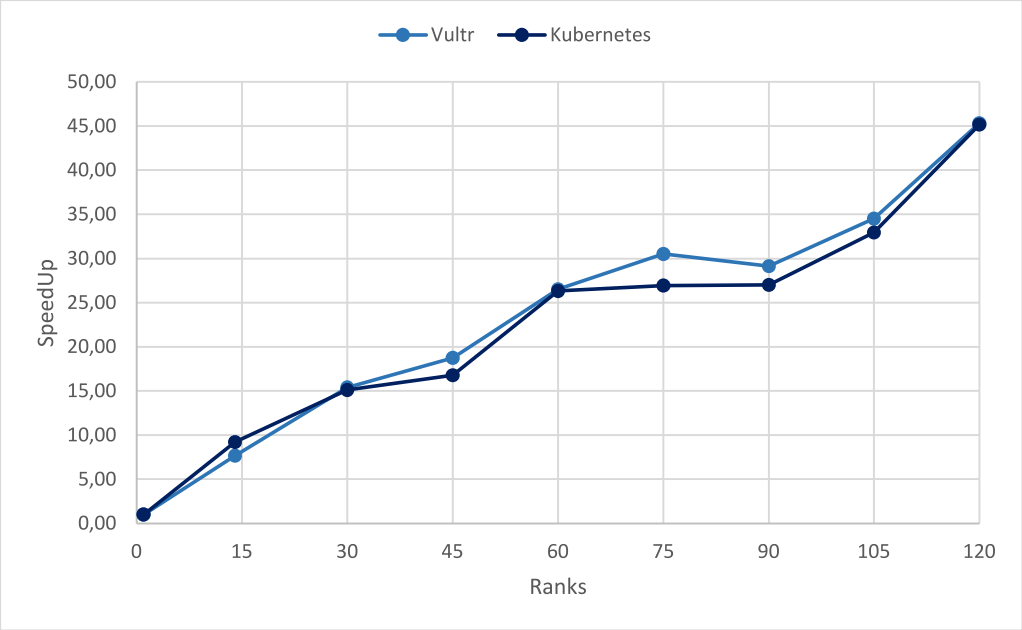

Figura 6.1: Escalamiento fuerte comparativa de *SpeedUp* MiniFE

MiniFE prueba escalamiento débil

La prueba de escalamiento débil fue realizada utilizando un tamaño base para 15 *ranks* de 20 millones mediante las dimensiones las dimensiones nx=2000, ny=1000 y nz=10. A partir de esta base se calculó un tamaño proporcional para el resto de número de *ranks*. Debido a que el ejecutable solo acepta tamaños enteros como entrada el tamaño de problema para el experimento utilizando un solo *rank* tuvo que ser reducido de 1333333,333 a 1330000.

Las tablas 6.15 y 6.16 muestran los resultados de tiempo de ejecución obtenidos en las diferentes repeticiones por número de *ranks* y tamaño de problema en el clúster de Vultr y el clúster de trabajo respectivamente.

Tabla 6.15: Tiempos de ejecución MiniFE en clúster de Vultr

Ranks Vultr	T1	T2	T3	T-De	T-Avg	P-Size
1	6,2202	5,8904	6,8129	0,3817	6,3079	1330000
14	14,5485	13,5746	14,3329	0,4177	14,1520	20000000
30	15,3205	14,7827	14,2006	0,4573	14,7679	40000000
45	21,9000	20,8490	20,6010	0,5631	21,1167	60000000
60	22,4341	19,4304	18,8678	1,5655	20,2441	80000000
75	27,7796	26,8032	26,9916	0,4229	27,1915	100000000
90	31,6845	31,4359	30,4257	0,5444	31,1820	120000000
105	34,7225	32,6968	32,4331	1,0228	33,2841	140000000
120	28,0903	28,6460	30,1192	0,8561	28,9518	160000000

Tabla 6.16: Tiempos de ejecución MiniFE en clúster de Kubernetes

Ranks K8S	T1	T2	T3	T-De	T-Avg	P-Size
1	5,6969	5,6902	5,6706	0,0112	5,6859	1330000
14	14,9189	13,6419	15,0222	0,6277	14,5277	20000000
30	19,2396	18,2026	17,4177	0,7462	18,2866	40000000
45	24,8207	26,2889	26,4483	0,7326	25,8526	60000000
60	23,2583	20,8737	21,5013	1,0092	21,8778	80000000
75	31,1844	31,7096	30,6953	0,4142	31,1964	100000000
90	36,1473	34,7974	36,8056	0,8359	35,9168	120000000
105	36,0852	35,5579	36,1042	0,2532	35,9158	140000000
120	27,5748	26,0005	29,7704	1,5460	27,7819	160000000

Si se analiza la desviación estándar en términos del coeficiente de variación se tiene que el valor máximo para el tiempo de ejecución fue del 7.73% en los *worker nodes* y del 5.64% en el clúster de trabajo. Estos valores indican que la media es representativa del conjunto de datos es decir es un conjunto de datos homogéneo. Además la variación fue menor en el *clúster de trabajo*.

La tabla comparativa 6.17 muestra el rendimiento alcanzado del clúster de trabajo con respecto al clúster de Vultr con respecto al tiempo medio de ejecución por número de *ranks*.

Tabla 6.17: Escalamiento débil comparativa de rendimiento MiniFE

Ranks	T-Avg-WN	T-Avg-CT	R-CT
1	6,3079	5,6859	109,8607
14	14,1520	14,5277	97,3455
30	14,7679	18,2866	76,1734
45	21,1167	25,8526	77,5724
60	20,2441	21,8778	91,9302
75	27,1915	31,1964	85,2712
90	31,1820	35,9168	84,8158
105	33,2841	35,9158	92,0934
120	28,9518	27,7819	104,0410

Analizando el rendimiento de los nodos del clúster de trabajo con respecto al rendimiento obtenido en los *worker nodes*. La media alcanzada de rendimiento fue del 91.01% siendo el valor mínimo 76%. El rendimiento en el caso de este *benchmark* no se ve afectado por el número de nodos utilizados. Esto se puede corroborar si se compara el rendimiento del 104.04% utilizando 120 ranks(8 nodos) contra otros resultados con menor número de ranks donde el rendimiento es menor.

Las tablas 6.18 y 6.19 muestran los resultados obtenidos de eficiencia utilizando el tiempo

medio de ejecución por número de *ranks* y tamaño de problema en el clúster de Vultr y el clúster de trabajo respectivamente. La métrica de eficiencia es un valor proporcional de acuerdo al tiempo de ejecución utilizando un solo nodo, el cual fue menor en el clúster de trabajo por lo que esta se ve influenciada por esta variación.

Tabla 6.18: Escalamiento débil MiniFE en clúster de Vultr

Ranks Vultr	**T-Avg**	**Eficiencia**	**P-Size**
1	6,3079	1,0000	1330000
14	14,1520	0,4457	20000000
30	14,7679	0,4271	40000000
45	21,1167	0,2987	60000000
60	20,2441	0,3116	80000000
75	27,1915	0,2320	100000000
90	31,1820	0,2023	120000000
105	33,2841	0,1895	140000000
120	28,9518	0,2179	160000000

Tabla 6.19: Escalamiento débil MiniFE en clúster de Kubernetes

Ranks K8S	**T-Avg**	**Eficiencia**	**P-Size**
1	5,6859	1,0000	1330000
14	14,5277	0,3914	20000000
30	18,2866	0,3109	40000000
45	25,8526	0,2199	60000000
60	21,8778	0,2599	80000000
75	31,1964	0,1823	100000000
90	35,9168	0,1583	120000000
105	35,9158	0,1583	140000000
120	27,7819	0,2047	160000000

El gráfico 6.2 compara visualmente la eficiencia obtenida en los nodos del clúster de trabajo contra la eficiencia obtenida en los *worker nodes* utilizando escalamiento débil. El eje x muestra los ranks utilizados mientras que el eje y muestra la eficiencia alcanzada. La curva de escalamiento con ambos ambientes es descendente y llega a punto a partir de los 90*ranks* donde la eficiencia vuelva a estabilizarse indistintamente del tamaño del problema.

Nuevamente en la prueba de escalamiento débil el rendimiento alcanzado utilizando 4(60 *ranks*) y 8(120 *ranks*) nodos en el clúster de trabajo en comparación con el rendimiento obtenido en los *worker nodes* con la misma cantidad de *ranks* es virtualmente idéntico.

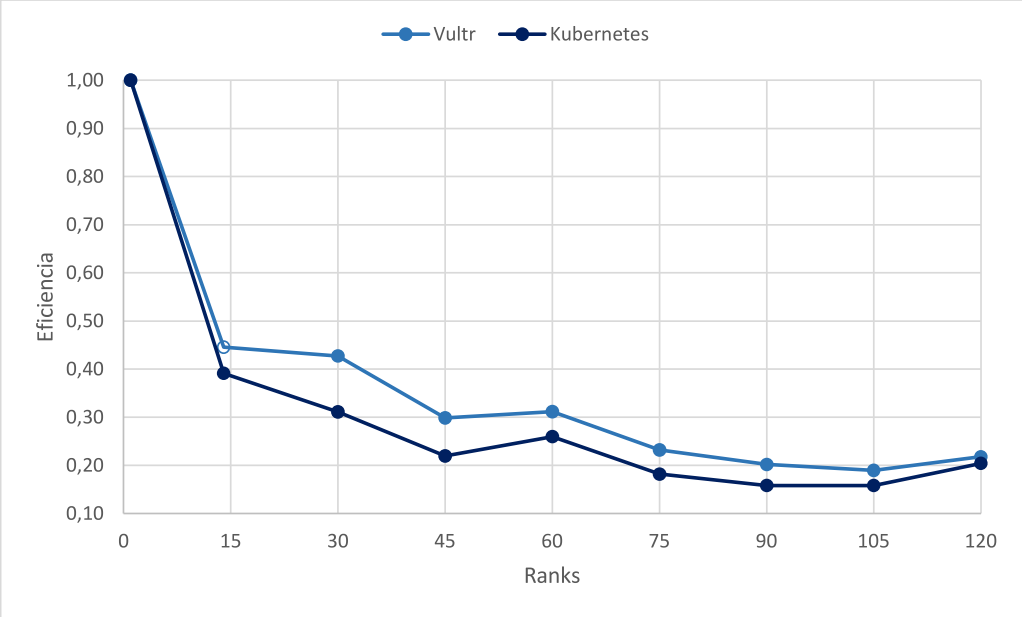

Figura 6.2: Escalamiento débil comparativa de eficiencia MiniFE

6.4.2 MiniMD

Para este *benchmark* fue utilizada la versión de referencia sin modificación y el código fue compilado utilizando el archivo de *Makefile* predeterminado con el argumento *openmpi* para compilar una versión compatible con MPI. Una vez compilado el código se genera un ejecutable llamado `miniMD_openmpi`. Este ejecutable recibe como argumento un archivo llamado `in.lj.miniMD` el cual especifica los parámetros a utilizar para el *benchmark*. Todos los parámetros del archivo `in.lj.miniMD` se mantuvieron constantes para el *benchmark* a excepción del tamaño del problema.

Durante el experimento el clúster de trabajo presentó un problema al ejecutar el *benchmark* con quince *ranks*. Este problema no se presentó en el clúster de Vultr. Se encontraron dos soluciones la primera era reducir el tamaño del problema y la segunda era ejecutar el *benchmark* con catorce *ranks* en vez de quince. Se optó por la segunda con el fin de no reducir el tamaño del problema.

MiniMD prueba escalamiento fuerte

La prueba de escalamiento fuerte fue realizada utilizando un tamaño de problema de 737280 mediante las dimensiones x=96, y=96 y z=80. Las tablas 6.20 y 6.21 muestran los resultados de tiempo de ejecución obtenidos en las diferentes repeticiones por número de *ranks* en el clúster de Vultr y el clúster de trabajo respectivamente.

Tabla 6.20: Tiempos de ejecución MiniMD en clúster de Vultr

Ranks Vultr	T1	T2	T3	T-De	T-Avg
1	127,1020	126,8920	130,2610	1,5411	128,0850
14	10,7360	10,8860	10,6230	0,1077	10,7483
30	5,6660	5,4600	5,3560	0,1288	5,4940
45	3,8350	4,0320	3,9420	0,0805	3,9363
60	3,1870	3,0820	3,0420	0,0611	3,1037
75	2,6810	2,6830	2,7100	0,0132	2,6913
90	2,4070	2,5070	2,4920	0,0440	2,4687
105	2,2960	2,2910	2,1190	0,0823	2,2353
120	2,2830	2,1160	2,1000	0,0828	2,1663

Tabla 6.21: Tiempos de ejecución MiniMD en clúster de Kubernetes

Ranks K8S	T1	T2	T3	T-De	T-Avg
1	124,6020	127,6840	125,9180	1,2627	126,0680
14	9,9520	9,9910	9,6690	0,1435	9,8707
30	6,6140	7,0560	7,6640	0,4304	7,1113
45	5,3460	5,4530	5,1070	0,1446	5,3020
60	3,9220	4,0500	3,6410	0,1708	3,8710
75	2,9480	3,0020	3,3930	0,1983	3,1143
90	2,6670	2,6890	2,7180	0,0209	2,6913
105	2,8360	2,6800	2,6270	0,0887	2,7143
120	2,4650	2,4100	2,3910	0,0314	2,4220

Si se analiza la desviación estándar en términos del coeficiente de variación se tiene que el valor máximo para el tiempo de ejecución fue del 3.82% en los *worker nodes* y del 6.36% en el clúster de trabajo. Estos valores indican que la media es representativa del conjunto de datos es decir es un conjunto de datos homogéneo. Además la variación fue menor en los *worker nodes*.

Analizando el rendimiento de los nodos del clúster de trabajo con respecto al rendimiento obtenido en los *worker nodes*. La media alcanzada de rendimiento fue del 84.76% siendo el valor mínimo 65.30%. Todos los resultados de escalamiento fuerte para el *bechnmark* MiniMD utilizando 30 *ranks* o más o dicho de otra manera utilizando más de 1 nodo fueron menores al rendimiento obtenido en los *worker nodes* con igual número de nodos.

La tabla 6.22 comparativa muestra el rendimiento alcanzado del clúster de trabajo con respecto al clúster de Vultr con respecto al tiempo medio de ejecución por número de *ranks*.

Tabla 6.22: Escalamiento fuerte comparativa de rendimiento MiniMD

Ranks	T-Avg-WN	T-Avg-CT	R-CT
1	128,0850	126,0680	101,5747
14	10,7483	9,8707	108,1656
30	5,4940	7,1113	70,5618
45	3,9363	5,3020	65,3061
60	3,1037	3,8710	75,2766
75	2,6913	3,1143	84,2829
90	2,4687	2,6913	90,9803
105	2,2353	2,7143	78,5714
120	2,1663	2,4220	88,1982

La prueba realizada con 3 nodos(45 *ranks*) es atípica al resto de pruebas realizadas no solo en este *benchmark*, sino, en comparación con todas las mediciones realizadas en los demás *benchmarks*. El rendimiento alcanzado en el clúster de trabajo fue del 65.30% con respecto al rendimiento en los *worker nodes* con la misma cantidad de nodos fue la única prueba abajo del 70% de entre todas las mediciones realizadas.

Las tablas 6.23 y 6.24 muestran los resultados obtenidos de *speedup* y eficiencia utilizando el tiempo medio de ejecución por número de *ranks* en el clúster de Vultr y el clúster de trabajo respectivamente. Las métricas de *speedup* y eficiencia son valores proporcionales de acuerdo al tiempo de ejecución utilizando un solo nodo, el cual fue menor en el clúster de trabajo por lo que estas se ven influenciadas por esta variación.

El gráfico 6.3 compara visualmente el *speedup* obtenido en los nodos del clúster de trabajo contra el *speedup* obtenido en los *worker nodes*. El eje x muestra los ranks utilizados mientras que el eje y muestra el *speedup* alcanzado. La curva de escalamiento con ambos ambientes es ascendente sin declinarse o aplanarse en ningún momento. Esto es debido a que MiniMD es un *benchmark* que escala eficientemente con gran cantidad de nodos.

Tabla 6.23: Escalamiento fuerte MiniMD en clúster de Vultr

Ranks Vultr	T-Avg	SpeedUp	Eficiencia
1	128,0850	1,0000	1,0000
14	10,7483	11,9167	0,8512
30	5,4940	23,3136	0,7771
45	3,9363	32,5392	0,7231
60	3,1037	41,2689	0,6878
75	2,6913	47,5917	0,6346
90	2,4687	51,8843	0,5765
105	2,2353	57,3002	0,5457
120	2,1663	59,1253	0,4927

Tabla 6.24: Escalamiento fuerte MiniMD en clúster de Kubernetes

Ranks K8S	T-Avg	SpeedUp	Eficiencia
1	126,0680	1,0000	1,0000
14	9,8707	12,7720	0,9123
30	7,1113	17,7278	0,5909
45	5,3020	23,7774	0,5284
60	3,8710	32,5673	0,5428
75	3,1143	40,4799	0,5397
90	2,6913	46,8422	0,5205
105	2,7143	46,4453	0,4423
120	2,4220	52,0512	0,4338

Analizando el gráfico y los datos de escalamiento se podría decir que para el escalamiento fuerte del *benchmark* MiniMD el clúster de trabajo proporciona un rendimiento inferior con respecto al rendimiento en los *worker nodes* al utilizar más de un nodo. Y en ningún fue posible obtener un rendimiento virualmente idéntico como ocurrió en el caso del *benchmark* MiniFE.

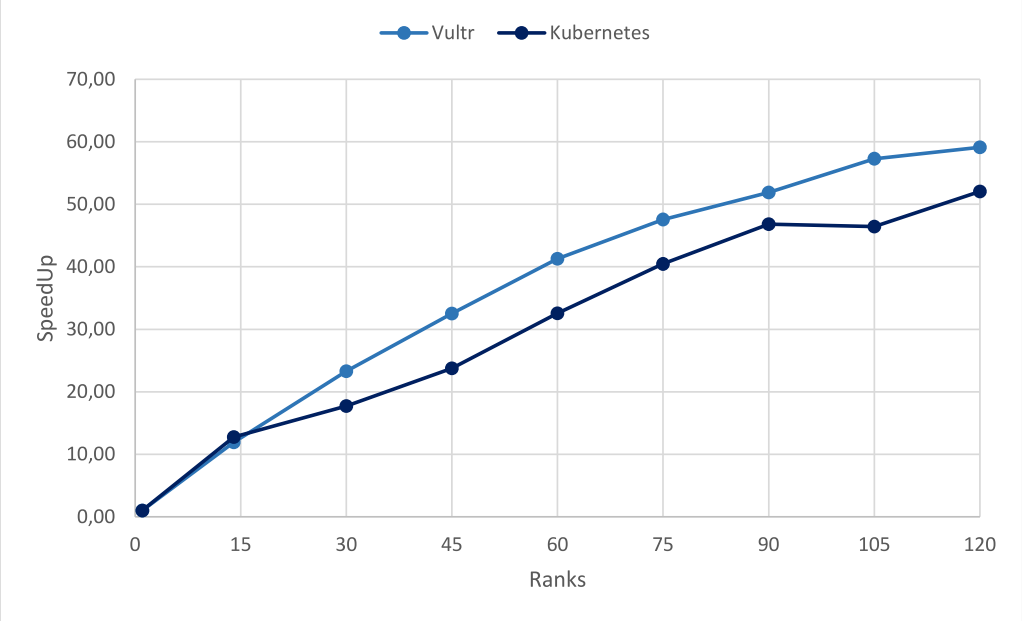

Figura 6.3: Escalamiento fuerte comparativa de *SpeedUp* MiniMD

MiniMD prueba escalamiento débil

La prueba de escalamiento débil fue realizada utilizando un tamaño base para 15 *ranks* de 2097152 mediante las dimensiones las dimensiones x=256, y=128 y z=64. A partir de esta base se calculó un tamaño proporcional para el resto de número de *ranks*. Debido a que

el ejecutable solo acepta tamaños enteros como entrada el tamaño de problema para el experimento utilizando un solo *rank* tuvo que ser reducido de 139810,1333 a 139264.

Las tablas 6.25 y 6.26 muestran los resultados de tiempo de ejecución obtenidos en las diferentes repeticiones por número de *ranks* y tamaño de problema en el clúster de Vultr y el clúster de trabajo respectivamente.

Tabla 6.25: Tiempos de ejecución MiniMD en clúster de Vultr

Ranks Vultr	**T1**	**T2**	**T3**	**T-De**	**T-Avg**	**P-Size**
1	23,5910	24,8460	23,5130	0,6108	23,9833	139264
14	28,7840	32,5830	29,3410	1,6751	30,2360	2097152
30	27,9280	28,3300	28,2040	0,1679	28,1540	4194304
45	28,3010	29,4500	28,0820	0,6000	28,6110	6291456
60	30,2310	30,9480	30,7160	0,2987	30,6317	8388608
75	32,2620	30,1250	31,6630	0,9001	31,3500	10485760
90	32,6060	32,4130	35,9060	1,6031	33,6417	12582912
105	30,7470	31,6060	30,7460	0,4052	31,0330	14680064
120	32,1050	31,4410	33,8800	1,0296	32,4753	16777216

Tabla 6.26: Tiempos de ejecución MiniMD en clúster de Kubernetes

Ranks K8S	**T1**	**T2**	**T3**	**T-De**	**T-Avg**	**P-Size**
1	23,2830	22,8390	22,9900	0,1843	23,0373	139264
14	28,1920	28,2290	28,1800	0,0209	28,2003	2097152
30	33,1060	31,2820	32,4250	0,7526	32,2710	4194304
45	31,3240	38,4610	34,0990	2,9376	34,6280	6291456
60	34,2190	31,6810	33,1720	1,0414	33,0240	8388608
75	32,2720	32,2080	33,3120	0,5060	32,5973	10485760
90	30,3410	29,7430	30,5340	0,3367	30,2060	12582912
105	30,7780	30,1060	31,1280	0,4241	30,6707	14680064
120	31,4690	31,6480	30,7530	0,3867	31,2900	16777216

Si se analiza la desviación estándar en términos del coeficiente de variación se tiene que el valor máximo para el tiempo de ejecución fue del 5.54% en los *worker nodes* y del 8.48% en el clúster de trabajo. Estos valores indican que la media es representativa del conjunto de datos es decir es un conjunto de datos homogéneo. Además la variación fue menor en el *clúster de trabajo*.

La tabla comparativa 6.27 muestra el rendimiento alcanzado del clúster de trabajo con respecto al clúster de Vultr con respecto al tiempo medio de ejecución por número de *ranks*.

Tabla 6.27: Escalamiento débil comparativa de rendimiento MiniMD

Ranks	T-Avg-WN	T-Avg-CT	R-CT
1	23,9833	23,0373	103,9444
14	30,2360	28,2003	106,7326
30	28,1540	32,2710	85,3769
45	28,6110	34,6280	78,9696
60	30,6317	33,0240	92,1900
75	31,3500	32,5973	96,0213
90	33,6417	30,2060	110,2125
105	31,0330	30,6707	101,1676
120	32,4753	31,2900	103,6499

Al Analizar el rendimiento de los nodos del clúster de trabajo con respecto al rendimiento obtenido en los *worker nodes*. La media alcanzada de rendimiento fue del 97.58% siendo el valor mínimo 78.96%. El rendimiento en el caso de este *benchmark* utilizando escalamiento débil no se ve afectado por el número de nodos utilizados. Esto se puede corroborar si se compara el rendimiento del 103.64% utilizando 120 ranks(8 nodos) contra otros resultados con menor número de ranks donde el rendimiento es menor.

Las tablas 6.28 y 6.29 muestran los resultados obtenidos de eficiencia utilizando el tiempo medio de ejecución por número de *ranks* y tamaño de problema en el clúster de Vultr y el clúster de trabajo respectivamente.

El gráfico **??** compara visualmente la eficiencia obtenida en los nodos del clúster de trabajo contra la eficiencia obtenida en los *worker nodes* utilizando escalamiento débil. El eje x muestra los ranks utilizados mientras que el eje y muestra la eficiencia alcanzada. La curva de escalamiento con ambos ambientes es descendente y llega a punto a partir de los 90*ranks* donde la eficiencia vuelva a estabilizarse indistintamente del tamaño del problema.

Tabla 6.28: Escalamiento débil MiniMD en clúster de Vultr

Ranks Vultr	T-Avg	Eficiencia	P-Size
1	23,9833	1,0000	139264
14	30,2360	0,7932	2097152
30	28,1540	0,8519	4194304
45	28,6110	0,8383	6291456
60	30,6317	0,7830	8388608
75	31,3500	0,7650	10485760
90	33,6417	0,7129	12582912
105	31,0330	0,7728	14680064
120	32,4753	0,7385	16777216

Tabla 6.29: Escalamiento débil MiniMD en clúster de Kubernetes

Ranks K8S	T-Avg	Eficiencia	P-Size
1	23,0373	1,0000	139264
14	28,2003	0,8169	2097152
30	32,2710	0,7139	4194304
45	34,6280	0,6653	6291456
60	33,0240	0,6976	8388608
75	32,5973	0,7067	10485760
90	30,2060	0,7627	12582912
105	30,6707	0,7511	14680064
120	31,2900	0,7363	16777216

En el gráfico 6.4 se aprecia como la eficiencia y por en de el rendimiento alcanzado por los nodos del clúster de trabajo utilizando 2 nodos(30 *ranks*), 3 nodos(45 *ranks*), 4 nodos(60 *ranks*) y 5 nodos(75 *ranks*) fue menor comparado con la eficiencia alcanzada con los *worker nodes* con igual número de *ranks*. Sin embargo, al utilizar 6 nodos(75 *ranks*), 7 nodos(105 *ranks*) y 8 nodos(120 *ranks*) el rendimiento obtenido por el clúster de trabajo fue virtualmente idéntico al rendimiento obtenido en los *worker nodes*.

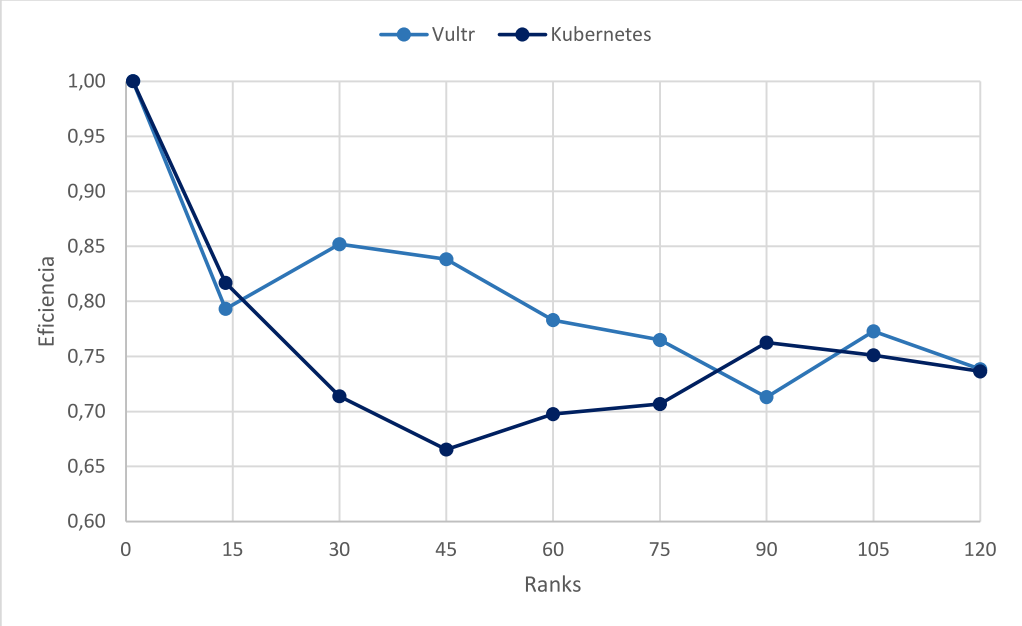

Figura 6.4: Escalamiento débil comparativa de eficiencia MiniMD

6.4.3 MiniXyce

Para este *benchmark* fue utilizada la versión 1.0 de referencia sin modificación y el código fue compilado utilizando el archivo de *Makefile* predeterminado. Una vez compilado el

código se genera un ejecutable llamado `miniXyce_ref.x`. Este ejecutable recibe como argumento la ruta de un archivo con la especificación del circuito de prueba.

Los archivos para los diferentes experimentos fueron creados mediante el *script* de generación de circuitos `RC_ladder.pl` el cual es parte del código fuente del *benchmark*. Para ejecutar el archivo es necesario tener una máquina que pueda capaz de compilar y ejecutar programa desarrollados con el lenguaje de programación *perl*. El *script* `RC_ladder.pl` recibe como argumento un número entero N y crea un archivo con un circuito de prueba el cual consiste en una escalera RC de N etapas.

MiniXyce prueba escalamiento fuerte

La prueba de escalamiento fuerte fue realizada utilizando un archivo de prueba con un circuito 100000 etapas. Las tablas 6.30 y 6.31 muestran los resultados de tiempo de ejecución obtenidos en las diferentes repeticiones por número de *ranks* en el clúster de Vultr y el clúster de trabajo respectivamente.

Tabla 6.30: Tiempos de ejecución MiniXyce en clúster de Vultr

Ranks Vultr	T1	T2	T3	T-De	T-Avg
1	36,6243	35,9556	34,8254	0,7424	35,8018
15	18,2054	18,5332	17,2791	0,5311	18,0059
30	18,7346	18,6926	19,1493	0,2061	18,8588
45	21,0021	21,2050	20,9212	0,1194	21,0428
60	22,6540	21,4747	21,5474	0,5396	21,8920
75	23,1360	22,9296	23,4642	0,2201	23,1766
90	24,2163	23,8795	23,8222	0,1739	23,9727
105	25,6739	25,7150	25,9897	0,1402	25,7929
120	25,1991	25,1817	24,0018	0,5604	24,7942

Tabla 6.31: Tiempos de ejecución MiniXyce en clúster de Kubernetes

Ranks K8S	T1	T2	T3	T-De	T-Avg
1	36,6785	37,0539	35,8781	0,4904	36,5368
15	17,6221	17,5117	18,1457	0,2765	17,7598
30	20,4866	20,4800	20,0286	0,2144	20,3317
45	23,1020	22,6707	23,1809	0,2242	22,9845
60	23,7935	23,3404	23,6567	0,1898	23,5969
75	26,2955	25,7864	26,3580	0,2560	26,1466
90	27,5419	27,0614	26,1394	0,5819	26,9142
105	28,6655	29,6754	30,0017	0,5688	29,4475
120	29,1339	28,5559	27,5311	0,6628	28,4070

Si se analiza la desviación estándar en términos del coeficiente de variación se tiene que el valor máximo para el tiempo de ejecución fue del 2.94% en los *worker nodes* y del 2.33% en el clúster de trabajo. Estos valores indican que la media es representativa del conjunto de datos es decir es un conjunto de datos homogéneo. Además la variación fue menor en el *clúster de trabajo*.

La tabla comparativa 6.32 muestra el rendimiento alcanzado del clúster de trabajo con respecto al clúster de Vultr con respecto al tiempo medio de ejecución por número de *ranks*.

Tabla 6.32: Escalamiento fuerte comparativa de rendimiento MiniXyce

Ranks	**T-Avg-WN**	**T-Avg-CT**	**R-CT**
1	35,8018	36,5368	97,9468
14	18,0059	17,7598	101,3666
30	18,8588	20,3317	92,1899
45	21,0428	22,9845	90,7723
60	21,8920	23,5969	92,2125
75	23,1766	26,1466	87,1852
90	23,9727	26,9142	87,7295
105	25,7929	29,4475	85,8307
120	24,7942	28,4070	85,4290

Analizando el rendimiento de los nodos del clúster de trabajo con respecto al rendimiento obtenido en los *worker nodes*. La media alcanzada de rendimiento fue del 91.18% siendo el valor mínimo 85.42%. El rendimiento en el caso de este *benchmark* se ve afectado conforme crece el número de nodos utilizados.

Las tablas 6.33 y 6.34 muestran los resultados obtenidos de *speedup* y eficiencia utilizando el tiempo medio de ejecución por número de *ranks* en el clúster de Vultr y el clúster de trabajo respectivamente.

Tabla 6.33: Escalamiento fuerte MiniXyce en clúster de Vultr

Ranks Vultr	**T-Avg**	**SpeedUp**	**Eficiencia**
1	35,8018	1,0000	1,0000
15	18,0059	1,9883	0,1326
30	18,8588	1,8984	0,0633
45	21,0428	1,7014	0,0378
60	21,8920	1,6354	0,0273
75	23,1766	1,5447	0,0206
90	23,9727	1,4934	0,0166
105	25,7929	1,3880	0,0132
120	24,7942	1,4440	0,0120

Tabla 6.34: Escalamiento fuerte MiniXyce en clúster de Kubernetes

Ranks K8S	T-Avg	SpeedUp	Eficiencia
1	36,5368	1,0000	1,0000
15	17,7598	2,0573	0,1372
30	20,3317	1,7970	0,0599
45	22,9845	1,5896	0,0353
60	23,5969	1,5484	0,0258
75	26,1466	1,3974	0,0186
90	26,9142	1,3575	0,0151
105	29,4475	1,2407	0,0118
120	28,4070	1,2862	0,0107

Al analizar los datos de *SpeedUp* el *benchmark* MiniXyce tiene un escalamiento realmente bajo conforme se aumenta el número de nodos, esto puede deberse a la alta dependencia del rendimiento de de E/S de red y disco. La infraestructura de *Vultr* no proporciona las velocidades necesarias para que el problema escale de forma adecuada. Sin embargo, este comportamiento se presenta tanto en los nodos del clúster de trabajo como en los *worker nodes*.

El gráfico 6.5 compara visualmente el *speedup* obtenido en los nodos del clúster de trabajo contra el *speedup* obtenido en los *worker nodes*. El eje x muestra los ranks utilizados mientras que el eje y muestra el *speedup* alcanzado. La curva de escalamiento con ambos ambientes es similar descendiendo conforme se aumenta el número de nodos.

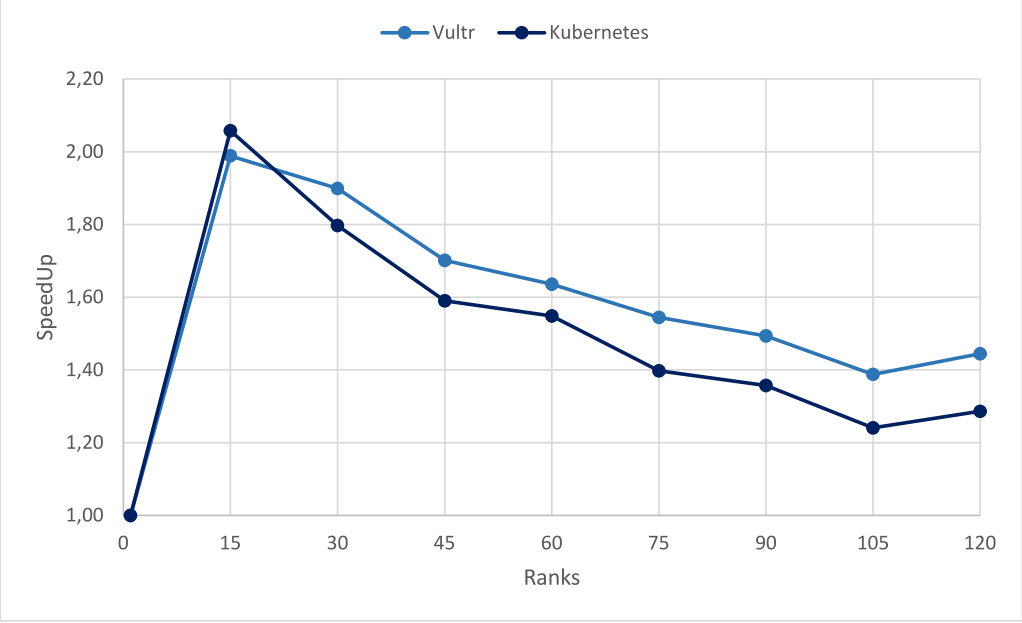

Figura 6.5: Escalamiento fuerte comparativa de *SpeedUp* MiniXyce

MiniXyce prueba escalamiento débil

La prueba de escalamiento débil fue realizada utilizando un archivo de prueba con un circuito de 64000 etapas para 15 *ranks*. A partir de esta base se calculó un tamaño proporcional para el resto de número de *ranks*. Debido a que el ejecutable solo acepta tamaños enteros como entrada el tamaño de problema para el experimento utilizando un solo *rank* tuvo que ser reducido de 4266,666667 a 4266.

La tablas 6.35 y 6.36 muestran los resultados de tiempo de ejecución obtenidos en las diferentes repeticiones por número de *ranks* y tamaño de problema en el clúster de Vultr y el clúster de trabajo respectivamente.

Tabla 6.35: Tiempos de ejecución MiniXyce en clúster de Vultr

Ranks Vultr	T1	T2	T3	T-De	T-Avg	P-Size
1	1,5065	1,4049	1,3714	0,0574	1,4276	4266
15	11,4026	11,2157	11,4376	0,0974	11,3520	64000
30	24,4460	24,5828	24,3115	0,1108	24,4468	128000
45	37,1966	37,9546	36,2862	0,6821	37,1458	192000
60	48,4308	49,7128	48,6250	0,5642	48,9229	256000
75	61,5548	59,6291	60,1496	0,8133	60,4445	320000
90	73,7098	72,6730	73,6448	0,4742	73,3425	384000
105	86,3920	84,7320	85,3090	0,6881	85,4777	448000
120	98,2989	97,4240	92,9262	2,3538	96,2164	512000

Tabla 6.36: Tiempos de ejecución MiniXyce en clúster de Kubernetes

Ranks K8S	T1	T2	T3	T-De	T-Avg	P-Size
1	1,3945	1,5274	1,3846	0,0651	1,4355	4266
15	10,9132	10,9537	11,6269	0,3273	11,1646	64000
30	24,2868	25,3883	24,6116	0,4621	24,7622	128000
45	39,4932	39,2061	38,1016	0,5999	38,9336	192000
60	50,8170	51,4560	50,1324	0,5405	50,8018	256000
75	63,2961	64,5751	63,1840	0,6310	63,6851	320000
90	76,6302	79,1115	76,0482	1,3283	77,2633	384000
105	87,8340	90,0349	92,5315	1,9190	90,1335	448000
120	109,4870	106,1240	98,9310	4,4030	104,8473	512000

Si se analiza la desviación estándar en términos del coeficiente de variación se tiene que el valor máximo para el tiempo de ejecución fue del 4.02% en los *worker nodes* y del 4.53% en el clúster de trabajo. Estos valores indican que la media es representativa del conjunto de datos es decir es un conjunto de datos homogéneo. Además la variación fue menor en los *worker nodes*.

La tabla comparativa 6.37 muestra el rendimiento alcanzado del clúster de trabajo con respecto al clúster de Vultr con respecto al tiempo medio de ejecución por número de *ranks*.

Tabla 6.37: Escalamiento débil comparativa de rendimiento MiniXyce

Ranks	T-Avg-WN	T-Avg-CT	R-CT
1	1,4276	1,4355	99,4468
14	11,3520	11,1646	101,6505
30	24,4468	24,7622	98,7096
45	37,1458	38,9336	95,1870
60	48,9229	50,8018	96,1594
75	60,4445	63,6851	94,6388
90	73,3425	77,2633	94,6542
105	85,4777	90,1335	94,5532
120	96,2164	104,8473	91,0296

Analizando el rendimiento de los nodos del clúster de trabajo con respecto al rendimiento obtenido en los *worker nodes*. La media alcanzada de rendimiento fue del 96.22% siendo el valor mínimo 91.02%. Al igual que en la prueba de escalamiento fuerte el rendimiento de este *benchmark* se ve afectado conforme crece el número de nodos utilizados.

El rendimiento en la mayoría de nodos del clúster de trabajo es muy similar al obtenido en los *worker nodes*. Sin embargo, esto puede deberse al hecho que el *benchmark* de MiniXyce depende del rendimiento de E/S de almacenamiento y red para un escalamiento eficiente.

Las tablas 6.38 y 6.39 muestran los resultados obtenidos de eficiencia utilizando el tiempo medio de ejecución por número de *ranks* y tamaño de problema en el clúster de Vultr y el clúster de trabajo respectivamente.

Tabla 6.38: Escalamiento débil MiniXyce en clúster de Vultr

Ranks Vultr	T-Avg	Eficiencia	P-Size
1	1,4276	1,0000	4266
15	11,3520	0,1258	64000
30	24,4468	0,0584	128000
45	37,1458	0,0384	192000
60	48,9229	0,0292	256000
75	60,4445	0,0236	320000
90	73,3425	0,0195	384000
105	85,4777	0,0167	448000
120	96,2164	0,0148	512000

Tabla 6.39: Escalamiento débil MiniXyce en clúster de Kubernetes

Ranks K8S	T-Avg	Eficiencia	P-Size
1	1,4355	1,0000	4266
15	11,1646	0,1286	64000
30	24,7622	0,0580	128000
45	38,9336	0,0369	192000
60	50,8018	0,0283	256000
75	63,6851	0,0225	320000
90	77,2633	0,0186	384000
105	90,1335	0,0159	448000
120	104,8473	0,0137	512000

El gráfico 6.6 compara visualmente la eficiencia obtenida en los nodos del clúster de trabajo contra la eficiencia obtenida en los *worker nodes* utilizando escalamiento débil. El eje x muestra los ranks utilizados mientras que el eje y muestra la eficiencia alcanzada. La curva de escalamiento con ambos ambientes es decrece y llega a punto a partir de los 90*ranks* donde la eficiencia es tan baja que tiende a estabilizarse indistintamente del tamaño del problema.

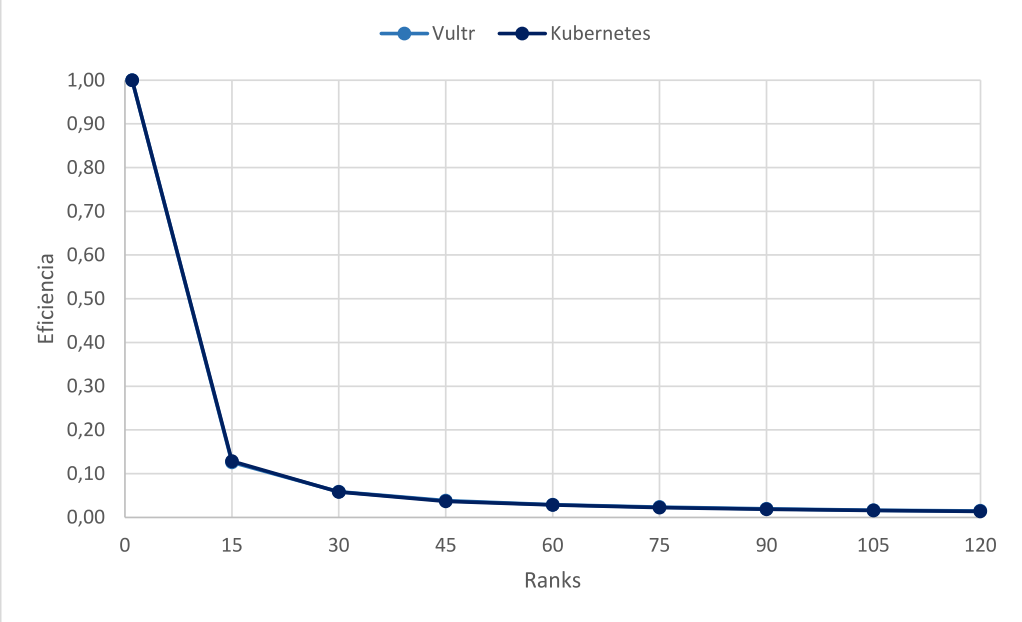

Figura 6.6: Escalamiento débil comparativa de eficiencia MiniXyce

Debido a que la prueba de escalamiento débil aumenta el tamaño del problema los requerimientos de E/S aumentarán también y el rendimiento se verá directamente afectado. Esto se evidencia en los tiempos de ejecución los cuales aumentan de manera significativa conforme aumenta el número de *ranks* y el tamaño del problema.

6.4.4 CloverLeaf

Para este *benchmark* fue utilizada la versión 1.1 de MPI sin modificación y el código fue compilado utilizando el archivo de *Makefile* predeterminado con el argumento *mpi* para compilar una versión compatible con MPI. Una vez compilado el código se genera un ejecutable llamado `clover_leaf`. Este ejecutable recibe como argumento un archivo llamado `clover.in` el cual especifica los parámetros a utilizar para el *benchmark*. Todos los parámetros del archivo `clover.in` se mantuvieron constantes para el *benchmark* a excepción del tamaño del problema.

CloverLeaf prueba escalamiento fuerte

La prueba de escalamiento fuerte fue realizada utilizando un archivo de prueba con un circuito 100000 etapas. Las tablas 6.40 y 6.41 muestran los resultados de tiempo de ejecución obtenidos en las diferentes repeticiones por número de *ranks* en el clúster de Vultr y el clúster de trabajo respectivamente.

Tabla 6.40: Tiempos de ejecución CloverLeaf en clúster de Vultr

Ranks Vultr	T1	T2	T3	T-De	T-Avg
1	320,8813	325,1644	336,3071	6,5018	327,4509
15	59,5927	63,2287	58,1490	2,1372	60,3235
30	30,7723	30,3028	35,9669	2,5666	32,3473
45	23,0382	22,5309	22,6588	0,2154	22,7426
60	16,0843	19,2466	17,8885	1,2953	17,7398
75	15,4579	13,5322	12,6103	1,1864	13,8668
90	10,4332	11,7107	13,2631	1,1571	11,8023
105	9,7661	12,5014	11,3826	1,1228	11,2167
120	7,3643	9,1542	8,3669	0,7325	8,2951

Tabla 6.41: Tiempos de ejecución CloverLeaf en clúster de Kubernetes

Ranks K8S	T1	T2	T3	T-De	T-Avg
1	314,5651	303,4708	302,6000	5,4468	306,8786
15	53,4080	58,2705	65,9631	5,1688	59,2139
30	31,8661	31,7339	33,9525	1,0161	32,5175
45	21,4263	21,9835	21,8072	0,2325	21,7390
60	17,6705	22,1855	22,6705	2,2514	20,8422
75	15,4488	16,6252	17,3251	0,7742	16,4664
90	14,6629	17,8877	17,3673	1,4136	16,6393
105	13,9768	13,3956	11,6835	0,9734	13,0186
120	9,7709	10,3029	12,2608	1,0706	10,7782

Si se analiza la desviación estándar en términos del coeficiente de variación se tiene que el valor máximo para el tiempo de ejecución fue del 10.01% en los *worker nodes* y del 10.80% en el clúster de trabajo. Estos valores indican que la media es representativa del conjunto de datos es decir es un conjunto de datos homogéneo. Además la variación fue menor en los *worker nodes*.

La tabla comparativa 6.42 muestra el rendimiento alcanzado del clúster de trabajo con respecto al clúster de Vultr con respecto al tiempo medio de ejecución por número de *ranks*.

Tabla 6.42: Escalamiento fuerte comparativa de rendimiento CloverLeaf

Ranks	**T-Avg-WN**	**T-Avg-CT**	**R-CT**
1	327,4509	306,8786	106,2826
14	60,3235	59,2139	101,8394
30	32,3473	32,5175	99,4739
45	22,7426	21,7390	104,4130
60	17,7398	20,8422	82,5118
75	13,8668	16,4664	81,2533
90	11,8023	16,6393	59,0169
105	11,2167	13,0186	83,9353
120	8,2951	10,7782	70,0660

Analizando el rendimiento de los nodos del clúster de trabajo con respecto al rendimiento obtenido en los *worker nodes*. La media alcanzada de rendimiento fue del 87.64% siendo el valor mínimo 70.06%. Para la prueba de escalamiento fuerte los resultados de tiempo de ejecución obtenidos con 2(30 *ranks*) y 3 nodos(45 *ranks*) obtuvieron un rendimiento virtualmente idéntico a los *worker nodes* con igual cantidad de *ranks*.

Las tablas 6.43 y 6.44 muestran los resultados obtenidos de *speedup* y eficiencia utilizando el tiempo medio de ejecución por número de *ranks* en el clúster de Vultr y el clúster de trabajo respectivamente.

Tabla 6.43: Escalamiento fuerte CloverLeaf en clúster de Vultr

Ranks Vultr	**T-Avg**	**SpeedUp**	**Eficiencia**
1	327,4509	1,0000	1,0000
15	60,3235	5,4283	0,3619
30	32,3473	10,1230	0,3374
45	22,7426	14,3981	0,3200
60	17,7398	18,4585	0,3076
75	13,8668	23,6140	0,3149
90	11,8023	27,7446	0,3083
105	11,2167	29,1932	0,2780
120	8,2951	39,4751	0,3290

Tabla 6.44: Escalamiento fuerte CloverLeaf en clúster de Kubernetes

Ranks K8S	T-Avg	SpeedUp	Eficiencia
1	306,8786	1,0000	1,0000
15	59,2139	5,1825	0,3455
30	32,5175	9,4373	0,3146
45	21,7390	14,1165	0,3137
60	20,8422	14,7239	0,2454
75	16,4664	18,6367	0,2485
90	16,6393	18,4430	0,2049
105	13,0186	23,5723	0,2245
120	10,7782	28,4722	0,2373

Las métricas de *speedup* y eficiencia son valores proporcionales de acuerdo al tiempo de ejecución utilizando un solo nodo, el cual fue menor en el clúster de trabajo por lo que estas se ven influenciadas por esta variación.

El gráfico 6.7 compara visualmente el *speedup* obtenido los nodos del clúster de trabajo contra el *speedup* obtenido en los *worker nodes*. El eje x muestra los ranks utilizados mientras que el eje y muestra el *speedup* alcanzado. La curva de escalamiento con ambos ambientes es ascendente sin declinarse o aplanarse en ningún momento. Esto es debido a que CloverLeaf es un *benchmark* que escala eficientemente con gran cantidad de nodos.

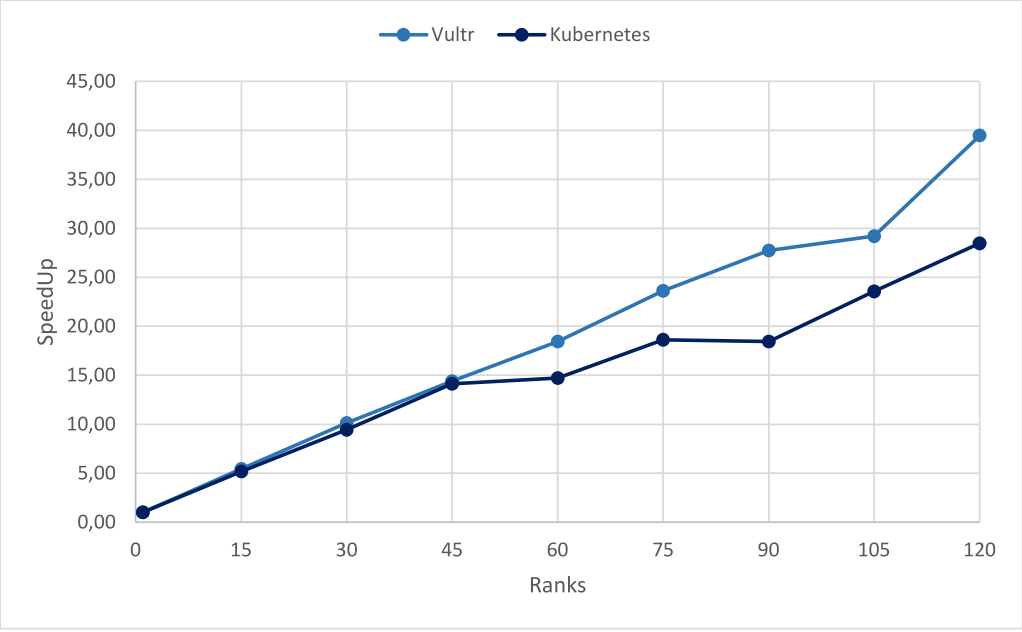

Figura 6.7: Escalamiento fuerte comparativa de eficiencia CloverLeaf

Otra característica de la prueba de escalamiento fuerte es que el rendimiento alcanzado por los nodos del clúster de trabajo con 2(30 *ranks*) y 3(60 *ranks*) nodos fue virtualmente

idéntico al rendimiento alcanzado en los *worker nodes* con la misma cantidad de *ranks*. Otra manera de ver este resultado es que podría existir una punto óptimo en el tamaño del problema y la cantidad de nodos para que ambos ambientes obtengan rendimientos similares.

CloverLeaf prueba escalamiento débil

La prueba de escalamiento débil fue realizada utilizando un tamaño base para 15 *ranks* de 29491200 mediante las dimensiones las dimensiones x=7680, y=3840. A partir de esta base se calculó un tamaño proporcional para el resto de número de *ranks*.

La tablas 6.45 y 6.46 muestran los resultados de tiempo de ejecución obtenidos en las diferentes repeticiones por número de *ranks* y tamaño de problema en el clúster de Vultr y el clúster de trabajo respectivamente.

Tabla 6.45: Tiempos de ejecución CloverLeaf en clúster de Vultr

Ranks Vultr	T1	T2	T3	T-De	T-Avg	P-Size
1	20,8413	20,4331	20,1483	0,2844	20,4742	1966080
15	58,9701	62,5206	59,8957	1,5038	60,4621	29491200
30	59,6647	72,9179	60,3081	6,1016	64,2969	58982400
45	63,3250	67,1996	64,0558	1,6809	64,8601	88473600
60	65,6467	66,4991	65,9694	0,3514	66,0384	117964800
75	72,6244	73,6179	73,7026	0,4895	73,3150	147456000
90	75,9134	77,4257	81,2733	2,2563	78,2041	176947200
105	83,8528	83,3156	78,2743	2,5127	81,8142	204902400
120	79,9476	80,6338	74,9766	2,5207	78,5193	235929600

Tabla 6.46: Tiempos de ejecución CloverLeaf en clúster de Kubernetes

Ranks K8S	T1	T2	T3	T-De	T-Avg	P-Size
1	20,2459	18,1941	18,1926	0,9676	18,8775	1966080
15	48,9784	50,5404	53,5912	1,9156	51,0367	29491200
30	63,0473	66,3869	62,5194	1,7123	63,9845	58982400
45	69,2995	69,4886	65,5304	1,8230	68,1062	88473600
60	61,1489	72,9056	67,5592	4,8062	67,2046	117964800
75	75,0424	71,7590	74,3906	1,4193	73,7307	147456000
90	68,7947	68,3358	72,1055	1,6794	69,7453	176947200
105	75,7460	77,0285	83,9505	3,6036	78,9083	204902400
120	71,5828	82,2614	74,3512	4,5248	76,0651	235929600

Si se analiza la desviación estándar en términos del coeficiente de variación se tiene que el valor máximo para el tiempo de ejecución fue del 9.48% en los *worker nodes* y del 7.15%

en el clúster de trabajo. Estos valores indican que la media es representativa del conjunto de datos es decir es un conjunto de datos homogéneo. Además la variación fue menor en el *clúster de trabajo*.

La tabla comparativa 6.47 muestra el rendimiento alcanzado del clúster de trabajo con respecto al clúster de Vultr con respecto al tiempo medio de ejecución por número de *ranks*.

Tabla 6.47: Escalamiento débil comparativa de rendimiento CloverLeaf

Ranks	T-Avg-WN	T-Avg-CT	R-CT
1	20,4742	18,8775	107,7986
14	60,4621	51,0367	115,5890
30	64,2969	63,9845	100,4858
45	64,8601	68,1062	94,9953
60	66,0384	67,2046	98,2341
75	73,3150	73,7307	99,4330
90	78,2041	69,7453	110,8163
105	81,8142	78,9083	103,5518
120	78,5193	76,0651	103,1256

Analizando el rendimiento de los nodos del clúster de trabajo con respecto al rendimiento obtenido en los *worker nodes*. La media alcanzada de rendimiento fue del 103.78% siendo el valor mínimo 94.99%. El rendimiento en el caso de este *benchmark* no se ve afectado por el número de nodos utilizados. Esto se puede corroborar si se compara el rendimiento del 103.12% utilizando 120 ranks(8 nodos) contra otros resultados con menor número de *ranks* donde el rendimiento es menor.

Las tablas 6.48 y 6.49 muestran los resultados obtenidos de eficiencia utilizando el tiempo medio de ejecución por número de *ranks* y tamaño de problema en el clúster de Vultr y el clúster de trabajo respectivamente. Además la imagen representa gráficamente el escalamiento obtenido utilizando la métrica de eficiencia.

Tabla 6.48: Escalamiento débil CloverLeaf en clúster de Vultr

Ranks Vultr	T-Avg	Eficiencia	P-Size
1	20,4742	1,0000	1966080
15	60,4621	0,3386	29491200
30	64,2969	0,3184	58982400
45	64,8601	0,3157	88473600
60	66,0384	0,3100	117964800
75	73,3150	0,2793	147456000
90	78,2041	0,2618	176947200
105	81,8142	0,2503	204902400
120	78,5193	0,2608	235929600

Tabla 6.49: Escalamiento débil CloverLeaf en clúster de Kubernetes

Ranks K8S	T-Avg	Eficiencia	P-Size
1	18,8775	1,0000	1966080
15	51,0367	0,3699	29491200
30	63,9845	0,2950	58982400
45	68,1062	0,2772	88473600
60	67,2046	0,2809	117964800
75	73,7307	0,2560	147456000
90	69,7453	0,2707	176947200
105	78,9083	0,2392	204902400
120	76,0651	0,2482	235929600

Las mediciones de rendimiento y eficiencia obtenidas en los nodos del clúster de trabajo en este *benchmark* en comparación a los *worker nodes* fueron las mejores entre todas las pruebas realizadas utilizando los *benchmarks* de Mantevo. a excepción del rendimiento obtenido con 3 nodos(45 *ranks*) el resto de resultados obtuvieron un rendimiento virtualmente idéntico o mayor en comparación a los resultados obtenidos en los *worker nodes* con igual número de nodos.

El gráfico 6.8 compara visualmente la eficiencia obtenida en los nodos del clúster de trabajo contra la eficiencia obtenida en los *worker nodes* utilizando escalamiento débil. El eje x muestra los *ranks* utilizados mientras que el eje y muestra la eficiencia alcanzada. La curva de escalamiento con ambos ambientes es similar y tiene un escalamiento lineal a partir de los 15 *ranks*.

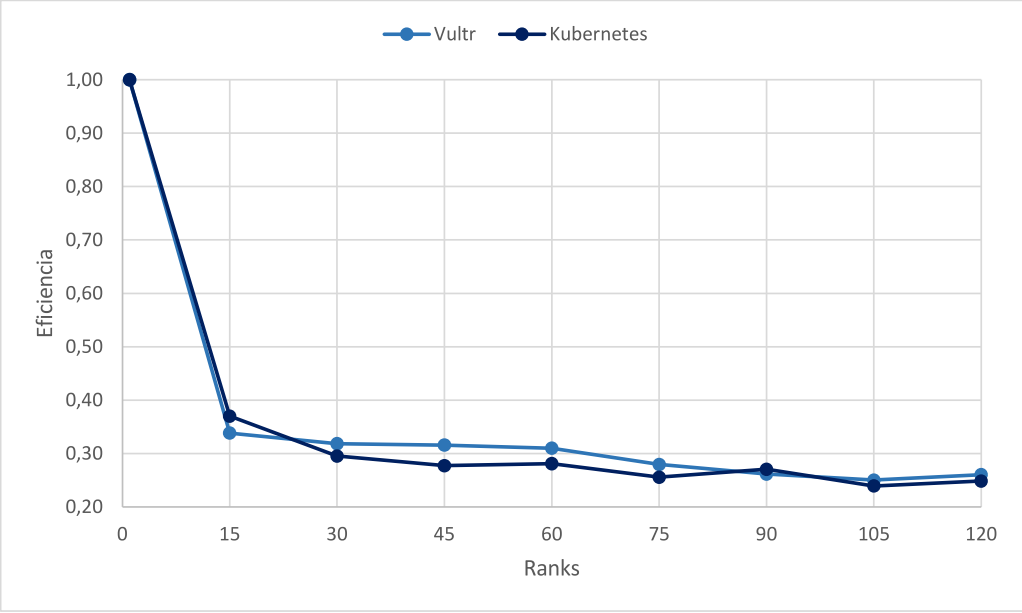

Figura 6.8: Escalamiento débil comparativa de eficiencia CloverLeaf

6.5 Rendimiento de entrenamiento de modelos de I.A

Para esta medición será utilizado el ambiente creado en la sección 5.3 del capítulo de implementación y un clúster de trabajo de dos nodos con GPU habilitada basado en la imagen de ambiente auto-contenido de OpenMPI con miniconda[1] utilizando el archivo YAML disponible en el repositorio de código de ejemplos para el operador [28]. La especificación del archivo YAML puede ser consultada en el anexo 20(8.20).

Para la medición del *benchmark* de I.A entrenando el modelo de ResNet-50 se utilizará el conjunto de datos de Cifar-10. Se utilizó el código del repositorio de ejemplos de LambdaLabsML[2] el cual fue copiado junto al conjunto de datos del Cifar-10 en el repositorio de código de *benchmarks* de I.A [15]. El código fue modificado para imprimir el tiempo total de ejecución al final de cada entrenamiento.

Para la medición del *benchmark AI Benchmark Alpha* se creó un *script* de Python el cual ejecuta el *benchmark* en su modalidad de entrenamiento y reporta el tiempo total transcurrido al final de la ejecución. Este *script* se encuentra en el repositorio de código de *benchmarks* de I.A [15].

Tanto en los nodos del clúster de trabajo como los *worker nodes* será configurado un ambiente virtual de miniconda capaz de ejecutar el software encargado de entrenar los modelos de I.A. Este ambiente de trabajo será creado con la especificación del archivo YAML la cual puede ser consultada en el anexo 21(8.21).

Todos los resultados de los experimentos incluyendo la salida de consola de cada comando serán guardados en las carpetas `AI-Experiments-Vultr` y `AI-Experiments-K8s` del repositorio de código de resultados de experimentos para el operador[27].

6.5.1 Resultados del entrenamiento de ResNet-50 con Cifar-10

Para esta medición se entrenó el modelo tres veces con un solo nodo y luego tres veces con dos nodos utilizando el mismo conjunto de datos sin entrenar. Las métricas a evaluar son las medias de tiempo de ejecución del programa de entrenamiento, *accuracy* y número de muestras por segundo procesadas.

El *accuracy* objetivo de esta medición fue de al menos un 70% con un *batch size* de 4096. Para lograrlo se utilizó un *learning rate* del 0.05 y 60 *epochs* con 1 nodo. Y un *learning rate* de 0.05 y 61 *epochs* con 2 nodos.

El entrenamiento distribuido utilizando dos nodos se lleve a cabo por medio de MPI utilizando Open MPI. En I.A el entrenamiento distribuido requiere que el *batch size* se

[1] Miniconda es un administrador de paquetes de Python. Permite crear ambientes virtuales a partir de una archivo de requerimientos donde se especifican los paquetes necesarios con sus respectivas versiones

[2] LambdaLabsML es un proveedor público de servicios en la nube especializado en brindar servicios de infraestructura de altas prestaciones para centros de investigación. Este proveedor ofrece una gran cantidad de configuraciones enfocadas a las necesidades de la I.A con redes de ultra alta velocidad

divida entre la cantidad de nodos ya que la idea es realizar el mismo entrenamiento con la ayuda de más recursos computacionales para reducir el tiempo de ejecución sin afectar el *accuracy* obtenido.

La tablas 6.50 y 6.51 muestran los tiempos de ejecución obtenidos en el clúster de trabajo y los *worker nodes* respectivamente. La primer columna muestra la cantidad de nodos utilizados. Las columnas T1, T2 y T3 muestran los tiempos de ejecución obtenidos en las diferentes repeticiones. Las columnas T-De y D-Avg muestran la desviación estándar y la media de los tiempos obtenidos respectivamente.

Tabla 6.50: Resnet-50 tiempo de entrenamiento en el clúster de Vultr

Nodes Vultr	T1	T2	T3	T-De	T-Avg
1	9,1236	9,1148	9,1344	0,0080	9,1243
2	5,6567	5,4413	5,4594	0,0976	5,5191

Tabla 6.51: Resnet-50 tiempo de entrenamiento en el clúster de Kubernetes

Nodes K8s	T1	T2	T3	T-De	T-Avg
1	9,4890	9,4992	9,4832	0,0066	9,4905
2	5,7182	5,7920	5,7845	0,0332	5,7649

Analizando la desviación estándar en términos del coeficiente de varianza se tiene un valor máximo de 1.76%. Esto significa que la media es significativa del conjunto de datos.

La tabla 6.52 muestra el rendimiento obtenido en los nodos del clúster de trabajo con respecto al rendimiento obtenido en los *worker nodes* en términos de tiempo de ejecución. Se obtuvo un rendimiento del 95.98% con 1 nodo y del con 95.54% 2 nodos.

Tabla 6.52: Resnet-50 comparativa tiempo de entrenamiento

Nodes	T-Avg-WN	T-Avg-CT	R-CT
1	9,1243	9,4905	95,9865
2	5,5191	5,7649	95,5470

La tablas 6.53 y 6.54 muestran el *accuracy* obtenido en el clúster de trabajo y los *worker nodes* respectivamente. La primer columna muestra la cantidad de nodos utilizados. Las columnas A1, A2 y A3 muestran el *accuracy* obtenido en las diferentes repeticiones de la prueba. Las columnas A-De y A-Avg muestran la desviación estándar y la media del *accuracy* obtenido respectivamente.

Tabla 6.53: Resnet-50 *accuracy* de entrenamiento en el clúster de Vultr

Nodes Vultr	A1	A2	A3	A-De	A-Avg
1	0,7191	0,7191	0,7191	0	0,7191
2	0,7125	0,7125	0,7125	0	0,7125

Tabla 6.54: Resnet-50 *accuracy* de entrenamiento en el clúster de Kubernetes

Nodes K8s	A1	A2	A3	A-De	A-Avg
1	0,7191	0,7191	0,7191	0	0,7191
2	0,7125	0,7125	0,7125	0	0,7125

En el caso de la métrica de *accuracy* la desviación estándar es cero y el resultado es el mismo en ambos ambientes. Esto es debido a que en iguales condiciones, es decir mismos parámetros de entrada y conjunto de datos es esperado que el *accuracy* se mantenga invariable.

La tabla 6.55 muestra el rendimiento obtenido en los nodos del clúster de trabajo con respecto al rendimiento obtenido en los *worker nodes* en términos de *accuracy*. El rendimiento fue de un 100% con 1 y 2 nodos. El *accuracy* se mantuvo constante y no vario lo que indica que la ejecución del entrenamiento en el clúster de trabajo no afecta calidad del mismo.

Tabla 6.55: Resnet-50 comparativa *accuracy*

Nodes	A-Avg-WN	A-Avg-CT	R-CT
1	0,7191	0,7191	100
2	0,7125	0,7125	100

La tablas 6.56 y 6.57 muestran la media de muestras por segundo procesadas por *epoch* en el clúster de trabajo y los *worker nodes* respectivamente. La primer columna muestra la cantidad de nodos utilizados. Las columnas MS-1, MS-2 y MS-3 muestran el resultado obtenido en las diferentes repeticiones de la prueba. Las columnas MS-De y MS-Avg muestran la desviación estándar y la media obtenida respectivamente.

Tabla 6.56: Resnet-50 muestras por segundo procesadas en el clúster de Vultr

Nodes Vultr	MS-1	MS-2	MS-3	MS-De	MS-Avg
1	6100,4504	6110,2683	6086,2515	9,8590	6098,9901
2	10757,6859	10788,0810	10743,1783	18,7100	10762,9817

Tabla 6.57: Resnet-50 muestras por segundo procesadas en el clúster de Kubernetes

Nodes K8s	MS-1	MS-2	MS-3	MS-De	MS-Avg
1	5863,6071	5862,2029	5860,3244	1,3448	5862,0448
2	10186,2327	10108,7958	10081,1160	44,4874	10125,3815

Analizando la desviación estándar en términos del coeficiente de varianza todos los valores son menores al 1%. Esto significa que la media es significativa del conjunto de datos.

La tabla 6.58 muestra el rendimiento obtenido en los nodos del clúster de trabajo con respecto al rendimiento obtenido en los *worker nodes* en términos de la media de las muestras por segundo procesadas por *epoch*. Se obtuvo un rendimiento del 96.11% con 1 nodo y del con 94.07% 2 nodos.

Tabla 6.58: Resnet-50 comparativa muestras procesadas por segundo

Nodes	MS-Avg-WN	MS-Avg-CT	R-CT
1	6098,9901	5862,0448	96,1150
2	10762,9817	10125,3815	94,0760

Debido a que el rendimiento obtenido en los nodos del clúster de trabajo fue muy superior al 70% la medición fue exitosa.

6.5.2 Resultados del *AI Benchmark Alpha*

Para esta medición se ejecutaron los *benchmarks* de *AI Benchmark Alpha* tres veces. Las pruebas se ejecutaron en un solo nodo debido a que este *benchmarks* no soporta entrenamiento distribuido. Las métricas a evaluar son el tiempo de ejecución, el cual es el tiempo que dura el *benchmark* entrenar los 19 modelos. Y el *Device Training Score* la cual es una puntuación que otorga el *benchmark* al dispositivo donde fue ejecutado el entrenamiento de acuerdo al rendimiento obtenido.

La tablas 6.59 y 6.60 muestran los tiempos de ejecución y *AI Training score* obtenidos en el clúster de trabajo y los *worker nodes* respectivamente. La primeras columnas muestran la métrica reportada y la cantidad de nodos utilizados. Las columnas R1, R2 y R3 muestran los los resultados obtenidos en las diferentes repeticiones. Las columnas De y Avg muestran la desviación estándar y la media de los resultados obtenidos respectivamente.

Tabla 6.59: *AI Benchmark Alpha* resultados en clúster de Vultr

Métrica	Nodos Vultr	R1	R2	R3	De	Avg
Tiempo	1	6,3687	6,2738	6,2225	0,0606	6,2883
AI Training Score	1	21239	21674	21708	213,5264	21540,3333

Tabla 6.60: *AI Benchmark Alpha* resultados en clúster de Kubernetes

Métrica	Nodos K8s	R1	R2	R3	De	Avg
Tiempo	1	6,3068	6,3203	6,333	0,0107	6,3200
AI Training Score	1	21684	21726	21549	75,5116	21653

En este *benchmark* el coeficiente de varianza calculado a partir de la desviaciones estándar fue menor al al 1% en todos los casos. Esto significa que la media es significativa del conjunto de datos.

La tabla 6.61 muestra el rendimiento obtenido en los nodos del clúster de trabajo con respecto al rendimiento obtenido en los *worker nodes*. El rendimiento obtenido fue virtualmente idéntico con una diferencia entre de menos del 1%. El resultado indica que el uso del método para la creación de ambientes virtuales de un solo nodo es una manera eficiente de entrenar modelos en nodos con las prestaciones necesarias.

Tabla 6.61: *AI Benchmark Alpha* comparativa de rendimiento

Métrica	Avg-WN	Avg- CT	R-CT
Tiempo	6,2883	6,3200	99,4959
AI Training Score	21540,3333	21653,0000	100,5230

Debido a que el rendimiento obtenido en los nodos del clúster de trabajo fue muy superior al 70% la medición fue exitosa.

Capítulo 7

Conclusiones y Trabajo Futuro

En este capítulo se presentan las conclusiones de la tesis en base a las mediciones realizadas en el capítulo de resultados y las observaciones efectuadas durante la implementación del método. Estas conclusiones contemplan la explicación y mención de los hallazgos de la investigación, la importancia de estos y sus implicaciones. Además se abarcan las limitaciones de la metodología y el método propuesto. Y se finaliza hablando de los posibles trabajos futuros que podría expandir y agregar valor al presente trabajo de investigación.

7.1 Conclusiones

El objetivo de esta investigación ha sido la definición de un método para la creación automatizada de ambientes que ejecuten de aplicaciones de computación paralela basadas en MPI en un ambiente distribuido dentro de un clúster de Kubernetes. En base a este objetivo se presentó el diseño e implementación de una prueba de concepto del método basado en un conjunto de CRDs controlados por un operador de Kubernetes.

El método presentado permite fácilmente crear un clúster para la ejecución de aplicaciones de computación paralela basadas en MPI en un clúster de Kubernetes. El tiempo de la creación del ambiente en un clúster de Kubernetes no excedió los 35 segundos por nodo para un clúster de 8 nodos. Cumpliendo de esta manera con uno de los objetivos trazados para la investigación, el cual era el que el tiempo de preparación del ambiente en el clúster de Kubernetes no excediera los 1 * N minutos siendo N el número de nodos deseados. El tiempo medio de creación de los nodos fue de 21.27 segundos superando con creces el objetivo trazado.

El rendimiento del método al ejecutar aplicaciones de computación paralela fue evaluado utilizando los *benchmarks* de Mantevo, comparando porcentualmente el rendimiento obtenido en el ambiente creado en el clúster de Kubernetes contra el obtenido en la misma infraestructura fuera de Kubernetes siendo este el 100%. La infraestructura utiliza para las pruebas fue la de un proveedor de servicios públicos en la nube llamado Vultr utilizando equipos de centros de datos de última generación.

De Matenvo se seleccionaron los *benchmarks* MiniFE, MiniMD, MiniXyce y CloverLeaf y se evaluaron las métricas de tiempo de ejecución, escalamiento fuerte y escalamiento débil. Se evaluaron un total de 72 mediciones en cada clúster. La media del rendimiento del tiempo de ejecución en el clúster de Kubernetes para las pruebas de escalamiento fuerte fue de un 90.27% y de un 97.1507% para las pruebas de escalamiento débil.

El rendimiento alcanzado superó el objetivo trazado que era obtener al menos una media del 70% en comparación a un clúster que utilizara la misma infraestructura fuera de Kubernetes. A destacar es que el rendimiento alcanzado por el ambiente creado en el clúster de Kubernetes superó el 90% en 49 de las 72 mediciones realizadas.

Además de Mantevo fue evaluado el rendimiento al entrenar modelos de I.A, comparando porcentualmente el rendimiento obtenido en el ambiente creado en el clúster de Kubernetes contra el obtenido en la misma infraestructura fuera de Kubernetes siendo este el 100%. Primeramente se evaluó el rendimiento al entrenar el modelo ResNet-50 y luego se evaluó el rendimiento al ejecutar los *benchmarks* de *AI Benchmark Alpha*. En ambos casos el rendimiento obtenido fue mayor al 94% con una media del 96.95%.

Respaldado por las mediciones obtenidas el método probó ser una opción efectiva para la ejecución de aplicaciones de computación paralela basadas en MPI en un clúster de Kubernetes. Principalmente al utilizar escalamiento débil y aplicaciones que tengan altos requerimientos computacionales y bajos o medios requerimientos de E/S. Además el método proporcionó un rendimiento excelente al utilizarse para la creación de ambientes para el entrenamiento de modelos de I.A.

Una de la problemáticas expuestas al inicio de esta investigación fue el difícil acceso de pequeñas o medianas empresas e individuos a recursos computacionales para la ejecución de aplicaciones de computación paralela. La implementación exitosa del presente método utilizando recursos computacionales en la nube al alcance de toda la población, lo coloca como una opción favorable para que estas compañías e individuos tengan la posibilidad de acceder a recursos computacionales y crear ambientes de forma sencilla. Permitiéndoles ejecutar las aplicaciones de computación paralela que requieran con un coste relativamente accesible.

En contraposición con las investigaciones de años anteriores estudiadas en los antecedentes de esta tesis, donde los rendimientos obtenidos al ejecutar cargas de computación de alto desempeño en ambientes de contenedores eran desalentadores. Esta investigación demuestra que hoy día es totalmente factible el utilizar ambientes en contenedores haciendo uso de orquestadores como Kubernetes para la creación automatizada de ambientes de computación de alto desempeño.

El uso de Kubernetes para la ejecución de cargas de computación de alto desempeño y computación paralela verá incrementado su uso. Especialmente con el rápido auge de la computación en la nube y los rápidos avances en hardware debido a la I.A. Hoy día los avances en las soluciones de comunicación permiten a los proveedores de servicios públicos en la nube ofrecer infraestructuras con velocidades de transferencia en la escala

de *terabytes* por segundo. Además por medio de estos proveedores distintas instituciones pueden tener acceso a miles de nodos de forma instantánea con los cuales se pueden crear clústers de computación de alto desempeño.

7.2 Limitaciones y trabajo futuro

Tanto la metodología como el método de este trabajo contienen ciertas limitaciones que deben ser expuestas. Al realizar la implementación del método se observo que al no utilizar las capacidades del CNI en este caso Calico, para utilizar la subred de los *worker nodes* el rendimiento de las velocidades de transferencia de red o escritura y lectura de disco se vieron drásticamente afectados. Con un rendimiento en el ambiente creado en el clúster de Kubernetes menor al 30% con respecto al clúster creado fuera de Kubernetes utilizando la misma infraestructura y *software*.

Esto limita el método a ser utilizado en un clúster de Kubernetes con *worker nodes* que se encuentren en una misma subred. Sin embargo al utilizar las capacidades del CNI para utilizar la misma subred de los *worker nodes* el rendimiento de velocidades de transferencia de red o escritura y lectura de disco no se vieron comprometidos. Alcanzando un rendimiento cercano al 100% en todas las mediciones realizadas utilizando las herramientas de medición dd y *iperf*.

Otra limitante fue un acceso limitado a hardware para las mediciones debido al presupuesto de esta investigación. El método fue validado en un clúster de 8 nodos, con 15 *ranks* por nodo. Y el protocolo de comunicación utilizado fue el de TCP/IP. Aunque todas las pruebas con la mayor cantidad de nodos fueron exitosas, no es posible afirmar el mismo comportamiento en clúster con cientos o miles de nodos.

Como trabajo futuro se considera importante el mejorar el tiempo de creación del ambiente paralelizando la creación de los nodos del clúster de trabajo creado en Kubernetes. Actualmente el método crea los nodos de forma secuencial por lo que hacerlo de manera paralela mejoraría el tiempo de creación especialmente en los casos de clústers con gran cantidad de nodos. Una forma de lograrlo sería crear un servicio *listener* en el último nodo que actualice constantemente el archivo de `/etc/hosts` en los diferentes nodos del clúster de trabajo utilizando el proceso actualmente implementado en el método.

Además validar el método en un clúster de Kubernetes que permita la creación de un ambiente de miles de nodos interconectados por medio de dispositivos de red de ultra alta velocidad como *InfinitiBand* podría probar la efectividad del método propuesto en comparación con súper computadores de similares características.

Capítulo 8

Anexos

8.1 Anexo 1. Definición de instancia de CRD de clúster

```
apiVersion: crisarias.com/v1alpha1
kind: MpiCluster
metadata:
  labels:
    app.kubernetes.io/name: irazucluster
    app.kubernetes.io/instance: irazucluster-instance
    app.kubernetes.io/part-of: kubernetes-mpi-operator
    app.kubernetes.io/managed-by: kustomize
    app.kubernetes.io/created-by: kubernetes-mpi-operator
  name: irazucluster
spec:
  clusterName: "irazu"
  clusterNodeImage: "docker.io/crisarias/mpicluster-openmpi-conda-operator-base:latest"
  clusterNodesCount: 8
  coresPerNode: 15
  memoryPerNode: 48000
  sharedMemoryPerNode: 2000
  sshPort: 22
  localVolumePath: "/mnt/beegfs"
  localVolumeCapacity: 1000
  enableNvidiaGPU: true
  numGPUs: 1
  nameGPUs: GRID-A100D-3-40C-MIG-3g.40gb
```

8.2 Anexo 2. Definición de instancia de CRD de job

```
apiVersion: crisarias.com/v1alpha1
kind: MpiJob
metadata:
```

```
  labels:
    app.kubernetes.io/name: testjob
    app.kubernetes.io/instance: testjob-instance
    app.kubernetes.io/part-of: kubernetes-mpi-operator
    app.kubernetes.io/managed-by: kustomize
    app.kubernetes.io/created-by: kubernetes-mpi-operator
  name: testjob
spec:
  jobName: "job1"
  clusterName: "irazu"
  command: |
    git clone https://github.com/crisarias/mpi-samples \
    cd mpi-samples/helloWorld \
    make \
    mpirun np 150 hello_world > output.txt
```

8.3 Anexo 3. Especificación del Clúster de Kubertenes para Kind

```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
name: tkb
networking:
  ipFamily: dual
nodes:
- role: control-plane
  kubeadmConfigPatches:
  - |
    kind: InitConfiguration
    nodeRegistration:
      kubeletExtraArgs:
        node-labels: "ingress-ready=true"
- role: worker
  extraMounts:
  - hostPath: /nfsshare
    containerPath: /nfs
- role: worker
  extraMounts:
  - hostPath: /nfsshare
    containerPath: /nfs
- role: worker
  extraMounts:
  - hostPath: /nfsshare
    containerPath: /nfs
```

8.4 Anexo 4. Especificación del espacio de nombres

```
apiVersion: v1
kind: Namespace
metadata:
  name: irazu-ns
```

8.5 Anexo 5. Especificación del servicio *headless*

```
# Headless Service
apiVersion: v1
kind: Service
metadata:
  namespace: irazu-ns
  name: irazu-headless-svc
  labels:
    cluster: irazu
spec:
  ports:
  - port: 22
    targetPort: 22
  clusterIP: None
  selector:
    cluster: irazu
```

8.6 Anexo 6. Especificación de la clase de almacenamiento

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  namespace: irazu-ns
  name: irazu-sc
provisioner: kubernetes.io/no-provisioner
reclaimPolicy: Delete
volumeBindingMode: Immediate
```

8.7 Anexo 7. Especificación del volumen persistente

```
apiVersion: v1
kind: PersistentVolume
metadata:
  namespace: irazu-ns
  name: irazu-pv
  labels:
    clusterStorageFolder: irazu-pv-1711675384424
spec:
  accessModes:
  - ReadWriteMany
  capacity:
```

```
    storage: 25Gi
  hostPath:
    path: /nfs/irazu-pv-1711675384424
    type: ""
  persistentVolumeReclaimPolicy: Delete
  storageClassName: irazu-sc
  volumeMode: Filesystem
```

8.8 Anexo 8. Especificación del *claim* de volumen persistente

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  namespace: irazu-ns
  name: irazu-pvc
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 25Gi
  storageClassName: irazu-sc
  volumeMode: Filesystem
  volumeName: irazu-pv
```

8.9 Anexo 9. Especificación del *statefulset*

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: irazu-sts
  namespace: irazu-ns
spec:
  podManagementPolicy: OrderedReady
  replicas: 3
  selector:
    matchLabels:
      cluster: irazu
  serviceName: irazu-headless-svc
  template:
    metadata:
      labels:
        cluster: irazu
    spec:
      containers:
      - env:
        - name: CLUSTERNODESCOUNT
          value: "3"
```

```
        - name: CORESPERNODE
          value: "4"
        - name: CLUSTERNAME
          value: irazu
        image: docker.io/crisarias/mpicluster-mpich-operator-base:latest
        imagePullPolicy: Always
        name: ctr-irazu
        ports:
        - containerPort: 22
          protocol: TCP
        resources:
          limits:
            cpu: "4"
            memory: 4000Mi
        startupProbe:
          exec:
            command:
            - bash
            - -c
            - /bin/testNodeSSH.sh $CLUSTERNODESCOUNT $CLUSTERNAME
          failureThreshold: 5
          initialDelaySeconds: 10
          periodSeconds: 10
          successThreshold: 1
          terminationGracePeriodSeconds: 10
          timeoutSeconds: 5
        volumeMounts:
        - mountPath: /nfs
          name: irazu-local-vol
      restartPolicy: Always
      schedulerName: default-scheduler
      securityContext: {}
      terminationGracePeriodSeconds: 10
      volumes:
      - name: irazu-local-vol
        persistentVolumeClaim:
          claimName: irazu-pvc
  updateStrategy:
    rollingUpdate:
      partition: 0
    type: RollingUpdate
```

8.10 Anexo 10. Especificación de CRD de clúster de MPICH

```
apiVersion: crisarias.com/v1alpha1
kind: MpiCluster
metadata:
  labels:
    app.kubernetes.io/name: irazucluster
```

```
    app.kubernetes.io/instance: irazucluster-instance
    app.kubernetes.io/part-of: kubernetes-mpi-operator
    app.kubernetes.io/managed-by: kustomize
    app.kubernetes.io/created-by: kubernetes-mpi-operator
  name: irazucluster
spec:
  clusterName: "irazu"
  clusterNodeImage: "docker.io/crisarias/mpicluster-mpich-operator-base:latest"
  clusterNodesCount: 8
  coresPerNode: 15
  memoryPerNode: 30
  sshPort: 22
  localVolumePath: "/mnt/beegfs"
  localVolumeCapacity: 500
```

8.11 Anexo 11. Especificación de CRD de clúster de OpenMPI

```
apiVersion: crisarias.com/v1alpha1
kind: MpiCluster
metadata:
  labels:
    app.kubernetes.io/name: barvacluster
    app.kubernetes.io/instance: barvacluster-instance
    app.kubernetes.io/part-of: kubernetes-mpi-operator
    app.kubernetes.io/managed-by: kustomize
    app.kubernetes.io/created-by: kubernetes-mpi-operator
  name: barvacluster
spec:
  clusterName: "barva"
  clusterNodeImage: "docker.io/crisarias/mpicluster-openmpi-operator-base:latest"
  clusterNodesCount: 8
  coresPerNode: 15
  memoryPerNode: 30
  sshPort: 22
  localVolumePath: "/mnt/beegfs"
  localVolumeCapacity: 500
```

8.12 Anexo 12. Especificación de CRD de clúster de SLURM

```
apiVersion: crisarias.com/v1alpha1
kind: MpiCluster
metadata:
  labels:
    app.kubernetes.io/name: poascluster
    app.kubernetes.io/instance: poascluster-instance
    app.kubernetes.io/part-of: kubernetes-mpi-operator
```

```
    app.kubernetes.io/managed-by: kustomize
    app.kubernetes.io/created-by: kubernetes-mpi-operator
  name: poascluster
spec:
  clusterName: "poas"
  clusterNodeImage: "docker.io/crisarias/mpicluster-mpich-slurm-operator-base:latest"
  clusterNodesCount: 8
  coresPerNode: 15
  memoryPerNode: 30
  sshPort: 22
  localVolumePath: "/mnt/beegfs"
  localVolumeCapacity: 500
```

8.13 Anexo 13. Especificación de CRD de job para el clúster de MPICH

```
apiVersion: crisarias.com/v1alpha1
kind: MpiJob
metadata:
  labels:
    app.kubernetes.io/name: mpijob
    app.kubernetes.io/instance: mpijob-instance
    app.kubernetes.io/part-of: kubernetes-mpi-operator
    app.kubernetes.io/managed-by: kustomize
    app.kubernetes.io/created-by: kubernetes-mpi-operator
  name: mpichjob
spec:
  jobName: "mpichjob"
  clusterName: "irazu"
  command: source ~/.bashrc && git clone https://github.com/Crisarias/mpich-examples && cd
      ↪ mpich-examples/mpi-hello-world/code && make && mpirun -np 120 ./mpi_hello_world
      ↪ >> /nfs/result.txt
```

8.14 Anexo 14. Especificación de CRD de job para el clúster de OpenMPI

```
apiVersion: crisarias.com/v1alpha1
kind: MpiJob
metadata:
  labels:
    app.kubernetes.io/name: mpijob
    app.kubernetes.io/instance: mpijob-instance
    app.kubernetes.io/part-of: kubernetes-mpi-operator
    app.kubernetes.io/managed-by: kustomize
    app.kubernetes.io/created-by: kubernetes-mpi-operator
  name: openmpijob
spec:
  jobName: "openmpijob"
```

```
  clusterName: "barva"
  command: source ~/.bashrc && git clone https://github.com/Crisarias/mpich-examples && cd
      ↪ mpich-examples/mpi-hello-world/code && make && mpirun -hostfile /nfs/hosts.openmpi
      ↪  -n 120 ./mpi_hello_world > /nfs/result.txt
```

8.15 Anexo 15. Especificación de CRD de job para el clúster de SLURM

```
apiVersion: crisarias.com/v1alpha1
kind: MpiJob
metadata:
  labels:
    app.kubernetes.io/name: mpijob
    app.kubernetes.io/instance: mpijob-instance
    app.kubernetes.io/part-of: kubernetes-mpi-operator
    app.kubernetes.io/managed-by: kustomize
    app.kubernetes.io/created-by: kubernetes-mpi-operator
  name: slurmjob
spec:
  jobName: "slurmjob"
  clusterName: "poas"
  command: source ~/.bashrc && git clone https://github.com/Crisarias/mpich-examples && cd
      ↪ mpich-examples/mpi-hello-world/code && make && srun -n 105 ./mpi_hello_world > /
      ↪ nfs/result.txt
```

8.16 Anexo 16. Versión de MPICH instalada por medio de APT

```
root@irazu-sts-0:/# mpiexec --version
HYDRA build details:
    Version: 4.0
    Release Date: Fri Jan 21 10:42:29 CST 2022
    CC: gcc -Wdate-time -D_FORTIFY_SOURCE=2 -g -O2 -ffile-prefix-map=/build/
        ↪ mpich-0xgrG5/mpich-4.0=. -flto=auto -ffat-lto-objects -flto=auto -ffat-lto
        ↪ -objects -fstack-protector-strong -Wformat -Werror=format-security -Wl,-
        ↪ Bsymbolic-functions -flto=auto -ffat-lto-objects -flto=auto -Wl,-z,relro
    Configure options: '--with-hwloc-prefix=/usr' '--with-device=ch4:ofi' 'FFLAGS=-
        ↪ O2 -ffile-prefix-map=/build/mpich-0xgrG5/mpich-4.0=. -flto=auto -ffat-lto
        ↪ -objects -flto=auto -ffat-lto-objects -fstack-protector-strong -fallow-
        ↪ invalid-boz -fallow-argument-mismatch' '--prefix=/usr' 'CFLAGS=-g -O2 -
        ↪ ffile-prefix-map=/build/mpich-0xgrG5/mpich-4.0=. -flto=auto -ffat-lto-
        ↪ objects -flto=auto -ffat-lto-objects -fstack-protector-strong -Wformat -
        ↪ Werror=format-security' 'LDFLAGS=-Wl,-Bsymbolic-functions -flto=auto -
        ↪ ffat-lto-objects -flto=auto -Wl,-z,relro' 'CPPFLAGS=-Wdate-time -
        ↪ D_FORTIFY_SOURCE=2'
    Process Manager: pmi
```

```
    Launchers available: ssh rsh fork slurm ll lsf sge manual persist
    Topology libraries available: hwloc
    Resource management kernels available: user slurm ll lsf sge pbs cobalt
    Demux engines available: poll select
root@irazu-sts-0:/#
```

8.17 Anexo 17. Especificación del archivo .dockerfile para imagen de MPICH

```
FROM ubuntu:22.04

# Update APT
RUN apt update

# Upgrade APT
RUN apt upgrade -y

# Install git
RUN apt install -y git

# Install curl
RUN apt install -y curl

#Install vim
RUN apt install -y vim

#Install dns utils
RUN apt install -y dnsutils

#Install cmake
RUN apt install -y cmake

#Install is python3
RUN apt -y install python-is-python3

# Install SSH Server
RUN apt install -y openssh-client openssh-server

# Configure ssh server for passwordless
RUN sed -i 's/#PasswordAuthentication yes/PasswordAuthentication no\
    ↪ nChallengeResponseAuthentication no\n/g' /etc/ssh/sshd_config
RUN sed -i 's/UsePAM yes/UsePAM no/g' /etc/ssh/sshd_config
RUN sed -i 's/#AuthorizedKeysFile/AuthorizedKeysFile/g' /etc/ssh/sshd_config
RUN sed -i 's/# PasswordAuthentication yes/ PasswordAuthentication no/g' /etc/ssh/ssh_config
RUN sed -i 's/ HashKnownHosts yes/ HashKnownHosts no/g' /etc/ssh/ssh_config

# Install build essentials
RUN apt install -y build-essential
```

```
RUN apt -y install gcc g++ gfortran

# Install MPICH
RUN apt -y install mpich

# Generate shell script to update .bashrc file
RUN printf "echo 'Generating bash inizialization file...' \
\nexport HYDRA_HOST_FILE=/nfs/hosts.mpich \
\ncat /dev/null > ~/.bashrc \
\necho export HYDRA_HOST_FILE=/nfs/hosts.mpich >> ~/.bashrc \
\necho 'bash inizialization file generated on ~/.bashrc successfully'" > /bin/generateBashInit.sh
RUN chmod a+x /bin/generateBashInit.sh

# Generate shell script to update mpich hosts file
COPY ../.misc/generateHostsFileScriptMpich.sh /bin/generateHostsFileScriptMpich.sh
RUN chmod a+x /bin/generateHostsFileScriptMpich.sh

# Generate shell script to generate ssh keys
COPY ../.misc/generateKeysFileScript.sh /bin/generateKeysFileScript.sh
RUN chmod a+x /bin/generateKeysFileScript.sh

# Generate shell script to prepare knownHosts file
COPY ../.misc/generateKnownHostsFile.sh /bin/generateKnownHostsFile.sh
RUN chmod a+x /bin/generateKnownHostsFile.sh

# Generate shell script to prepare etc/hosts file
COPY ../.misc/generateEtcHostsFileScript.sh /bin/generateEtcHostsFileScript.sh
RUN chmod a+x /bin/generateEtcHostsFileScript.sh

# Generate shell script to start ssh server
COPY ../.misc/startServerSSH.sh /bin/startServerSSH.sh
RUN chmod a+x /bin/startServerSSH.sh

# Generate shell script to probe node
COPY ../.misc/testNodeSSH.sh /bin/testNodeSSH.sh
RUN chmod a+x /bin/testNodeSSH.sh

# Generate shell script to start cluster
RUN printf "echo 'Initializing cluster...' \
\n/bin/generateHostsFileScriptMpich.sh \$CLUSTERNODESCOUNT \$CLUSTERNAME \
    ↪ $CORESPERNODE \
\n/bin/generateKeysFileScript.sh \$CLUSTERNAME\
\n/bin/startServerSSH.sh \
\n/bin/generateKnownHostsFile.sh \$CLUSTERNODESCOUNT \$CLUSTERNAME \
\n/bin/generateEtcHostsFileScript.sh \$CLUSTERNODESCOUNT \$CLUSTERNAME \
\n/bin/generateBashInit.sh \
\necho Keep node running waiting for jobs... \
\ntrap : TERM INT; sleep infinity & wait" > /bin/startClusterScript.sh
RUN chmod a+x /bin/startClusterScript.sh

# Keep container alive
CMD exec /bin/bash -c "/bin/startClusterScript.sh"
```

8.18 Anexo 18. Especificación del archivo .dockerfile para imagen OpenMPI

```
FROM ubuntu:22.04

# Update APT
RUN apt update

# Upgrade APT
RUN apt upgrade -y

# Install git
RUN apt install -y git

# Install curl
RUN apt install -y curl

#Install vim
RUN apt install -y vim

#Install dns utils
RUN apt install -y dnsutils

#Install cmake
RUN apt install -y cmake

#Install is python3
RUN apt -y install python-is-python3

# Install SSH Server
RUN apt install -y openssh-client openssh-server

# Configure ssh server for passwordless
RUN sed -i 's/#PasswordAuthentication yes/PasswordAuthentication no\
    ↪ nChallengeResponseAuthentication no\n/g' /etc/ssh/sshd_config
RUN sed -i 's/UsePAM yes/UsePAM no/g' /etc/ssh/sshd_config
RUN sed -i 's/#AuthorizedKeysFile/AuthorizedKeysFile/g' /etc/ssh/sshd_config
RUN sed -i 's/# PasswordAuthentication yes/ PasswordAuthentication no/g' /etc/ssh/ssh_config
RUN sed -i 's/ HashKnownHosts yes/ HashKnownHosts no/g' /etc/ssh/ssh_config

# Install build essentials
RUN apt install -y build-essential
RUN apt -y install gcc g++ gfortran

# Install Open MPI
RUN mkdir /opt/src/ && cd /opt/src/ && curl -o openmpi-4.1.6.tar.bz2 https://download.open-
    ↪ mpi.org/release/open-mpi/v4.1/openmpi-4.1.6.tar.bz2 && tar -jxf openmpi-4.1.6.tar.bz2
RUN cd /opt/src/openmpi-4.1.6 && ./configure --prefix=/opt/.openmpi && make all && make
    ↪ install

# Generate shell script to update .bashrc file
```

```
RUN printf "echo 'Generating bash inizialization file...' \
\nexport PATH="\$PATH:/opt/.openmpi/bin" \
\nexport LD_LIBRARY_PATH="\$LD_LIBRARY_PATH:/opt/.openmpi/lib/" \
\nexport OMPI_ALLOW_RUN_AS_ROOT=1 \
\nexport OMPI_ALLOW_RUN_AS_ROOT_CONFIRM=1 \
\ncat /dev/null > ~/.bashrc \
\necho export PATH="\$PATH:/opt/.openmpi/bin" >> ~/.bashrc \
\necho export LD_LIBRARY_PATH="\$LD_LIBRARY_PATH:/opt/.openmpi/lib/" >> ~/.bashrc \
\necho export OMPI_ALLOW_RUN_AS_ROOT=1 >> ~/.bashrc \
\necho export OMPI_ALLOW_RUN_AS_ROOT_CONFIRM=1 >> ~/.bashrc \
\necho 'bash inizialization file generated on ~/.bashrc successfully'" > /bin/generateBashInit.sh
RUN chmod a+x /bin/generateBashInit.sh

# Generate shell script to update openmpi hosts file
COPY ../.misc/generateHostsFileScriptOpenmpi.sh /bin/generateHostsFileScriptOpenmpi.sh
RUN chmod a+x /bin/generateHostsFileScriptOpenmpi.sh

# Generate shell script to generate ssh keys
COPY ../.misc/generateKeysFileScript.sh /bin/generateKeysFileScript.sh
RUN chmod a+x /bin/generateKeysFileScript.sh

# Generate shell script to prepare knownHosts file
COPY ../.misc/generateKnownHostsFile.sh /bin/generateKnownHostsFile.sh
RUN chmod a+x /bin/generateKnownHostsFile.sh

# Generate shell script to prepare etc/hosts file
COPY ../.misc/generateEtcHostsFileScript.sh /bin/generateEtcHostsFileScript.sh
RUN chmod a+x /bin/generateEtcHostsFileScript.sh

# Generate shell script to start ssh server
COPY ../.misc/startServerSSH.sh /bin/startServerSSH.sh
RUN chmod a+x /bin/startServerSSH.sh

# Generate shell script to probe node
COPY ../.misc/testNodeSSH.sh /bin/testNodeSSH.sh
RUN chmod a+x /bin/testNodeSSH.sh

# Generate shell script to start cluster
RUN printf "echo 'Initializing cluster...' \
\n/bin/generateHostsFileScriptOpenmpi.sh \$CLUSTERNODESCOUNT \$CLUSTERNAME \
    ↪ $CORESPERNODE \
\n/bin/generateKeysFileScript.sh \$CLUSTERNAME\
\n/bin/startServerSSH.sh \
\n/bin/generateKnownHostsFile.sh \$CLUSTERNODESCOUNT \$CLUSTERNAME \
\n/bin/generateEtcHostsFileScript.sh \$CLUSTERNODESCOUNT \$CLUSTERNAME \
\n/bin/generateBashInit.sh \
\necho Keep node running waiting for jobs... \
\ntrap : TERM INT; sleep infinity & wait" > /bin/startClusterScript.sh
RUN chmod a+x /bin/startClusterScript.sh

# Keep container alive
CMD exec /bin/bash -c "/bin/startClusterScript.sh"
```

8.19 Anexo 19. Especificación del archivo .dockerfile para imagen SLURM

```
FROM ubuntu:22.04

# Update APT
RUN apt update

# Upgrade APT
RUN apt upgrade -y

# Install git
RUN apt install -y git

# Install curl
RUN apt install -y curl

#Install vim
RUN apt install -y vim

#Install dns utils
RUN apt install -y dnsutils

#Install cmake
RUN apt install -y cmake

#Install is python3
RUN apt -y install python-is-python3

# Install SSH Server
RUN apt install -y openssh-client openssh-server

# Configure ssh server for passwordless
RUN sed -i 's/#PasswordAuthentication yes/PasswordAuthentication no\
    ↪ nChallengeResponseAuthentication no\n/g' /etc/ssh/sshd_config
RUN sed -i 's/UsePAM yes/UsePAM no/g' /etc/ssh/sshd_config
RUN sed -i 's/#AuthorizedKeysFile/AuthorizedKeysFile/g' /etc/ssh/sshd_config
RUN sed -i 's/# PasswordAuthentication yes/ PasswordAuthentication no/g' /etc/ssh/ssh_config
RUN sed -i 's/ HashKnownHosts yes/ HashKnownHosts no/g' /etc/ssh/ssh_config

# Install build essentials
RUN apt install -y build-essential
RUN apt -y install gcc g++ gfortran

# Create users for munge and slurm
RUN export SLURMUSER=990 && export MUNGEUSER=991 && groupadd -g $SLURMUSER
    ↪ slurm && useradd -m -c "SLURM workload manager" -d /var/lib/slurm -u
    ↪ $SLURMUSER -g slurm -s /bin/bash slurm && groupadd -g $MUNGEUSER munge &&
    ↪ useradd -m -c "MUNGE Uid 'N' " -d /var/lib/munge -u $MUNGEUSER -g munge -s /
    ↪ sbin/nologin munge
```

```
# Install Munge it will autogenerate key
RUN apt -y install munge

# Copying misc files
COPY ../.misc/slurm.conf /opt/operator/misc/slurm.conf

# Generate shell script to update .bashrc file
RUN printf "echo 'Generating bash inizialization file...' \
\nexport HYDRA_HOST_FILE=/nfs/hosts.mpich \
\ncat /dev/null > ~/.bashrc \
\necho export HYDRA_HOST_FILE=/nfs/hosts.mpich >> ~/.bashrc \
\necho 'bash inizialization file generated on ~/.bashrc successfully'" > /bin/generateBashInit.sh
RUN chmod a+x /bin/generateBashInit.sh

# Generate shell script to update mpich hosts file
COPY ../.misc/generateHostsFileScriptMpich.sh /bin/generateHostsFileScriptMpich.sh
RUN chmod a+x /bin/generateHostsFileScriptMpich.sh

# Generate shell script to generate ssh keys
COPY ../.misc/generateKeysFileScript.sh /bin/generateKeysFileScript.sh
RUN chmod a+x /bin/generateKeysFileScript.sh

# Generate shell script to prepare knownHosts file
COPY ../.misc/generateKnownHostsFile.sh /bin/generateKnownHostsFile.sh
RUN chmod a+x /bin/generateKnownHostsFile.sh

# Generate shell script to prepare etc/hosts file
COPY ../.misc/generateEtcHostsFileScript.sh /bin/generateEtcHostsFileScript.sh
RUN chmod a+x /bin/generateEtcHostsFileScript.sh

# Generate shell script to start ssh server
COPY ../.misc/startServerSSH.sh /bin/startServerSSH.sh
RUN chmod a+x /bin/startServerSSH.sh

# Generate shell script to probe node
COPY ../.misc/testNodeSSH.sh /bin/testNodeSSH.sh
RUN chmod a+x /bin/testNodeSSH.sh

COPY ../.misc/installSlurm.sh /bin/installSlurm.sh
RUN chmod a+x /bin/installSlurm.sh

# Generate shell script to start cluster
RUN printf "echo 'Initializing cluster...' \
\n/bin/generateHostsFileScriptMpich.sh \$CLUSTERNODESCOUNT \$CLUSTERNAME \
    ↪ $CORESPERNODE \
\n/bin/generateKeysFileScript.sh \$CLUSTERNAME\
\n/bin/startServerSSH.sh \
\n/bin/generateKnownHostsFile.sh \$CLUSTERNODESCOUNT \$CLUSTERNAME \
\n/bin/generateEtcHostsFileScript.sh \$CLUSTERNODESCOUNT \$CLUSTERNAME \
\n/bin/installSlurm.sh \$CLUSTERNODESCOUNT \$CLUSTERNAME \$CORESPERNODE \
\n apt -y install mpich \
\n/bin/generateBashInit.sh \
```

```
\necho Keep node running waiting for jobs... \
\ntrap : TERM INT; sleep infinity & wait" > /bin/startClusterScript.sh
RUN chmod a+x /bin/startClusterScript.sh

# Keep container alive
CMD exec /bin/bash -c "/bin/startClusterScript.sh"
```

8.20 Anexo 20. Especificación de CRD de clúster de OpenMPI con miniconda para entrenamiento de I.A

```
apiVersion: crisarias.com/v1alpha1
kind: MpiCluster
metadata:
  labels:
    app.kubernetes.io/name: barvacluster
    app.kubernetes.io/instance: barvacluster-instance
    app.kubernetes.io/part-of: kubernetes-mpi-operator
    app.kubernetes.io/managed-by: kustomize
    app.kubernetes.io/created-by: kubernetes-mpi-operator
  name: barvacluster
spec:
  clusterName: "barva"
  clusterNodeImage: "docker.io/crisarias/mpicluster-openmpi-conda-operator-base:latest"
  clusterNodesCount: 2
  coresPerNode: 5
  memoryPerNode: 58000
  sharedMemoryPerNode: 2000
  sshPort: 22
  localVolumePath: "/mnt/beegfs"
  localVolumeCapacity: 500
  enableNvidiaGPU: true
  numGPUs: 1
  nameGPUs: GRID-A100D-3-40C-MIG-3g.40gb
```

8.21 Anexo 21. Especificación de ambiente virtual de conda para entrenamiento de I.A

```
apiVersion: crisarias.com/v1alpha1
kind: MpiCluster
metadata:
  labels:
    app.kubernetes.io/name: barvacluster
    app.kubernetes.io/instance: barvacluster-instance
    app.kubernetes.io/part-of: kubernetes-mpi-operator
    app.kubernetes.io/managed-by: kustomize
    app.kubernetes.io/created-by: kubernetes-mpi-operator
```

```
  name: barvacluster
spec:
  clusterName: "barva"
  clusterNodeImage: "docker.io/crisarias/mpicluster-openmpi-conda-operator-base:latest"
  clusterNodesCount: 2
  coresPerNode: 5
  memoryPerNode: 58000
  sharedMemoryPerNode: 2000
  sshPort: 22
  localVolumePath: "/mnt/beegfs"
  localVolumeCapacity: 500
  enableNvidiaGPU: true
  numGPUs: 1
  nameGPUs: GRID-A100D-3-40C-MIG-3g.40gb
```

Bibliografía

[1] Consul by hashicorp. https://www.consul.io/. Accessed: July 25th, 2023.

[2] containerd. https://containerd.io/. Accessed: 2023-07-31.

[3] June 2022 — top500. URL https://www.top500.org/lists/top500/2022/06/. Accessed: 2023-08-13.

[4] Open mpi frequently asked questions. https://www.open-mpi.org/faq/?category=building. Accessed: April 20, 2024.

[5] Operator framework official documentation. URL https://sdk.operatorframework.io/docs/overview/. Accessed: 2023-08-07.

[6] Singularity user guide. https://docs.sylabs.io/guides/3.5/user-guide/introduction.html. Accessed: 2023-07-29.

[7] Slurm workload manager. https://slurm.schedmd.com/overview.html. Accessed: 2023-07-29.

[8] Sylabs wlm operator. https://github.com/sylabs/wlm-operator/blob/master/README.md. Accessed: 2023-07-29.

[9] Top500 — the linpack benchmark. URL https://www.top500.org/lists/top500/2022/06/. Accessed: 2024-02-25.

[10] Top500 rank. URL https://en.wikipedia.org/wiki/TOP500. Accessed: 2023-08-13.

[11] Torque resource manager. https://github.com/adaptivecomputing/torque. Accessed: 2023-07-29.

[12] Rawan Aljamal, Ali El-Mousa, and Fahed Jubair. A comparative review of high-performance computing major cloud service providers. In *2018 9th International Conference on Information and Communication Systems (ICICS)*, pages 181–186, 2018.

[13] Alphabet. Google cloud, cloud computing services. urlhttps://cloud.google.com/. Accessed: 2023-08-19.

[14] Amazon. Amazon web services, cloud computing services. urlhttps://aws.amazon.com/. Accessed: 2023-08-19.

[15] LambdaLabsML Arias Chaves Cristian. Ai benchmarks. https://github.com/Crisarias/ai-benchmarks, 2024.

[16] The Kubernetes Authors. Kubernetes official documentation, Oct 2022. URL https://kubernetes.io/docs. Accessed: 2023-08-07.

[17] AWE and the University of Warwick. Cloverleaf. https://uk-mac.github.io/CloverLeaf/. Version 1.3, Accessed: 2024-03-28.

[18] DH Bailey, E Barszcz, JT Barton, DS Browning, RL Carter, RA Fatoohi, PO Frederickson, TA Lasinski, HD Simon, V Venkatakrishnan, et al. The nas parallel benchmarks rnr-94-007. *NASA Advanced Supercomputing Division, Tech. Rep*, 1994.

[19] Pavan Balaji, Wesley Bland, William Gropp, Rob Latham, Huiwei Lu, Antonio J Pena, Ken Raffenetti, Sangmin Seo, Rajeev Thakur, and Junchao Zhang. Mpich user's guide. *Argonne National Laboratory*, 2014.

[20] Forest Baskett and John L. Hennessy. Microprocessors: From desktops to supercomputers. *Science*, 261(5123):864–871, 1993. URL https://www.science.org/doi/abs/10.1126/science.261.5123.864.

[21] Angel Beltre, Pankaj Saha, Madhusudhan Govindaraju, Andrew J. Younge, and Ryan Eric Grant. Enabling hpc workloads on cloud infrastructure using kubernetes container orchestration mechanisms. In *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*. IEEE, 2019.

[22] Brendan Burns, Joe Beda, Kelsey Hightower, and Lachlan Evenson. *Kubernetes: Up and Running*. O'Reilly Media, Inc., 3 edition, 2022. URL https://www.oreilly.com/library/view/kubernetes-up-and/9781098110192/.

[23] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *ACM Queue*, 14:70–93, 2016.

[24] Ralph H. Castain, David Solt, Joshua Hursey, and Aurelien Bouteiller. Pmix: Process management for exascale environments. In *Proceedings of the 24th European MPI Users' Group Meeting*, EuroMPI '17, New York, NY, USA, 2017. Association for Computing Machinery. URL https://doi.org/10.1145/3127024.3127027.

[25] Kubernetes Community. *The Kubebuilder Book*. 2024. URL https://book.kubebuilder.io/.

[26] Arias Chaves Cristian. Kubernetes mpi operator. https://github.com/Crisarias/kubernetes-mpi-operator, 2024.

[27] Arias Chaves Cristian. Kubernetes mpi operator experiments. https://github.com/Crisarias/kubernetes-mpi-operator-experiments, 2024.

[28] Arias Chaves Cristian. Kubernetes mpi operator samples. https://github.com/Crisarias/kubernetes-mpi-operator-samples, 2024.

[29] Paul Stewart Crozier, Heidi K Thornquist, Robert W Numrich, Alan B Williams, Harold Carter Edwards, Eric Richard Keiter, Mahesh Rajan, James M Willenbring, Douglas W Doerfler, and Michael Allen Heroux. Improving performance via mini-applications. 9 2009. URL https://www.osti.gov/biblio/993908.

[30] M. de Bayser and Renato Cerqueira. Integrating mpi with docker for hpc. In *2017 IEEE International Conference on Cloud Engineering (IC2E)*, 2017.

[31] Jason Dobies and Joshua Wood. *Kubernetes Operators: Automating the Container Orchestration Platform*. O'Reilly Media, Inc., 2020.

[32] Jack J Dongarra. The linpack benchmark: An explanation. In *International Conference on Supercomputing*, pages 456–474. Springer, 1987.

[33] Michael Eder. Hypervisor-vs. container-based virtualization. *Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM)*, 1, 2016.

[34] Steven Farrell, Murali Emani, Jacob Balma, Lukas Drescher, Aleksandr Drozd, Andreas Fink, Geoffrey Fox, David Kanter, Thorsten Kurth, Peter Mattson, Dawei Mu, Amit Ruhela, Kento Sato, Koichi Shirahata, Tsuguchika Tabaru, Aristeidis Tsaris, Jan Balewski, Ben Cumming, Takumi Danjo, Jens Domke, Takaaki Fukai, Naoto Fukumoto, Tatsuya Fukushi, Balazs Gerofi, Takumi Honda, Toshiyuki Imamura, Akihiko Kasagi, Kentaro Kawakami, Shuhei Kudo, Akiyoshi Kuroda, Maxime Martinasso, Satoshi Matsuoka, Henrique Mendonça, Kazuki Minami, Prabhat Ram, Takashi Sawada, Mallikarjun Shankar, Tom St. John, Akihiro Tabuchi, Venkatram Vishwanath, Mohamed Wahib, Masafumi Yamazaki, and Junqi Yin. Mlperf™ hpc: A holistic benchmark suite for scientific machine learning on hpc systems. In *2021 IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC)*, pages 33–45, 2021.

[35] Huiyu Feng, Rob F Van der Wijngaart, Rupak Biswas, and Catherine Mavriplis. Unstructured adaptive (ua) nas parallel benchmark, version 1.0. *NASA Technical Report NAS-04*, 6, 2004.

[36] Michael A Frumkin and Leonid Shabano. Arithmetic data cube as a data intensive benchmark. 2003.

[37] Roberto Gozalo-Brizuela and Eduardo C Garrido-Merchan. Chatgpt is not all you need. a state of the art review of large generative ai models. *arXiv preprint arXiv:2301.04655*, 2023.

[38] Richard L. Graham, Timothy S. Woodall, and Jeffrey M. Squyres. Open mpi: A flexible high performance mpi. In Roman Wyrzykowski, Jack Dongarra, Norbert Meyer, and Jerzy Waśniewski, editors, *Parallel Processing and Applied Mathematics*, pages 228–239, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[39] William Gropp. *Tutorial on mpi: The message-passing interface*. Argonne National Laboratory, 1995.

[40] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Computing*, 22(6):789–828, 1996. URL https://www.sciencedirect.com/science/article/pii/0167819196000245.

[41] Georg Hager and Gerhard Wellein. *Introduction to high performance computing for scientists and engineers*. CRC Press, 2010.

[42] Jan Heichler. An introduction to beegfs. *Introductionto BeeGFS by ThinkParQ. pdf*, 2014.

[43] Michael Allen Heroux, Jack Dongarra, and Piotr Luszczek. Hpcg benchmark technical specification. Technical Report SAND2013-8752, Sandia National Lab. (SNL-NM), Albuquerque, NM (United States), 2013. URL ^1^.

[44] Raphaël Hertzog and Roland Mas. *The Debian Administrator's Handbook*. Freexian SARL, 2017. URL https://debian-handbook.info/.

[45] Joshua Hursey. Design considerations for building and running containerized mpi applications. In *2020 2nd International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*, pages 35–44, 2020.

[46] Andrey Ignatov. Ai benchmark alpha. https://ai-benchmark.com/alpha.html, 2024.

[47] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[48] Sandia National Laboratories. Mantevo benchmarks. https://github.com/Crisarias/mantevo-benchmarks, 2024.

[49] Canocinal Ltd. Ubuntu documentation. urlhttps://docs.ubuntu.com/. Accessed: 2024-03-23.

[50] Zoya Masih. On demand file systems with beegfs. 2023.

[51] Peter Mattson, Christine Cheng, Gregory Diamos, Cody Coleman, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, et al. Mlperf training benchmark. *Proceedings of Machine Learning and Systems*, 2:336–349, 2020.

[52] Esteban Meneses. Parallel computing (mc-8836). Course of Computer Science Master Degree, 2021. Costa Rica Institute of Technology, San José. Contact: esteban.meneses@acm.org.

[53] Microsoft. Microsoft azure, cloud computing services. urlhttps://azure.microsoft.com/en-us/. Accessed: 2023-08-19.

[54] Nvidia Microsoft. Supercomputing performance in the cloud. Technical report, Microsoft, 2021.

[55] mpitutorial. Mpi tutorial. https://github.com/mpitutorial/mpitutorial. Accessed: 2024-03-28.

[56] Nigel Poulton. *The Kubernetes Book: 2023 Edition*. Amazon.com Services LLC, 2023.

[57] Cristian Ramon-Cortes, Albert Serven, Jorge Ejarque, Daniele Lezzi, and Rosa M Badia. Transparent orchestration of task-based parallel applications in containers platforms. *Journal of Grid Computing*, 16:137–160, 2018.

[58] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115:211–252, 2015.

[59] Drew Schmidt, Junqi Yin, Michael Matheson, Bronson Messer, and Mallikarjun Shankar. Defining big data analytics benchmarks for next generation supercomputers. *arXiv preprint arXiv:1811.02287*, 2018.

[60] Muhammad Shafiq and Zhaoquan Gu. Deep residual learning for image recognition: A survey. *Applied Sciences*, 12(18):8972, 2022.

[61] Fact Sheet. Collaboration of oak ridge, argonne, and livermore (coral), 2021.

[62] Lizhen Shi and Zhong Wang. Computational strategies for scalable genomics analysis. *Genes*, 10(12):1017, 2019.

[63] Rob F Van der Wijngaart and Michael Frumkin. Nas grid benchmarks version 1.0. *NASA Ames Research Center TR NAS-02-005*, 2002.

[64] Rob F vanderWijngaart and Jin Haopiang. Nas parallel benchmarks, multi-zone versions. In *Supercomputing 2003*, 2003.

[65] Vultr. About vultr, a global cloud for you. https://www.vultr.com/company/about-us/, 2014. Accessed: 2024-03-28.

[66] Parkson Wong and R Der Wijngaart. Nas parallel benchmarks i/o version 2.4. *NASA Ames Research Center, Moffet Field, CA, Tech. Rep. NAS-03-002*, page 91, 2003.

[67] Zhiwei Xu, Xuebin Chi, and Nong Xiao. High-performance computing environment: a review of twenty years of experiments in China. *National Science Review*, 3(1):36–48, 01 2016. URL https://doi.org/10.1093/nsr/nww001.

[68] Hsi-En Yu and Weicheng Huang. Building a virtual HPC cluster with auto scaling by the docker. *CoRR*, abs/1509.08231, 2015. URL http://arxiv.org/abs/1509.08231.

[69] Naweiluo Zhou, Yiannis Georgiou, Marcin Pospieszny, Li Zhong, Huan Zhou, Christoph Niethammer, Branislav Pejak, Oskar Marko, and Dennis Hoppe. Container orchestration on hpc systems through kubernetes. *Journal of Cloud Computing*, 10(16), 2021. URL https://journalofcloudcomputing.springeropen.com/articles/10.1186/s13677-021-00231-z.