Coplanificación de procesos maleables de aprendizaje automático mediante contenedores

Universidad Nacional de La Plata

Leandro Ariel Libutti
Directores: Dr. Francisco Igual
Dra. Laura De Giusti

30 de mayo de 2023

Agradecimientos

Quiero agradecer a toda mi familia y seres queridos por el apoyo incondicional en todo el transcurso de esta carrera, en especial a mi madre Graciela, a mi padre Vicente, a mis hermanos Melina y Mauro, a mi abuela Tita y a mis sobrinos Fausto y Luca. A mi novia y compañera de vida Fátima. A mis hermanos del alma Nicolás, Jeremías, Christian, Camila y Eugenia.

Agradecer al Laboratorio de Investigación en Informática LIDI por darme la oportunidad de formar parte de un equipo de trabajo, que no solo me ha permitido perfeccionarme, sino que también se han generado grandes amistades.

A todo el grupo de Algoritmos Paralelos del LIDI por la buena predisposición y compañerismo, en especial Franco, Enzo, Laura y Marcelo.

A la Universidad Complutense de Madrid por recibirme con los brazos abiertos y dedicarme tiempo, siempre dispuestos a cooperar en la formación de recursos humanos y con un equipo de trabajo con el que disfruté las estadías. Especial agradecimiento a Francisco y Luis, gracias a ellos es posible el desarrollo de esta tesis y por compartir lindos momentos en las estancias en Madrid.

Resumen

En las últimas décadas, el avance de los algoritmos de Aprendizaje Automático (*Machine Learning, ML)* ha despertado el interés en la búsqueda de estrategias que logren acelerar los procesos de entrenamiento e inferencia típicos de este ámbito, especialmente cuando éstos surgen en servidores con un elevado grado de paralelismo, complejidad y heterogeneidad. Típicamente, estos procesos se realizan a través de entornos de trabajo (*frameworks*) de propósito específico tales como Tensoflow, Keras, Caffe o Pytorch. A día de hoy, Tensorflow es uno de los frameworks más utilizados por parte de los desarrolladores de algoritmos de ML. Desde el punto de vista de rendimiento computacional, existen entre sus parámetros de configuración diversas opciones de configuración relativas al grado de paralelismo, que pueden ser fijadas a priori, pero no pueden ser reconfiguradas durante el proceso de entrenamiento o inferencia, por lo que se consideran parámetros *rígidos*. En situaciones en las que múltiples instancias del *framework* se ejecutan en una misma máquina, dicha rigidez puede derivar en problemas tales como *oversuscription*, degradamiento del rendimiento del sistema y/o aplicación e infrautilización de los recursos computacionales. Por lo tanto, resulta importante agregar un grado de *elasticidad* en Tensorflow, permitiendo aumentar la productividad del sistema en entornos dinámicos multiprogramados.

Por otro lado, la utilización de contenedores como método de virtualización ligera permite una mejor administración de los recursos y portabilidad. Existen múltiples planificadores que permiten aprovechar los beneficios de los contenedores, pero solo permiten llevar a cabo una asignación estática de recursos en el momento de su creación, y en algunos casos reasignación de recursos en tiempo de ejecución; en cualquier caso, las aplicaciones en ejecución dentro del contenedor no se encuentran preparadas para reaccionar ante dicho evento, y por tanto no se adaptarán en ningún caso a la modificación en los recursos asignados al contenedor.

Por todo lo comentado anteriormente, este trabajo propone el diseño e implementación de un mecanismo completo de elasticidad en el uso de recursos computacionales en el *framework* Tensorflow, permitiendo la reasignación dinámica de núcleos de cómputo durante la ejecución del algoritmo de ML. Además, se extiende el uso de la elasticidad a contenedores con la implementación de un controlador/cliente que permita administrar los recursos

computacionales asignados a los algoritmos de ML que ejecutan internamente. Por último, se implementa un planificador de contenedores elásticos con el fin de gestionar dinámicamente los recursos del sistema entre todos los contenedores activos y definir políticas de planificación que favorezcan el rendimiento global del sistema.

Índice general

1. Introducción 1
- 1.1. Motivación 1
- 1.2. Objetivos 3
- 1.3. Organización 4

2. Marco Teórico 6
- 2.1. Contenedores 6
- 2.2. Elasticidad/(re)asignación de recursos en contenedores 10
- 2.3. Elasticidad en cloud 11
- 2.4. Elasticidad en aplicaciones 12

3. Integración de elasticidad en Tensorflow 14
- 3.1. El framework Tensorflow 14
  - 3.1.1. Grafo computacional 14
  - 3.1.2. Modelo de ejecución 16
  - 3.1.3. Optimizaciones 20
  - 3.1.4. Visualización de trazas de ejecución 21
- 3.2. Diseño e integración de elasticidad en Tensorflow 23
  - 3.2.1. Diseño de la solución 23
  - 3.2.2. Implementación 25

4. Coplanificación de aplicaciones elásticas 28
- 4.1. Diseño de la solución 29
  - 4.1.1. Funcionamiento del planificador 30
- 4.2. Políticas de planificación y reasignación de recursos 33
  - 4.2.1. Política de planificación 33
  - 4.2.2. Políticas de asignación/reasignación de recursos 34
- 4.3. Implementación de la solución 37
  - 4.3.1. Módulo del cliente 37

4.3.2. Módulo del servidor del planificador 42
4.3.3. Módulo del visor de trazas 57

5. Resultados **63**
5.1. Especificaciones para las pruebas 63
5.2. Elasticidad en Tensorflow 64
5.2.1. Elasticidad del inter paralelismo 64
5.2.2. Elasticidad del Intra Paralelismo 66
5.3. Evaluación de *oversubscription* 68
5.4. Coplanificación de contenedores elásticos 70
5.4.1. Elasticidad en contenedores 70
5.4.2. Planificación de contenedores 71

6. Conclusiones y trabajo futuro **78**
6.1. Conclusiones 78
6.2. Trabajos Futuros 82

A. Instrucciones de uso del planificador **83**
A.1. Requerimientos de sistema 83
A.2. Lanzamiento del servidor 84

B. Tablas de métricas de los experimentos **87**

Índice de figuras

2.1. Contenedores vs Virtualización clásica. 7
2.2. Ciclo de orquestación de contenedores. 9

3.1. Grafo computacional en TF. 16
3.2. Planificación de nodos listos para ejecutar. 18
3.3. Proceso de ejecución de nodos planificados. 19
3.4. Poda del grafo computacional. 20
3.5. Grafo computacional en Tensorboard. 22
3.6. Ventana Profile de Tensorboard. 23
3.7. Función *Trace Viewer* en Tensorboard. 24
3.8. Nueva planificación de nodos listos. 27

4.1. Modelo del planificador de contenedores. 31
4.2. Esquema de planificación de contenedores. 31
4.3. Modelo del planificador de contenedores. 32
4.4. Proceso de planificación de la política FCFS 34
4.5. Reasignación de recursos. 36
4.6. Diagrama de Gantt que visualiza el porcentaje completado de cada trabajo. 59
4.7. Gráfico de contenedor y las épocas del algoritmo de ML. . . . 60

5.1. Batches para la evaluación del inter paralelismo. 66
5.2. Batches para la evaluación del intra paralelismo. 67
5.3. Traza de ejecución de TF original en contenedor sin oversubscription. 70
5.4. Traza de ejecución de TF original en contenedor con oversubscription. 70
5.5. Trazas de ejecución para validar la elasticidad de TF en contenedores. 72

A.1. Interacción de hilos del planificador y contenedor lanzado. . . 86

Índice de tablas

5.1. Tiempos de ejecución de batch por etapa en cada versión TF. 67
5.2. Tiempos de ejecución para asignación fija de recursos en contenedor y variación de paralelismo de la aplicación TF. 69
5.3. Tiempos medio de ejecución para lanzamiento de contenedor con diferente versión de TF. 71
5.4. Métricas de escenario de alta demanda de peticiones y baja cantidad de recursos por contenedor. 74
5.5. Métricas de escenario de alta demanda de peticiones y baja cantidad de recursos por contenedor. 75
5.6. Métricas de escenario de baja demanda de peticiones y baja cantidad de recursos por contenedor. 76
5.7. Métricas de escenario baja demanda de peticiones y alta cantidad de recursos por contenedor. 77

B.1. Tabla completa de las métricas para escenario de alta demanda de peticiones y baja cantidad de recursos por contenedor. 87
B.2. Tabla completa de las métricas para escenario de alta demanda de peticiones y alta cantidad de recursos por contenedor. . 88
B.3. Tabla completa de las métricas para escenario de baja demanda de peticiones y baja cantidad de recursos por contenedor. 88
B.4. Tabla completa de las métricas para escenario de baja demanda de peticiones y alta cantidad de recursos por contenedor. . 89

Lista de acrónimos

CPU Unidad Central de Procesamiento. 2, 8, 9, 11, 17, 18, 22, 23, 63, 83

FPGA Arreglo de Compuertas Lógicas Programables. 2

GPU Unidad de Procesamiento Gráfico. 2, 15, 17, 18, 21–23, 82

HPC Cómputo de Alto Rendimiento. 7, 8, 10, 78

INTER Inter Paralelismo. 19, 24, 43, 47, 48, 53, 64–69, 79, 81, 85

INTRA Intra Paralelismo. 19, 24, 43, 47, 48, 53, 54, 64, 66–69, 79, 81, 85

ML Machine Learning. 1–4, 10, 14, 15, 17, 21–23, 29, 30, 32–34, 37, 40, 41, 57, 59, 60, 63, 64, 78–82

TF Tensorflow. 14–17, 19, 20, 23–26, 29, 30, 38–41, 43, 47–49, 52, 53, 63, 64, 67–77, 79–81, 84, 85

TPU Unidades de Procesamiento Tensorial. 17, 22

Capítulo 1

Introducción

1.1. Motivación

En las últimas décadas, los algoritmos de ML están siendo usados en una amplia variedad de aplicaciones tales como reconocimiento de imágenes, segmentación, reconocimiento de voz, procesamiento de idiomas, entre otros [1, 2, 3, 4]. Este crecimiento exponencial está directamente relacionado con tres avances fundamentales [5]:

- El desarrollo de **mejores algoritmos** con aplicaciones directas en muchos campos de la ciencia y la ingeniería.
- La disponibilidad de **cantidades masivas de datos** y la factibilidad de almacenarlos y analizarlos de manera eficiente.
- La aparición de **nuevas arquitecturas de hardware**, típicamente paralelas y/u heterogéneas, que permiten una adecuada explotación de nuevos algoritmos en grandes conjuntos de datos en un tiempo adecuado.

En general, construir un modelo de ML efectivo es un proceso complejo y lento que implica determinar el algoritmo adecuado y modelar la arquitectura ajustando los hiperparámetros del modelo [6]. En la actualidad existen dos tipos de parámetros del modelo:

- Los que pueden ser actualizados en el proceso de entrenamiento del modelo, por ejemplo, los pesos de las red neuronal.
- Los especificados antes del entrenamiento del modelo porque definen la arquitectura del mismo, por ejemplo, la tasa de aprendizaje.

Además, los procesos de entrenamiento de redes neuronales presentan típicamente características que los hacen susceptibles de ser acelerados en plataformas de alto rendimiento: presentan un grado de paralelismo masivo y homogéneo, y permiten ser planteados en base a primitivas de cómputo de alto rendimiento. Típicamente, el entrenamiento de los modelos se realiza a través de *frameworks* de ML tales como Tensorflow [7], Caffe [8], Keras [9] y Pytorch [10] que permiten ocultar detalles de implementación al usuario manteniendo un alto rendimiento. Hoy en día, Tensorflow es uno de los frameworks de mayor utilización por parte de los programadores de algoritmos de ML. Entre sus hiperparámetros de ejecución de un modelo se encuentra la selección de las arquitecturas de hardware que se utilizarán (CPU, GPU, FPGA) y cuántos recursos computacionales de cada una de ellas se desea utilizar, típicamente en base al número de núcleos de computación a explotar. Uno de los principales problemas que presenta la selección de los recursos computacionales en este tipo de *software* es que se realizan antes de la ejecución del algoritmo, por lo cual, una vez lanzado no es posible su modificación. Es posible afirmar que Tensorflow (al igual que otros entornos de similar naturaleza), son **estáticos** desde el punto de vista del uso de recursos computacionales.

El uso estático de recursos no es en si misma problemática, siempre que las aplicaciones se ejecuten de forma aislada en el computador. Sin embargo, en entornos en los que múltiples aplicaciones conviven en el mismo computador, y tanto sus puntos de llegada al sistema como los requisitos a nivel de recurso que éstas presentan puedan variar, puede llevar a situaciones de *oversuscription* (utilización de más recursos computacionales de los realmente disponibles), degradando el rendimiento global del sistema y en particular de cada aplicación. Así, si fuese posible modificar la cantidad de recursos dinámicamente a lo largo de la ejecución de la aplicación con simples comunicaciones con el *framework*, sería posible ajustar, bajo demanda, los niveles de paralelismo reduciendo las posibilidades de *oversuscription*; del mismo modo, existe el caso en que una aplicación utilice (por petición del usuario) menos recursos de los realmente disponibles, bien sea porque éstos han sido liberados por otras aplicaciones, o porque el usuario ha solicitado una cantidad menor de la que realmente se le puede ofrecer. En cualquiera de los dos casos, la adición de capacidades de **maleabilidad** o **elasticidad** en *software* paralelo parece un requisito indispensable para aumentar la productividad del sistema en entornos dinámicos con múltiples aplicaciones conviviendo (usualmente referidos como *multi-tenant systems* [11].

Cuando tenemos varios algoritmos de ML ejecutando en una misma máquina, es importante una planificación eficiente de los recursos compu-

tacionales para evitar situaciones que penalicen el rendimiento de los mismos, por ejemplo, el oversubscription generado cuando los recursos utilizados son mayores que los que presenta la máquina.

Así, parece necesario no solo aplicar técnicas de planificación de procesos y gestión de recursos avanzadas que permitan optimizar el rendimiento global del sistema y particular de las aplicaciones, respetando a la vez los requisitos de calidad de servicio (QoS, *quality of service*), sino además siendo consciente de la posibilidad de gestionar aplicaciones maleables. Si se dispone de *frameworks* que permitan administrar dinámicamente los recursos computacionales, estamos en condiciones de desarrollar un planificador orientado a aplicaciones de ML, que sea consciente de esta ventaja y tome decisiones que favorezcan el rendimiento global del sistema.

La utilización de contenedores para la ejecución de algoritmos de ML (u otro tipo de aplicaciones) se ha popularizado como una solución de virtualización ligera que brinda, entre otras ventajas, una mejor administración de los recursos y mayor portabilidad. Existen múltiples técnicas de orquestación de contenedores para proveer un proceso de gestión automatizado, incluyendo administración de recursos, desarrollo, supervisión del estado, equilibrado de carga, seguridad y configuración de la red [12]. En su mayoría basándose en Docker [13], estas técnicas son utilizadas por *frameworks* de orquestación tales como Docker Swarm [14], Kubernetes [15] y Marathon [16].

Sin embargo, y aunque tanto Docker como los anteriores orquestadores permiten la asignación **estática** de recursos en el momento de la creación de los contenedores, y en algunos casos la reasignación dinámica de recursos en tiempo de ejecución, ésta no es efectiva si el software confinado en el contenedor no es consciente de dichos cambios para adaptar (aumentando o disminuyendo) el uso efectivo de recursos en la búsqueda de evitar situaciones de *oversuscription* o baja utilización de recursos. En otras palabras, no existe, a día de hoy, ningún mecanismo para interactuar con *aplicaciones maleables confinadas en contenedores*.

1.2. Objetivos

El objetivo principal que se plantea es lograr una infraestructura que permita, de forma holística, implementar mecanismos de coplanificación de contenedores que ejecuten aplicaciones maleables o elásticas, específicamente enfocado a algoritmos de ML sobre una versión modificada de Tensorflow que permita una selección dinámica de los recursos computacionales.

El objetivo principal se divide en tres objetivos secundarios:

- Modificación del esquema de gestión de recursos dentro de la infraestructura del *framework* Tensorflow para permitir selección dinámica del paralelismo de las operaciones que conforman el modelo de ML.

- Diseño e implementación de un controlador interno de cada contenedor que permita gestionar los recursos computacionales asignados dinámicamente al algoritmo de ML, y de un mecanismo de comunicación entre el sistema y la aplicación confinada en el contenedor.

- Diseño e implementación de un planificador de contenedores que ejecutan algoritmos de ML sobre Tensorflow elástico utilizando técnicas de orquestación que permitan gestionar los recursos computacionales del sistema eficientemente.

1.3. Organización

El resto de este trabajo final se compone de cinco capítulos:

- El capítulo 2 establece el marco teórico del trabajo desarrollado, presentando el uso de contenedores como herramienta para reducir la complejidad y mejorar el costo de migración de aplicaciones entre sistemas con diferentes arquitecturas de hardware, A su vez, se presenta el concepto de elasticidad en contenedores, donde se mencionan los tipos que existen, y su uso en el *cloud* y aplicaciones generales.

- El capítulo 3 introduce los conceptos generales del *framework* Tensorflow tales como la representación de las operaciones de un algoritmo de ML, el modelo de ejecución, optimizaciones aplicadas con librerías especificas y la herramienta para visualización de los modelos de ML. Una vez entendido el funcionamiento del *framework*, se plantea el diseño e implementación de la elasticidad dentro del mismo explicando todos los componentes que fueron necesarios modificar para su funcionamiento.

- El capítulo 4 plantea la necesidad de un planificador de contenedores elástico que permita variar sus recursos dinámicamente, y que a su vez, estos cambios sean reconocidos por las aplicaciones ejecutadas internamente logrando una elasticidad completa, generando mejor uso de los recursos del sistema. Se describe el diseño del planificador, las políticas

necesarias para la atención de los contenedores y su implementación con explicación de sus componentes internos.

- El capítulo 5 se divide en tres secciones. En la primera se mencionan los experimentos y resultados que permiten verificar el correcto funcionamiento de la elasticidad en Tensorflow. La segunda sección, justifica la necesidad de controlar los recursos asignados a los contenedores para evitar el problema de oversubscription. En la última sección, se describen las pruebas realizadas tanto con la versión original como con la versión elástica de TF. Además, se definen las métricas para comparar las pruebas y un análisis de los valores obtenidos para las diferentes políticas de asignación del planificador.

- El capítulo 6 expone las conclusiones de la tesis y el trabajo futuro.

Capítulo 2

Marco Teórico

2.1. Contenedores

La portabilidad del software es una preocupación desde hace mucho tiempo para los desarrolladores. Una de las primeras iniciativas que trató de abordar la complejidad y el costo de migrar software a nuevas plataformas tuvo lugar a finales de la década de 1950 con la creación del lenguaje de programación COBOL [17]. Han pasado más de 60 años desde ese momento y todavía sigue siendo un desafío vigente.

El uso de contenedores permite reforzar la portabilidad incrustando una pila de software completa para ejecutar una aplicación en varios contextos. Al igual que la virtualización clásica, la ejecución de aplicaciones dentro de contenedores provee aislamiento del sistema host y de otros contenedores. Por lo tanto, los derechos de administrador se pueden asignar a los usuarios dentro de un contenedor sin afectar al host.

La virtualización clásica puede conducir a gastos generales significativos, especialmente para procesos relativamente livianos que pueden requerir solo decenas de mega bytes. Estos procesos extras y el procesamiento adicional también impacta en el tiempo de inicio dado de una maquina virtual ya que requiere arrancar un kernel completo, lo cual puede tomar varios minutos. Por el contrario, iniciar un contenedor es esencialmente ejecutar un proceso por lo que normalmente requiere fracciones de segundo. Este inicio rápido puede ser muy útil para entornos muy dinámicos donde las cargas de trabajo pueden necesitar cambiar rápidamente los recursos entre diferentes componentes en función de la demanda[18].

Los contenedores empaquetan directamente la aplicación y sus bibliotecas requeridas para ejecutar su servicio en el motor de Docker que contenido

dentro de un sistema operativo host y una infraestructura de hardware. Sin embargo, la tecnología de virtualización clásica requiere una instalación adicional de un sistema operativo invitado por cada aplicación lo que genera un alto costo de inicio (fig. 2.1).

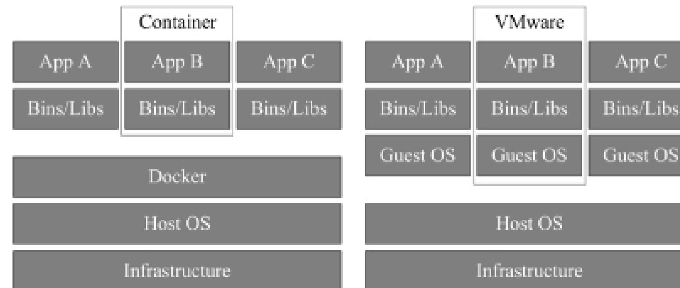


Figura 2.1: Contenedores vs Virtualización clásica.

Los contenedores, además de ayudar a los esfuerzos de portabilidad y evitar un alto overhead de la virtualización [19], también traen otros beneficios a los usuarios de HPC que vale la pena mencionar [20]:

- Encapsulamiento de entornos completos de software; los contenedores incluyen los servicios y bibliotecas necesarias para su correcto despliegue.
- Empaquetado de aplicaciones con diferentes versiones de software y dependencias, evitando software con conflicto de versiones en el host.
- Colaboración simplificada, ya que los contenedores como unidad de software se puede compartir fácilmente.
- Reproducibilidad mejorada, ya que los contenedores son auto-contenidos y se pueden utilizar para replicar resultados consistentemente.
- Portabilidad mejorada de las aplicaciones; los contenedores se ejecutan con la misma configuración en diferentes entornos.
- Flexibilidad de desarrollo mejorada; los contenedores permiten el desarrollo de aplicaciones en entornos que no son de alto rendimiento.
- Despliegue rápido y ejecución a escala; al ser una virtualización liviana, el lanzamiento de los contenedores toma minutos o segundos en la

mayoría de los casos. Asimismo, al ocupar menos espacio de memoria, es posible ejecutar múltiples contenedores simultáneamente.

La portabilidad es también una gran preocupación en el área de HPC debido a que la mayoría de estas aplicaciones tienen que ejecutarse en múltiples plataformas y entornos manteniendo niveles altos de rendimiento computacional. Tras la aparición de Docker 1 en 2013, han surgido varias implementaciones de contenedores destinados al alto rendimiento entre los que se destacan Singularity [21] y Shifter [22].

El panorama de la tecnología de contenedores se está desarrollando y expandiendo rápidamente; sin embargo, estamos lejos de la etapa de madurez y aún quedan muchos desafíos por resolver, por ejemplo: la reducción de los gastos generales de red en comparación con los hipervisores; el intercambio seguro de recursos y el aislamiento para permitir múltiples contenedores; la mejora de las metodologías y herramientas de seguimiento de contenedores; la mejora de las capacidades de migración y ampliación del tiempo de ejecución [23].

Docker es una de las soluciones más populares de contenedores. Aunque enfoques como contenedores de Linux [24] han existido durante mucho tiempo, este nuevo framework es el que popularizó la *contenedorización* como una mejor y más eficiente solución para proporcionar aislamiento para aplicaciones, especialmente para despliegues de entornos en la nube [25].

Para permitir una monitorización y control dinámico de múltiples aplicaciones empaquetadas en contenedores se necesita un orquestador/planificador encargado de dicha tarea. La orquestación de contenedores se ocupa de la gestión en tiempo de ejecución para admitir las fases de implementación, ejecución y mantenimiento. Por lo general, ofrece control de límite de recursos, programación, equilibrio de carga, verificación de estado, tolerancia a fallas y escalado automático (fig. 2.2) [23].

Los orquestadores están compuestos por 4 partes encargadas de administrar diferentes características de los contenedores:

- El controlador de recursos presente en el orquestador permite definir la cantidad de CPU y memoria que se le asigna a cada contenedor, teniendo en cuenta los limites impuestos en la configuración para evitar la interferencia entre todos los contenedores activos.
- El planificador es el encargado de definir la cantidad de contenedores que se ejecutarán por nodo. Se pueden definir políticas donde cada nodo tiene afinidad por ciertas cargas de trabajo o aplicaciones para lograr un reparto que aumente el rendimiento total del sistema.

- El balanceador de carga distribuye la carga de trabajo entre múltiples instancias del contenedor. Existen diferentes políticas y entre las más usadas se encuentra round-robin. Es posible incluir nuevas políticas de balance de carga si el orquestador lo permite.

- El analizador de vida se encarga de controlar que un contenedor sea capaz de responder peticiones. Principalmente chequea las conexiones de los puertos (TCP/UDP/SSH) y la comprobación de recepción y envío de solicitudes por la red.

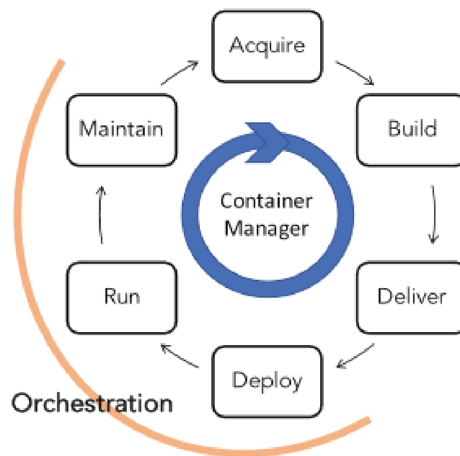


Figura 2.2: Ciclo de orquestación de contenedores.

Además, los orquestadores integran dos funcionalidades importantes cuando se gestiona la ejecución de múltiples contenedores concurrentes:

- La tolerancia a fallas se puede implementar como control de réplica y/o alta disponibilidad controlador. Permite especificar y mantener un número deseado de contenedores. El control de estado se utiliza para determinar cuándo se debe reparar un contenedor defectuoso por el cual se debe destruir y lanzar uno nuevo.

- El escalado automático permite agregar y eliminar contenedores automáticamente. Existen políticas para mantener el umbral de utilización de los recursos del sistema (CPU y memoria). También, en algunos

orquestadores se pueden agregar nuevas políticas que se adapten a las necesidades del usuario.

Entre los frameworks de orquestación de contenedores podemos encontrar el uso de soluciones locales o *cloud*. En las soluciones locales se presenta Docker *swarm*, entorno nativo que engloba funcionalidades de agrupamiento de contenedores y administración del sistema a través del propio motor de Docker. Además se encuentra Kubernetes, un sistema de orquestación para contenedores Docker (con soporte para otras tecnologías de contenedores y máquinas virtuales) que permite planificar y administrar las cargas de trabajo en función de los parámetros definidos por el usuario. Maraton, un orquestador de contenedores para Apache Mesos con características que permiten ejecutar aplicaciones en un entorno de clúster. En cuanto a soluciones en la nube se encuentra Cloudify, un orquestador que permite el modelado de aplicaciones y servicios y la automatización de su ciclo de vida [23].

2.2. Elasticidad/(re)asignación de recursos en contenedores

Los contenedores presentan grandes ventajas para el desarrollo y ejecución de aplicaciones de HPC en las que podemos incluir a los algoritmos de ML. Permiten adaptar fácilmente el despliegue de este tipo de aplicaciones a través de la elasticidad horizontal y vertical. La elasticidad horizontal permite aumentar y disminuir el número de contenedores asignados a cada aplicación. Por otro lado, la elasticidad vertical permite aumentar y disminuir la cantidad de recursos computacionales asignados a cada contenedor de la aplicación. La mayoría de las soluciones existentes consideran la elasticidad horizontal [26] o vertical [27] [28]. Al aprovechar al máximo la elasticidad, una aplicación puede reaccionar más rápidamente a pequeñas variaciones de carga de trabajo, a través de una elasticidad vertical, así como a picos repentinos de carga de trabajo, a través de una escala horizontal. Sin embargo, hasta ahora solo un número limitado de trabajos ha explorado los beneficios de combinar las dos dimensiones de elasticidad para aplicaciones basadas en contenedores [29].

Es de interés la administración de recursos de contenedores Docker debido a su uso masivo en el desarrollo y ejecución de aplicaciones de HPC. Una característica importante de esta herramienta es el aislamiento de recursos utilizando cgroups en el kernel de Linux el cual permite limitar, contabilizar y aislar el uso de recursos de los procesos y contenedores logrando inde-

pendencia entre ellos dentro de un mismo sistema operativo [30]. Cgroups no solo administra el uso de procesos, sino que también permite calcular métricas de CPU, memoria y del bloque de entrada/salida.

Docker permite seleccionar la cantidad de recursos (CPU, memoria, entre otros) asignados a cada contenedor. Al centrarse en el uso de CPU, de forma predeterminada cada contenedor tiene acceso a todos los núcleos de la arquitectura. La mayoría de las veces no controlar esta configuración lleva a una sobrecarga de trabajo que termina perjudicando el rendimiento de las aplicaciones y sistema [31]. Docker presenta un archivo donde se configura los limites llamados *suaves* y *duros*. Cuando se configura el límite *suave*, el contenedor puede usar todos los recursos de la máquina host. También, hay otros parámetros que se pueden controlar aquí como la proporción de CPU que puede utilizar el contenedor. Por otro lado, los límites *duros* definen la cantidad específica de recursos que puede utilizar el contenedor.

El acceso a la CPU se programa mediante el uso del Completely Fair Scheduler (CFS) o usando Real-Time Scheduler (RTS). En CFS, los ciclos de la CPU se dividen proporcionalmente entre los contenedores. Por otro lado, RTS proporciona una manera de configurar los límites *duros* a los contenedores. En [27] se propone utilizar RTS para controlar los contenedores de manera elástica acorde a la demanda de trabajo existente.

2.3. Elasticidad en cloud

La elasticidad en la nube es una característica deseable a tener en cuenta con gran importancia. Las técnicas de virtualización son la clave para lograr la escalabilidad en la nube, permitiendo explotar el uso de los recursos físicos (CPU, memoria, almacenamiento o red, entre otros). Generalmente, el lanzamiento de múltiples aplicaciones en diferentes contenedores no es un tarea sencilla ya que se deben analizar las cargas de trabajo para lograr una distribución equitativa de los recursos disponibles y evitar problemas como escasez u *oversuscription* de recursos.

En los servicios de cloud, los usuarios solo pagan los recursos utilizados, lo que comúnmente se conoce como esquema de *pay-as-you-go*. Los desarrolladores pueden ampliar o reducir los recursos del servidor de forma sencilla aprovechando las API proporcionadas por los frameworks. Se puede acceder a ellas en cuestión de horas o incluso minutos.

Cuando ocurren muchas peticiones en un corto plazo de tiempo, el sistema administrador del cloud necesita ajustar los recursos del servidor para evitar ineficiencia. Manejar de forma manual este problema es dificultoso

y, por lo tanto existen mecanismos automáticos de asignación de recursos llamados *autonomous elastic cloud* que permiten asignar dinámicamente los recursos teniendo en cuenta el número de solicitudes. Cuando el número de peticiones aumenta, este software es capaz de adicionar más recursos para las aplicaciones [32].

En un principio, se utilizaban maquinas virtuales para el lanzamiento de aplicaciones en cloud lo cual era muy costoso en rendimiento como principal desventaja. Además, no es necesario el despliegue de un sistema operativo ya que alcanza con tener un servidor web (apache, nginx) y/o los lenguajes de programación y librerías necesarias por cada aplicación. Otra desventaja es el costo de cambiar los recursos de la aplicación debido a que no es un proceso directo y debe realizarse desde el sistema operativo. Por lo tanto, desplegar aplicaciones cloud en máquinas virtuales produce degradación del sistema y mayor tiempo de desarrollo.

2.4. Elasticidad en aplicaciones

A nivel de aplicación, se entiende por elasticidad o maleabilidad a la capacidad intrínseca de las mismas para modificar su comportamiento (y en muchos casos, el uso de recursos que éstas hacen) durante su ejecución, bajo demanda de agentes externos o bien de forma autónoma.

No todas las aplicaciones, bibliotecas o infraestructuras en tiempo de ejecución (*runtimes*) soportan este tipo de característica. De hecho, una gran parte de aplicaciones permiten configurar las características de ejecución *a priori*, fijando las mismas y manteniéndolas durante toda la vida del programa. En aplicaciones que conllevan un tiempo de ejecución considerablemente largo (por ejemplo, procesos de entrenamiento de redes neuronales), esta falta de elasticidad puede suponer un doble problema:

1. Una decisión incorrecta en tiempo de arranque del programa puede conllevar una ejecución subóptima, sin posibilidad de solucionarla durante la vida del mismo.

2. La situación en cuanto a recursos disponibles en el sistema puede variar durante la vida del programa en ejecución. Así, es posible que durante la misma, se liberen recursos que podrían ser utilizados para mejorar la ejecución del programa, o se reserven recursos para otras aplicaciones, de modo que se generen situaciones de *oversuscription*.

Las características de elasticidad/maleabilidad suelen exponerse al exterior por medio de un conjunto de sintonizadores (o *knobs*), que son suscep-

tibles de ser consultados y/o modificados externamente, de forma asíncrona. Estos *knobs* pueden ser *específicos* de la aplicación (por ejemplo, en el caso del entrenamiento de redes neuronales, podrían variarse dinámicamente aspectos como el tamaño de *batch* mientras la red está entrenando), o bien *genéricos* (por ejemplo, modificando dinámicamente el número de hilos mientras una sección paralela está siendo ejecutada). En cualquier caso, la modificación del valor puede conllevar una variación tanto en las métricas internas de la aplicación (por ejemplo, velocidad de convergencia en el proceso de entrenamiento) como en el uso efectivo de recursos por parte de la aplicación.

Capítulo 3

Integración de elasticidad en Tensorflow

3.1. El framework Tensorflow

TF es uno de los *frameworks* de código abierto más utilizado por los desarrolladores en el ámbito de ML. Está compuesto por un conjunto de herramientas, librerías y recursos implementados por la comunidad que lo utiliza permitiendo implementar y ejecutar nuevos algoritmos de ML de forma fácil y con un alto rendimiento computacional. TF permite crear modelos de ML utilizando suites de alto nivel como Keras, con prototipado rápido, reduciendo el tiempo de desarrollo, seguimiento y análisis del modelo en tiempo de ejecución y una depuración sencilla.

También ofrece el entrenamiento y la implementación de modelos en servidores/dispositivos en el borde (*edge computing*) a través de versiones ligeras del entorno o en la web, sin importar el lenguaje o la arquitectura de hardware utilizada. Para dispositivos móviles cuenta con una versión liviana llamada *TF Lite*.

La arquitectura interna de TF esta compuesta de tres partes fundamentales que serán comentadas en las siguientes subsecciones.

3.1.1. Grafo computacional

En TF, los algoritmos de ML están representados por grafos computacionales, compuesto principalmente de 4 elementos:

- Nodos. representan las operaciones, típicamente matemáticas, las cuales se alimentan de datos de entrada para su procesamiento generando

datos de salida.

- Tensores. Es un conjunto de valores de un mismo tipo (como por ejemplo de valores enteros o flotantes). El número de dimensiones de un tensor se denomina *rank* y la cantidad de elementos de la dimensión se llama *shape*. Por lo tanto, podemos decir que los tensores se representan por matrices multidimensionales. Es importante entender que un tensor solo realiza el intercambio de datos entre operaciones y no realiza almacenamiento en memoria. Existen tensores que almacenan valores constantes, los cuales no se permiten modificar durante la ejecución del grafo.

- Vértices. Representan los datos que fluyen entre los nodos a través de los tensores.

- Variables. En diferentes algoritmos de ML, el modelo generado es ejecutado múltiples veces. Generalmente, es necesario mantener el estado de la evaluación anterior del modelo, tales como los pesos y otros parámetros de la red neuronal. Debido a que la mayoría de los tensores se destruyen entre las distintas ejecuciones del modelo, es necesario la utilización de variables.

Para comprender el grafo computacional de TF, se describe un ejemplo sencillo. En la figura 3.1 se visualiza la representación de la función $f(x,y) = x^2y + y + 2$. La operación x^2 es definida por el *nodo 1* que recibe como entradas la variable x duplicada. El *nodo 2* realiza la multiplicación entre la salida del *nodo 1* y la variable y. El *nodo 3* suma la variable y y la constante con valor 2. El *nodo 4* realiza la sumatoria de las salidas del *nodo 2* y *nodo 3* lo que genera el resultado final de la función.

La ejecución de los nodos/operaciones y el flujo de datos a través de los tensores se puede realizar solamente utilizando *sesiones*. La principal tarea es la asignación de recursos y variables. Además, permite definir la ejecución de un grafo o subgrafos que componen el modelo. Tras el comienzo de la ejecución, se realiza el análisis de las operaciones del grafo buscando dependencias y definiendo cuáles se ejecutarán en cada dispositivo permitiendo elegir la asignación si el usuario lo desea. Por ejemplo, si se quiere ejecutar una operación que presenta un rendimiento mayor en GPU, podemos indicarlo en la sesión. Para realizar esta asignación, TF cuenta con un algoritmo de alojamiento (*allocation*), que permite mantener el control de las dependencias.

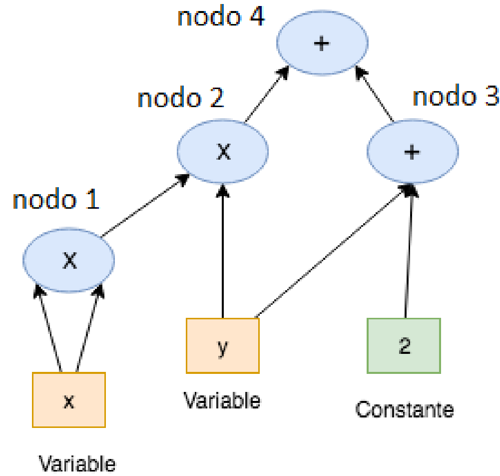


Figura 3.1: Grafo computacional en TF.

3.1.2. Modelo de ejecución

Para la ejecución del grafo computacional explicado anteriormente, TF esta compuesto de cuatro actores: el cliente o *client* , el maestro o *master*, los trabajadores o *workers* y los dispositivos o *devices*. El *client* es el encargado de crear una sesión con un determinado grafo. Luego, este grafo es enviado para su ejecución al *master* encargado de tres tareas principales:

1. Realiza una poda del grafo para obtener el subgrafo que permita evaluar los nodos solicitados por el cliente.

2. Divide el grafo en subgrafos para cada dispositivo participante.

3. Almacena en caché estos subgrafos para que puedan utilizarse en pasos posteriores.

Los *workers* se encargan de las siguientes tareas:

1. Maneja las solicitudes del *master*.

2. Programa la ejecución de los núcleos para las operaciones que componen un subgrafo local.

3. Supervisa uno o más *devices*.

Los *devices* se encargan del procesamiento de las operaciones, ya que para ellos se encuentran implementados los *kernels*. Son las unidades mas pequeñas donde eventualmente deben ser planificadas las operaciones del grafo de ejecución. Entre los *devices* utilizados se encuentran la CPU y la GPU. Recientemente, de la mano del avance del hardware y la aparición de placas aceleradoras específicas para algoritmos de ML, TF acepta el uso de las TPU desarrolladas por Google denominadas **Coral**, entre otros dispositivos.

Cada *device* cuenta con una unidad denominada *executor* donde se definen las colas de planificación de nodos/operaciones para cada *thread* dentro de un *threadpool*. Por lo tanto, varios *threads* pueden estar programando las tareas de la cola de nodos listos dentro del *executor*.

Cada *thread* es el encargado de planificar y ejecutar las operaciones asociadas a los nodos. El *thread* analiza si puede ejecutar cada nodo comprobando varios criterios de decisión. Obtiene los nodos a planificar desde una cola propia denominada $\mathrm{Q}_{\mathrm{ready}}$. Si se cumplen los criterios para dicho nodo, es almacenado en otra cola denominada $\mathrm{Q}_{\mathrm{inline}}$, que contiene los nodos listos para ser ejecutados por el *thread* . De lo contrario, si uno de los criterios de decisión no se cumple, se delega el nodo para que se ejecute en otro *thread*. La planificación de los nodos listos se describe en los siguientes pasos (ver Fig. 3.2):

Paso 1: Comprueba si $\mathrm{Q}_{\mathrm{ready}}$ está vacía. Si no esta vacía, continua al siguiente paso. De lo contrario, finaliza la planificación.

Paso 2: Se obtiene el próximo nodo $\mathrm{N}_{\mathrm{next}}$, que contiene la operación correspondiente al grafo, de la cola $\mathrm{Q}_{\mathrm{ready}}$.

Paso 3: Si $\mathrm{N}_{\mathrm{next}}$ es un nodo no costoso, se encola en $\mathrm{Q}_{\mathrm{inline}}$ para indicar que está listo para su ejecución. Se entiende por nodo costoso a aquel que TF etiqueta como *expensive* debido a un mayor tiempo para resolverse por su alto cómputo intensivo.

Paso 4: Si el nodo es costoso y el *thread* actual tiene un nodo de este tipo planificado para su ejecución, se asigna el nodo a otro *thread*. Se vuelve a la evaluación del paso 1.

El procedimiento de ejecución de los nodos se describe en los siguiente pasos (ver figura 3.3):

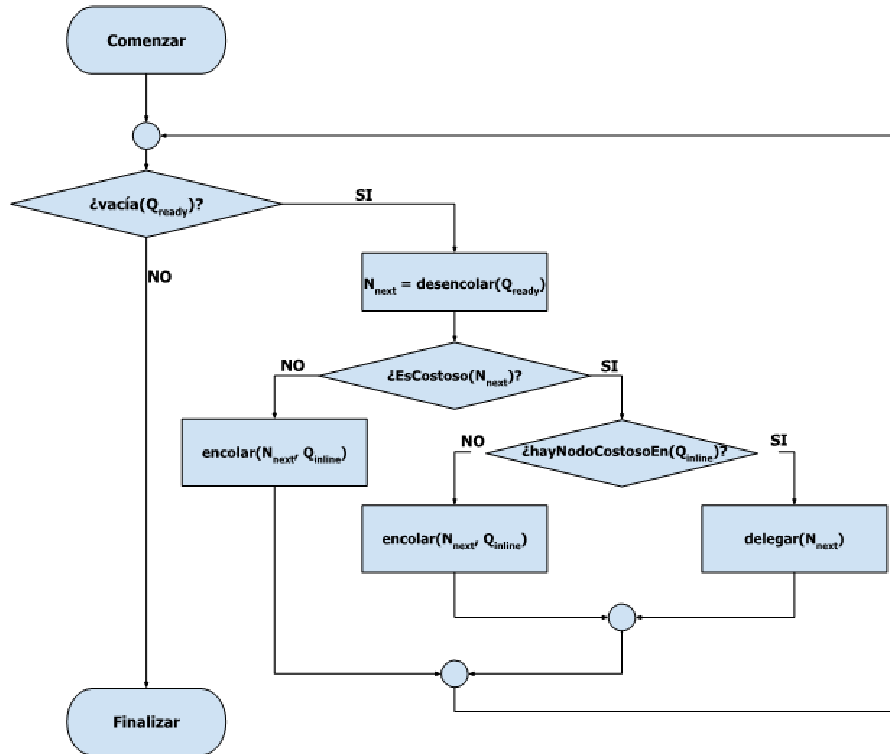


Figura 3.2: Planificación de nodos listos para ejecutar.

Paso 1: Se evalúa si la cola Q_{inline} contiene nodos para ejecutar. En el caso que esté vacía, finaliza el proceso de ejecución de nuevos nodos. De lo contrario. Si hay nodos en la cola, se obtiene el más antiguo.

Paso 2: Se verifica que las dependencias y las condiciones de ejecución se cumplan para el nodo.

Paso 3: Se ejecuta el nodo utilizando la implementación de kernel para la arquitectura de hardware del sistema (CPU, GPU o acelerador de propósito específico).

Paso 4: Se decrementan las dependencias de los nodos que tienen relación con la ejecución del nodo actual.

Paso 5: Se chequea si hay nuevos nodos disponibles para planificar. En caso

afirmativo, se realiza dicho procedimiento utilizando los pasos mencionados en la planificación de nodos listos.

Paso 6: Se vuelve a realizar el paso 1.

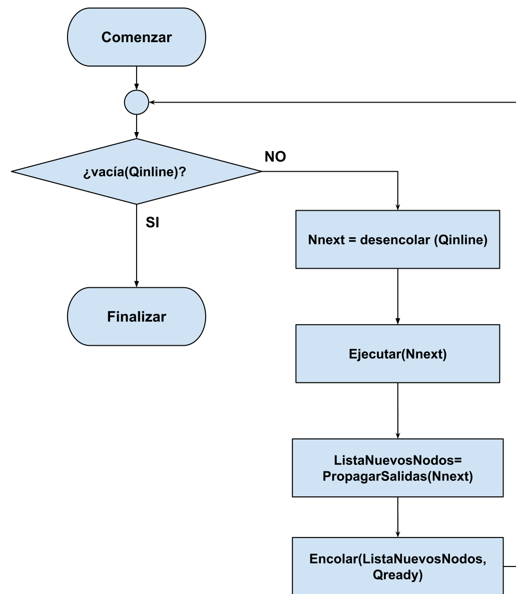


Figura 3.3: Proceso de ejecución de nodos planificados.

Por otra parte, TF permite explotar dos tipos de paralelismo llamados *Intra Paralelismo* e *Inter Paralelismo*. Es responsabilidad del usuario su configuración (aunque puede detectarse una configuración óptima de forma automática, de forma individual o combinada).

- **Intra paralelismo**. Permite el control del número de subprocesos/threads que se utilizarán para la ejecución de una tarea asociada a un nodo. Para lograr esto, las implementaciones de las tareas (*kernels*) deben admitir este tipo de paralelización.
- **Inter paralelismo**. Permite el control de la cantidad de operaciones de kernel independientes que se pueden ejecutar simultáneamente.

TF delega el manejo de estos tipos de paralelismo a la implementación desarrollada en la librería Eigen [33] en dispositivos CPU, aprovechando su flexibilidad y eficiencia.

3.1.3. Optimizaciones

Cuando se desea obtener el máximo rendimiento computacional, es necesario aplicar optimizaciones al algoritmo. TF tiene la ventaja de contar con librerías específicas que se encargan de dicha tarea facilitando el uso a los usuarios.

Principalmente, cuenta con tres optimizaciones: poda del grafo de ejecución, planificación de operaciones y compresión con pérdidas.

- **Poda del grafo de ejecución**. Es una optimización realizada por muchos compiladores donde se busca en el grafo operaciones que sean iguales para evitar que se calcule múltiples veces. Para esto se utilizan variables temporales que almacenan el valor de la operación y permiten la reutilización. En los grafos de TF es común que haya operaciones iguales, por lo que si no se realiza esta optimización tendremos un costo considerable en rendimiento. Asimismo, permite el uso de menos memoria ya que no debemos almacenar múltiples resultados del cálculo de la operación. Por ejemplo si hay dos subgrafos que calculan la misma operación y tienen el mismo tensor de salida, se realiza la reducción de dos subgrafos a uno. Este nuevo subgrafo contiene dos tensores de salida (que contienen el mismo resultado). En la Fig. 3.4 se observa un ejemplo de poda de grafo.

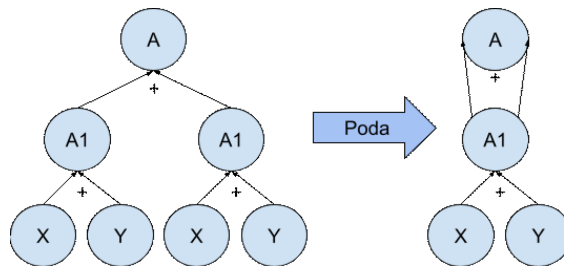


Figura 3.4: Poda del grafo computacional.

- **Planificación de operaciones**. La planificación de las operaciones de TF puede resultar en un mejor rendimiento del sistema, en particu-

lar con respecto a las transferencias de datos y el uso de la memoria. Realizar una optimización en las operaciones puede reducir la ventana de tiempo durante la cual los resultados intermedios deben mantenerse en la memoria entre operaciones y, por lo tanto, el consumo máximo de memoria. Esta reducción es principalmente importante para los dispositivos con GPU donde la memoria es escasa. Además, se debe intentar reducir la comunicación de datos entre dispositivos para evitar la competencia por los recursos de la red. Una optimización importante es la planificación de los nodos para que comiencen en el momento que sus dependencias se cumplen y evitar inicios tempranos. Para estimar el momento de inicio se analizan las rutas críticas de los nodos del grafo. También, se utilizan librerías preexistentes que optimizan multiplicaciones de matrices en diferentes dispositivos permitiendo mejorar el cómputo de las operaciones, incluidos BLAS [34], cuBLAS [35] o librerías de redes neuronales en GPU como cuda-convnet [36] y cuDNN [37].

- **Compresión con pérdidas (Lossy Compression)**. Algunos algoritmos de ML, incluidos los que se utilizan normalmente para entrenar redes neuronales, toleran el ruido y la aritmética de precisión reducida. A menudo se usa compresión con pérdida cuando hay envíos de datos entre dispositivos (a veces dentro de la misma máquina pero especialmente a través de distintas máquinas). Por ejemplo, generalmente se insertan nodos de conversión especiales que convierten representaciones de punto flotante de 32 bits en una representación de punto flotante de 16 bits (no el estándar de punto flotante IEEE de 16 bits propuesto, sino solo un formato flotante IEEE 794 de 32 bits, pero con 16 bits menos de precisión en la mantisa), y luego volver a convertir a una representación de 32 bits en el otro lado del canal de comunicación (simplemente llenando con ceros para la parte perdida de la mantisa, ya que eso es menos costoso computacionalmente que hacer matemáticamente el redondeo probabilístico en la conversión de $32 \rightarrow 16 \rightarrow 32$ bits) [7].

3.1.4. Visualización de trazas de ejecución

Los modelos de ML generalmente utilizan estructuras complejas de redes neuronales, como por ejemplo Resnet 50 donde el número (50) representa la cantidad de capas de profundidad. Para tener una visión precisa sobre este tipo de redes, facilitar la depuración de modelos y la inspección de valores

intermedios con alto nivel de detalle, se requiere una herramienta sofisticada. Tensorboard es una interfaz web que permite la visualización y manipulación de los grafos de los modelos de ML. Permite que el usuario entienda como fluyen los datos a través del grafo. Además, divide el modelo en subgrafos los cuales pueden verse como bloques. En la Fig. 3.5 se visualiza el modelo Resnet en la herramienta.

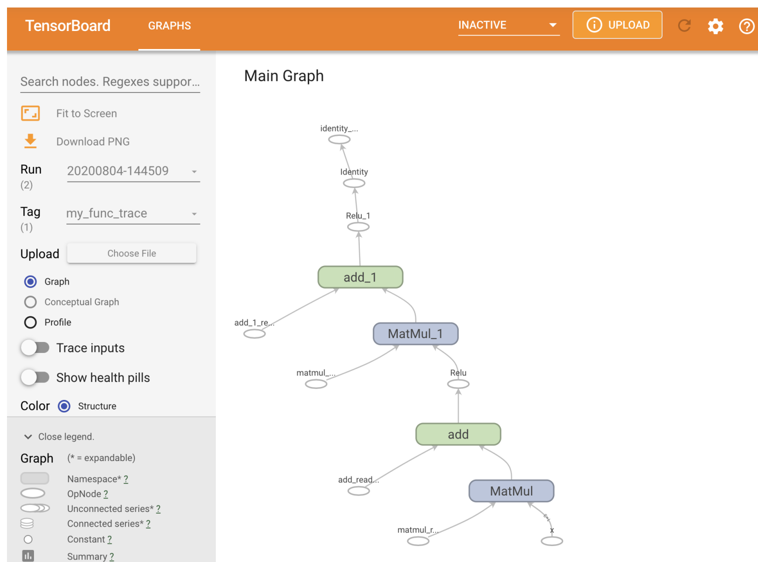


Figura 3.5: Grafo computacional en Tensorboard.

Una función potente que presenta esta herramienta es la visualización de valores de tensores a lo largo de la ejecución del grafo. Para cada tensor se pueden visualizar dos tipos de resúmenes: escalar e histogramas. Los resúmenes escalares muestran la progresión de un valor de tensor escalar, que se puede muestrear en ciertos recuentos de iteraciones. De esta forma, se podría, por ejemplo, observar la precisión o pérdida del modelo con el tiempo. Los nodos de resumen de histograma permiten al usuario realizar un seguimiento de las distribuciones de valores, como las de los pesos de las redes neuronales o las estimaciones softmax finales.

En la sección "Profile" se encuentra una descripción general del rendimiento del modelo, especialmente los diferentes tiempos para el cómputo, entrada/salida de datos, comunicación entre dispositivos, compilación, entre otros. También presenta un visor de seguimiento donde se muestran los diferentes eventos que ocurrieron en los dispositivos (CPU/GPU/TPU) durante

el periodo de creación de perfiles permitiendo entender dónde se producen los cuellos de botella (Fig. 3.6).

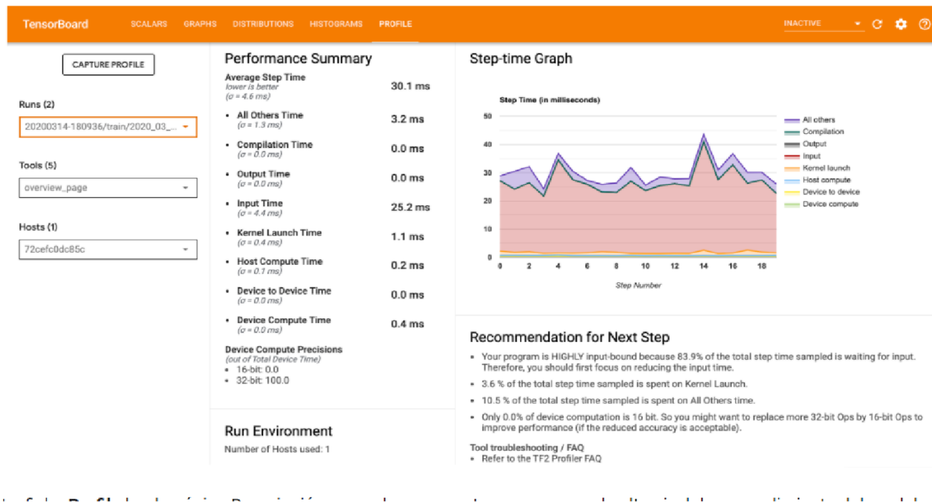


Figura 3.6: Ventana Profile de Tensorboard.

Por último, una función muy interesante para ver el perfil de rendimiento es *Trace Viewer* la cual permite visualizar los tiempos y momentos de cada operación ejecutada en los dispositivos. Por ejemplo, en la Fig. 3.7 se observa que la etapa seleccionada (de 22 ms) se está ejecutando en la CPU la operación *tf-data-iterator-get-next* responsable de procesar los datos de entrada y enviarlos a la GPU. En ese tiempo, la misma se encuentra inactiva. Encontrar estos puntos que degradan el rendimiento sirven para mejorar el modelo de ML intentando mantener siempre activos los dispositivos.

3.2. Diseño e integración de elasticidad en Tensorflow

3.2.1. Diseño de la solución

El usuario puede elegir ambos grados de paralelismo provistos por TF; esta elección, sin embargo, es puramente estática, ya que la configuración se realiza en el código del algoritmo o a través de variables de entorno, y en cualquier caso, previamente a su ejecución y de forma constante durante la misma.

Figura 3.7: Función *Trace Viewer* en Tensorboard.

El modelo de ejecución explicado en la sección anterior nos permite entender cómo funciona la distribución de los nodos de un grafo entre los dispositivos participantes y a su vez el grado de paralelismo de cada operación. Si analizamos los dos tipos de paralelismo que existen en el framework, la distribución de nodos entre *threads* se encuentra relacionada con el *INTER* y la ejecución de cada nodo con el *INTRA*. Ambos tipos son manejados por la librería Eigen. También hemos visto en el código fuente que se encuentran directivas del estándar OpenMP [38] pero no son utilizadas a la hora de la ejecución de los modelos ya que requiere de la instalación de alguna implementación de este estándar y su posterior configuración.

En primer lugar, se necesita que el usuario pueda comunicarse con el framework para indicar en cualquier momento de la ejecución el nuevo paralelismo que desea. Una forma sencilla es utilizando señales, es decir comunicando un evento de un proceso a otro. El usuario modifica un archivo de texto donde se indican los nuevos valores de paralelismo y se envía la señal. El sistema operativo, en este caso Unix, es el encargado de que el proceso correspondiente a TF reciba la señal. Este proceso se encargará de atender la petición y cambiar el grado de ambos paralelismos comunicándose con la librería Eigen.

Hasta el momento de la realización de este trabajo, la comunicación con Eigen para el cambio de paralelismo se debe desarrollar, ya que esta función no se encuentra disponible. Cuando esta librería recibe la notificación realiza la acción de despertar o dormir los *threads* creados en el dispositivo para el tipo de paralelismo indicado ajustando a lo solicitado por el usuario.

3.2.2. Implementación

Cada *thread* perteneciente a un *device* se encarga de planificar los nodos listos de un grafo computacional. Cada uno de los *threads* en caso de recibir una señal del sistema le indicará al controlador del *threadpool* ubicado en la librería Eigen que se debe realizar un aumento o disminución del paralelismo. El controlador le asignará un estado a cada *thread* que define si debe seguir ejecutando operaciones o no. Para incorporar la maleabilidad dentro de TF, se requiere cambios en la en la librería Eigen y en el núcleo de TF.

Modificaciones en el threadpool de Eigen

La biblioteca Eigen responsable de administrar el grupo de *threads* de un determinado dispositivo, no permite una control dinámico del número de *threads* activos en cualquier momento arbitrario. Por lo tanto, se requiere agregar información adicional de estado por *threads* para activar/desactivar su comportamiento normal.

La nueva versión del controlador del *threadpool* permite recibir cambios de paralelismo. Cuando se recibe un cambio, puede suceder dos casos:

- **Aumento de paralelismo**. En este caso —. En caso que se pueda cumplir, se cambian a activo y se despierta la cantidad de hilos necesarios para satisfacer los requerimientos del usuario.

- **Disminución de paralelismo**. En primer lugar se verifica si la disminución deja activo como mínimo a un *thread*. En caso de que se cumpla esta condición, de forma aleatoria se cambia el estado de los hilos necesarios para cumplir con los solicitados. Si las colas de los *threads* marcados como inactivos quedaron con tareas pendientes, eventualmente serán atendidas por los que quedan activos.

Cuando un *thread* comienza la espera de trabajo (en este caso nodos de cómputo) evalúa si debe mantenerse activo o no dependiendo de su estado actual. En el caso que su estado es activo, analiza si es posible ejecutar nuevos nodos que se encuentran en la cola propia de nodos o en la cola de

otro *thread*. En el caso de que las colas estén vacías (propia o perteneciente a otro) se duerme esperando a que le envíen la señal de que se planificaron nuevos nodos. En el caso que su estado sea inactivo, antes de dormirse verifica si hay trabajo pendiente para despertar a un *thread* activo.

Estos cambios se realizaron sobre el código fuente de la biblioteca Eigen, específicamente en el archivo `NonBlockingThreadPool.cc`.

Modificaciones en el núcleo de Tensorflow

Hasta el momento, el estado de un *thread* no se puede controlar desde el núcleo de TF. La única comunicación posible permite delegar la ejecución de nodos a otro *thread*. Para agregar este control adicional, se debe modificar el proceso de planificación de nodos listos agregando la consulta del estado del *thread* antes de enviar un nodo de la cola de nodos listos Q_{ready} a la cola Q_{inline}. En la Fig. 3.8 se muestra el nuevo proceso de planificación. Cuando se obtiene un nodo de la cola Q_{ready} se verifica si es costoso en cuanto a tiempo de ejecución y el estado del *thread*. Si debe ejecutar y no es costoso en cómputo, se planifica igual que en el mecanismo anterior. Caso contrario, pueden suceder dos casos:

- El *thread* deba seguir ejecutando. En este caso, al igual que en el proceso de planificación original, debe analizar el costo de cómputo del nodo para decidir si lo ejecuta o lo delega a otro *thread*.
- El *thread* debe dormirse y, por lo tanto, delegar el nodo.

Esta modificación se realizó en el código fuente del archivo `executor.cc`, encargado de realizar la asignación de nodos de un grafo computacional entre los dispositivos.

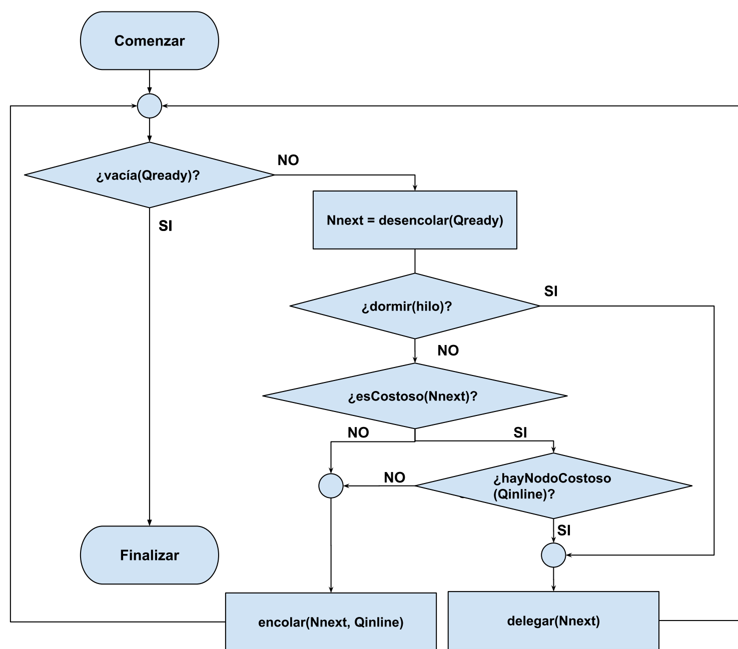


Figura 3.8: Nueva planificación de nodos listos.

Capítulo 4

Coplanificación de aplicaciones elásticas

El objetivo final de cualquier política de asignación elástica de recursos es hacer coincidir en todo momento la capacidad del sistema con la demanda de los usuarios: los recursos pueden ampliarse cuando otras tareas dejaron de utilizarlos y reducirse con precisión para asignarlos a nuevas tareas. Aunque las técnicas de virtualización existentes, como las máquinas virtuales (VM) y los contenedores, admiten la reconfiguración de recursos sobre la marcha, explotar realmente la elasticidad de los recursos sigue siendo un desafío.

La tecnología basada en contenedores permite variar la cantidad de recursos asignados a un contenedor en cualquier momento de la ejecución (a día de hoy, la reasignación de recursos en tecnologías como Docker se limita al número de núcleos y cantidad de memoria asignada por contenedor). Para que este cambio se vea reflejado en las aplicaciones internas, se necesita que las mismas puedan variar sus recursos en tiempo de ejecución. En caso contrario, ocurrirán fenómenos no deseados y contraproducentes de cara a un correcto aprovechamiento de los recursos disponibles, tales como:

Desperdicio de recursos, en caso de aumentar la cantidad de recursos (núcleos) asignados a una aplicación por encima de los inicialmente configurados y por tanto explotados por la misma.

***Oversuscription*,** al reducir la cantidad de recursos por debajo de los inicialmente configurados para la aplicación, llevando a situaciones de contención y por tanto degradando el rendimiento.

Para reducir el impacto de estos fenómenos, es necesario que no solo la tecnología de virtualización/contenedores soporte la reasignación dinámica

de recursos (característica ya disponible), sino que las aplicaciones que en ellas se ejecutan contengan:

1. Variación dinámica de los recursos computacionales. En el caso del número de núcleos, por ejemplo, variando el grado de paralelismo que en un instante determinado pueden llegar a explotar.
2. Capacidad de comunicación con un agente externo tanto para recibir notificaciones asíncronas de solicitud de reasignación dinámica de recursos, como para notificar cualquier cambio en el uso de recursos por parte de la propia aplicación.

En este trabajo, se exploran ambas características integrando el uso de Tensorflow como aplicación confinada en contenedores dinámicamente configurables. Específicamente, se utilizará la versión elástica de TF presentada en el Capítulo 3 como prueba de concepto. Utilizar una versión elástica del framework TF permite contar con una aplicación que responde ante cambios en los recursos computacionales. Si es ejecutada dentro de contenedores, permite aprovechar la posibilidad de modificar los recursos computacionales del contenedor en tiempo de ejecución, logrando beneficios en su rendimiento.

Por otra parte, los planificadores de contenedores actuales no tienen en cuenta la elasticidad de los contenedores y aplicaciones. Cuando el usuario o el sistema solicita un cambio en la cantidad de recursos computacionales, el planificador realiza el proceso de reasignación indicando al contenedor los nuevos recursos. Este nuevo cambio no es informado a las aplicaciones que conviven dentro del contenedor, generando pérdida de beneficios o degradamiento del rendimiento del sistema si no hay un planificador que orqueste dicho cambio.

Por estos motivos, se implementa un planificador de aplicaciones elásticas orientado a algoritmos de ML ejecutados en el framework TF.

4.1. Diseño de la solución

El diseño propuesto parte de la base de un sistema para gestionar solicitudes simultáneas de múltiples usuarios para la ejecución de aplicaciones con una cantidad de recursos computacionales específica. Además, los usuarios ejecutarán dichas aplicaciones confinadas en contenedores.

En primer lugar, se debe analizar qué tipo de peticiones se recibirán por parte del usuario. Para facilitar esta tarea, es conveniente que el propio usuario sea el encargado de generar una imagen de un contenedor utilizando

un esqueleto de imagen donde solo deba agregar la ruta o directorio donde se encuentra su ejecutable.

Así, un **planificador** será el encargado de atender las peticiones de los usuarios que solicitan la ejecución de un algoritmo (modelo) de ML sobre TF. Esta petición será evaluada teniendo en cuenta las políticas de planificación y los recursos disponibles en el sistema, en función de los recursos solicitados. Si la evaluación es satisfactoria, se generará un contenedor Docker que internamente cuenta con una versión de TF elástica (véase Capítulo 3), capaz de ejecutar el algoritmo desarrollado por el usuario.

A su vez, los contenedores contarán con un **cliente** capaz de comunicarse con el planificador para manejar diferentes eventos que ocurran durante su vida. Este cliente se añade automáticamente al contenedor y se ejecuta de forma autónoma en su arranque, siendo transparente al usuario. Principalmente, es importante que el cliente sea capaz de informar cuando finaliza la ejecución del algoritmo de ML. Además, debe permitir la recepción de cambios en el uso de recursos (nivel de paralelismo) indicados por el planificador. Por último, debe contar con algún mecanismo de control de vida, permitiendo tomar decisiones si el algoritmo no finalizó correctamente debido a algún error en cualquier punto de su ejecución. Cuando este evento ocurra debe ser informado al planificador para tomar a una decisión sobre este problema. La Fig. 4.1 muestra el diseño general del planificador de contenedores de ML propuesto.

4.1.1. Funcionamiento del planificador

Una vez definidos los principales actores en el diseño, es necesaria la definición de un proceso de planificación de los contenedores. Descrito a alto nivel, el planificador implementa de forma cíclica los siguientes pasos:

1. En primer lugar, el planificador evalúa si existen peticiones pendientes para la liberación de recursos. El cliente en cada contenedor informa a través de una petición cuando finaliza su ejecución y el planificador la atenderá antes de realizar el lanzamiento de nuevos contenedores. En la Fig. 4.2 se visualizan los pasos a seguir para esta tarea de forma esquemática.

2. El segundo paso del planificador consiste en atender las peticiones de ejecución pendientes que ya fueron analizadas con anterioridad y no pudieron planificarse. Esto ocurre cuando los recursos disponibles no cumplen los límites mínimos impuestos por la petición. En la Fig. 4.3 se observa el proceso de atención. Lo primero que se analiza es si hay

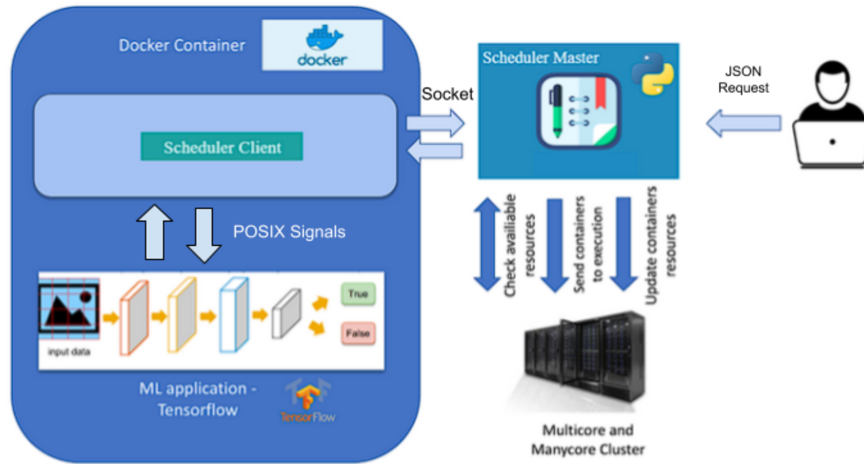


Figura 4.1: Modelo del planificador de contenedores.

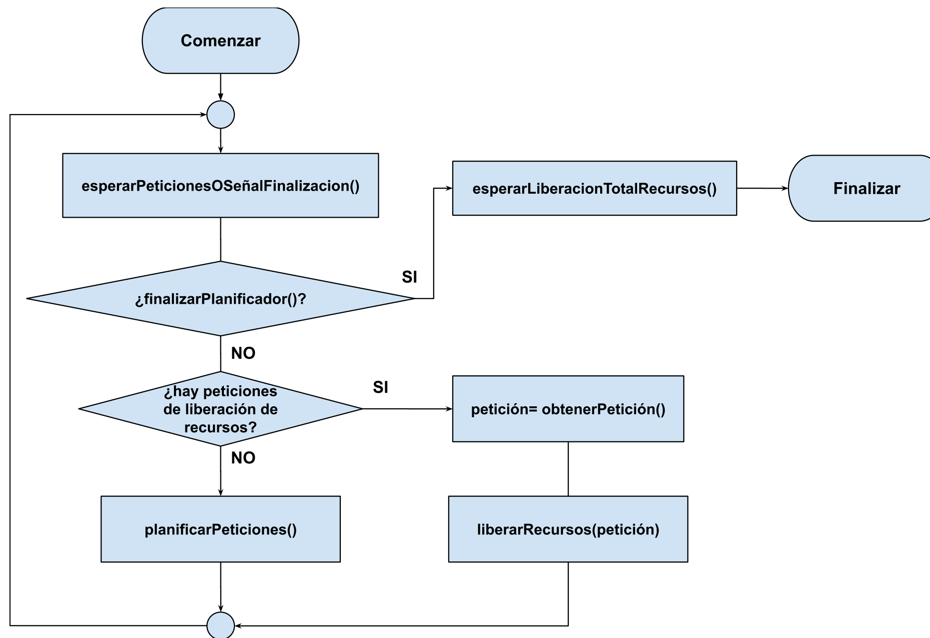


Figura 4.2: Esquema de planificación de contenedores.

recursos disponibles. En caso que no los haya, se procede al siguiente paso del planificador. En caso afirmativo, se obtiene una petición y se solicitan los recursos necesarios. Si esto se cumple se lanza el contenedor Docker con el algoritmo de ML; en caso contrario, la petición es colocada nuevamente en una estructura de peticiones pendientes.

En este punto, también pueden suceder errores de lanzamiento del contenedor, como por ejemplo, falla de la red de comunicación entre cliente y servidor, falla de algoritmo por error de alguna librería, entre otros. Ante estos errores, el cliente del contenedor informa el evento y se genera una nueva petición que es enviada a la estructura de peticiones pendientes, actualmente definida con un cola.

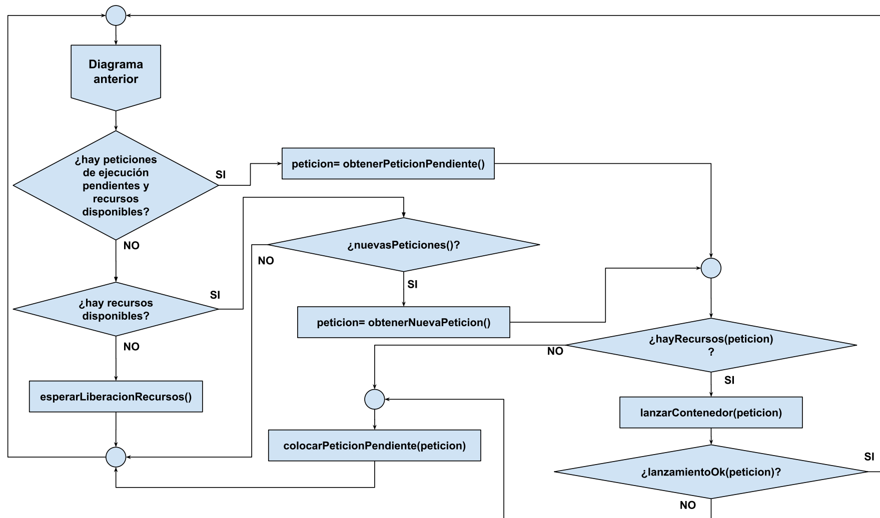


Figura 4.3: Modelo del planificador de contenedores.

3. El tercer paso contempla la planificación de nuevas peticiones. Es idéntico que el procesamiento de peticiones pendientes, donde solo cambia la estructura de la cual se obtienen las mismas.

4. Como último paso, luego de realizar la planificación de todas las peticiones, se analiza si es necesario una reasignación de recursos. Esto permitirá utilizar los recursos no asignados para distribuirlos entre los contenedores activos. Este procesamiento debe ser opcional y el usuario especificará al comienzo de la ejecución del planificador si la misma

está habilitada o no. Para este diseño, se dará prioridad a los contenedores activos más antiguos, en los cuales se aplicará un criterio definido por la política que se encuentre configurada en el planificador.

Por otra parte, pueden producirse fallos en el lanzamiento y/o ejecución de un contenedor. Estos fallos generalmente son producidos por errores de comunicación entre el cliente del contenedor y el servidor/planificador, o algún problema con el framework de ML mientras se ejecuta el algoritmo. Por lo tanto, se debe contar con un mecanismo sencillo para el control de errores. Para monitorizar estos eventos inesperados, el planificador enviará un mensaje al contenedor consultando sobre el estado de ejecución del mismo cada un tiempo determinado. El cliente del contenedor revisará cada intervalos fijos de tiempo si el algoritmo se encuentra ejecutando y almacenará ese estado. Cuando reciba una consulta del planificador le contestará con el estado actual el cual permitirá definir si el contenedor debe ser cancelado y replanificado en el futuro teniendo en cuenta las políticas de planificación.

Para el control de errores de comunicación, el planificador decidirá si el contenedor debe ser cancelado cuando envía una consulta de estado y no recibe una respuesta en un intervalo de tiempo definido.

4.2. Políticas de planificación y reasignación de recursos

Un punto importante a tener en cuenta en el planificador es la definición de las políticas que se desean aplicar para la planificación y asignación de recursos de los contenedores. El usuario será el encargado de definir cuáles utilizar en el momento de lanzar el planificador y durante toda su vida se mantendrán activas las mismas. Se debe contar con diferentes políticas que intenten cumplir con los requisitos solicitados por el usuario manteniendo un uso adecuado de los recursos computacionales presentes en el sistema.

Para esta solución se plantea una política de planificación y tres de asignación/reasignación de recursos, con la intención de añadir nuevas y mas sofisticadas en un futuro.

4.2.1. Política de planificación

First Come, First Served o FCFS

Está política utiliza una cola donde se almacenan de manera ordenada las peticiones de ejecución que van generando los usuarios del planificador.

En la Fig. 4.4 se visualiza el proceso de planificación utilizando esta política. Cuando se obtiene una petición de la cola se lanza el contenedor asociado a la misma y ante cualquier error en el lanzamiento se vuelve a colocar al final de la cola. Es importante destacar que cada petición no respeta ningún tipo de prioridad.

4.2.2. Políticas de asignación/reasignación de recursos

Strict

El contenedor asociado a la petición atendida se lanzará solo si el sistema cuenta estrictamente con la cantidad de recursos computacionales indicados por el usuario. Por ejemplo, si el usuario solicita la ejecución de un modelo de ML con 8 núcleos, el planificador verifica si esa cantidad se encuentra libre y solo en caso afirmativo realiza el lanzamiento del contenedor Docker. En caso de que el sistema no presenta los recursos solicitados, la petición será atendida posteriormente dependiendo de la política de planificación utilizada.

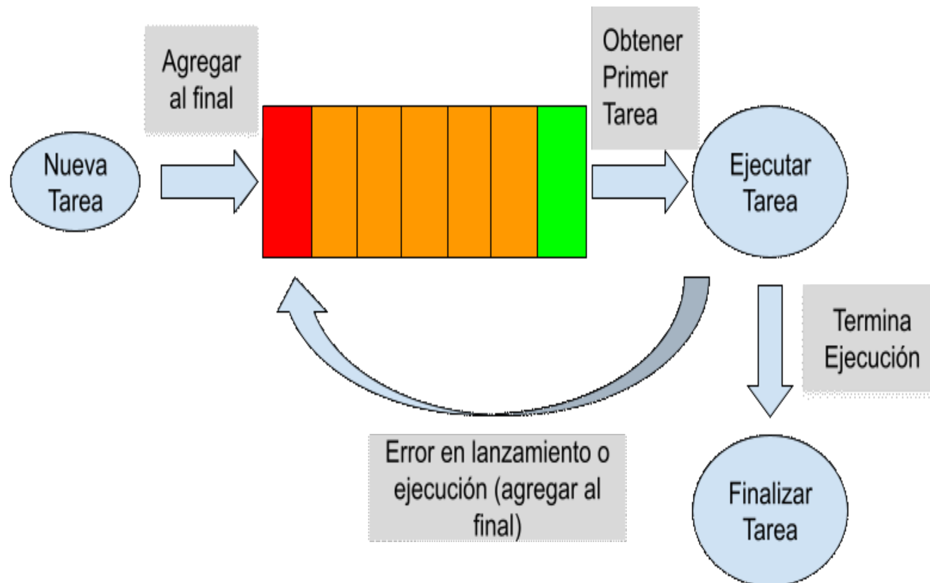


Figura 4.4: Proceso de planificación de la política FCFS

Always Attend

Esta política tiene como objetivo la planificación de la máxima cantidad de peticiones teniendo en cuenta los recursos disponibles en el sistema. En este caso puede suceder que la cantidad de recursos asignada a un contenedor sea menor que la cantidad solicitada por el usuario.

La principal diferencia con la política *strict* es que si el sistema no posee los recursos solicitados se intentará asignar los disponibles en ese momento. Por ejemplo, si el usuario solicita 8 núcleos en la petición pero el sistema solo cuenta con 4 disponibles, se le asignará esta última cantidad.

Max Prop o asignación máxima de recursos

Al igual que la política *Always Attend*, la cantidad de recursos asignados puede ser diferente a la especificada por el usuario. En este caso se intentará asignar una proporción de los recursos disponibles teniendo en cuenta todas las peticiones de lanzamiento pendientes. Esta proporción se calcula como la relación entre los recursos disponibles y la cantidad total solicitada por todas las peticiones. Por ejemplo, si en el sistema hay 8 núcleos libres y hay 2 peticiones pendientes, una que solicita 10 núcleos y otra que solicita 6, el factor de proporción es 0.5. La asignación de recursos sera 5 y 3 respectivamente.

Reasignación de recursos

Una función muy importante que debe tener el planificador es modificar los recursos computacionales de un contenedor en cualquier punto de su ejecución. Cuando se liberan recursos debido a la finalización de un contenedor, éstos podrían utilizarse para asignarlos a los otros contenedores que se encuentren activos. Esta función será habilitada por el usuario en el caso que desee usarla. Además se debe especificar si cuando hay recursos disponibles se dará prioridad a los contenedores activos o se utilizarán para nuevas peticiones. Esta decisión no afectará al proceso de reasignación, sino que solo especifica en qué momento se intentará reasignar recursos.

En la Fig. 4.5 se visualiza el proceso de reasignación de recursos. En caso de encontrarse habilitada la funcionalidad se analiza si se cuenta con recursos libres. Si esta condición se cumple, se tomarán contenedores activos mientras haya recursos libres. El orden de procesamiento de los contenedores activos viene establecido por una cola FIFO; por lo tanto, siempre se comenzará analizando los mas antiguos que se encuentran en ejecución. Para cada contenedor se obtiene la información acerca de los recursos utilizados y los

solicitados por el usuario en la petición inicial. En caso que la diferencia sea mayor a 0 se debe analizar si los recursos disponibles alcanzan para igualar a los solicitados por el usuario o no. Si esto sucede, el contenedor aumenta sus recursos al total que especificó el usuario en la petición inicial. Caso contrario, se le asignan todos los recursos disponibles. Este último caso finaliza el proceso de reasignación ya que no se disponen de más recursos libres.

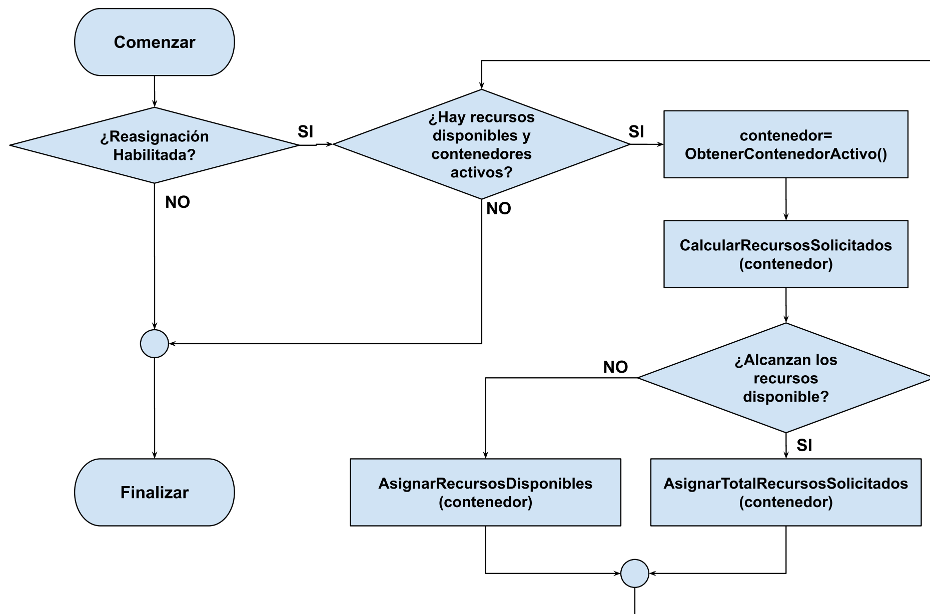


Figura 4.5: Reasignación de recursos.

La estrategia de reasignación para la política *max_prop*, en primer lugar, calcula el factor de proporción que consiste en la relación entre los recursos libres y la cantidad de contenedores activos. Si este factor es 0, de forma predeterminada se le asigna el valor 1. Mientras haya recursos para asignar, se recorren los contenedores incrementando el paralelismo indicado por el factor de proporción. Por ejemplo, si el contenedor actual utiliza 2 núcleos y el factor de proporción es 1, se incrementan los recursos del mismo a 3 núcleos.

4.3. Implementación de la solución

Esta sección describe el desarrollo de la solución, principalmente, de los diferentes módulos que intervienen teniendo en cuenta el diseño propuesto. Además, se comentarán las herramientas y bibliotecas utilizadas para los módulos.

Todas las descripciones suponen el uso de Python en su versión 3 como lenguaje de programación, y Docker como infraestructura de gestión de contenedores. La implementación de todo el proyecto se divide en tres módulos principales:

1. *Módulo del cliente*, donde se encuentra la lógica de ejecución de un contenedor utilizando Docker y la comunicación con el servidor del planificador.

2. Módulo del servidor del planificador, encargado del manejo de peticiones de ejecución y de la comunicación con los contenedores.

3. *Visor de trazas de ejecución*, que permite visualizar los tiempos de cada contenedor y obtener métricas de rendimiento de las aplicaciones. Cada una de estas implementaciones se explican en las siguientes secciones.

.

4.3.1. Módulo del cliente

El cliente cuenta principalmente de dos partes principales que se describen a continuación.

Creación del contenedor Docker

En primer lugar, la definición de una imagen de contenedor Docker que permite cumplir con todos lo requerimientos necesarios para su lanzamiento. Para esto se cuenta con la definición de un fichero de configuración de imagen denominado `Dockerfile` donde se especifican las herramientas y librerías utilizadas por el modelo de ML a utilizar y por el cliente del contenedor. Dentro de este archivo es importante describir los diferentes tipos de comandos que se puede ejecutar:

- **ARG**. Define los argumentos utilizados en la ejecución del fichero.

- **FROM**. Crea una capa a partir de una imagen base de sistema. Por ejemplo, permite especificar una versión de Ubuntu como base.
- **RUN**. Esta instrucción ejecuta cualquier comando en una capa nueva encima de la imagen base especificada y realiza un commit de los resultados. Esa nueva imagen intermedia es usada para el siguiente paso en el fichero.
- **ENV**. Configura las variables de ambiente. Estos valores estarán disponibles en los ambientes de los siguientes comando del fichero. Estos valores persisten en el lanzamiento de la imagen Docker.
- **ENTRYPOINT**. Define el punto inicial de la imagen final. Permite especificar un comando junto con los parámetros, que se ejecutará cuando se lanza un contenedor.

En el listado 1 se visualizan los comandos iniciales utilizados para la generación de la imagen. Los dos comandos iniciales permiten generar una imagen de Ubuntu 20.04. Posteriormente se almacena en el directorio de inicio los archivos fuentes del cliente. Luego, se instalan todas las librerías necesarias para el correcto funcionamiento de TF dentro del contenedor. Se completan las dependencias necesarias realizando la instalación de Python con el comando *RUN* como se visualiza en el listado 2.

A través del gestor de paquetes `pip3` se realiza la instalación de TF usando un paquete de instalación con extensión `.whl` que integra la versión elástica del framework descrita en el Capítulo 3. En los últimos dos comandos del fichero `Dockerfile`, el usuario debe especificar la carpeta donde se encuentra almacenado el algoritmo desarrollado con sus parámetros correspondientes. Para evitar la descarga de datos durante su ejecución, el comando `RUN python3 ...` realiza un *warmup* o calentamiento del algoritmo que tiene como principal funcionalidad la descarga de todos los *datasets* necesarios. A continuación, el comando `ENTRYPOINT python3 ...` realiza la ejecución del cliente del planificador cuando se lanza el contenedor (véase listado 3) .

Para la construcción de la imagen Docker utilizando el fichero `Dockerfile` se define un *script* llamado *create_docker_image.sh* encargado de ejecutar el comando *docker build* que recibe como parámetros el nombre definido para la imagen y el nombre del fichero `Dockerfile` (véase listado 4).

```
ARG UBUNTU_VERSION=20.04
FROM ubuntu:${UBUNTU_VERSION}
ADD . /scheduler_src
ARG DEBIAN_FRONTEND=noninteractive
RUN apt-get update && apt-get install -y \
    --no-install-recommends \
    build-essential \
    curl \
    git \
    libcurl3-dev \
    libfreetype6-dev \
    libhdf5-serial-dev \
    libzmq3-dev \
    pkg-config \
    rsync \
    software-properties-common \
  sudo \
        unzip \
        zip \
        zlib1g-dev \
        openjdk-8-jdk \
        openjdk-8-jre-headless \
        iproute2 \
        && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/*
```

Listing 1: Comandos iniciales para la creación de la imagen.

Creación del módulo de control

La segunda parte del cliente es el módulo encargado del lanzamiento y control del contenedor Docker. Este módulo se encuentra desarrollado en Python y cuenta con diferentes funciones donde cada una está asignada a un hilo concurrente.

El hilo principal, en primer lugar, espera la recepción de la información relativa al grado de paralelismo deseado y la versión de TF, ambas enviadas por el servidor del planificador. A continuación, se desarrolla el proceso de ejecución del algoritmo, compuesto por varios pasos:

```
RUN chmod a+w /etc/passwd /etc/group
ARG USE_PYTHON_3_NOT_2=1
ARG _PY_SUFFIX=${USE_PYTHON_3_NOT_2:+3}
ARG PYTHON=python3
ARG PIP=pip${_PY_SUFFIX}
# See http://bugs.python.org/issue19846
ENV LANG C.UTF-8
RUN apt-get update && apt-get install -y \
    ${PYTHON} \
    ${PYTHON}-pip
RUN ${PIP} --no-cache-dir install --upgrade \
    pip setuptools
# Some TF tools expect a "python" binary
RUN ln -s $(which ${PYTHON}) /usr/local/bin/python
RUN apt-get update && apt-get install -y \
    build-essential curl \
    git wget openjdk-8-jdk \
    ${PYTHON}-dev virtualenv swig
RUN ${PIP} --no-cache-dir install \
    Pillow h5py keras_applications \
    keras_preprocessing matplotlib \
    mock numpy scipy sklearn \
    pandas future portpicker \
    && test "${USE_PYTHON_3_NOT_2}" -eq 1 && true ||
    ${PIP} --no-cache-dir install enum34
```

Listing 2: Comandos para la instalación de python.

- Lanzamiento del algoritmo de ML definido por el usuario en un fichero Python.
- Envío del grado de paralelismo al *framework* TF elástico.
- Obtención del identificador de proceso asignado a TF.

Por último, realiza la creación de 3 hilos donde cada uno tiene asignado una tarea distinta:

- *Actualización de recursos del contenedor*. Se encarga de recibir la información del nuevo grado de paralelismo del contenedor y enviar la

```
RUN pip3 --version
RUN cd /scheduler_src && \
    pip3 install tensorflow-2.4.0-cp38-cp38-linux_x86_64.whl
RUN pip3 install packaging && \
    pip3 install tensorflow-datasets && \
    pip3 install tensorboard && \
    apt-get install htop && \
    apt-get install nano

RUN pip3 install plotly && \
    pip3 install pandas

RUN apt-get update
RUN apt install docker.io -y
RUN docker --version
#TF Warmup
RUN python3 /scheduler_src/models/keras_example_resnet_warmup.py
# Insert Tensorflow algorithm and parameters
ENTRYPOINT python3 /home/Scheduler/Client/client.py TF_Algorithm
```

Listing 3: Comandos para instalación de TF y declaración del algoritmo.

```
# $1: name of image, for ex. tf_scheduler
# $2: docker image, for ex. tensorflow-program.Dockerfile
docker build -t $1 -f  $2 .
```

Listing 4: Comando para la construcción del contenedor.

señal correspondiente a TF. Debido a que el *framework* define dos tipos de paralelismo (inter e intra), es necesario enviar las señales para modificar uno o ambos, según lo solicite el sistema.

- *Comunicación con el servidor del planificador*. Espera la recepción de mensajes del sistema, asociados a cambios de paralelismo o eventos inesperados.
- *Control de ejecución del algoritmo de ML*. Cada intervalos finitos de tiempo realiza la consulta del estado del algoritmo. Para identificar si

está activo o no, evalúa si el algoritmo modificó el fichero de registro/-log. Si no fue modificado luego de un tiempo definido por un umbral, se procede a la cancelación del contenedor y el aviso al sistema para que tome una decisión.

El cliente también cuenta con una rutina de manejo de evento, la cual se encarga de recibir la señal de finalización del algoritmo, avisar a los demás hilos de este evento y enviar un mensaje al sistema para indicar su finalización.

4.3.2. Módulo del servidor del planificador

El servidor es el principal actor del planificador encargado de múltiples tareas necesarias para el control total de las peticiones y de los contenedores activos. La lógica para estas tareas se encuentra en el fichero principal y en ficheros secundarios.

El fichero principal presenta la lógica para la atención de las peticiones, el control general de los recursos del sistema y el control de cada contenedor. Excepto esta última tarea, las demás son asignadas a un hilo de ejecución de Python. Cada contenedor es controlado por un hilo que tiene asignada una función. Por ejemplo, si se cuenta con 4 contenedores activos, se asignará un hilo por contenedor para controlarlos. A continuación se describe cada una de las funciones que se asignan a los hilos.

Atención de peticiones

Para esta tarea se define una función que centraliza el procesamiento de las diferentes peticiones que se generan en el sistema, ya sea por los contenedores activos, cuando finalizan su atención o por parte del usuario del planificador para indicar el lanzamiento o actualización de sus recursos. Esta función finaliza su trabajo cuando modifican el valor de una variable compartida que indica el estado del planificador. Para explicar su comportamiento, se puede separar en tres tareas principales.

La primera de ellas es la encargada de atender las peticiones de finalización de los contenedores activos. Se obtiene cada una de las peticiones de una cola específica. Esta petición contiene el número de hilo asignado y los recursos que se liberan. Luego, se invoca a una función del sistema para liberar los recursos y, por último, se espera a que el hilo que controla al contenedor finalice. Su implementación se visualiza en el listado 5.

La segunda tarea realizada por esta función es la atención de las peticiones de lanzamiento/actualización de contenedores que fueron procesadas en

```
while(not cola_peticiones.empty()):
  container_thread_id, resources = cola.get()
  system_info.free_resources(resources)
  container_thread_id.join()
```

Listing 5: Atención de peticiones de finalización de contenedores

el pasado pero por escasez de recursos disponibles no pudieron completarse. Para ambos tipos de peticiones se comienza con la reserva de recursos. En caso que sea una petición de lanzamiento, se obtienen los recursos solicitados de la misma. Caso contrario, si es una actualización, se realiza una búsqueda del contenedor en una estructura que almacena la información de cada uno. Si se encuentra, se calcula la cantidad de recursos solicitados como la diferencia entre los recursos de la petición y los utilizados actualmente por el contenedor. Una vez obtenido este valor, se analiza si es posible reservar los recursos del sistema. Si el valor solicitado es mayor a 0 y es menor a la cantidad disponibles en el sistema se cumple la solicitud. Caso contrario, si la petición solicita incremento de los recursos, se reservan los disponibles en el sistema aunque sea menor. Por último, en el caso que la petición solicite un decremento de los recursos utilizados, se libera dicha cantidad. La función se llama *reserva_recursos* y el pseudocódigo se visualiza en el listado 6.

El siguiente paso es realizar el lanzamiento del contenedor en caso que la política de planificación lo permita. Para esto, si la petición requiere un aumento del paralelismo se invoca la política de planificación para que retorne los valores para ambos tipos de paralelismos de TF (INTER e INTRA). En el caso de las peticiones de inicio, si la política retorna ambos valores mayores a cero, se realiza el lanzamiento del contenedor. Si es una petición de actualizaciones pueden suceder dos casos; que se solicite un aumento o un decremento del paralelismo. En el primer caso, si la política de planificación retorna ambos paralelismos con un valor nos indica que debemos aumentar tanto el INTER como el INTRA. Si retorna un tipo de paralelismo con valor y otro no, solo se actualiza el que contiene valor. Si ambos no tienen valor no se actualiza ninguno de los dos. En el listado 7 se visualiza la implementación de dicha función llamada *planificacion_peticion*.

Para atender las peticiones pendientes se recorre dicha cola mientras hay recursos disponibles y peticiones para analizar. Cada una de las peticiones es planificada y retorna un valor que indica si la misma pudo cumplirse.

```
def reserva_recursos(peticion, rec_disp):
  if(isinstance(peticion, 'Inicio'):
    rec_pedido= peticion.inter_p + peticion.intra_p
  else:
    for(lista_contenedores in c):
      if(contenedor.id = peticion.id):
        rec_pedido=(peticion.inter_p+peticion.intra_p)
                -(c.inter_p+c.intra_p)
  if(rec_pedido>0) and (rec_pedido < rec_disp):
    system.aplicar_recursos(rec_pedido)
  else:
    if(rec_pedido>0):
      system.aplicar_recursos(rec_disp)
    else:
      system.liberar_recursos(rec_pedido)
```

Listing 6: Método para la reserva de recursos.

En caso que no se pudo planificar, es almacenada en una cola auxiliar. Una vez que se termina el procesamiento de estas peticiones en caso que haya peticiones en la cola auxiliar, son almacenadas nuevamente en la cola de peticiones pendientes. El pseudocódigo se visualiza en el listado 8.

La tercera y última tarea desarrollada corresponde al procesamiento de las nuevas peticiones de lanzamiento/actualización de contenedores enviadas por el usuario del planificador. Esta tarea es similar a la atención de peticiones pendientes con la diferencia de que si la planificación retorna que no se pudo completar, es colocada en la cola de peticiones pendientes. El pseudocódigo de la implementación se observa en el listado 9.

Entre las tareas distribuidas en ficheros secundarios, se encuentra la aplicación de una política de planificación a una petición que se desea lanzar, el almacenamiento de la información de los contenedores activos para un mayor control de su ejecución y la toma de decisiones cuando finaliza o suceden eventos inesperados como un fallo en su ejecución.

Por otra parte, estos ficheros definen el control de los recursos del sistema para evitar sobrecarga de los mismos manteniendo siempre un umbral que puede ser especificado por el usuario.

Asimismo, es necesario utilizar un protocolo para la comunicación entre el servidor y el cliente alojado en cada contenedor que permita intercam-

```python
def planificacion_peticion(peticion):
  ok=False
  if(isinstance('Inicio'):
    inter_p, intra_p = politica.planificar(peticion)
    if (inter_p >0) and (intra_p > 0):
      #Lanzar contenedor docker
      ok=True
  else:
    for(lista_contenedores in c):
      if(c.id = peticion.id):
        if(rec_pedido < 0): #Decrementar el paralelismo
          c.actualizar(inter_p, intra_p)
        else: #Aumentar el paralelismo
          inter_p, intra_p = politica.planificar(peticion)
          if(inter_p>0) or (intra_p>0):
            if(inter_p>0) and (intra_p>0):
              #Actualizar ambos paralelismos
              c.actualizar(inter_p, intra_p)
            else:
              if(inter_p>0):
                #Solo actualizar inter
                c.actualizar(inter_p, 0)
              else:
                #Solo actualizar intra
                c.actualizar(0, intra_p)
        ok=True
  return ok
```

Listing 7: Método para planificación de petición

biar información, controlar el estado de ejecución e indicar nuevos valores de paralelismo. También es importante que se encargue de almacenar la información de ejecución de cada contenedor, como los tiempos de inicio y finalización, y los recursos que se fueron asignando a lo largo de su vida. Estos datos permiten realizar cálculos de métricas de rendimiento al final de la ejecución del planificador.

A continuación, se ilustra en detalle cada una de estas funcionalidades definidas en los ficheros secundarios.

```
while(not cola_pendientes.empty()) and (recursos_disponibles >0):
  peticion = cola_pendientes.get()
  recursos_solicitados = planificacion_recursos(peticion)
  ok= planificacion_peticion(peticion)
  if( not ok ):
    cola_aux.put(peticion)
while(not cola_aux.empty()):
  cola_pendientes.put(cola_aux.get())
```

Listing 8: Atención de peticiones de peticiones pendientes.

```
while(not cola_nuevas.empty()) and (recursos_disponibles >0):
  peticion = cola.get()
  recursos_solicitados = planificacion_recursos(peticion)
  ok= planificacion_peticion(peticion, recursos_solicitados)
  if( not ok):
    cola_pendientes.put(peticion)
```

Listing 9: Atención de nuevas peticiones.

Generación de peticiones

El planificador es capaz de atender peticiones para ejecutar un nuevo contenedor o actualizar los recursos de uno que se encuentre activo. Para la construcción de estas peticiones se define una clase genérica con los campos necesarios para su posterior atención. También se definen las clases hijas *Start* y *Update* que permiten definir una petición de lanzamiento y de actualización de un contenedor respectivamente. En el listado 10 se visualiza su implementación.

Información de contenedor

Para el almacenamiento de la información de los contenedores se define una clase en Python que cuenta principalmente con los siguientes campos:

- Nombre del contenedor.
- Numero de proceso de Docker.

```python
class Request():
    def __init__(self, request_id, inter_p=-1, intra_p=-1):
        self.__request_id=request_id
        self.__inter_parallel=inter_p
        self.__intra_parallel=intra_p

    def get_request_id(self):
        return  self.__request_id

    def get_inter_parallelism(self):
        return self.__inter_parallel

    def get_intra_parallelism(self):
        return self.__intra_parallel

class Start(Request):
    def __init__(self, request_id, image, inter_p, intra_p):
        super().__init__( request_id,inter_p, intra_p)
        self.__image= image

    def get_image(self):
        return self.__image

class Update(Request):
    def __init__(self, request_id, inter_p=-1, intra_p=-1):
        super().__init__(request_id,inter_p, intra_p)
```

Listing 10: Clases para la generación de peticiones.

- Grados de paralelismo INTER e INTRA definidos para TF por el servidor del planificador.
- Grados de paralelismo INTER e INTRA indicados por el usuario. No necesariamente corresponde a los definidos por el servidor del planificador, ya que hay contenedores que no pueden atenderse con la cantidad de recursos solicitada o se asignan más si hay disponibles.
- Socket de comunicación con el contenedor.
- Imagen de Docker utilizada.

Esta clase contiene los métodos necesarios para la manipulación de estos campos. Además, define un método encargado de actualizar el paralelismo del contenedor, el cual recibe por parámetro los nuevos valores del mismo. Su función es evaluar cuáles tipos de paralelismos de TF se deben modificar (INTER, INTRA o ambos). Luego, se genera un paquete de información con los nuevos valores del paralelismo y se envía al contenedor utilizando sockets. Por último, se actualizan los campos del contenedor que almacenan dicha información. En el listado 11 se visualiza la implementación de dicho método.

Manejo de recursos del sistema

Para la gestión de los recursos se define una clase encargada de almacenar la cantidad total que presenta el sistema expresado en núcleos y la cantidad utilizados actualmente por los contenedores expresado en la misma unidad.

La clase cuenta con los métodos necesarios para chequear, reservar, y liberar los recursos solicitados por el servidor.

La verificación de recursos solo realiza la resta entre los recursos totales y los que se encuentran en uso. Luego, si el servidor desea reservar una determinada cantidad se analizará si es posible. En este caso, si los recursos solicitados son menores o iguales a los disponibles se reservan y devuelve como exitosa la operación. Cuando recibe un llamado para la liberación de recursos aumenta la cantidad de recursos disponibles sin realizar chequeos. Esta clase también presenta métodos adicionales para consultar la cantidad total de recursos del sistema, la ocupación actual y el uso de memoria.

En el listado 12 se visualiza la definición de la clase con los métodos explicados anteriormente.

Comunicación servidor-cliente

Para comunicar el servidor con cada cliente se utiliza la librería `Socket`. Python proporciona una API conveniente y consistente que se asigna directamente a las llamadas del sistema. La función `socket()` devuelve un objeto cuyos métodos implementan las diversas llamadas al sistema para la comunicación. Los tipos de parámetros tienen un nivel más alto que en la interfaz en el lenguaje C: al igual que con las operaciones de lectura (`read`) y escritura (`write`) en los archivos de Python, la asignación del búfer en las operaciones de recepción es automática y la longitud del búfer está implícita en las operaciones de envío [39].

Luego de importar esta API, se definen dos funciones para el envío y

```python
# Actualizar el paralelismo total del contenedor en ejecución
# Retorna si la operación de actualización finaliza correctamente
def update_parallelism(self, inter_parallel=0, intra_parallel=0):
    # Nuevo paralelismo total soportado por el contenedor
    if inter_parallel > 0 and intra_parallel > 0:
        new_parallel= inter_parallelism + intra_parallelism
    else:
        if(inter_parallel>0):
            new_parallel=inter_parallel+self.intra_exec_parallel
        else:
            new_parallel=intra_parallel+self.inter_exec_parallel
    # Comando de actualización del paralelismo del contenedor
    run_command= 'docker update ' + str(self.docker_ps)
    + ' --cpus ' + str(new_parallel)
    # Generar objeto JSON para enviar actualizacion
    data= {
        "container": self.container_number,
        "inter_parallelism": inter_parallel,
        "intra_parallelism": intra_parallel
    }
    # Enviar objeto JSON al cliente
    json_data_socket._send(self.clientsocket, data)
    # Actualizar informacion del paralelismo del contenedor
    if inter_parallel > 0:
        # Actualizar info del inter paralelismo del contenedor
        self.inter_user_parallel= inter_parallel
        self.inter_exec_parallel= inter_parallel
    if intra_parallel >0:
        # Actualizar info del intra paralelismo del contenedor
        self.intra_user_parallel= intra_parallel
        self.intra_exec_parallel= intra_parallel
```

Listing 11: Método para actualización del paralelismo de TF.

recepción de los datos. El formato elegido para los datos es JSON (*Java Script Object Notation*) el cual es uno de los más extendidos para almacenamiento y transferencia de datos. Esto se debe a la facilidad de entender su estructura y al bajo costo de espacio requerido.

```python
class systemInfo:
    def __init__(self):
        self.cores= multiprocessing.cpu_count()
        self.cores_used=0
    def total_cores(self):
        return self.cores
    def check_resources(self):
        return self.cores - self.cores_used
    def apply_resources(self, parallelism, not_control=False):
        if ((not not_control) and (parallelism >
        (self.cores - self.cores_used))):
            return False
        else:
            self.cores_used= self.cores_used + parallelism
            return True
    def free_resources(self, parallelism):
        self.cores_used= self.cores_used - parallelism
    def system_occupation(self):
        return (self.cores_used/self.cores)*100
```

Listing 12: Clase para el manejo de los recursos computacionales.

Al tratar con JSON, a menudo nos encontramos con dos términos conocidos como *serialización* y *deserialización* de datos. El formato básico para escribir JSON es solo un tipo de datos de cadena que contiene datos en pares clave-valor. Para que la máquina entienda esta cadena, debe convertirse en un objeto que luego pueda ser consumido por el intérprete. El proceso de convertir una cadena JSON en un objeto Python se llama *deserialización* y el proceso de convertir un objeto Python a JSON se llama *serialización*. Cuando enviemos los datos se debe realizar la serialización para luego enviar dos mensajes a través del socket. Caso contrario, cuando recibimos un paquete debemos deserializarlo.

Para usar este formato en Python se importa la librería JSON. Hay cuatro métodos básicos en esta biblioteca de la siguiente manera:

- `json.dump`: este método se usa para serializar un objeto Python de la memoria en una secuencia con formato JSON que se puede escribir en un archivo.

- `json.dumps`: se usa para serializar los objetos de Python en la memoria en una cadena que está en formato JSON. La diferencia entre ambos es que en el primero se produce un flujo de datos mientras que en el segundo se crea un tipo de datos de cadena.

- `json.load`: puede usar este método para cargar datos de un archivo JSON que existe en el sistema de archivos. Analiza el archivo y luego deserializa los datos en un objeto python.

- `json.loads`: es similar a json.load, la única diferencia es que puede leer una cadena que contiene datos en formato JSON.

En el listado 13 se visualiza la implementación de las funciones para el envio y recepcion de datos a través del socket en Python. Para el envío se serializan los datos usando la función `dumps` de la librería JSON y luego se envía la dimensión del paquete y los datos serializados usando la función `send` de la librería `socket`. Para la recepción, es necesario el proceso inverso en el cual se obtiene la dimensión del paquete y sus datos para luego realizar la deserialización y retornarlos para su lectura/procesamiento.

Políticas de planificación

Para definir el comportamiento de las políticas de planificación se crea una nueva clase esqueleto con los campos genéricos que requiere cualquier política del planificador. Entre los campos se encuentran:

- *Colas de peticiones nuevas.* Dependiendo de la política utilizada, es posible que se defina más de una cola, por ejemplo, en el caso de utilizar colas con diferentes prioridades. Es por esto que se declara una variable para almacenar un arreglo de colas.

- *Colas de peticiones pendientes.* La estructura de las colas es igual a la de nuevas peticiones. La diferencia es que en estas se almacenan las peticiones que no se planificaron debido a la falta de recursos.

- *Tipo de asignación de recursos.* Indica cuál es la política para asignar los recursos libres a los contenedores activos. Muchas veces varias políticas tienen la misma estructura pero se diferencian en la asignación de recursos. Para no crear una nueva clase para cada una, se crea la misma para esas políticas y se especifica en un campo cuál se desea utilizar.

```python
def _send(socket_, data):
  try:
    serialized = json.dumps(data)
  except (TypeError, ValueError) as e:
    raise Exception('You can only send JSON-serializable data')
  # send the length of the serialized data first
  size_data= str(len(serialized)) + '\n'
  socket_.send(bytes(size_data, 'utf-8'))
  # send the serialized data
  socket_.send(bytes(serialized, 'utf-8'))

def _recv(socket_):
  # read the length of the data
  length_str = ''
  char = socket_.recv(1).decode('utf-8')
  while char != '\n':
    length_str += str(char)
    char = socket_.recv(1).decode('utf-8')
  total = int(length_str)
  # read data from socket
  data = socket_.recv(total).decode('utf-8')
  try:
    deserialized = json.loads(data)
  except (TypeError, ValueError) as e:
    raise Exception('Data received was not in JSON format')
  return deserialized
```

Listing 13: Métodos para envío y recepción de datos usando sockets en python.

- *Factor de proporción*. Indica cómo se distribuyen los recursos libres entre los dos tipos de paralelismo que presenta el *framework* TF.

Para la definición de cada política se crea una clase que hereda de la especificada anteriormente donde el constructor debe especificar la cantidad de colas, el tipo de asignación y el factor de proporción.

Esta versión inicial del planificador cuenta con la política FCFS (First Come - First Serve) con una cola de peticiones nuevas y pendientes, factor de proporción igual a 1, y con tres variantes de políticas de asignación de

recursos explicadas en el diseño de la solución.

Por otro lado, esta nueva clase cuenta con un método que retorna una lista con los recursos para cada paralelismo de una determinada petición, recibiendo por parámetro ambos paralelismos solicitados y teniendo en cuenta la política de asignación utilizada:

- **FCFS original**. Comprueba si el INTER e INTRA recibido por parámetro es mayor 0. Si se cumple, se verifica que ambos paralelismos sean menores o iguales a los recursos disponibles para guardar en una lista definitiva los valores. Caso contrario, si es menor a 0 esta indicando una disminución del paralelismo para un determinado contenedor activo. El código se visualiza en el Listado 14.

```
def strict_scheduling(inter_request, intra_request)
  parallelism_request= inter_request+intra_request
  if (parallelism_request > 0):
        if resources_availables >= parallelism_request:
      # Se asigna el total de recursos solicitados
      return [inter_request, intra_request]
        else:
          # No se pueden asignar recursos
          return []
  else:
    # Es una peticion de actualizacion y disminucion de recursos
    # del contenedor (se asignan los valores negativos para luego
    # liberar los recursos)
    return [inter_request, intra_request]
```

Listing 14: Codigo para planificar recursos de una peticion con la politica *FCFS*.

- **Always attend**. Comprueba que los recursos disponibles sean como mínimo 2, ya que no es posible asignar un valor menor a 1 al INTER e INTRA de TF. Al igual que la anterior política, si los recursos disponibles son mayores o iguales a los solicitados se guardan en una lista definitiva para posteriormente asignarlos al contenedor. Caso contrario, si ambos paralelismos solicitados son válidos (mayor a 0) y no alcanzan los recursos disponibles, se calcula para cada tipo de paralelismo la relación entre su valor y la sumatoria de ambos paralelismos.

Por ejemplo, si el INTRA y el total de paralelismo solicitados es 4 y 12 respectivamente, la proporción INTRA es 0.3. Si los recursos libres son 9, se le asignará 3. Pueden suceder problemas de redondeo cuando se calcula la cantidad asignada a cada paralelismo, por lo que, luego se realiza una verificación para que ambos como mínimo sean 1. En caso que alguno de ellos sea 0, se le asigna 1 y se decrementa el valor del otro paralelismo. El código de la implementación se visualiza en el listado 15.

- **Max prop**. Inicialmente realiza la misma comprobación de la política anterior, es decir, si se cuenta con un mínimo de 2 recursos disponibles. Si se cumple, se obtiene el factor de proporción teniendo en cuenta los recursos solicitados por todas las peticiones pendientes de lanzamiento. Para cada nivel de paralelismo se calcula la relación entre el valor solicitado y el factor de proporción. Es necesario truncar el valor del calculo anterior para evitar problemas de mayor asignación de recursos que los disponibles. El código de la implementación se visualiza en el listado 16.

  Es importante comentar que el factor de proporción es calculado una sola vez por el planificador antes de atender todas las peticiones pendientes. Para el cálculo se recorren todas las peticiones para obtener la sumatoria total de recursos solicitados. Luego se calcula la relación entre esa sumatoria y la cantidad de recursos disponibles. Este valor es almacenado en un campo de la clase que contiene las políticas.

Manejo de eventos de los contenedores

Almacenar la información de todos los eventos que ocurren en cada contenedor es importante para analizar el rendimiento del sistema y aplicar posibles mejoras para planificaciones posteriores. Actualmente interesa almacenar los siguientes eventos:

- Llegada de una petición.
- Atención de petición.
- Cambio de paralelismo de un contenedor.
- Finalización de contenedor.

```
def always_scheduling(inter_request, intra_request)
if (resources >= 2):
    # se asigna como mínimo 1 hilo inter y 1 hilo intra
    if (inter_parallel+intra_parallel <= resources):
        return [inter_request, intra_request]
    else:
        if (inter_parallel>0) and (intra_parallel >0):
            resources_request=inter_parallel+intra_parallel
            # Petición con asignación de ambos paralelismos
            inter_fraction= inter_request/resources_request
            intra_fraction= intra_parallel/resources_request
            inter_p= int(round(inter_fraction*resources))
            intra_p= int(round(intra_fraction*resources))
        else:
            # Actualizacion de un solo tipo de paralelismo
            if(inter_parallel>0):
                inter_p=resources
                intra_p=0
            else:
                inter_p=0
                intra_p=resources
        # Se consulta por ambos valores luego de asignar
        # proporción para evitar problemas de redondeo
        if inter_p == 0:
            inter_p= inter_p+1
            intra_p= intra_p-1
        if intra_p == 0:
            intra_p= intra_p+1
            inter_p= inter_p-1
    return [inter_p, intra_p]
else:
  # No se pueden asignar recursos
  return []
```

Listing 15: Código para planificar recursos de una petición con la política *Always*.

Para el manejo se define una clase llamada `TraceLog` encargada del almacenamiento de todos los eventos de los contenedores. Cada evento es un

```python
def max_prop_scheduling(inter_parallel, intra_parallel)
    if (resources > 2):
        # Obtener el factor de proporcion
        factor_prop= get_factor_prop()
        # Calcular la proporcion de inter e intra paralelismo.
        # El round a veces asigna mas recursos
        # que el total disponible.
        # Directamente truncar el valor flotante de la división.
        inter_p= int(inter_parallel/factor_prop)
        intra_p= int(intra_parallel/factor_prop)
        if(inter_p == 0) and (inter_parallel > 0): inter_p=1
        if(intra_p == 0)and (intra_parallel > 0): intra_p=1
        return [inter_p, intra_p]
    else:
        if(resources == 2):
          # Se asigna 1 a cada tipo de paralelismo
            return [1,1]
        else:
            # No se pueden asignar recursos a la petición.
            return []
```

Listing 16: Codigo para planificar recursos de una petición con la política Max prop.

diccionario de Python que contiene un conjunto de campos principales:

- Identificador de petición del cliente del planificador.
- Identificador de contenedor Docker.
- Tiempo de comienzo del evento del contenedor.
- Tiempo de finalización del evento del contenedor.

Cada evento de un contenedor es almacenado utilizando un diccionario en Python donde se accede a cada valor a través de una clave. En el listado 17 se visualizan los datos del diccionario con su clave-valor.

La clase `TraceLog` cuenta, principalmente, con un campo de tipo lista para almacenar todos los eventos que ocurren durante la vida del planificador y dos métodos para la creación y finalización de eventos. El método

```
dict(Task=container, Start=datetime.datetime.now(),
Finish=-1, Cores=threads, RequestId= request_id)
```

Listing 17: Diccionario definido para cada evento de un contenedor.

`init_container_event()` recibe la cantidad de recursos asignados por la política de planificación, el identificador de petición y de contenedor para crear un diccionario en Python que almacena los parámetros indicados y el tiempo de comienzo del evento.

Por otro lado, el método `finish_container_event()` presenta una lógica de mayor complejidad que el primero, principalmente porque debe realizar la búsqueda del evento para un identificador de contenedor y asignar el tiempo de finalización. Asimismo, define un parámetro denominado `time_epcohs` que contiene los tiempos de cada época del algoritmo de ML. Este es el número de veces que se ejecutarán los algoritmos de *Forward Propagation* y *Back Propagation*. En cada ciclo (época) todos los datos de entrenamiento pasan por la red neuronal para que esta aprenda sobre ellos. Por ejemplo, si existen 10 ciclos y 1000 datos, en cada ciclo o época los 1000 datos pasarán por la red neuronal. El formato de este parámetro en la función es un vector de épocas y por defecto tiene valor nulo. Por ejemplo, si un algoritmo presenta 5 épocas de 50 segundos cada una, el parámetro será *[50,50,50,50]*. Cada época es separada en un evento para que la visualización de la traza de ejecución sea correcta. Esta solución se explica en detalle en el apartado 4.3.3. En la listado 18 se visualiza la implementación de los métodos de inicio y finalización de un contenedor.

4.3.3. Módulo del visor de trazas

El método estándar para analizar el rendimiento de un programa es utilizar herramientas de generación de perfiles. La información sobre el comportamiento de un programa se recopila durante la ejecución y, a menudo, se escribe en archivos para un estudio posterior. Después de la ejecución, tenemos un archivo con datos sobre qué eventos han ocurrido, cuándo y dónde. Eso es lo que llamamos *generación de trazas de ejecución*.

Posteriormente, la información recopilada es procesada estadísticamente por un software de análisis (interpretación de trazas) y el resultado se puede presentar (de forma gráfica o textual) al programador (representación de trazas).

Hoy en día, existen varias herramientas que permiten generar un visor

de eventos, y particularmente Python cuenta con bibliotecas específicas que facilitan la generación de diferentes tipos de gráficos. Plotly es una biblioteca gráfica interactiva de alto nivel y de código abierto que incluye más de 30 tipos de gráficos, incluidos gráficos científicos, gráficos en 3D, gráficos estadísticos, mapas SVG o gráficos financieros, entre otros.

En particular, interesa el módulo *Plotly Express* que proporciona mas de 30 funciones para crear diferentes tipos de figuras. Esta API se desarrolló con el objetivo de ser lo más consistente y fácil de aprender, lo que facilita el cambio de un gráfico de dispersión a uno de barras o a un histograma. Entre todas las opciones, el diagrama de Gantt permite representar barras de tiempo para diferentes trabajos. Por lo tanto, se desarrolla un visor de trazas que utilice este gráfico.

Para generar un diagrama de Gantt con Plotly es necesario que los eventos de los contenedores se encuentren almacenados en diccionarios donde cada uno presenta el tiempo de inicio y finalización, el número de trabajo/-contenedor y opcionalmente se puede elegir un color a la barra. Un ejemplo básico en la wiki de Plotly muestra como definir 3 trabajos (A, B y C), sus tiempos y un valor entre 0 y 100 que define el color de la barra de cada trabajo. Todos estos trabajos se almacenan en un *dataframe* o lista para representarlos gráficamente.

En el listado 19 se visualiza la definición de cada trabajo almacenado en el *dataframe*. La invocación a la función `timeline` de Plotly Express permite generar la traza donde cada fila del *dataframe* representa una barra rectangular, donde la longitud es definida por el tiempo de inicio y finalización que se indican con las claves definidas en los parámetros `x_start` y `x_end`. Además, el parámetro `y` define la clave usada para indicar el número de tarea (o contenedor en nuestro trabajo) y el color usando el parámetro `color`. Por lo tanto, para el ejemplo cada fila que representa un trabajo debe contener un valor para cada clave explicada anteriormente. Por último, se visualiza el gráfico invocando a la función *show*. El gráfico de ejemplo usando Plotly se visualiza en la Fig. 4.6.

Utilizando como base el ejemplo de la web de Plotly, se utilizan los eventos almacenados en la clase `TraceLog` para representarlos en el diagrama de Gantt. Es importante mencionar que cada época correspondiente al algoritmo de cada contenedor se encuentra almacenada en un evento separado. Estos eventos no utilizan el mismo identificador que el contenedor al cual corresponden para que en el gráfico no se muestre en la misma fila. Por ejemplo, los eventos de épocas de un algoritmo para el contenedor 1 se visualizarán en la fila 1.5 en vez de la 1. Esto permite observar los tiempos de cada época debajo de la barra que indica el tiempo total del contenedor. Es-

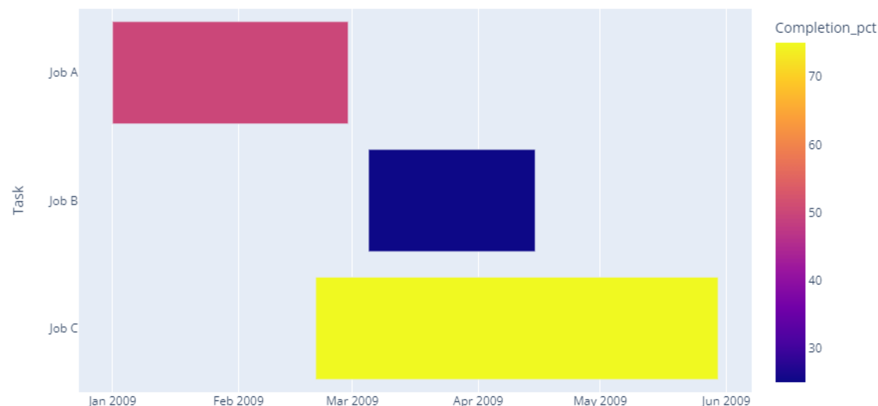


Figura 4.6: Diagrama de Gantt que visualiza el porcentaje completado de cada trabajo.

ta implementación se encuentra en el método *finish_container_event()* (véase listado 18) donde se observa que a cada época se le asigna como valor de fila la suma entre el numero de contenedor y 0.5. En la Fig. 4.7 se visualiza el diagrama de Gantt para un contenedor, donde la posición 0 del eje Y indica el tiempo total del contenedor. En este caso, en un momento de su ejecución se modificaron los recursos, generando dos colores para la representación de la barra. El azul representa el uso de 16 núcleos y el naranja 2. En la posición 0.5 del mismo eje se representan las épocas del algoritmo de ML ejecutado dentro del contenedor, que para este ejemplo contiene 2. La leyenda que describe los colores no representa la cantidad de núcleos para el caso de las épocas y solo se utiliza para separar los tiempos entre cada una de ellas.

Para graficar el diagrama de Gantt se define una función similar a la utilizada en el ejemplo de Plotly. Se utiliza la lista de eventos para indicar de dónde procesarlos, la clave que contiene el tiempo de inicio y finalización de cada evento (`Start` y `Finish`), la clave que pertenece al numero de contenedor (`Task`) y la clave que contiene el color con el cual se representará el evento. Para nuestro trabajo, el color define la cantidad de recursos/núcleos utilizados por el contenedor en ese momento indicado con la clave `Cores`. En el listado 20 se visualiza su implementación.

En el apéndice A se encuentran las instrucciones de uso del planificador.

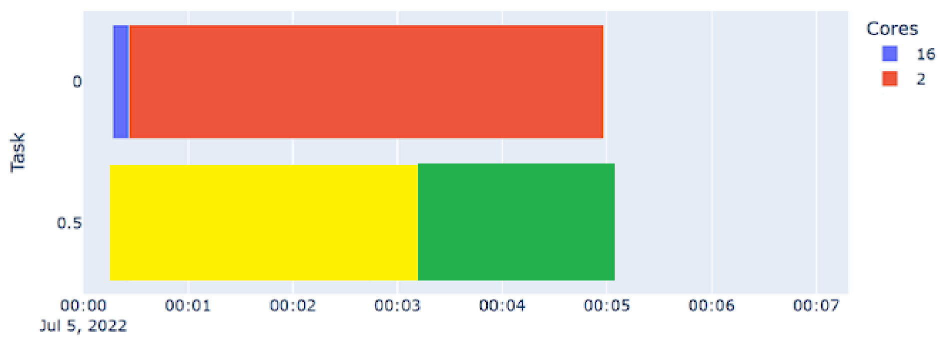


Figura 4.7: Gráfico de contenedor y las épocas del algoritmo de ML.

```python
# Almacenar comienzo de ejecución del contenedor en dataframe
def init_container_event(self,container, threads, request_id):
self.__df_events.append(dict(Task=container,
    Start=datetime.datetime.now(), Finish=-1,
    Cores=threads, RequestId= request_id))

# Asignar tiempo de finalizacion del contenedor en el dataframe
# y generar el timeline de los steps de tensorflow
def finish_container_event(self, container_number, time_steps):
container_num = int(container_num)
start_datetime_step=datetime.datetime.now()
if time_steps:
    for container in self.__df_events:
        if ((int(container["Task"]) == container_num) and
        (not isinstance(container["Finish"], datetime.datetime))):
            start_datetime_step= container["Start"]
    n_container= container_number + 0.5
    colour=1
    for time_step in time_steps:
        added_sec= datetime.timedelta(0, int(time_step))
        finish_datetime_step= start_datetime_step + added_sec
        self.__df_events.append(dict(Task=n_container,
        Start=start_datetime_step, Finish=finish_datetime_step,
            Cores=colour))
        if colour==1:
            colour=2
        else:
            colour=1
        start_datetime_step= finish_datetime_step
else:
    print("No time steps in container: ", str(container_num))
for container in self.__df_events:
    if ((int(container["Task"]) == container_num) and
    (not isinstance(container["Finish"], datetime.datetime))):
        container["Finish"] = datetime.datetime.now()
        print("Changes finish container: ", str(container_num))
```

Listing 18: Métodos para el manejo de eventos en la clase *TraceLog*.

```
import plotly.express as px
import pandas as pd

df = pd.DataFrame([
    dict(Task="Job A", Start='2009-01-01', Finish='2009-02-28'
    , Completion_pct=50),
    dict(Task="Job B", Start='2009-03-05', Finish='2009-04-15'
    , Completion_pct=25),
    dict(Task="Job C", Start='2009-02-20', Finish='2009-05-30'
    , Completion_pct=75)
])

fig = px.timeline(df, x_start="Start", x_end="Finish", y="Task",
color="Completion_pct")
fig.show()
```

Listing 19: Ejemplo de visualización de trabajos con Plotly.

```
def plot_gantt(self, day):
    fig = px.timeline(self.__df_events, x_start="Start",
    x_end="Finish", y="Task", color="Cores")
    fig.show()
```

Listing 20: Función en clase *TraceLog* para graficar diagrama.

Capítulo 5

Resultados

5.1. Especificaciones para las pruebas

Para extraer los resultados se realizaron las pruebas sobre una arquitectura de servidor NUMA. El mismo esta compuesto por dos procesadores Intel(R) Xeon(R) CPU E5-2670 con 8 núcleos físicos, los cuales se pueden extender a 16 núcleos lógicos vía tecnología *HyperThreading*. Cada núcleo presenta una frecuencia nominal de 2.6 GHz y máxima de 3.3 GHz. La memoria RAM es tecnología DDR3 con capacidad de 64 Gbytes. El sistema operativo base es Ubuntu 18.04 LTS.

Para evaluar la elasticidad en TF se utilizó la versión 2.0, la cual fue lanzada en el año 2019 y presenta cambios significativos con respecto a la primera versión. Particularmente, agrega un modo de ejecución *eager* donde se evalúan las operaciones inmediatamente cuando están disponibles (se cumplen sus dependencias) y no se genera el grafo de ejecución. Para nuestras pruebas, este modo es deshabilitado y se utiliza el clásico modelo de ejecución donde se genera un grafo de ejecución de las operaciones del modelo de ML. Para habilitar la elasticidad, se realizaron los cambios necesarios en los ficheros explicados en el Capítulo 3.

El algoritmo seleccionado para nuestras pruebas es Resnet50 definido a través de Keras. Se realiza el entrenamiento de dicha red neuronal profunda definiendo 5 épocas, con 20 pasos por época. El conjunto de datos utilizados pertenecen al set CIFAR-100 que contiene imágenes con una dimensión de 32x32 y 3 canales. El número de clases es un valor fijo definido en 10, con un batch size de 128. En cualquier caso, la mayor parte de los resultados observados son fácilmente extrapolables a otros modelos o condiciones de experimentación.

En las siguientes secciones se desarrollan los experimentos de la tesis. En la sección 5.2 se evalúa el funcionamiento del paralelismo en la versión elástica de TF para verificar que el INTER e INTRA paralelismo permite modificar su valor en tiempo de ejecución del algoritmo de ML. Luego, en la sección 5.3 se demuestra el problema de oversubscription generado cuando las aplicaciones no cuentan con mecanismos de elasticidad en contenedores Docker. Por último, en la sección 5.4 se evalúa el planificador de contenedores donde se plantean diferentes escenarios de demanda de recursos para analizar el rendimiento de los algoritmos de ML en contenedores con y sin elasticidad.

5.2. Elasticidad en Tensorflow

5.2.1. Elasticidad del inter paralelismo

Para comprobar que los cambios realizados en el paralelismo INTER de TF funcionan correctamente es necesario lanzar el algoritmo con la versión elástica del *framework* y realizar modificaciones de éste paralelismo en tiempo de ejecución. La validación de la elasticidad se visualiza con las trazas de tiempos de los hilos por batch retornadas por Tensorboard. En esta prueba obtuvimos 3 trazas en puntos específicos de la ejecución. Interesa al comienzo para comprobar que tiene asignado los recursos aplicados en el lanzamiento; en el intermedio, luego de realizar el primer cambio de recursos; en el final, cuando se asignan nuevamente los recursos iniciales. En cada etapa mencionada calculamos el tiempo promedio de un conjunto de batches para obtener la tendencia central.

Para esta prueba, se utiliza un *script* que realiza dicha tarea. En el listado 21 se visualiza la implementación.

Como primer paso del script, se escribe en un archivo de texto el paralelismo inicial que se asigna al framework, que para esta prueba, es de 16 hilos INTER y 1 hilo INTRA. Luego se ejecuta el algoritmo en background y se obtiene su identificador de proceso (PID) necesario para informar a TF los cambios de paralelismo a través de la señal número 10 de la lista de eventos del sistema. Luego de 90 segundos, se realiza el primer cambio de recursos, disminuyendo los hilos INTER a 1. Se espera el mismo tiempo y se vuelve incrementar a 16 el mismo tipo de paralelismo.

Cada ejecución del script almacena una traza para el procesamiento de un batch de imágenes. Por lo tanto es necesario realizar 3 ejecuciones para obtener las trazas de hilos al inicio, a la mitad y al final. En cada lanzamiento se varía el número de batch en el código del algoritmo TF, el cual cuenta con un procesamiento total de 100 batches. Para obtener la muestra del

```
#!/bin/bash
# Ejecutar 3 veces variando el profile batch en keras
para que tome un batch inicial, intermedio y final
# Indicar paralelismo inter e intra en fichero
fichero='/root/tf_parallelism.txt'
echo 16 1 > $fichero
# Envio a ejecución primer algoritmo
python3 keras_example_resnet.py 16 1 &> execution1.txt &
BACK_PID1=$!
echo "PID: $BACK_PID1"
sleep 90
# Disminuyo paralelismo inter a la mitad
echo 1 1 > $fichero
sudo kill -10 $BACK_PID1
sleep 90
# Incremento paralelismo inter a 16
echo 16 1 > $fichero
sudo kill -10 $BACK_PID1
#Espero finalizacion del algoritmo
wait $BACK_PID1
```

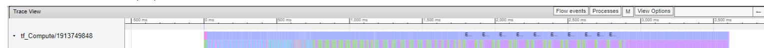
Listing 21: Script en bash para validación del INTER.

comienzo se eligen los primeros 10 batches. El intermedio corresponde al rango 45-55, y el último entre 90-100.

En la Fig. 5.1a se visualiza que la cantidad de recursos al comienzo de la ejecución del algoritmo es 16 hilos INTER. En la Fig. 5.1b se observa que el cambio de paralelismo INTER a 1 se realiza correctamente. Por ultimo, la Fig. 5.1c confirma que el cambio final correspondiente al aumento de hilos INTER a 16 ha funcionado.

Con respecto a los tiempos de los batches analizados, es importante mencionar cual es el overhead o costo introducido al utilizar una versión elástica del framework. En la tabla 5.1 se visualizan los tiempos de ejecución de la etapa evaluada en versión elástica y los tiempos de la etapa inicial asignando 1 y 16 hilos INTER en la versión no elástica. Se observa que el overhead introducido por la reducción del paralelismo al mínimo es de aproximadamente 1.5 por ciento si se compara los tiempos de batch intermedio en versión elástica e inicial con 1 hilo INTER en la no elástica. Si analizamos el costo de

(a) Traza de hilos INTER en el tercer batch.



(b) Traza de hilos INTER en batch intermedio.

(c) Traza de hilos INTER en batch final.

Figura 5.1: Batches para la evaluación del inter paralelismo.

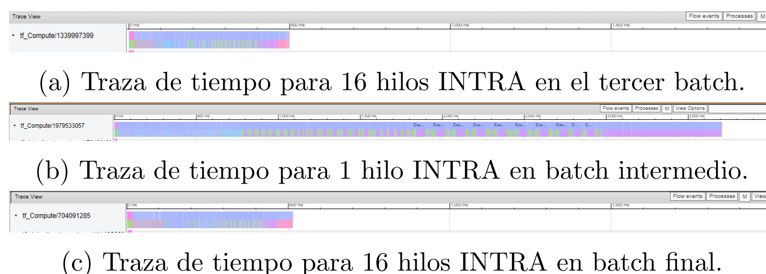
aumentar del minimo al máximo la cantidad de recursos no se observa una diferencia entre ambas implementaciones, ya que el batch de la etapa final de la versión elástica tarda 2021 milisegundos y la ejecución de un batch con la misma cantidad de recursos en la versión no elástica presenta un tiempo de 2067 milisegundos.

5.2.2. Elasticidad del Intra Paralelismo

La verificación de la elasticidad en el INTRA es similar al INTER. El script utilizado tiene pequeñas modificaciones, principalmente, en la cantidad de hilos INTER e INTRA que para esta prueba serán 1 y 16 respecti-

TF	N° Batch	Hilos Inter	Tiempo Batch [ms]
Elástico	Inicio	16	2081
	Intermedio	1	3606
	Final	16	2021
No Elástico	Inicio	1	3553
		16	2067

Tabla 5.1: Tiempos de ejecución de batch por etapa en cada versión TF.



(a) Traza de tiempo para 16 hilos INTRA en el tercer batch.

(b) Traza de tiempo para 1 hilo INTRA en batch intermedio.

(c) Traza de tiempo para 16 hilos INTRA en batch final.

Figura 5.2: Batches para la evaluación del intra paralelismo.

vamente. Por otra parte, los tiempos de espera son distintos debido a que ejecutar con alto grado de paralelismo INTRA incrementa el rendimiento del algoritmo. Otro cambio con respecto al script anterior es que se usa la señal POSIX número 12 del sistema Linux para indicar a TF el cambio de paralelismo. Los tiempos de espera son de 10 segundos para decrementar el paralelismo al minimo; luego se espera 60 segundos para indicar nuevamente el valor inicial de paralelismo asignado. En el listado 22 se visualiza su implementación.

Se obtienen las trazas para los mismos números de batch utilizados para el otro tipo de paralelismo. En la Fig. 5.2 se visualiza un solo hilo INTER que tiene asignado un conjunto de hilos INTRA. Se observa que los tiempos de los batches 3 y 50 (figura 5.2a y 5.2b) es 498 y 3704 milisegundos respectivamente. Por lo tanto, se observa que la disminución del INTRA funciona. Por otra parte, el tiempo del último batch (figura 5.2c) es de 514 milisegundos, lo que evidencia que el aumento de paralelismo es aplicado correctamente por TF.

```
# Ejecutar 3 veces variando el profile batch en keras
para tomar traza en batch inicial, intermedio y final
# Indicar paralelismo inter e intra en fichero
fichero='/root/tf_parallelism.txt'
echo 1 16 > $fichero
# Envio a ejecución primer algoritmo
python3 keras_example_resnet.py 1 16 &> execution1.txt &
BACK_PID1=$!
echo "PID: $BACK_PID1"
sleep 10
# Disminuyo paralelismo inter a la mitad
echo 1 1 > $fichero
sudo kill -12 $BACK_PID1
sleep 60
# Incremento paralelismo inter a 16
echo 1 16 > $fichero
sudo kill -12 $BACK_PID1
#Espero finalizacion del algoritmo
wait $BACK_PID1
```

Listing 22: Script en bash para validación del Intra Paralelismo.

5.3. Evaluación de *oversubscription*

El fenómeno de *oversubscription* es un problema presente cuando los recursos (específicamente el número de núcleos) asignados a una aplicación son inferiores a los realmente utilizados por la misma. En escenarios en los que los recursos asignados a una aplicación se reduzcan dinámicamente por debajo de los inicialmente fijados, puede surgir junto a su consiguiente degradación de rendimiento. Sólo si las aplicaciones son elásticas, será posible aliviar este problema, reduciendo dinámicamente el número de hilos a los nuevos recursos asignados de forma dinámica.

En esta sección, el objetivo es ilustrar dicho fenómeno. Para ello, se plantea un experimento en el que se lanza un contenedor Docker que internamente ejecuta el algoritmo Resnet sobre TF no elástico. En la Tabla 5.2 se visualiza el conjunto de pruebas realizadas. La columna de *Cores* es la cantidad de recursos asignados al contenedor. *Hilos TF* es el producto de los hilos INTER e INTRA creados por el *framework*, donde el primero siempre

es fijo y el segundo varía entre 1 y 5. Por ultimo, *Tiempo Batch* es el tiempo de ejecución del modelo utilizando el primer batch. Se observa que en las ejecuciones con 6 cores asignados al contenedor no se produce *oversubscription*, mostrando una buena escalabilidad. Por otra parte, si se asignan 2 cores al contenedor y se incrementa el numero de hilos TF en sucesivas ejecuciones, se produce *oversubscription*, ya que los hilos comienzan a competir por el acceso a los recursos degradando el rendimiento del algoritmo.

Cores	Hilos TF	Tiempo Batch [ms]
6	2	3056
	3	2339
	4	1764
	5	1436
	6	1268
2	2	3032
	3	4320
	4	4512
	5	5195
	6	5490

Tabla 5.2: Tiempos de ejecución para asignación fija de recursos en contenedor y variación de paralelismo de la aplicación TF.

Para validar la utilización de contenedores elásticos, en este trabajo realizamos dos pruebas donde cada una lanza un contenedor Docker con asignación de 12 núcleos físicos e internamente contiene TF ejecutando el algoritmo Resnet. En ambas pruebas, un segundo después del lanzamiento del contenedor se disminuye la cantidad de recursos asignados a 1.

En la primera prueba se lanza un contenedor con TF elástico ejecutando el algoritmo con 1 hilo INTER y 11 hilos INTRA. Es importante mencionar que TF recibe el cambio de recursos y modifica la cantidad de hilos INTRA a 1 cuando al contenedor se le disminuye la cantidad de núcleos asignados. En la figura 5.3 se visualiza la traza de ejecución para el último el batch del algoritmo con una duración de 3642 milisegundos.

En la segunda prueba se lanza el contenedor con la diferencia que ejecuta el algoritmo TF no elástico. En la figura 5.4 se visualiza la traza de ejecución del algoritmo con un tiempo de batch igual a 5194 milisegundos.

Con estas pruebas se observa que disminuyendo la cantidad de recursos al contenedor sin aplicar el cambio a la aplicación interna, el tiempo de ejecución es mayor en comparación con lanzar la misma prueba con una versión

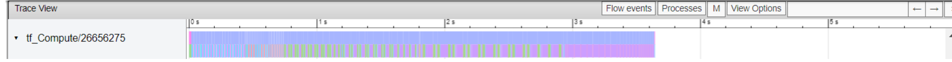


Figura 5.3: Traza de ejecución de TF original en contenedor sin oversubscription.

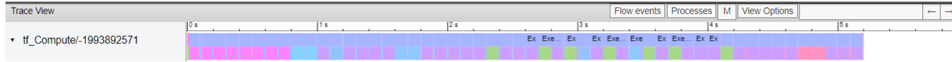


Figura 5.4: Traza de ejecución de TF original en contenedor con oversubscription.

elástica del contenedor/aplicación los cuales reciben el aviso sobre los cambios de recursos. De esta manera, evitamos el fenómeno de *oversuscription* generado por la versión no elástica.

Obsérvese que, de modo contrario al fenómeno de *oversuscription*, el incremento dinámico de recursos por encima de los realmente usados por una aplicación/contenedor puede conllevar un desperdicio de recursos, ya que algunos de los nuevos recursos jamás serán utilizados por la aplicación. De nuevo, el uso de contenedores y aplicaciones maleables mejorará el rendimiento para adaptarlos realmente a la cantidad de recursos disponibles.

5.4. Coplanificación de contenedores elásticos

5.4.1. Elasticidad en contenedores

Para validar que la elasticidad de TF funciona dentro de contenedores, se realiza un experimento en el cual se desea visualizar que un cambio de paralelismo de un contenedor en tiempo de ejecución tiene impacto sobre el rendimiento del mismo. Para esto, se define una carga de trabajo para el planificador que contempla el lanzamiento de un contenedor al comienzo (tiempo cero de comenzar a escuchar peticiones) con un requerimiento de 2 cores. Luego de 30 segundos, se envía una petición de actualización de recursos aumentando los mismos a 16 cores. La diferencia entre ambas pruebas es la versión de TF, donde la primera se realiza con la elástica y la segunda con la original. La política de planificación es *FCFS* explicada en la sección 4.

En la Tabla 5.3 se observan los tiempos para ambas pruebas donde se verifica que el tiempo del contenedor lanzado con la versión elástica de TF es

menor en comparación con la original debido a que el cambio de recursos se aplica tanto al contenedor como a la aplicación interna. Este comportamiento se visualiza en las trazas de ejecución de la Figura 5.5 donde se observa en la fila 0 el tiempo del contenedor separado en dos colores donde la etapa con asignación de 2 cores se encuentra en azul y con 16 cores en rojo para cada subfigura. En la fila 0.5 se observan los tiempos de las épocas del algoritmo donde el color verde corresponde a la primera y el azul a la segunda. En ambas trazas el tiempo antes de realizar el cambio de paralelismo es igual con un valor de 28 segundos. Luego del cambio de paralelismo aplicado, la traza de la figura 5.5a correspondiente a la versión original de TF, presenta un tiempo de 325 segundos. Por otro lado, la traza de la figura 5.5b donde es utilizada la versión elástica de TF, luego de cambio de paralelismo presenta un tiempo de 47 segundos. Esta mejora del rendimiento también se observa en el tiempo de las épocas del algoritmo.

Política	N° Contenedores	Versión TF	Tiempo ejecución [s]
FCFS	1	Elástica	75
		Original	353

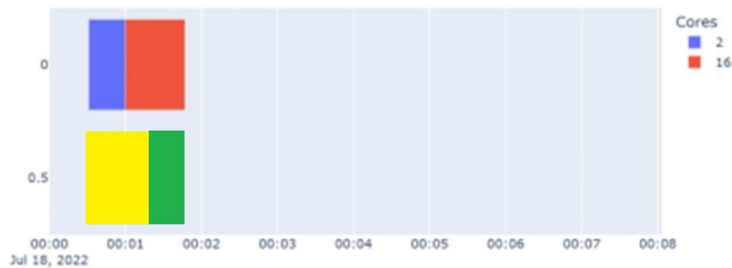
Tabla 5.3: Tiempos medio de ejecución para lanzamiento de contenedor con diferente versión de TF.

5.4.2. Planificación de contenedores

La evaluación del planificador de contenedores se centra en comparar el comportamiento de las dos versiones de TF y observar las diferencias entre las políticas implementadas. Para esto, en primer lugar, es necesaria la definición de métricas del planificador que permitan detectar el comportamiento del mismo. A continuación, se describen cada una de ellas:

- *Tiempo medio de ejecución (TME)*. Promedio del tiempo en el cual el contenedor se encuentra ejecutando el algoritmo en TF. La unidad de medición es segundos.
- *Tiempo medio de respuesta (TMR)*. Promedio del tiempo que demora el planificador en atender las peticiones de lanzamiento. La unidad de medición es segundos.
- *Tiempo medio de ejecución real (TMER)*. Sumatoria de las dos métricas anteriores. La unidad de medición es segundos.

(a) Tiempo de ejecución de contenedor con TF original.



(b) Tiempo de ejecución de contenedor con TF elástico.

Figura 5.5: Trazas de ejecución para validar la elasticidad de TF en contenedores.

- *Tiempo total (TT)*. Tiempo total de duración del planificador. La unidad de medición es segundos.
- *Recursos asignados (RA)*. Relación entre la cantidad de recursos utilizados por los contenedores y el total que presenta el sistema. No presenta unidad de medición.
- *Productividad (PRO)*. Cantidad de contenedores finalizados por hora. La unidad de medición es Contenedores/Hora (C/Hr).

Además, es importante definir escenarios para la evaluación donde se varían los tiempos de llegada de las peticiones en búsqueda de caracterizar los casos que suceden en la vida real. Particularmente, se definen los siguientes escenarios con variantes:

1. *Alta demanda de peticiones*. El tiempo de llegada entre cada una es bajo.

2. *Baja demanda de peticiones.* El tiempo de llegada entre cada una es alto.

En cada uno de estos escenarios se realizaron dos pruebas donde se varía la cantidad de recursos solicitados en cada petición, particularmente, en la búsqueda de caracterizar las siguientes situaciones:

1. *Baja demanda de recursos por petición.* Solicita por debajo de la mitad de recursos que presenta el sistema.

2. *Alta demanda de recursos por petición*, donde solicita arriba de la mitad de recursos que presenta el sistema.

En todos los experimentos se utilizan las tres variantes de asignación de recursos (*FCFS, Always, Max Prop*). Además, se habilita la resignación de recursos priorizando su aplicación cuando se liberan recursos en el sistema. En el apéndice B se visualizan las tablas completas donde se utilizan variantes en la cantidad de contenedores de 8 a 128. Por simplicidad, solo se realizan comentarios cuando se lanzan las pruebas con 128.

Escenario de alta demanda de peticiones

Para modelar una alta demanda de peticiones se analizó la duración de entrenamiento del algoritmo TF con la menor cantidad de recursos asignados. El tiempo para este caso es de 300 segundos. La generación de una sobrecarga de peticiones se realiza con la llegada de peticiones cada 60 segundos debido a que este tiempo es una quinta parte de la duración del algoritmo, provocando acumulación de peticiones pendientes.

En el experimento con baja cantidad de recursos solicitados por petición, se elige de forma aleatoria entre 2 y 8 cores, es decir, como máximo se asigna a un contenedor la mitad de los recursos que se encuentran en el servidor. En la tabla 5.4 se visualizan las métricas para cada política de asignación y sus variantes de versión de TF.

Analizando la métrica TME, en la política *FCFS* el valor se incrementa 8 segundos en la versión elástica debido a que no hay resignación de recursos en ningún contenedor, por lo tanto, esa diferencia corresponde al sobrecoste introducido por la elasticidad. En las políticas *Always Attend* y *Max Prop* esta métrica mejora 18 y 36 segundos respectivamente, ya que hay contenedores que elevan su paralelismo en tiempo de ejecución. El TMR es similar en las diferentes versiones de TF, excepto en la política *Always Attend* donde la diferencia es considerable, aproximadamente 800 segundos, producido principalmente porque cada contenedor presenta un tiempo de ejecución mayor

Política	TF	TME [s]	TMR [s]	TMER [s]	TT [s]	RA	PRO [C/Hr]
FCFS	Elástica	206	1382	1588	6910	0.88	66.6
	Original	198	1237	1435	6652	0.86	69.2
Always Attend	Elástica	217	1230	1447	6853	0.87	67.2
	Original	253	2028	2282	8243	0.92	55.9
Max Prop	Elástica	255	1232	1488	6760	0.92	68.1
	Original	373	1530	1903	7151	0.94	64.4

Tabla 5.4: Métricas de escenario de alta demanda de peticiones y baja cantidad de recursos por contenedor.

y genera retardo en la atención de nuevas peticiones. Analizando en conjunto las métricas RA y PRO, se observa que las versiones originales de TF tienden a asignar mayor cantidad de recursos a sus contenedores sin generar aumento en la productividad, ya que el planificador asigna nuevos recursos al contenedor pero no se aplican al algoritmo TF ejecutado internamente. Esto se ve reflejado, por ejemplo, en la política *Max Prop* donde la versión elástica presenta mayor PRO (68.1 vs 64.4) y menor RA (0.92 vs 0.94) que la original.

Si se comparan las versiones elásticas de las políticas, *Max Prop* presenta un mayor TME, debido a la alta cantidad de contenedores activos en cada momento, que impacta en un aumento del TMER. Pero el PRO y el RA aumentan y el TT disminuye, producido por la alta relación entre contenedores activos y distribución de recursos.

El segundo experimento, donde se busca incrementar la demanda de recursos, es similar al anterior con la diferencia que las peticiones solicitan mayor cantidad de recursos, el cual es un valor aleatorio definido entre 8 y 16 cores, correspondiente a la mitad y el total de recursos del sistema. En la tabla 5.5 se visualizan las métricas para cada política de asignación y sus variantes de TF.

En la métrica TME se observa un comportamiento similar al explicado en el experimento anterior con la diferencia que los valores son menores para las políticas *FCFS* y *Always Attend* debido al aumento de recursos asignados a cada contenedor. El TMR mejora en la versión elástica de las políticas ya que aplica la reasignación de recursos tanto al contenedor como al algoritmo interno. La disminución del TMR impacta en el TMER provocando un aumento del PRO. Para el caso del RA, también sucede el mismo comportamiento, donde es mayor para la versión original sin generar aumento

Política	TF	TME [s]	TMR [s]	TMER [s]	TT [s]	RA	PRO [C/Hr]
FCFS	Elástica	74	3187	3261	10579	0.63	43.5
	Original	70	2946	3016	9910	0.65	46.4
Always Attend	Elástica	95	1705	1801	7300	0.86	63.1
	Original	139	2903	3042	10237	0.9	45
Max Prop	Elástica	347	1425	1772	7079	0.93	65
	Original	375	1550	1925	7252	0.94	63.5

Tabla 5.5: Métricas de escenario de alta demanda de peticiones y baja cantidad de recursos por contenedor.

en el PRO.

Con respecto a la comparación entre políticas, nuevamente *Max Prop* genera mayor cantidad de contenedores activos y distribución de recursos (mayor TME y RA) provocando una disminución en el TT y un aumento en el PRO.

Escenario de baja demanda de peticiones

El escenario de baja demanda de peticiones consiste en generar *huecos* entre peticiones que favorezcan la reasignación de recursos. Existe dos estrategias para el lanzamiento de este escenario: aumentar el tiempo de llegada de las peticiones o disminuir la duración del algoritmo TF. En este caso para evitar demoras en los experimentos se decide disminuir la duración del algoritmo cambiando la cantidad de épocas de 2 a 5 y los pasos por épocas de 20 a 10. Si es lanzado con la menor cantidad de recursos, el tiempo del mismo es de 150 segundos.

El primer experimento, en el cual las peticiones solicitan baja demanda de recursos es similar al escenario anterior con la diferencia que se define un valor estático de 6 recursos. Este valor fue elegido ya que es menor a la mitad de la cantidad de recursos del sistema y no es múltiplo del mismo, generando que luego de planificar contenedores se generen espacios de reasignación de recursos. En la tabla 5.6 se visualizan los valores de las métricas para dicho experimento.

Se observa que los contenedores de las versiones elásticas tienen un TME menor (entre 12 y 14 seg.) excepto, al igual que los experimentos anteriores, la política *FCFS*. Con respecto al TMR, sucede el mismo comportamiento, generando un TMER y TT menor y un PRO mayor para la versión elástica

Política	TF	TME [s]	TMR [s]	TMER [s]	TT [s]	RA	PRO [C/Hr]
FCFS	Elástica	52	787	839	4321	0.63	106.6
	Original	50	726	776	4086	0.61	112.7
Always Attend	Elástica	58	288	346	3246	0.82	141.9
	Original	70	481	551	3666	0.84	125.6
Max Prop	Elástica	128	203	331	3211	0.9	143.5
	Original	142	234	376	3335	0.89	138.1

Tabla 5.6: Métricas de escenario de baja demanda de peticiones y baja cantidad de recursos por contenedor.

en las políticas *Always Attend* y *Max Prop* con un aumento de 15 C/Hr aproximadamente. Los recursos tienen una mayor distribución en estas políticas con incremento de hasta un 30 %, proporcionado por el alto nivel de reasignación. Si se comparan las versiones elásticas de las políticas, *Max Prop* penaliza la duración media de los contenedores (TME) con aumento de 70 segundos, provocado por la alta cantidad de contenedores ejecutando con bajo uso de recursos. Por otro lado, presenta un TMR menor que las demás políticas (entre 85 y 580 seg. menos). La política *Always Attend* presenta un menor TME pero sufre una penalización elevada en el TMR, generando un aumento de 15 segundos en el TMER y una disminución en el PRO de 2 C/Hr con respecto a *Max Prop*.

El experimento con alta demanda de recursos por petición solo varía el valor solicitado que sube de 6 a 12 cores. No se elige como valor la máxima cantidad de recursos del sistema para generar etapas de reasignación luego de lanzar contenedores. En la tabla 5.7 se visualizan las métricas para las diferentes políticas de asignación y versiones de TF.

Se observa que las versiones de TF elástico presentan mejor TME (entre 1 y 17 seg.), con un incremento mayor en las políticas *Always Attend* y *Max Prop*. Lo mismo sucede con el TMR con una diferencia aun mayor, llegando a 700 segundos menos en la política *Always Attend*. Tanto el TME como el TMR generan un menor tiempo del TMER y TT tanto para *Always Attend* y *Max Prop*. El RA es levemente superior en las versiones originales de TF pero al no aprovechar la resignación de recursos generan un mayor PRO en todas las políticas con versión elástica con un aumento entre 12 y 34 C/Hr.

Comparando las versiones elásticas de las políticas, se visualiza un comportamiento similar a las pruebas anteriores donde se obtiene mayor PRO en la política *Max Prop* generado por la significativa disminución del TMER en

Política	TF	TME [s]	TMR [s]	TMER [s]	TT [s]	RA	PRO [C/Hr]
FCFS	Elástica	31	1477	1508	5611	0.56	82.1
	Original	32	1583	1616	5771	0.57	79.8
Always Attend	Elástica	44	580	624	3792	0.73	121.5
	Original	69	1323	1392	5296	0.87	87
Max Prop	Elástica	128	208	336	3257	0.88	141.4
	Original	145	221	367	3297	0.89	139.7

Tabla 5.7: Métricas de escenario baja demanda de peticiones y alta cantidad de recursos por contenedor.

300 seg. aproximadamente comparado con la política *Always Attend*. Aunque esta ultima presenta un mejor TME, el TMR penaliza la duración total de los contenedores provocando una disminución de 20 C/HR del PRO. En cuanto a distribución de recursos, *Max Prop* es levemente mas elevada que *Always Attend* comparando las versiones elásticas y un 30 % mayor que *FCFS* dejando en evidencia que la resignación de recursos favorece el RA.

Capítulo 6

Conclusiones y trabajo futuro

En este capítulo se exponen las conclusiones de este trabajo y luego se plantean posibles líneas trabajos futuros.

6.1. Conclusiones

La aparición de arquitecturas de hardware con mayor cantidad de recursos disponibles on-chip permite explotar los algoritmos de cómputo intensivo, principalmente los encargados de procesar grandes volúmenes de datos. La elección del grado de paralelismo para la ejecución de estos algoritmos no es trivial y hace años que se busca la manera óptima, especialmente cuando se habla de algoritmos de ML. Los frameworks existentes para este tipo de cómputo requieren la asignación de recursos computacionales antes de su lanzamiento, lo que conlleva a que el usuario deba realizar un análisis en la búsqueda del valor mas adecuado.

Por otra parte, la gran capacidad de cómputo que presentan los sistemas HPC generalmente conlleva a asignar mayor cantidad de recursos a las aplicaciones, particularmente por la escalabilidad que pueden obtener, generando escenarios donde quedan recursos ociosos o una repartición con baja carga de trabajo entre ellos. Para mejorar la utilización, el rendimiento y la eficiencia energética de estos sistemas aparece el concepto de *multi-tenant* donde diferentes aplicaciones comparten los recursos del sistema.

Si en el sistema se encuentran conviviendo múltiples aplicaciones, es importante una asignación eficiente de los recursos para evitar sobrecarga de trabajo que derive en una degradación del rendimiento. Por lo tanto,

se requieren aplicaciones capaces de modificar sus recursos en tiempo de ejecución, ya sea para aumentar o disminuir el paralelismo según el estado del sistema. Al ser un requisito indispensable, los frameworks de ML deben añadir este comportamiento.

Además, la aparición de herramientas de virtualización ligera como Docker han permitido una mayor administración de los recursos debido a la posibilidad de modificar los recursos asignados a los contenedores activos y una mayor portabilidad entre sistemas.

Teniendo en cuenta los anteriores antecedentes, se detallan a continuación los aportes realizados con este trabajo:

- *Modificación del esquema de gestión de recursos dentro de la infraestructura del* framework *TF para permitir selección dinámica del paralelismo de las operaciones que conforman el modelo de ML.* En el capitulo 3 se describieron los conceptos principales del framework, así como la arquitectura interna para generar los modelos compuesta por cuatro partes principales (grafo computacional, modelo de ejecución, optimizadores y visualización de trazas).

  Luego de entender su funcionamiento, se presentó el modelo de ejecución para explicar el diseño de la solución que permite modificar los recursos computacionales asignados al framework TF para la ejecución del algoritmo de ML, destacando los cambios necesarios en los hiperparámetros del framework que definen el paralelismo denominados *INTER* e *INTRA*. Finalmente, se describen los cambios realizados en el framework, con énfasis en las modificaciones de los ficheros encargados del manejo de los hilos y la especificación de aspectos técnicos de la implementación de la librería Eigen usada por TF.

- *Diseño e implementación de un controlador interno de cada contenedor que permita gestionar los recursos computacionales asignados dinámicamente a TF , y de un mecanismo de comunicación entre el cliente del contenedor y TF.* En el capítulo 4 se describe la implementación de un cliente situado dentro de contenedores Docker encargado de las funciones principales, tales como el control de los recursos asignados al framework TF, lanzamiento del algoritmo de ML y, comunicación con el sistema para envío y recepción de eventos relacionados con los recursos asignados. Esta implementación permite introducir elasticidad dentro del contenedor, ya que cualquier actualización de recursos enviada por el sistema a través de comandos Docker permite informarlo a las aplicaciones internas, que para este trabajo es TF.

- *Diseño e implementación de un planificador de contenedores que ejecutan algoritmos de ML sobre TF elástico utilizando técnicas de orquestación que permitan administrar los recursos computacionales del sistema eficientemente.* El capitulo 4 plantea el problema presente en los planificadores de contenedores actuales, los cuales no controlan dinámicamente los recursos asignados a la aplicaciones internas. Esto puede generar sobrecarga de trabajo en los recursos del contenedor o bajo aprovechamiento de los mismos.

  Por lo mencionado anteriormente, se define el diseño de un planificador que contemple esta situación. Se modela los actores intervenientes, explicando las principales tareas que debe cumplir cada uno. Luego, se describen los pasos necesarios para la atención y liberación de contenedores dentro del planificador. También se menciona la necesidad de incorporar un mecanismo de control de fallos que pueden producirse durante la ejecución de un contenedor elástico.

  En la sección 4.2 se comentaron las políticas de asignación y reasignación de recursos que puede aplicar el planificador hasta el momento de la escritura de esta tesis. Todas las políticas de asignación se basan en FCFS. La primera de ellas, es aplicarla estrictamente donde se lanza el contenedor si hay recursos disponibles de acuerdo a la cantidad solicitada por el mismo. La primer variante denominada *Always Attend* permite el lanzamiento aunque los recursos disponibles sean menores a los solicitados. La variante *Max Prop* calcula la relación entre la cantidad de recursos solicitados por todas las peticiones y los disponibles en el sistema para lograr una distribución equitativa, lo cual puede provocar una menor asignación de recursos que los solicitados por cada contenedor. Por último, se comentaron las estrategias de resignación de recursos con imágenes y ejemplos.

  Se definió una sección en el capitulo que describe la implementación de cada componente principal del planificador. Se comenzó con el cliente del contenedor comentado en el aporte anterior. Luego el servidor con las funciones de generación y atención de peticiones, almacenamiento de la información de los contenedores, manejo de los recursos, la comunicación con el cliente de cada contenedor y el funcionamiento de las políticas de asignación/reasignación. Cada una de estas partes del servidor ilustra con códigos en Python su implementación para más detalle. El siguiente componente descrito es el visor de trazas que permite generar diagramas de Gantt para visualizar los tiempos de los contenedores y del algoritmo de ML descompuesto en épocas para

mayor detalle de su comportamiento.

La última sección del capitulo, describió los requerimientos del sistema necesarios para lanzar el planificador e instrucciones para el lanzamiento del mismo. Además, se incluyó una figura para demostrar cómo es la interacción de los hilos del planificador y del cliente del contenedor cuando se lanza una petición generada por un usuario.

- *Evaluación de la elasticidad implementada en el framework TF y su funcionamiento en contenedores.* Se comprobó que la elasticidad de los dos tipos de paralelismo (INTER e INTRA) de TF funcionan correctamente mediante la realización de pruebas de variación de estos hiperparámetros en tiempo de ejecución. El análisis se realizó con las trazas de hilos obtenidas de diferentes etapas de la ejecución del algoritmo, donde se logra visualizar que la cantidad de hilos intervenientes antes y después del cambio de paralelismo es correcta.

  Un medida importante del beneficio de la elasticidad en TF es el overhead introducido por su implementación. Este coste fue evaluado mediante la comparación de dos pruebas lanzadas, una TF original y otra con la versión elástica. Los valores devueltos arrojan un incremento del tiempo de total de procesamiento de cada batch en 1.5 %, el cual no aporta pérdidas de rendimiento cuando se aplica la elasticidad.

  Luego se comprobó mediante lanzamiento de experimentos la falta de interacción entre un contenedor y sus aplicaciones internas. Se realizó la ejecución de contenedores con TF original. Cuando se aumentaban los recursos, TF no se beneficiaba de dicha cambio, ya que el rendimiento del algoritmo no incrementaba. Si se disminuían los recursos, provocaba una sobrecarga de trabajo generando una degradación del rendimiento del contenedor. Por lo tanto, con estas pruebas se evidenció la necesidad de aplicar elasticidad de recursos en los contenedores.

- *Comparación de los tiempos del planificador utilizando versión original y elástica de TF. Evaluación de las diferentes políticas de planificación implementadas.* Para evaluar el comportamiento del planificador, se definieron un conjunto de métricas. Se plantearon dos escenarios de llegada de peticiones y se ejecutaron las pruebas para obtener los valores. En primer lugar, se observó que la versión elástica de TF permite aumentar el rendimiento de los contenedores que ejecutan algoritmos de ML. En los experimentos donde hay alta distribución y reasignación de recursos, la versión elástica de TF supera a la original en tiempo medio de ejecución de cada contenedor y productividad.

Comparando las políticas implementadas, se observó que *Max Prop* tiende a mejorar la distribución de recursos entre los contenedores activos, provocando una mayor productividad que las políticas *FCFS* y *Always Attend*. Por otro lado, es notorio el aumento del tiempo medio de ejecución cuando se aplica *Max Prop*, pero se ve beneficiado por la disminución del tiempo de respuesta del planificador, ya que siempre que sea posible intentará distribuir los recursos liberados entre las nuevas peticiones de lanzamiento.

Con todos los aportes mencionados anteriormente, se considera que se han alcanzado los objetivos propuestos para este trabajo. Luego del análisis de resultados se concluyó que la elasticidad de las aplicaciones y de las herramientas de *contenedorización* son fundamentales para lograr una óptima administración de los recursos de los sistemas actuales, donde cada vez presentan mayores unidades de cómputo y el costo de no gestionarlos correctamente afecta directamente al rendimiento y eficiencia energética, ya sea por sobre o infrautilización de los recursos.

6.2. Trabajos Futuros

Las líneas de trabajo que surgen de esta investigación son:

- Incorporar en el planificador soporte de sistemas heterogéneos. Integrar la administración de recursos de aceleradores tales como GPU y placas de borde es un aporte significativo por el alto rendimiento y ancho de banda que soportan superando en dos veces su valor en cada nueva generación. Por lo tanto, con tales tendencias de escalamiento en GPU, la ejecución de una sola aplicación o modelo de ML no explota la utilización completa de los recursos de la placa.
- Evaluar la eficiencia energética del planificador de aplicaciones elásticas y compararlo con otros planificadores del mercado tales como Kubernetes, Maraton, Cloudify, entre otros.
- Analizar el uso de funciones desarrolladas por proveedores de GPU, como Multi-Stream [40], Multi-Process Service (MPS) [41], Multi-Instance GPU (MIG) [42] y GPUs virtuales(vCS) [43] para soporte de planificación en tiempo de ejecución y administración de recursos.
- Explorar la administración de recursos en otros frameworks de ML tales como PyTorch y Caffe, con el fin de extender la elasticidad a mas aplicaciones y realizar estudios comparativos entre ellos.

Apéndice A

Instrucciones de uso del planificador

A.1. Requerimientos de sistema

Para la utilización del planificador, en primer lugar, es necesario contar con un sistema operativo Linux. Luego, a traves de un gestor de paquetes (por ejemplo apt) se debe instalar python y el gestor de contenedores Docker.

Una vez instalado python, se requieren una serie de librerías que pueden instalarse utilizando el gestor de paquetes pip:

- numpy: es el paquete fundamental para la computación científica en Python. Es una biblioteca que proporciona un objeto de matriz multidimensional, varios objetos derivados (como matrices y matrices enmascaradas) y una variedad de rutinas para operaciones rápidas en matrices, que incluyen manipulación matemática, lógica, de formas, clasificación, selección, E/S , transformadas discretas de Fourier, álgebra lineal básica, operaciones estadísticas básicas, simulación aleatoria y mucho más [44].

- psutil: es una biblioteca multiplataforma para recuperar información sobre procesos en ejecución y utilización del sistema (CPU, memoria, discos, red, sensores) en python. Es útil principalmente para la supervisión del sistema, la creación de perfiles y la limitación de los recursos del proceso y la gestión de los procesos en ejecución [45].

- json: biblioteca que permite serializar objetos python en json legibles (dictados o cadenas) y deserializarlos nuevamente [46].

- plotly: biblioteca gráfica interactiva y de código abierto. Su funcionamiento fue explicado en la sección 4.3.3 [47].

Con todos estos requerimientos cumplidos, se puede realizar el lanzamiento del planificador.

A.2. Lanzamiento del servidor

Para ejecutar el planificador en el sistema, luego de cumplir con los requerimientos mencionados anteriormente, se debe descargar la carpeta que contiene todos los códigos desde el siguiente repositorio:

`https://github.com/\leanlibutti/Tensorflow-Container-Scheduler/tree/master/scheduler`

En primer lugar, el usuario debe copiar su algoritmo desarrollado en TF en la carpeta *models*. También, es necesario que almacene el dataset utilizado por el algoritmo dentro de la carpeta del planificador.

El próximo paso es la modificación del archivo dockerfile para especificar cuál es el algoritmo que se desea ejecutar. Para esto, se accede al archivo *tensorflow-program.dockerfile* y se modifica la última línea indicando el nombre del algoritmo TF. En el código siguiente se visualiza la definición del algoritmo resnet. Esta línea realiza la ejecución del cliente del planificador dentro del contenedor donde recibe por parámetro cuál es el algoritmo que se ejecuta.

```
ENTRYPOINT python3 /home/Scheduler/Client/client.py
    keras_example_resnet
```

Una vez configurado el dockerfile, se debe crear la imagen. Para ello, a través de una terminal de comandos del sistema, se utiliza la directiva *build* de Docker como se muestra en la siguiente línea:

```
docker build -t image_name .
```

Es necesario situarse en la carpeta principal del planificador para realizar la ejecución de este comando.

Antes de lanzar el servidor, se deben definir los contenedores que se ejecutarán en un archivo de texto ubicado en la carpeta **Scheduler-Host** con el nombre **request_file.txt**. Las peticiones almacenadas en este archivo presenta un formato específico. Cada línea de texto contiene el tipo de petición (ejecución o actualización), número de peticion/contenedor, imagen de docker, cantidad de paralelismo y espera de tiempo de atención de la próxima petición. Por ejemplo, si se desea ejecutar un contenedor de la

imagen *tf_scheduler* con 4 hilos INTRA y 1 hilo INTER se debe especificar la siguiente línea:

```
execution,0,tf_scheduler,4,1,0
```

Para lanzar el servidor se ejecuta el siguiente comando:

```
python3.8 run_scheduler.py param1 param2
```

El primer parámetro indica la cantidad de veces que se desea ejecutar el planificador y el segundo la cantidad de contenedores que se van a crear en cada ejecución del mismo.

Al finalizar cada ejecución del planificador, se genera una carpeta en */Data/log* con el nombre correspondiente a la cantidad de contenedores que se lanzaron. Por ejemplo, si se lanza un solo contenedor la carpeta se llama "1Contenedores". Dentro de esa carpeta se observan diferentes archivos que permiten calcular las métricas definidas. Además, se almacenan los timeline del algoritmo TF de cada contenedor en la carpeta *Outputs*.

En la figura A.1 se visualiza cómo intervienen los diferentes actores del planificador en la atención de una petición del usuario. Cuando el usuario genera una petición es atendida por el hilo encargado de la recepción el cual evalúa si es valida y notifica al hilo que procesa si se cumplen los requerimientos para lanzar la petición. Si se cumple, se crea un hilo controlador del contenedor encargado del envió y recepción de eventos. Al comienzo informa al contenedor sobre los recursos asignados para la ejecución del algoritmo TF. Luego, si en algún momento el hilo de atención del planificador debe actualizar sus recursos, notificará al hilo que controla el contenedor y éste se encargará de la comunicación del evento. Dentro del contenedor se encuentra el hilo que controla y notifica eventos al algoritmo. Cualquier comunicación recibida por el planificador es atendida por un hilo creado específicamente para dicha tarea. En el caso de actualización de recursos, el hilo de comunicación del contenedor recibe dicho cambio el cual es notificado al hilo controlador de TF que envía la señal de cambio al algoritmo.

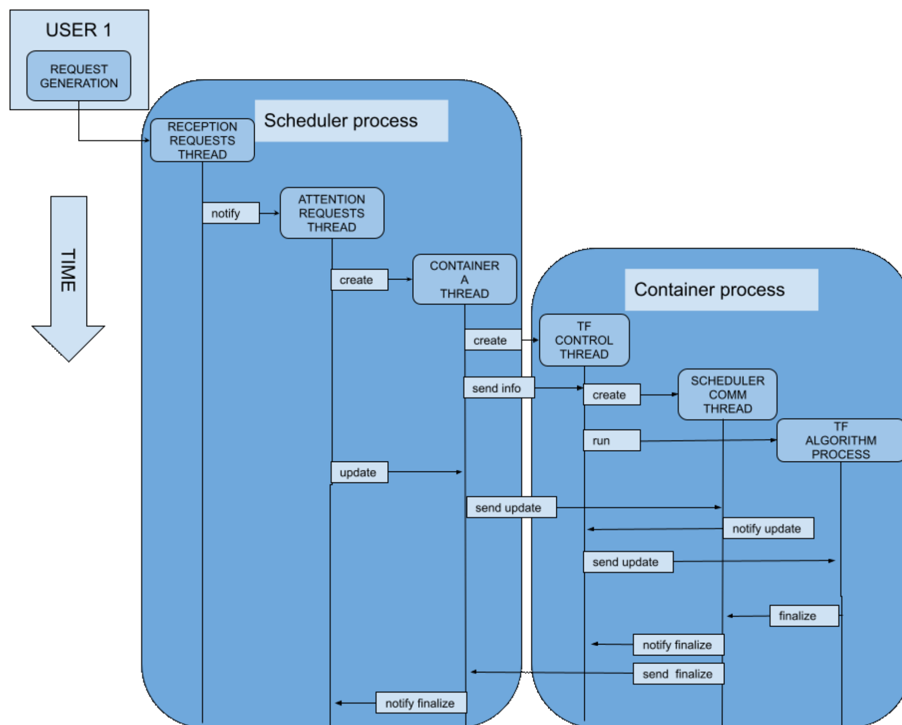


Figura A.1: Interacción de hilos del planificador y contenedor lanzado.

Apéndice B

Tablas de métricas de los experimentos

En este apéndice se visualizan las tablas de las métricas de los cuatro experimentos realizados en el planificador variando la cantidad de contenedores lanzados desde 8 a 128.

Recursos	T llegada [s]	Contenedores	Politica	TF	TME [s]	TMR [s]	TMER [s]	TT [s]	RA	PRO [C/Hr]
8	Uniforme (20.5)	8	FCFS	maleable	90.125	90	180.125	445	80	64.71910112
				original	91.125	91.75	182.875	433	80	66.51270208
			always attend	maleable	110.875	101.25	212.125	563	84.2105263	51.15452931
				original	93.625	93.75	187.375	445	80	64.71910112
			max prop	maleable	135.5	67.25	202.75	407	88.3928571	70.76167076
				original	244.5	64.25	308.75	630	90.4761905	45.71428571
		16	FCFS	maleable	88.8125	221.1875	310	827	83.9285714	69.64933495
				original	92.8125	239	331.8125	882	86.2068966	65.30612245
			always attend	maleable	87.25	220.8125	308.0625	810	83.9285714	71.11111111
				original	92.25	238.125	330.375	879	84.4827586	65.52901024
			max prop	maleable	271.375	152.75	424.125	873	87.5	65.97938144
				original	320	238.3125	558.3125	1058	91.4285714	54.44234405
		32	FCFS	maleable	105.59375	569.1875	674.78125	2102	83.8235294	54.80494767
				original	92.84375	503.15625	596	1733	86.6071429	66.47432198
			always attend	maleable	93.78125	568.65625	662.4375	1788	89.6551724	64.4295302
				original	92.625	499.8125	592.4375	1719	89.2857143	67.01570681
			max prop	maleable	235.03	424.03	659.06	1725	92.6535088	66.7826087
				original	307.28125	507.84375	815.125	1876	93.2459677	61.40724947
		64	FCFS	maleable	102.140625	1227.32813	1329.46875	3875	90	59.45806452
				original	92.171875	1004.39063	1096.5625	3386	89.0909091	68.04489073
			always attend	maleable	103.453125	1331.5625	1435.01563	3872	90.4	59.50413223
				original	92.328125	1008.6875	1101.01563	3385	89.5454545	68.06499261
			max prop	maleable	368.42	932.37	1300.79	3744	93.75	61.53846154
				original	363.015625	883.671875	1246.6875	3470	93.6383929	66.39769452
		128	FCFS	maleable	92.40625	2041.9375	2134.34375	6964	88.4955752	66.16886847
				original	92.0390625	2084.84375	2176.88281	6745	88.5844749	68.31727205
			always attend	maleable	95.515625	2280.71094	2376.22656	7129	90.04329	64.63739655
				original	92.4453125	2110.80469	2203.25	6801	90.4545455	67.75474195
			max prop	maleable	339.15625	2185.03125	2524.1875	7337	93.0907173	62.80496116
				original	378.203125	1985.5	2363.70313	6882	95.3195067	66.95727986

Tabla B.1: Tabla completa de las métricas para escenario de alta demanda de peticiones y baja cantidad de recursos por contenedor.

Recursos	T llegada [s]	Contenedores	Politica	TF	TME [s]	TMR [s]	TMER [s]	TT [s]	RA	PRO [C/Hr]
16	Uniforme (20.5)	8	FCFS	maleable	60.625	189.125	249.75	597	80	48.24120603
				original	56.375	176.375	232.75	583	80	49.39965695
			always attend	maleable	60.375	189	249.375	584	84.2105263	49.31506849
				original	57.125	180.125	237.25	576	80	50
			max prop	maleable	135.375	80.125	215.5	514	88.3928571	56.0311284
				original	142.75	97.375	240.125	481	90.4761905	59.87525988
		16	FCFS	maleable	60.5	388.5	449	1190	83.9285714	48.40336134
				original	56.6875	362.125	418.8125	1136	86.2068966	50.70422535
			always attend	maleable	64.75	427.3125	492.0625	1287	83.9285714	44.75524476
				original	56.125	357.0625	413.1875	1090	84.4827586	52.8440367
			max prop	maleable	175.4375	169.5	344.9375	864	87.5	66.66666667
				original	269	164.1875	433.1875	893	91.4285714	64.50167973
		32	FCFS	maleable	64.4375	914.3125	978.75	2600	83.8235294	44.30769231
				original	56.96875	788.1875	845.15625	2234	86.6071429	51.56669651
			always attend	maleable	60.8125	845.625	906.4375	2380	89.6551724	48.40336134
				original	56.375	775.90625	832.28125	2226	89.2857143	51.75202156
			max prop	maleable	317.84375	530.28125	848.125	1918	92.6535088	60.06256517
				original	331.15625	427.9375	759.09375	1765	93.2459677	65.26912181
		64	FCFS	maleable	61.375	1657.6875	1719.0625	4777	90	48.23110739
				original	56.59375	1522.75	1579.34375	4432	89.0909091	51.98555957
			always attend	maleable	61.453125	1706.46875	1767.92188	4811	90.4	47.89025151
				original	56.625	1520.6875	1577.3125	4474	89.5454545	51.49754135
			max prop	maleable	371.3125	936.78125	1308.09375	3672	93.75	62.74509804
				original	359.796875	829.59375	1189.39063	3481	93.6383929	66.18787705
		128	FCFS	maleable	62.96875	3567.30469	3630.27344	9924	88.4955752	46.43288996
				original	56.65625	3095.98438	3152.64063	8859	88.5844749	52.0149001
			always attend	maleable	62.71875	3523.05469	3585.77344	9829	90.04329	46.88167667
				original	56.421875	3080.125	3136.54688	8835	90.4545455	52.15619694
			max prop	maleable	395.890625	2093.15625	2489.04688	7329	93.0907173	62.87351617
				original	378.25	1945.15625	2323.40625	6947	95.3195067	66.33079027

Tabla B.2: Tabla completa de las métricas para escenario de alta demanda de peticiones y alta cantidad de recursos por contenedor.

Recursos	T llegada [s]	Contenedores	Politica	TF	TME [s]	TMR [s]	TMER [s]	TT [s]	RA	PRO [C/Hr]
6	Uniforme (20.5)	8	FCFS	maleable	49.625	38.375	88	263	60	109.505703
				original	49.875	40.875	90.75	301	54.5454545	95.6810631
			Always Attend	maleable	63	27.75	90.75	288	67.5	100
				original	64.125	16.75	80.875	267	65	107.865169
			Max Prop	maleable	49.375	29.75	79.125	247	65	116.59919
				original	47.875	33.625	81.5	281	68.125	102.491103
		16	FCFS	maleable	50	72.625	122.625	516	47.3684211	111.627907
				original	50.125	77.125	127.25	531	49.3421053	108.474576
			Always Attend	maleable	54.25	15.5625	69.8125	409	70.8333333	140.831296
				original	68.0625	41.875	109.9375	499	72.9166667	115.430862
			Max Prop	maleable	60.1875	38	98.1875	459	84.375	125.490196
				original	58.875	37.75	96.625	444	82.8125	129.72973
		32	FCFS	maleable	50.78125	196.125	246.90625	1067	57.2916667	107.966261
				original	50.375	201.53125	251.90625	1065	60	108.169014
			Always Attend	maleable	56.6875	65.75	122.4375	811	74.1071429	142.046856
				original	69.125	128.625	197.75	920	80.859375	125.217391
			Max Prop	maleable	86.53125	66.03125	152.5625	792	72.7678571	145.454545
				original	97.90625	70.5625	168.46875	848	84.4827586	135.849057
		64	FCFS	maleable	50.84375	364.84375	415.6875	2079	56.5217391	110.822511
				original	50.703125	376.625	427.328125	2081	63.4191176	110.716002
			Always Attend	maleable	57.25	114.34375	171.59375	1619	75.2314815	142.310068
				original	70.28125	250.15625	320.4375	1840	82.1721311	125.217391
			Max Prop	maleable	107.546875	222.65625	330.203125	1847	83.7090164	124.742826
				original	133.4375	142.5	275.9375	1694	86.9419643	136.009445
		128	FCFS	maleable	52.390625	787.4375	839.828125	4321	63.2978723	106.641981
				original	50.3515625	726.09375	776.445313	4086	61.1842105	112.77533
			Always Attend	maleable	58.890625	288.421875	347.3125	3246	82.5934579	141.959335
				original	70.703125	481.09375	551.796875	3666	84.1386555	125.695581
			Max Prop	maleable	128.578125	203.710938	332.289063	3211	90.4166667	143.506696
				original	142.046875	234.890625	376.9375	3335	89.3348624	138.170915

Tabla B.3: Tabla completa de las métricas para escenario de baja demanda de peticiones y baja cantidad de recursos por contenedor.

Recursos	T llegada [s]	Contenedores	Politica	TF	TME [s]	TMR [s]	TMER [s]	TT [s]	RA	PRO [C/Hr]
12	Uniforme (20.5)	8	FCFS	maleable	31.5	66.875	98.375	48.2142857	77.6280323	371
				original	32.5	72.375	104.875	42.8571429	78.9041096	365
			always attend	maleable	39.5	6	45.5	45	108.270677	266
				original	64.75	35.75	100.5	58.9285714	79.338843	363
			max prop	maleable	39	28.5	67.5	59.0909091	101.408451	284
				original	38.875	25.625	64.5	54.5454545	98.9690722	291
		16	FCFS	maleable	31.375	178.6875	210.0625	54	77.5235532	743
				original	32.6875	189	221.6875	57	79.0123457	729
			always attend	maleable	40.9375	50.875	91.8125	63.2352941	114.97006	501
				original	67.0625	137.375	204.4375	69.7916667	83.4782609	690
			max prop	maleable	120.25	64.625	184.875	85	98.9690722	582
				original	62.125	41.625	103.75	79.296875	126.315789	456
		32	FCFS	maleable	31.4375	404.25	435.6875	54.6875	80.0555942	1439
				original	32.46875	427.03125	459.5	57.8125	80.5594406	1430
			always attend	maleable	41.625	142.75	184.375	73.3870968	129.583802	889
				original	68.375	333.09375	401.46875	74.4444444	85.3333333	1350
			max prop	maleable	118	92.03125	210.03125	79.375	128.85906	894
				original	116.8125	76.4375	193.25	86.2068966	134.894614	854
		64	FCFS	maleable	31.40625	733.46875	764.875	56.4516129	81.644224	2822
				original	32.390625	779.15625	811.546875	56.8421053	78.9852588	2917
			always attend	maleable	45.453125	287.171875	332.625	73.1060606	116.129032	1984
				original	68.40625	620.53125	688.9375	77.8735632	85.8100559	2685
			max prop	maleable	86.40625	64.140625	150.546875	84.1346154	145.638432	1582
				original	127.125	103.984375	231.109375	89.7727273	138.544799	1663
		128	FCFS	maleable	31.3359375	1477.22656	1508.5625	56.147541	82.1243985	5611
				original	32.5546875	1583.44531	1616	57.4468085	79.8475134	5771
			always attend	maleable	44.28125	580.765625	625.046875	73.8	121.518987	3792
				original	69.1953125	1323.5	1392.69531	79.3352601	87.0090634	5296
			max prop	maleable	128.632813	208.335938	336.96875	88.2009346	141.479889	3257
				original	145.289063	221.890625	367.179688	89.0186916	139.763421	3297

Tabla B.4: Tabla completa de las métricas para escenario de baja demanda de peticiones y alta cantidad de recursos por contenedor.

Bibliografía

[1] X. Zheng, H. Chen, and T. Xu, "Deep learning for chinese word segmentation and pos tagging," in *Proceedings of the 2013 conference on empirical methods in natural language processing*, pp. 647–657, 2013.

[2] H. Seo, M. Badiei Khuzani, V. Vasudevan, C. Huang, H. Ren, R. Xiao, X. Jia, and L. Xing, "Machine learning techniques for biomedical image segmentation: an overview of technical aspects and introduction to state-of-art applications," *Medical physics*, vol. 47, no. 5, pp. e148–e167, 2020.

[3] L. Verde, G. De Pietro, and G. Sannino, "Voice disorder identification by using machine learning techniques," *IEEE access*, vol. 6, pp. 16246–16255, 2018.

[4] M. Wu and L. Chen, "Image recognition based on deep learning," in *2015 Chinese Automation Congress (CAC)*, pp. 542–546, IEEE, 2015.

[5] L. A. Libutti, F. D. Igual, L. Piñuel, L. De Giusti, and M. Naiouf, "Towards a malleable tensorflow implementation," in *Conference on Cloud Computing, Big Data & Emerging Topics*, pp. 30–40, Springer, 2020.

[6] L. Yang and A. Shami, "On hyperparameter optimization of machine learning algorithms: Theory and practice," *Neurocomputing*, vol. 415, pp. 295–316, 2020.

[7] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pp. 265–283, 2016.

[8] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*, pp. 675–678, 2014.

[9] A. Gulli and S. Pal, *Deep learning with Keras.* Packt Publishing Ltd, 2017.

[10] N. Ketkar, "Introduction to pytorch," in *Deep learning with python*, pp. 195–208, Springer, 2017.

[11] J. Mace, P. Bodik, R. Fonseca, and M. Musuvathi, "Retro: Targeted resource management in multi-tenant distributed systems," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pp. 589–603, 2015.

[12] Z. Zhong, M. Xu, M. A. Rodriguez, C. Xu, and R. Buyya, "Machine learning-based orchestration of containers: A taxonomy and future directions," *arXiv preprint arXiv:2106.12739*, 2021.

[13] "Docker." `https://www.docker.com/`. (Accessed on 07/07/2022).

[14] "Docker swarm." `https://docs.docker.com/engine/swarm/`. Accedido en Enero de 2022.

[15] "Kubernetes." `https://kubernetes.io/es/`. Accedido en Enero de 2022.

[16] "Marathon." `https://mesosphere.github.io/marathon/`. Accedido en Enero de 2022.

[17] N. Ensmenger, "The computer boys take over: Computers, programmers, and the politics of technical expertise (history of computing)," 2010.

[18] D. M. Jacobsen and R. S. Canon, "Contain this, unleashing docker for hpc," *Proceedings of the Cray User Group*, pp. 33–49, 2015.

[19] R. Morabito, J. Kjällman, and M. Komu, "Hypervisors vs. lightweight virtualization: a performance comparison," in *2015 IEEE International Conference on Cloud Engineering*, pp. 386–393, IEEE, 2015.

[20] L. Benedicic, F. A. Cruz, A. Madonna, and K. Mariotti, "Portable, high-performance containers for hpc," *arXiv preprint arXiv:1704.03383*, 2017.

[21] G. M. Kurtzer, V. Sochat, and M. W. Bauer, "Singularity: Scientific containers for mobility of compute," *PloS one*, vol. 12, no. 5, p. e0177459, 2017.

[22] D. Jacobsen and R. Canon, "Shifter: Containers for hpc," in *Cray Users Group Conference (CUG'16)*, 2016.

[23] E. Casalicchio, "Container orchestration: A survey," *Systems Modeling: Methodologies and Tools*, pp. 221–235, 2019.

[24] "Lxc: Linux containers." `https://linuxcontainers.org/lxc/`. Accedido en Enero de 2022.

[25] S. Abraham, A. K. Paul, R. I. S. Khan, and A. R. Butt, "On the use of containers in high performance computing environments," in *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, pp. 284–293, IEEE, 2020.

[26] S. Horovitz and Y. Arian, "Efficient cloud auto-scaling with sla objective using q-learning," in *2018 IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud)*, pp. 85–92, IEEE, 2018.

[27] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Autonomic vertical elasticity of docker containers with elasticdocker," in *2017 IEEE 10th international conference on cloud computing (CLOUD)*, pp. 472–479, IEEE, 2017.

[28] S. Shekhar, H. Abdel-Aziz, A. Bhattacharjee, A. Gokhale, and X. Koutsoukos, "Performance interference-aware vertical elasticity for cloud-hosted latency-sensitive applications," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pp. 82–89, IEEE, 2018.

[29] F. Rossi, M. Nardelli, and V. Cardellini, "Horizontal and vertical scaling of container-based applications using reinforcement learning," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pp. 329–338, IEEE, 2019.

[30] E. Preeth, F. J. P. Mulerickal, B. Paul, and Y. Sastri, "Evaluation of docker containers based on hardware utilization," in *2015 international conference on control communication & computing India (ICCC)*, pp. 697–700, IEEE, 2015.

[31] "Docker: Resources contraints." https://docs.docker.com/config/containers/resource_constraints. Accedido en Enero de 2022.

[32] H. T. Ciptaningtyas, B. J. Santoso, and M. F. Razi, "Resource elasticity controller for docker-based web applications," in *2017 11th International Conference on Information & Communication Technology and System (ICTS)*, pp. 193–196, IEEE, 2017.

[33] "Eigen." https://eigen.tuxfamily.org/index.php?title=Main_Page. (Accessed on 04/06/2022).

[34] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff, "A set of level 3 basic linear algebra subprograms," *ACM Transactions on Mathematical Software (TOMS)*, vol. 16, no. 1, pp. 1–17, 1990.

[35] M. Fatica, "Cuda toolkit and libraries," in *2008 IEEE hot chips 20 symposium (HCS)*, pp. 1–22, IEEE, 2008.

[36] A. Krizhevsky, "Google code archive - long-term storage for google code project hosting.."

[37] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.

[38] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald, *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[39] "Python socket." https://docs.python.org/3/library/socket.html. Accessed on 26/05/2022.

[40] "Nvidia multi streams." https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency/. (Accessed on 21/07/2022).

[41] "Nvidia multi process service (mps)." https://docs.nvidia.com/deploy/mps/index.html. (Accessed on 21/07/2022).

[42] "Nvidia multi instances (mig)." https://docs.nvidia.com/datacenter/tesla/mig-user-guide/. (Accessed on 21/07/2022).

[43] "Nvidia virtual compute server (vcs)." https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/solutions/resources/documents1/. (Accessed on 21/07/2022).

[44] "Numpy library." https://numpy.org. Accessed on 1/06/2022.

[45] "Psutil library." https://pypi.org/project/psutil/. Accessed on 1/06/2022.

[46] "Json library." https://jsons.readthedocs.io/en/latest/. Accessed on 1/06/2022.

[47] "Plotly library." https://plotly.com/. (Accessed on 1/06/2022).