PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

Escuela de Posgrado

Desarrollo de un algoritmo de *Instance Placement* en nubes privadas que soporte cargas de Alta Performance

Tesis para obtener el grado académico de Maestro en Informática con mención en Ciencias de la Computación que presenta:

Rubén Francisco Córdova Alvarado

Asesores:

Dr. César Augusto Santiváñez Guarniz

Dr. César Armando Beltrán Castañón

Lima, 2023

Informe de Similitud

Yo, ***César Augusto SANTIVÁÑEZ GUARNIZ***, docente de la Escuela de Posgrado de la Pontificia Universidad Católica del Perú, asesor de la tesis titulada " Desarrollo de un algoritmo de Instance Placement en nubes privadas que soporte cargas de Alta Performance" de el autor **Rubén Francisco CÓRDOVA ALVARADO**, dejo constancia de lo siguiente:

- El mencionado documento tiene un índice de puntuación de similitud de 04%. Así lo consigna el reporte de similitud emitido por el software *Turnitin* el 3/07/2024.
- He revisado con detalle dicho reporte y la tesis, y no se advierte indicios de plagio.
- Las citas a otros autores y sus respectivas referencias cumplen con las pautas académicas.

Santivañez Guarniz, Cesar Augusto

Lugar y fecha: San Miguel, 3 de Julio de 2024

Apellidos y nombres del asesor / de la asesora: ***SANTIVÁÑEZ GUARNIZ, César Augusto***	
DNI: 09339312	Firma 
ORCID: 0000-0002-0050-3631	

Dedicatoria

A mi familia, en particular mi mamá Elva y mi hermana Elizabeth, por ser quienes me motivaron a realizar la maestría. Gracias por enseñarme la importancia de perseverar a pesar de los momentos complicados que se presenten.

A mi papá José, que desde el cielo me acompañó en este camino con altos y bajos. Sé lo orgulloso que estarías de mí.

A mi esposa Heldy por su amor, comprensión y apoyo incondicional durante todo el tiempo de desarrollo de la tesis. Gracias por ser el soporte emocional en los momentos complicados que se presentaron.

Resumen

El aumento de la capacidad computacional ha permitido el uso cada vez mayor de métodos computacionales para resolver problemas complejos de diferentes áreas, logrando tal incremento en la eficiencia y productividad que se dice que hemos empezado una nueva revolución industrial (la era del conocimiento). En esta nueva era, el uso de aplicaciones de alta, *High-Performance Computing* en inglés (HPC), es cada vez más común. Una forma de utilizar de manera eficiente los recursos computacionales es desplegar estas aplicaciones sobre recursos compartidos (paradigma de computo en la nube, sea esta pública o privada) en lugar de asignarlos a servidores de manera exclusiva, lo que puede resultar en tiempos muertos en el uso de alguno o todos los recursos. El problema de decidir la mejor forma de compartir recursos asignados a servidores ya sea como máquinas virtuales (VMs), contenedores, o en modo dedicado (*bare metal*) es llamado el problema de *Instance Placement*, y es fundamental para la performance de una plataforma de computo en la nube. El subproblema que se presenta cuando ya se decidió una asignación via VMs es el de *VM Placement*.

El problema de *Instance Placement* es actualmente un problema abierto debido a que la solución *online* requiere el conocimiento no sólo de las demandas actuales y sus parámetros, sino también de las demandas futuras. Como un primer acercamiento a una solución, esta tesis busca diseñar e implementar un algoritmo de *Offline Instance Placement* donde el conjunto de demandas, su inicio y duración, así como sus estadísticas de uso son conocidas. El algoritmo busca asignar –de la mejor manera posible– los recursos de cómputo a instancias en una nube privada, considerando el tipo de carga a la que estas pertenecen y su nivel de servicio.

Debido a que OpenStack es una de las soluciones más empleadas para nubes privadas, se toma como referencia el *scheduler* de OpenStack para comparar la utilidad de el algoritmo propuesto. Luego de realizar las pruebas, se obtuvo que el *scheduler* propuesto presenta una mayor utilidad que el *scheduler* de OpenStack para distintos tipos de cargas.

Palabras clave: *Virtual Machine Placement*, *High Performance*, *Cloud Computing*, Optimización Combinatoria.

Índice

Pág

Índice de Tablas

Pág

Lista de Figuras

Pág

PRIMERA PARTE: MARCO DE LA INVESTIGACIÓN

CAPÍTULO I. GENERALIDADES

1.1. Contexto de las Aplicaciones Actuales

En los últimos años, el modelo de cómputo llamado Computación en la Nube (*Cloud Computing*) ha sido adoptado en diferentes áreas, cada vez con mayor acogida. Entre diversos motivos, esto se debe a que les permite a las empresas tener un menor costo de despliegue de sus servicios (modelo de pago *Pay-as-you-go*), así como brindar la posibilidad de escalar rápidamente en la cantidad de recursos que requieran las aplicaciones, acceder desde múltiples ubicaciones que tengan acceso a Internet y ofrecer soluciones de recuperación de datos ante desastre o pérdidas de ellos. En contraste, en el pasado las empresas debían invertir en la adquisición y mantenimiento de servidores; en algunos casos se encontraban en situaciones donde la cantidad de recursos utilizados en los servidores era menor a los disponibles, generando un desperdicio de energía por tener servidores subutilizados. Adicionalmente, las empresas requerían de un equipo técnico capacitado para la administración de los servidores, capaz de reconfigurar la infraestructura ante nuevos requerimientos y de responder ante fallas en el sistema los tiempos establecidos por el Acuerdo de Nivel de Servicio (SLA).

Tradicionalmente, los entornos de *Cloud Computing* han sido empleados para desplegar aplicaciones web y de bases de datos (ámbito empresarial), así como soluciones de almacenamiento en línea (ámbito personal). Si bien en un inicio bastaba con poder acceder a los recursos en la nube, a medida que se intensificó su uso aparecieron otros requerimientos, tales como acceder de manera más rápida a ellos. Con la aparición de nuevas aplicaciones, como los servicios de *Streaming* en vivo, hicieron que el modelo de *Cloud Computing* inicial fuera evolucionando y ofreciendo un procesamiento particular para ciertos tipos de aplicaciones (por ejemplo, transcodificación de video). Un caso de ello fue la aparición del modelo de *Edge Computing*, donde los recursos de cómputo se movieron más cerca de los usuarios para poder realizar el procesamiento de datos más cerca de donde son consumidos.

Por otro lado, aplicaciones científicas y de alta performance, las cuales usualmente se ejecutaban en *clusters* de servidores dedicados para obtener una mejor performance, también han ido migrando hacia entornos de computación en la nube pública y privada, para aprovechar las ventajas que ofrece. Entre ellas podemos encontrar al Cómputo de Alta Performance (HPC), Big Data, Inteligencia Artificial y a las redes de operadores móviles.

Existen diversos motivos por los que aplicaciones de alta performance –las cuales inicialmente requerían un uso exclusivo de equipos de cómputo y red– en la actualidad son desplegadas sobre entornos de *Cloud Computing*. En primer lugar, hoy en día existen en el mercado servidores con una gran capacidad de cómputo, como se muestra en la Tabla I. Esto, junto con tecnologías como SR-IOV o DPDK, permite a los operadores móviles realizar el procesamiento de tráfico de red en servidores con arquitectura x86 en vez de utilizar equipos especializados con ASICs como CAMs (en *switches*) o TCAMs (*routers*).

Tabla I. Recursos por servidor en la actualidad

Procesador	Memoria	Almacenamiento	Tarjeta de Red	Tarjeta Gráfica
≥ 20 núcleos ≥ 100 MB caché	≥ 128 GB DDR5 ≥ 4800 MT/s	SSD SAS ≥ 1 TB	100 Gbps	≥ 50 teraFLOPS de precisión simple ≥ 25 teraFLOPS de precisión doble

En segundo lugar, los *clusters* de servidores de alta performance dedicados para HPC o Big Data suelen tener recursos sin utilizar, generando un desperdicio en el consumo de energía por estar prendidos sin carga de procesamiento), lo que a su vez genera problemas con emisiones de CO2 en centros de datos con gran cantidad de servidores y, por consiguiente, el calentamiento global. Por el contrario, al desplegar estos *clusters* de alta performance en un entorno de nube, los recursos de cómputo son asignados mientras se realiza el procesamiento; una vez concluido, estos se liberan y pueden ser utilizados por otras aplicaciones, aprovechando mejor los recursos. Adicionalmente, si consideramos que los servidores de la nube pueden disponer de componentes especializados como GPUs o TPUs, estos pueden ser asignados de forma exclusiva o compartida por las aplicaciones de Inteligencia Artificial o Procesamiento de Señales.

Por último, en la actualidad existen técnicas que permiten reducir el *overhead* de virtualización, el cual afecta la performance de las aplicaciones en la nube. Técnicas como CPU *pinning*, modificar el *niceness* y modificar el tipo de *scheduler* de procesos en Linux permiten que las aplicaciones puedan ejecutarse sobre CPUs específicos con alta prioridad, minimizando la interferencia con otros procesos. Por el lado de la red, técnicas como PCI-passthrough y DMA permiten que el tráfico de las aplicaciones se escriba directamente de la memoria a la NIC sin tener que pasar por el *kernel* de Linux (cuello de botella del tráfico de red).

La sección de Ingeniería de Telecomunicaciones de la Pontificia Universidad Católica del Perú (PUCP) cuenta con una plataforma de emulación de escenarios académico llamada Hybrid Academic Scenario Testbed (HAST), el cual dispone de 1 TB de RAM, 192 *cores*, una SAN de 80 TB de almacenamiento y 2 redes de datos de 10 Gbps, el cual está basado en una solución de nube privada llamada OpenStack. El HAST tiene como objetivo albergar escenarios académicos (por ejemplo, *cluster* de HPC donde se realicen simulaciones de Química o Física, laboratorios de los cursos de pregrado, etc.). Actualmente el HAST utiliza el *scheduler* de OpenStack para realizar la ubicación de máquinas virtuales (VMs) en servidores físicos; sin embargo, el *scheduler* de OpenStack asume que todas las VMs presentan un patrón similar de consumo de recursos. Debido a que existen escenarios que requieren una alta performance –con un patrón de uso de recursos diferente a las cargas tradicionales de Cloud– se requiere hacer un mapeo de instancias a servidores físicos que considere los requerimientos de cada tipo de instancias.

1.2. Problemática

Tradicionalmente las aplicaciones de alta performance se han ejecutado en *clusters* de servidores dedicados a la ejecución de cada aplicación. De esta manera, se aseguraba que los recursos de cómputo eran utilizados única y exclusivamente por estas aplicaciones, evitando la interferencia con otras. Sin embargo, debido a la gran capacidad de cómputo que las aplicaciones de alta performance demandan, estos *clusters* poseen una gran huella de carbono. Además, la cada vez mayor adopción del modelo de *Cloud Computing* debido a sus beneficios en términos de

escalabilidad, alta disponibilidad y opciones de recuperación ante desastres, hizo que se explorara la opción de desplegar estos entornos de alta performance sobre la nube.

Si bien en una nube privada existen soluciones para desplegar servidores dedicados a aplicaciones de alta performance (p. ej. Ironic en OpenStack), existen casos donde se disponen de pocos servidores físicos y no se pueda reservar servidores completos. Además, esto podría generar que los servidores se encuentren subutilizados, pues las ejecuciones de aplicaciones de alta performance no utilizan todos los recursos disponibles en los servidores o que ocurran tiempos muertos de ejecución. Debido a ello, el modelo de *Cloud Computing* permite aprovechar la mayor cantidad de recursos de cómputo disponibles al ejecutar diferentes aplicaciones en un servidor. En la Figura 1 se muestra una comparación de los casos donde se reservan servidores para aplicaciones distintas (izquierda, quedan recursos sin utilizar) y donde se despliegan múltiples aplicaciones en los mismos servidores (derecha, se aprovechan todos los recursos).

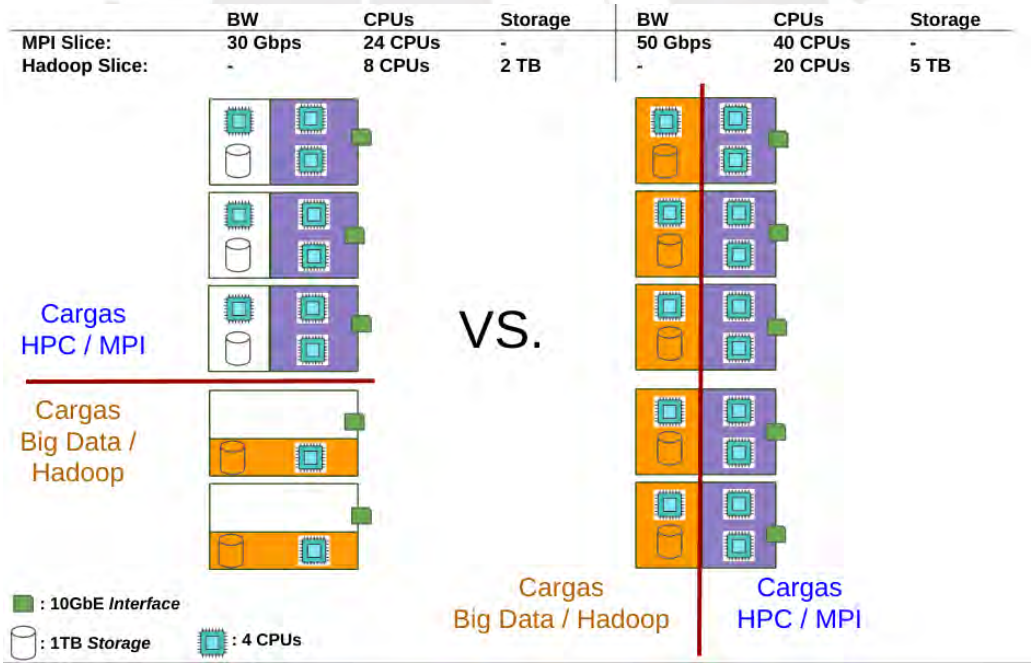

Figura 1. Dedicar vs. compartir servidores para diferentes aplicaciones.

Fuente: Elaboración propia

Por otro lado, los entornos de *Cloud Computing* (p. ej. OpenStack) suelen realizar un *overprovisioning* de recursos: es decir, asumiendo que el consumo de recursos es menor a la cantidad de recursos asignados, se asignan más recursos de los que realmente disponen los servidores; así los recursos pueden ser aprovechados por otras instancias. Esto se basa en que, en la práctica, las instancias de cómputo utilizan los recursos asignados por instantes y/o solo una fracción de ellos (multiplexación estadística). Aunque realizar *overprovisioning* permite utilizar de forma más eficiente los recursos de los servidores, esto podría generar eventos de contención de recursos, donde una aplicación de alta performance compita por acceder al mismo recurso con otra aplicación, incumpliendo posibles SLAs.

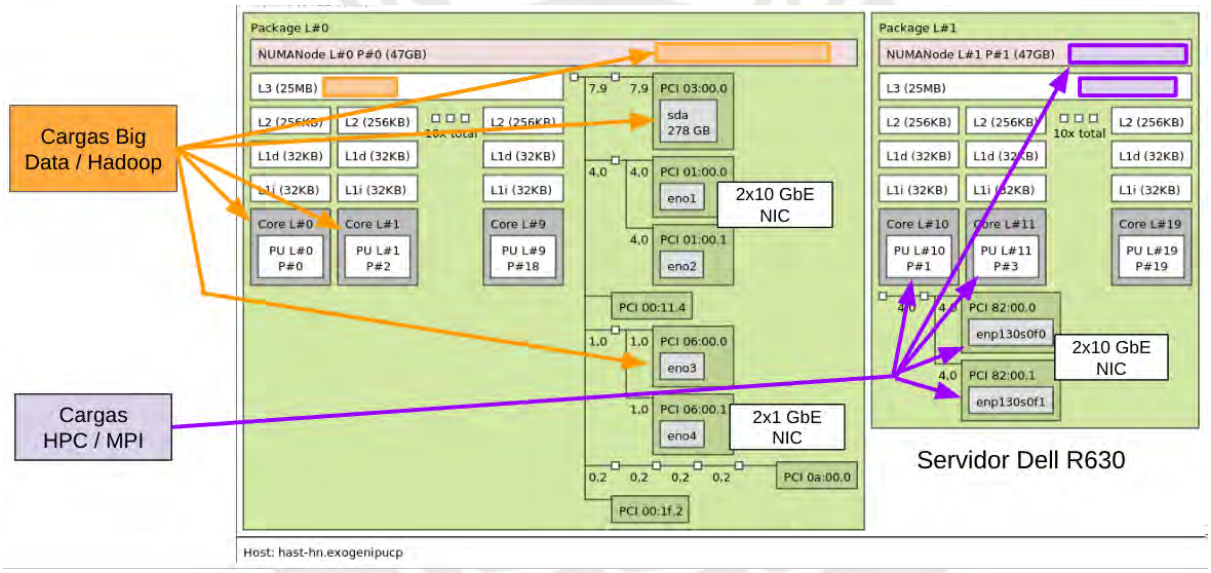

Figura 2. Asignación de recursos en un servidor a múltiples aplicaciones.
Fuente: Elaboración propia

Por otro lado, la forma en cómo se asignan los recursos puede impactar en la cantidad y performance de las instancias de cómputo, a su vez impactando en la en el número de aplicaciones y en su performance, como se muestra en el diagrama de la Figura 2. Respecto al número de aplicaciones, se observa que las carga HPC está utilizando solamente el procesador 2 (nodo NUMA #1) y la interfaz de red conectada a él, mientras que el procesador 1 (nodo NUMA #0) y los dispositivos PCI conectados a él quedarían libres si no fuesen usados por otra aplicación, como en este caso la

carga de Hadoop (problema con reservar servidores para una única aplicación). Más aun, se observa que quedan algunos recursos sin utilizar (núcleos en ambos nodos NUMA y 2 puertos de 10 GbE en el nodo NUMA 0) que podrían ser asignados a otras aplicaciones.

Más aún, para obtener una buena performance en Linux (*kernel* del sistema operativo) se debe considerar el funcionamiento del *scheduler* de procesos. *Completely Fair Scheduler* o CFS (*scheduler* por defecto desde la versión de *kernel* 2.6.23) asigna un núcleo disponible al proceso que lo solicite (*The Linux Kernel documentation*, s/f); si en un siguiente instante el proceso quiere realizar otra ejecución, el *scheduler* de Linux podría asignarle un núcleo diferente al anterior (Lozi et. al., 2016). Esto puede afectar el desempeño del proceso, ya que los datos e instrucciones que estuvo utilizando se puede encontrar en la caché L1/L2 o en la caché L3 del nodo NUMA del núcleo anterior (asumiendo que el nuevo núcleo se encuentre en un nodo NUMA distinto), por lo que se generarían *caché misses* (no se encuentra la información en la caché) y se buscaría en la memoria principal, cuyo acceso es más lento que el de las cachés. En caso los núcleos empleados por los procesos se encuentren en nodos NUMA distintos, para acceder a los datos e instrucciones se debe pasar por el bus de interconexión de procesadores (Intel QPI o AMD HyperTransport), el cual es más lento que el bus de comunicación entre la RAM y el procesador, ocasionando posibles cuellos de botella.

1.3. Objetivos

A continuación, se presentan tanto el objetivo general como los objetivos específicos del presente trabajo de investigación.

1.3.1. Objetivo General

Tomando en consideración que existen distintos tipos de cargas y que la forma en cómo se realiza la asignación de instancias puede impactar en la performance de las aplicaciones ejecutadas sobre ellas, el objetivo del presente trabajo de tesis es diseñar e implementar un algoritmo de *Instance Placement* que permita asignar de la

mejor manera posible los recursos de cómputo de servidores –núcleos del procesador, memoria, *bandwidth*, almacenamiento, GPU, entre otros– a instancias de cómputo en una nube privada. El algoritmo de *Instance Placement* debe considerar el tipo de carga de las instancias (cada tipo tiene diferente patrón de uso de recursos), debe asegurar que satisfaga el nivel de servicio de las instancias (caso contrario denegar su creación) y permita que las instancias sean máquinas virtuales, contenedores o servidores físicos (dependiendo del nivel de servicio requerido).

1.3.2. Objetivos Específicos

- Clasificar los posibles escenarios (taxonomía) y determinar los de interés
- Modelar los escenarios de interés como problemas de Optimización Combinatoria, donde se considere el *slice* de instancias completo como la unidad de asignación (no solo una instancia individual) y la prioridad de cada tipo de carga.
- Identificar la complejidad del problema de asignación de recursos para los escenarios de interés.
- Implementar un algoritmo de *Instance Placement* que se ejecute en tiempo real.
- Utilizando la métrica de utilidad, evaluar y comparar el algoritmo propuesto con el *scheduler* de OpenStack.

1.4. Metodología

El presente trabajo de tesis se va a desarrollar en 5 etapas, las cuales permitirán que se realicen los objetivos anteriormente mencionados. A continuación, se presenta cada una:

a) Revisión del estado del arte: en esta etapa se revisarán las publicaciones relacionadas al problema de *VM Placement*, así como los trabajos que enfoquen

esta asignación en entornos de alta performance y que planteen cargas diferenciadas.

b) Revisión de conceptos teóricos: en la presente etapa se revisarán la teoría de optimización combinatoria, partiendo desde la clase a la cual pertenece el problema, la complejidad de los algoritmos de optimización hasta el uso de heurísticas para hallar la mejor solución posible.

c) Formulación del problema y planteamiento de la solución: tomando como referencia lo revisado en las etapas anteriores, se formulará el escenario de asignación de recursos a instancias como un problema de optimización combinatoria y se planteará un algoritmo basados en optimización y/o heurísticas para resolverlo.

d) Desarrollo del algoritmo en base a optimización y/o heurísticas: en esta etapa se desarrollará el algoritmo propuesto usando los lenguajes de programación Python y Matlab.

e) Evaluación del algoritmo: ejecutará el algoritmo desarrollado y se evaluará la utilidad que provee respecto al *scheduler* de OpenStack.

1.5. Alcances

El presente trabajo de tesis tiene los siguientes alcances:

- El diseño, implementación y evaluación del algoritmo de *Instance Placement* considere la disponibilidad de los siguientes recursos: núcleos de procesador, memoria RAM y almacenamiento.
- Se considera que el problema de *Instance Placement* (asignación de recursos a instancias) es de tipo *Offline*: se asume que los patrones de utilización de recursos (p. ej. qué días, a qué horas y por cuánto tiempo se utilizan) se conocen de antemano, para realizar la asignación de instancias a servidores.
- Debido a que no se tiene registro de cuál es el patrón óptimo de asignación de instancias a servidores del HAST, el presente trabajo de tesis tiene como objetivo

servir como punto de partida para obtener un registro de dicho patrón. Para ello, se emplearán técnicas de optimización combinatoria.

- A futuro, se considerará el problema de *Instance Placement* de tipo *Online*: teniendo como base los patrones de uso y asignación de recursos, se emplearán técnicas de *Machine Learning* para resolver los problemas de estimación (predicción de uso de recursos) y clasificación (determinar si es que se permita la creación de VMs y en qué servidor se colocan).
- La métrica para evaluar el algoritmo de *Instance Placement* propuesto es la utilidad, el cual indica cuál es el beneficio de crear determinadas VMs sobre el resto.
- Para evaluar el algoritmo propuesto, se utilizará como referencia el *scheduler* de OpenStack, el cual emplea 3 criterios para determinar en qué servidor crear las VMs (número de *cores*, RAM y almacenamiento) con un factor de *overprovisioning* para cada uno.

CAPÍTULO II. ESTADO DEL ARTE

En el presente capítulo se presentan las publicaciones revisadas, las cuales se han enfocado en resolver problemas de *Instance Placement* con sus diferentes variantes, tanto en entornos donde no se priorizan aplicaciones como en aplicaciones de alta performance. Se considera como término de búsqueda *VM Placement*, pues en un inicio las máquinas virtuales (VM) eran la manera en cómo se agrupaban recursos dentro de servidores, en un contexto de Centro de Datos o *Cloud Computing*. No obstante, el concepto de una VM como grupo de recursos se puede generalizar hacia el concepto de instancia.

2.1. Investigación sobre el Problema de VMP

En esta sección se describen las publicaciones más relevantes sobre el problema de *Virtual Machine Placement* (VMP), el cual consiste en que –dado un grupo de máquinas virtuales (VMs)– se debe determinar cuál es el servidor físico (PM) más adecuado para ubicar las VMs, en un contexto de *Cloud Computing*. Si bien el nombre del problema está definido en la literatura como *Virtual Machine Placement* (p. ej. Masdari et al., 2016; Usmani & Singh, 2016; y Bharathi et al., 2017), algunos autores también se refieren a él como *Virtual Machine Allocation* (Luo et al., 2019), *Resource Allocation* (Barán & López-Pires, 2017 y Madni et al., 2017), *Virtual Machine Scheduling* (Liu & Qiu, 2016) o *Resource Scheduling* (Singh & Chana, 2016).

Barán y López-Pires (2017) presentan una taxonomía para clasificar los trabajos de investigación sobre VMP, en base al entorno (con los criterios de orientación, arquitectura de despliegue y tipos de formulación), y a la formulación y resolución del problema (con los criterios de optimización, función objetivo y técnica de solución). Con respecto al entorno, la orientación indica si se enfoca al *Cloud Service Provider* o CSP (qué VM se ubica en qué PM) o si se enfoca al *Broker* (qué VM se coloca en qué CSP); la arquitectura de despliegue determina si se utiliza una nube con un solo centro de datos o una nube distribuida; y la formulación indica si el planteamiento del problema es *offline* (el algoritmo conoce todos los futuros eventos) u *online* (el algoritmo no conoce los eventos futuros). Con respecto a la formulación y

resolución del problema, la optimización indica si el enfoque tiene un solo objetivo, multiobjetivo o un multiobjetivo planteado como un solo objetivo; la función objetivo determina qué se quiere minimizar/maximizar (p.ej. el consumo de energía, o la utilización de recursos); y las técnicas de solución indican si se utilizan algoritmos determinísticos (p. ej. Programación Lineal o Programación Dinámica), heurísticas (p. ej. *First-Fit* o *Best-Fit*) meta-heurísticas (p. ej. Algoritmo Genético o *Ant-Colony Optimization*) y algoritmos de aproximación (p. ej. *p-aproximation*).

En Talebian et. al. (2020) los autores presentan una definición formal del problema de VMP, donde se plantea el problema como uno de optimización en el cual se pueden identificar los objetivos y restricciones, los cuales se representan como ecuaciones y/o inecuaciones, y varían de acuerdo al contexto del VMP de interés. En los objetivos se plantea qué se quiere minimizar, en base a ciertos supuestos: por ejemplo, asumiendo que el consumo de energía es proporcional con el número de PMs prendidos en un *datacenter*, entonces el objetivo de reducir el consumo de energía se traduce en minimizar la cantidad de PMs utilizados para ubicar las VMs. Por el lado de las restricciones, estas indican el espacio de búsqueda de la solución: por ejemplo, una restricción al problema de VMP es que una VM sólo puede estar ubicada en un único PM. Luego se muestra que las posibles soluciones se representan como matriz de dimensiones $m \times n$, donde m representa el número de VMs a ubicar y n representa el número de PMs existentes. Adicionalmente se muestran algunos casos de VMP junto con una revisión de literatura para el problema de VMP en base al número (uno solo o múltiples) y tipo de objetivos, así como la técnica empleada para resolverlos.

2.2. Investigación sobre las técnicas de solución del Problema de VMP

En esta sección se describen las publicaciones más relevantes sobre las técnicas empleadas para resolver el problema de *Virtual Machine Placement*. Según lo indicado por Seyyedsalehi y Khansari (2022), este problema es de tipo $\mathcal{NP}$ -hard, por lo no resulta factible encontrar la solución óptima cuando el tamaño del problema es grande (Abohamama & Hamouda, 2020). Es por ello que en la revisión de literatura

principalmente se encontraron trabajos que proponen algoritmos basados en heurísticas y meta-heurísticas.

En Mosa y Sakellariou (2019) se muestra un algoritmo para el caso de VMP dinámico, donde las VMs pueden ser reubicadas en función a la demanda de recursos de las VMs. Esto tiene como objetivo minimizar los escenarios de subutilización y sobreutilización de recursos en un centro de datos. Para ello, los autores desarrollan una solución basada en un algoritmo genético que toma en cuenta los recursos de CPUs (núcleos del procesador) y memoria. Esta solución es simulada con CloudSim (*framework* para la simulación de infraestructura de *Cloud Computing*) con trazas de consumo de memoria y CPUs que siguen una distribución uniforme. Debido al comportamiento dinámico del consumo de recursos de las VMs, se generan las trazas y se realiza la reubicación de VMs cada 5 minutos (intervalo de *scheduling*) con un tiempo de simulación de 1 día. Para evaluar la solución se utilizan las métricas de AOUU (promedio de subutilización de CPUs y memoria) y AOSLAV, (promedio del número de violaciones de SLA), y se compara con la heurística *Best-Fit Decreasing* (BFD). Los resultados muestran que para diferentes números de VMs y PMs, el valor de AOUU y AOSLAV en el algoritmo propuesto es siempre menor al de BFD.

En Wu y Shen (2017) se presenta un algoritmo de VMP que tiene como objetivo minimizar el costo de PMs, el cual es proporcional no solo al número de PMs prendidos sino también al tiempo en que lo están. Para ello, se modela el problema como uno de optimización combinatoria, donde se desea minimizar el tiempo total de ejecución de los PMs. Luego los autores proponen algoritmos para los escenarios *online* y *offline*. La diferencia entre ambos casos es que –asumiendo que las solicitudes de creación de VMs contienen la demanda de recursos y su tiempo de ejecución– en el primero se conoce de antemano la información de todas las VMs, mientras que en el segundo se conoce la información a medida que van llegando las solicitudes de creación. Para el escenario *online* se utiliza una heurística que toma en cuenta el ratio de utilización de recursos en un periodo de tiempo, mientras que para el escenario *offline* se utiliza un algoritmo *Greedy* basado en el algoritmo *Best-Fit*. Luego de realizar las simulaciones, se obtiene que los algoritmos propuestos requieren un menor número de PMs y un menor tiempo de ejecución, tanto para el caso *offline* (en comparación con *First-Fit Decresing* y *Best-Fit Decreasing*) como para el caso *online* (en comparación con *First-Fit* y *Best-Fit*).

En Xing et al. (2022) se muestra un escenario de VMP multiobjetivo, donde se desea minimizar el consumo total de energía de PMs y *switches*, así como la utilización de la red. El escenario considerado es un centro de datos con una topología *Fat-Tree*, donde una parte importante del consumo de energía proviene de los *switches*. Para ello, los autores modelan el problema como uno de optimización combinatoria, donde se minimice el costo (objetivo compuesto el consumo de energía y red) con las restricciones de no exceder el total de recursos (CPUs, memoria y capacidad de puerto de red) del servidor. Los autores proponen el algoritmo *Energy- and Traffic-Aware Ant-Colony Optimization* (ETA-ACO) con 3 esquemas de solución: *Energy- and Bandwidth-Aware PM Selection* (EBAPMS), el cual primero filtra los PMs con menor consumo de energía, y luego aplica un segundo filtro para obtener los PMs con menor consumo de red; *Traffic-based VM Ordering* (TVMO), el cual ubica primero las VMs que mayor demanda de red; y *Direct Information Exchange* (DIEX), el cual construye nuevas soluciones al propagar los componentes de las mejores soluciones encontradas por las hormigas. Luego de probar el algoritmo ETA-ACO con diferentes parámetros, se observa que el esquema DIEX obtiene menores valores de costo, consumo de energía y consumo de red. Al comparar ETA-ACO con otras heurísticas y meta-heurísticas (estado del arte para el problema de VMP) también se obtienen mejores resultados de las métricas de interés.

2.3. Investigación sobre HPC en la Nube

Si bien inicialmente se tomó como referencia los trabajos que hablan acerca de la taxonomía y revisiones sistemáticas del ejecutar aplicaciones HPC sobre la Nube (Netto et al., 2019 y Talebian et al., 2020), en la presente sección se detallan las publicaciones más relevantes de sobre qué consideraciones tener tanto al momento de ejecutar aplicaciones HPC sobre entornos de *Cloud Computing* como al realizar la ubicación de VMs para este tipo de aplicaciones.

En Gupta et al. (2013) se aborda el tema de VMP en un contexto donde se tienen aplicaciones HPC, por lo que el algoritmo de VMP debe considerar que una solicitud de creación involucra varias VMs, las cuales componen la infraestructura de un tipo de aplicación HPC. Esto se debe a que tradicionalmente los proveedores de

nube utilizaban algoritmos de *scheduling* agnósticos a la aplicación que ejecutan las VMs, lo que resultaba en una baja performance. En base a algunos experimentos donde evalúan el impacto de colocar aplicaciones HPC en los mismos servidores, los autores proponen 3 consideraciones al momento de realizar la ubicación de VMs en PMs para mejorar la performance de las aplicaciones HPC: *Cross-Application Interference* (colocar juntos algunos tipos de aplicaciones mejora la performance de al menos una de ellas –interferencia positiva–), *Topology Awareness* (ubicar VMs de aplicaciones que se comunican constantemente en servidores que tienen conexiones de alta velocidad) y *Hardware Awareness* (colocar las VMs de una aplicación en servidores con procesadores que tengan las mismas características). También proponen una clasificación de aplicaciones HPC: ExtremeHPC (aplicaciones sensibles a la topología, requieren servidores dedicados), SyncHPC (soportan un pequeño grado de interferencia que ExtremeHPC) AsyncHPC (menos sensibles a la comunicación, soportan mayor interferencia que SyncHPC) y NonHPC (soportan más interferencia que AsyncHPC y pueden ser ubicados en servidores heterogéneos).

En Gupta et al. (2016) se complementa el trabajo de investigación anterior al presentar un análisis del cuello de botella en performance de la Nube, así como técnicas para optimizar la Nube a fin de mejorar el desempeño de aplicaciones HPC. Los autores utilizan el NAS Parallel Benchmarks (NPB) para comparar la performance de las aplicaciones HPC disponibles en NPB al ejecutarlas en supercomputadoras y en la Nube (pública y privada). En las simulaciones se observa que la mayoría de aplicaciones escalan bien en supercomputadoras, mientras que en los entornos de *Cloud Computing* utilizados no, tal como se muestra en la variabilidad en su performance. Luego de identificar que las aplicaciones HPC son sensibles a la latencia y *bandwidth* de red, los autores encuentran que las redes virtuales en las nubes empleadas (públicas y privadas) presentan una mayor latencia y menor *bandwidth* en comparación con las supercomputadoras, así como interferencia con dichas aplicaciones. Para poder optimizar la Nube para HPC, los autores presentan las técnicas de *Lightweight Virtualization* (*thin* VMs con PCI-*passthrough* a dispositivos de I/O y contenedores), CPU *Affinity* (se vincula un proceso a un CPU específico) sin superposición y uso de *Link Aggregation* (múltiples enlaces físicos funcionan como uno solo enlace lógico).

En Melo Alves et al. (2018) se considera el problema de VMP para escenarios de HPC, donde es frecuente que ocurra interferencia cruzada debido a que se comparte un PM. Para ello, se define una versión modificada del problema original llamado *Interference-aware Virtual Machine Placement Problem* (IVMPP), el cual tiene como objetivo minimizar tanto la interferencia de las aplicaciones HPC como el número de PMs empleadas para ubicar las VMs. Los autores utilizan el modelo propuesto en Melo Alves y Drummond (2017) para predecir la interferencia sufrida por aplicaciones que comparten el mismo servidor, el cual se basa en la intensidad de acceso a la *Shared Last-Level Cache* (SLLC), DRAM y red virtual. Los autores realizan una formulación matemática del IVMPP que –al ser una variación del problema VMP original– es de clase $\mathcal{NP}$-Hard, por los que se propone una solución basada en la meta-heurística de *Iterated Local Search* (ILS). Luego de comparar el nivel de interferencia obtenido con la solución propuesta, las heurísticas más conocidas –*Best Fit* (BF), *First Fit* (FF), *Worst Fit* (WF), *Best Fit Decreasing* (BFD), *First Fit Decreasing* (FFD) y *Worst Fit Decreasing* (WFD)– y *Multidimensional Online Bin Packing* (propuesto en Gupta et al., 2013) obtienen un menor nivel.

CAPÍTULO III. MARCO TEÓRICO

En el presente capítulo se muestran las definiciones y conceptos necesarios para entender el contexto del problema de *VM Placement* para aplicaciones de alta performance. Primero se presentan las aplicaciones que usualmente requieren un alto desempeño de los recursos asignados; luego se presentan algunas técnicas empleadas para mejorar la performance de las aplicaciones corriendo en VMs; por último, se presenta el tema de optimización combinatoria, donde se muestran las clases de complejidad que pueden tener los problemas, los problemas combinatorios de interés y los métodos para solucionarlos.

3.1. Aplicaciones de Alta Performance

Las aplicaciones de alta performance son un conjunto de aplicaciones que tienen la característica de emplear una gran cantidad de recursos para realizar sus ejecuciones. Por lo general, estas aplicaciones se ejecutan en un hardware dedicado para evitar la interferencia con otras aplicaciones; con ello, se asegura que los recursos estén disponibles para cuando se requiera acceder a ellos. A continuación, se presentan algunas de las aplicaciones de alta performance más conocidas.

3.1.1. High Performance Computing

En el artículo web *Getting started with high performance computing* (2021), se define a la Computación de Alta Performance (HPC por sus siglas en inglés) como el procesamiento de datos mediante el uso de cálculos complejos a una alta velocidad (en un periodo de tiempo corto). En la actualidad, las aplicaciones de HPC se despliegan sobre *clusters* de computadoras (también llamados nodos), las cuales proveen capacidad computacional mediante el uso de sus CPUs y GPUs y se encuentran interconectadas por una red de alta velocidad y baja latencia (por ejemplo, Infiniband). Para ejecutar un trabajo de HPC en el *cluster*, se hace uso de Message Passing Interface (MPI), el cual consiste de una librería y protocolo que permitan

distribuir la ejecución de un trabajo sobre múltiples equipos, así como la comunicación de a nivel de aplicación en el *cluster*. Un elemento importante en los entornos HPC es el *scheduler* de trabajos (p. ej. Slurm o TORQUE), el cual permite asignar de forma eficiente las solicitudes de trabajo a los recursos de cómputo y hacer el seguimiento de la disponibilidad de los recursos.

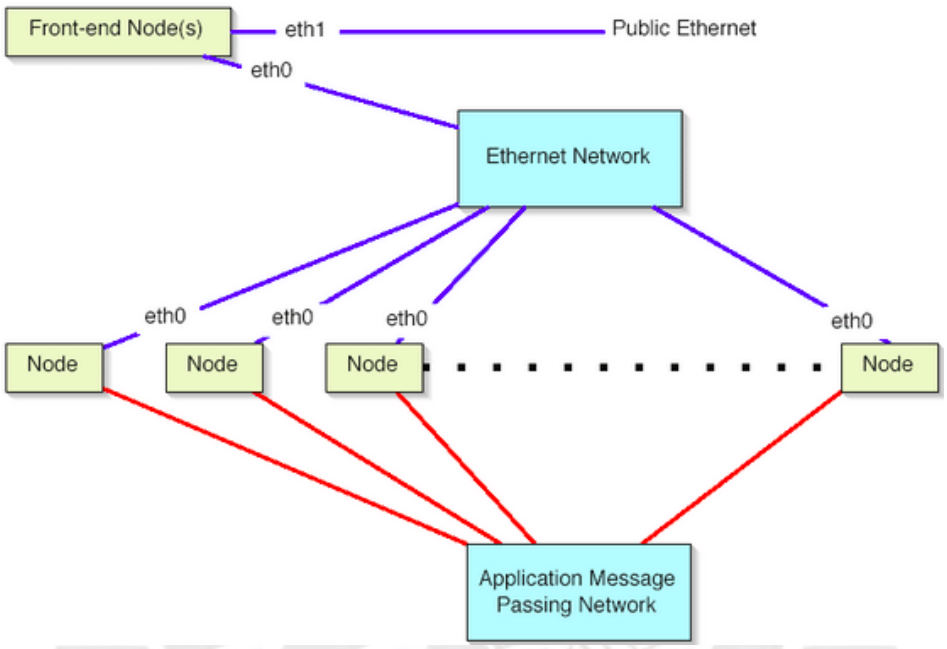

Figura 3. Arquitectura de un clúster de computadoras para HPC
Fuente: Gamess y Ortiz-Zuazaga (2016)

3.1.2. High Throughput Computing

High Throughput Computing (HTC) es un modelo de cómputo donde se ejecutan múltiples trabajos independientes sobre recursos de cómputo heterogéneos (*racks* de servidores, computadoras de escritorio, recursos en la nube, etc.) con dueños distintos (*Center for High Throughput Computing*, s.f.). HTC aprovecha de forma eficiente el poder computacional disponible en estos equipos durante los periodos de tiempo donde el uso de recursos es bajo (p. ej. las computadoras se encuentran en estado *idle* o hibernando) para ejecutar los trabajos. A diferencia de los entornos HPC, donde el objetivo es ejecutar lo más rápido posible un solo trabajo

(medido en FLOPS), HTC se enfoca en completar la mayor cantidad de trabajos en un tiempo prolongado (medido en FLOPY). Para ello, se disponen de herramientas como BOINC o HTCondor, los cuales permiten distribuir y ejecutar los trabajos en los equipos disponibles.

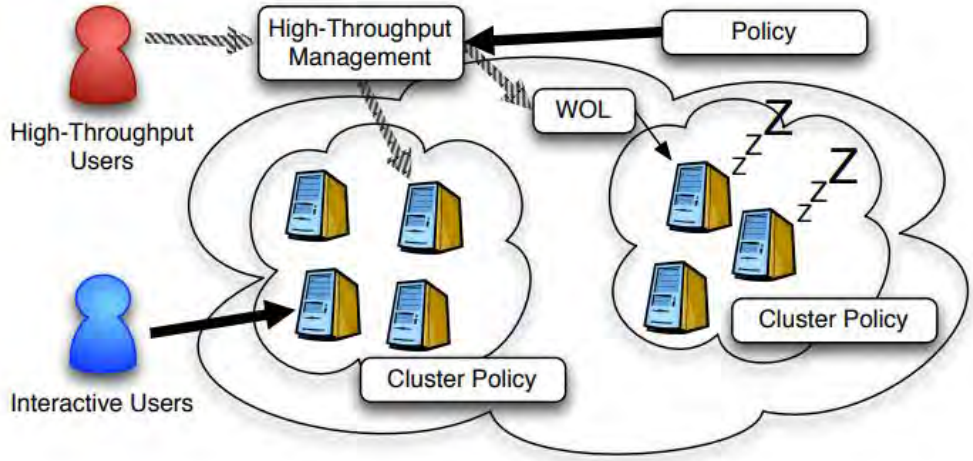

Figura 4. Modelo de un sistema HTC en un entorno multi-usuario
Fuente: Forshaw et al. (2014)

3.1.3. Big Data

Big Data es un término que hace referencia a un conjunto de datos con una gran variedad, el cual presenta un gran volumen que se va incrementando cada vez a una mayor velocidad (Oracle, s/f). Como se indica en Alla (2018), este conjunto de datos presenta ciertas características, inicialmente llamadas las 3 Vs (Variedad, Volumen y Velocidad), las cuales fueron incrementadas hasta llegar a 7 Vs: adicionalmente a las 3 Vs, se adicionaron la Veracidad, Variabilidad, Visualización y Valor. Una de las soluciones más empleadas en los entornos de Big Data es Apache Hadoop, el cual es un *framework* de código abierto que ofrece un sistema donde almacenar y procesar la data. Para el almacenamiento, Hadoop utiliza el Hadoop Distributed File System (HDFS) –el cual es un sistema de archivos distribuido, que soporta réplicas y es tolerante a fallas–, y para el procesamiento utiliza el *framework* MapReduce V2 (basado en el *scheduler* YARN).

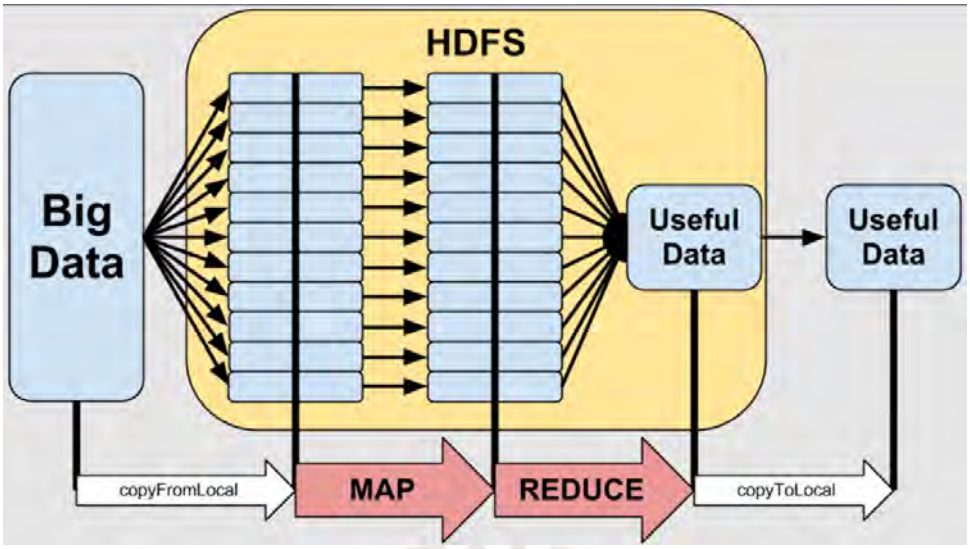

Figura 5. Workflow de trabajos con Map-Reduce y data en HDFS
Fuente: Lockwood (2015)

3.1.4. Redes Móviles

Tradicionalmente, los equipos que conforman una red móvil (tanto los equipos del *backhaul* –*switches* y *routers*– como los del *core* de la red –SGW, PGW, entre otros–) eran provistos por fabricantes que manejaban su propia arquitectura. Para evitar problemas de compatibilidad entre las tecnologías de los equipos, los operadores móviles optaban por adquirir equipos del mismo fabricante, lo cual generaba una situación de *vendor lock-in*. Con la mayor adopción del modelo *Cloud Computing* y el avance de las tecnologías relacionadas, fueron apareciendo soluciones de código abierto y/o software libre para los equipos de las redes móviles, como Aether del *Open Networking Foundation* o Magma Core de *The Linux Foundation*. Respecto a la red de acceso, el enfoque de *Edge Computing* y la arquitectura Open RAN permitieron que se puedan virtualizar los equipos de control de la capa de acceso (p. ej. el BBU paso a ser vBBU, incluso desagregándose en BU+CU). Por otro lado, la aparición de *Network Function Virtualization* (OpenStack Team, 2020) permitió que se desplieguen servicios de red (p. ej. firewalls, NAT, IPS, *transcoders*) sobre servidores con arquitectura x86 como *Virtual Network Functions* (VNFs) interconectadas (también llamado *Service Function Chaining*).

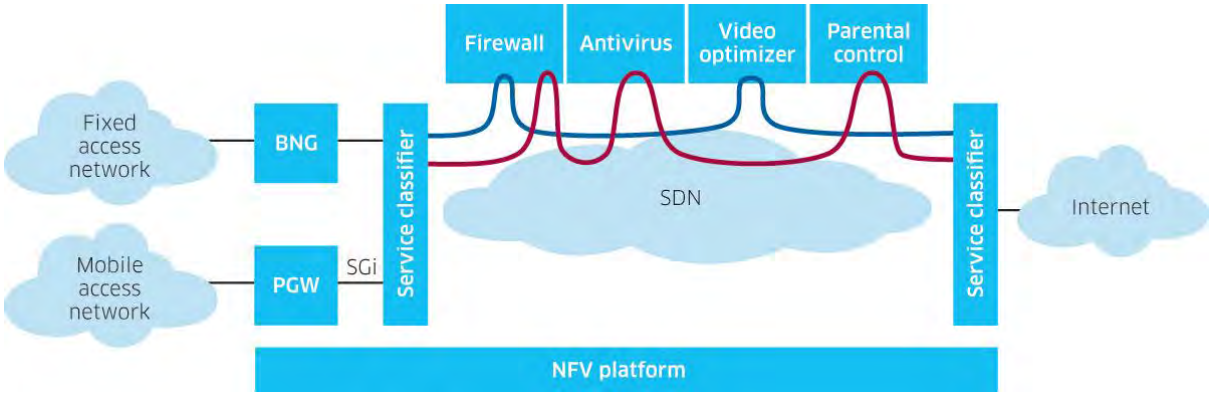

Figura 6. Ejemplo de *Service Function Chaining* (SFC).
Fuente: Advanced Networking Research Lab (s/f)

3.2. Optimizaciones en Hardware para Alta Performance

Con el uso de las tecnologías de virtualización, los recursos presentes en los servidores se comenzaron a utilizar de forma más eficiente; no obstante, se introdujo un *overhead* asociado a la virtualización. Debido a ello, la performance obtenida en los trabajos ejecutados en entornos virtuales es menor a los obtenidos al ejecutarlos en entornos físicos. A continuación, se presentan algunas técnicas que existen para minimizar dicho *overhead* en los entornos virtuales.

3.2.1. Process Affinity

En los sistemas operativos con kernel de Linux, cada hilo de ejecución e interrupción tiene una propiedad llamada *processor affinity* (Klech et al., 2020). El *scheduler* de procesos de Linux utiliza esta información para determinar qué procesos e interrupciones pueden ejecutarse en determinados núcleos del procesador (CPU *pinning*). Modificando adecuadamente esta propiedad (junto con el *niceness*), se puede optimizar la performance de los procesos. Esto se debe a que cuando los procesos compiten por obtener acceso a los recursos de hardware (en particular al procesador), se puede indicar que un proceso de alta prioridad se ejecute en un núcleo de procesador, mientras que los otros procesos utilicen el resto de núcleos.

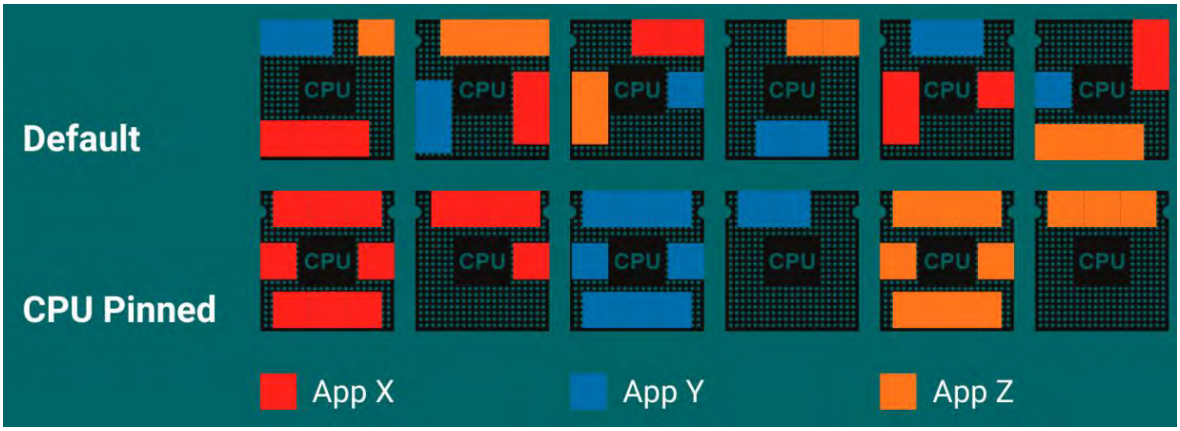

Figura 7. *Scheduling* de procesos por defecto vs. con CPU *Pinning*
Fuente: Srivastava (2021)

3.2.2. Process Priority

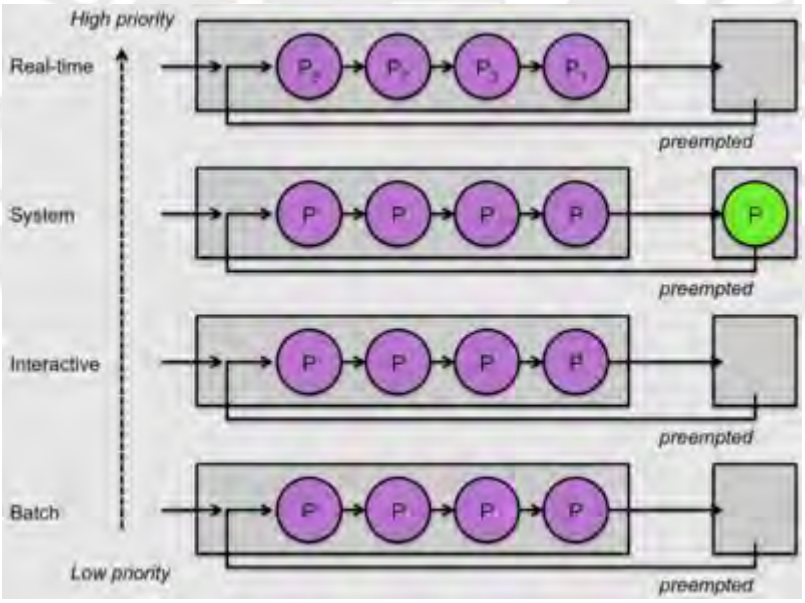

Figura 8. Ejemplo de un *Scheduler* de Procesos con Colas Multi-nivel
Fuente: Krzyzanowski (2015)

Los sistemas operativos ejecutan múltiples procesos, incluso más que el número de unidades de procesamiento (núcleos del procesador) disponibles. Debido a ello, es importante contar con un *scheduler* de procesos que permita indicar qué tareas de los procesos se ejecutan en determinado instante de tiempo. Como se

indica en Carrigan (2021), el *scheduler* del sistema crea la ilusión de que múltiples procesos se están ejecutando en simultáneo. En los sistemas con kernel de Linux, a cada proceso se le asigna un puntaje llamado *niceness*, el cual varía entre -20 y 19. Este indica la prioridad relativa de un proceso, teniendo un valor de -20 para procesos de alta prioridad y 19 para procesos de baja prioridad. Si existen múltiples procesos que requieren ser ejecutados, el *scheduler* de procesos priorizará los procesos con mayor prioridad sobre los de menor prioridad.

3.2.3. PCI passthrough

Este es un método que permite a las VMs (*guests*) conectarse a dispositivos PCI de los servidores (hosts), de manera que tengan acceso exclusivo a ellos (Radvan et al., 2014). Así, los dispositivos PCI operan como si estuvieran directamente a las VMs sin pasar por el *hypervisor*, mejorando la performance (en algunos casos *near-native*) principalmente de las aplicaciones que requieren una gran cantidad de operaciones de I/O (Jones, 2009). Para ello, se debe tanto habilitar en la BIOS la extensión *Virtualization Technology for Directed I/O* (VT-d, en procesadores Intel) o I/O *Memory Management Unit* (IOMMU, en procesadores AMD), como activar la opción Intel VT-d en el kernel del host. Para poder conectarlo al *guest*, se debe desconectar el dispositivo PCI del *host*.

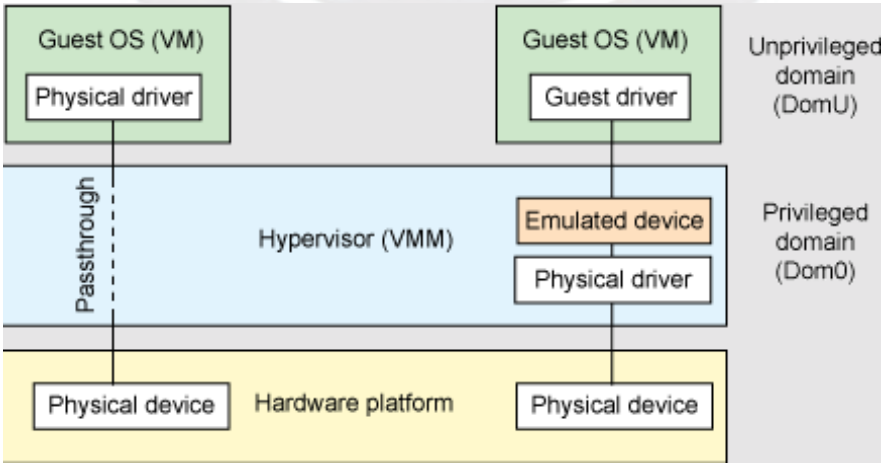

Figura 9. PCI passthrough vs. emulación de dispositivos

Fuente: Jones (2009)

3.2.4. SR-IOV

Single Root I/O Virtualization (SR-IOV) es una tecnología de virtualización de I/O para dispositivos PCIe, de manera se divide lógicamente un controlador Ethernet PCIe físico (*Physical Function* o PF) en múltiples dispositivos PCIe (*Virtual Function* o VF). A diferencia de PCI-*passthrough*, cada uno de los VFs se pueden conectar directamente a una VM, lo que evita pasar por el *hypervisor* y el *switch* virtual, y permite obtener una baja latencia y un rendimiento muy cercano a *wire-speed* (OpenStack Documentation, 2020).

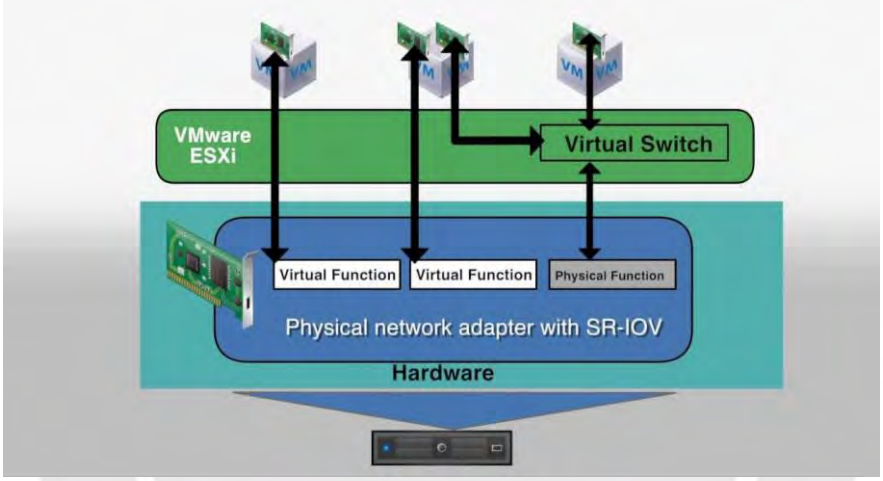

Figura 10. PCI passthrough vs. emulación de dispositivos
Fuente: VMware Docs (2013)

3.3. Optimización Combinatoria

Los problemas combinatorios son tales que tratan de encontrar grupos o asignaciones de un conjunto finito y discreto de objetos que satisfaga determinadas condiciones o restricciones (Hoos y Stutzle, 2004). Las combinaciones de estos objetos (llamados componentes de la solución) conforman el conjunto de soluciones candidatas del problema combinatorio. En la mayoría de problemas combinatorios, el espacio de soluciones candidatas (potenciales soluciones encontradas al resolver la instancia del problema, pero puede que no satisfagan todas las restricciones) es por lo menos exponencial respecto al tamaño de la instancia del problema. Por otro lado,

los problemas de optimización son problemas combinatorios cuyas soluciones no solo deben cumplir las condiciones (problema de decisión), sino que son evaluadas por una función objetivo, y la finalidad es hallar soluciones con valores óptimos de dicha función. Debido a ello, los problemas de optimización combinatoria se definen en base a la función objetivo (a minimizar o maximizar) y a las condiciones.

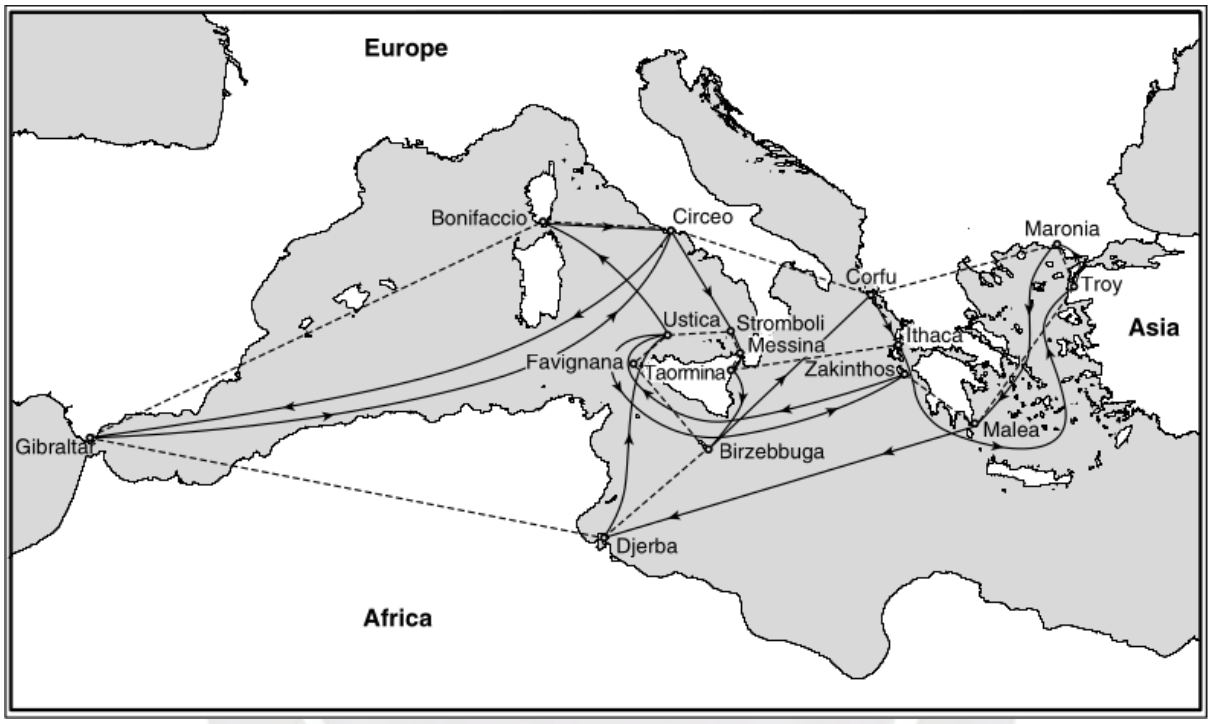

Figura 11. Ejemplo del problema de Optimización Combinatoria (TSP).
Fuente: Hoos y Stutzle (2004)

3.3.1. Clases de Complejidad de Problemas

Mientras que la complejidad computacional de un algoritmo está relacionado solo a una instancia del problema, la complejidad del problema puede definirse como la complejidad que tiene el mejor algoritmo para el caso con el peor tiempo (Hoos y Stutzle, 2004). En base a ello, se definen las siguientes clases de problemas:

- $\mathcal{P}$: clase de problemas que pueden ser resueltos por una máquina determinista en tiempo polinomial; adicionalmente $\mathcal{P} \subset \mathcal{NP}$, pues una máquina determinista puede ser emulada por una máquina no-determinista.

- $\mathcal{NP}$: clase de problemas que pueden ser resueltos por una máquina no-determinista en tiempo polinomial; además, se pueden verificar en tiempo polinomial.
- $\mathcal{NP}$-hard: un problema es de clase $\mathcal{NP}$-hard si cada problema en $\mathcal{NP}$ puede reducirse en tiempo polinomial a él; esto quiere decir que por lo menos son tan difíciles como cualquier problema en $\mathcal{NP}$.
- $\mathcal{NP}$-complete: son los problemas $\mathcal{NP}$-hard que están contenidos en $\mathcal{NP}$; es decir, son los problemas $\mathcal{NP}$ más difíciles.

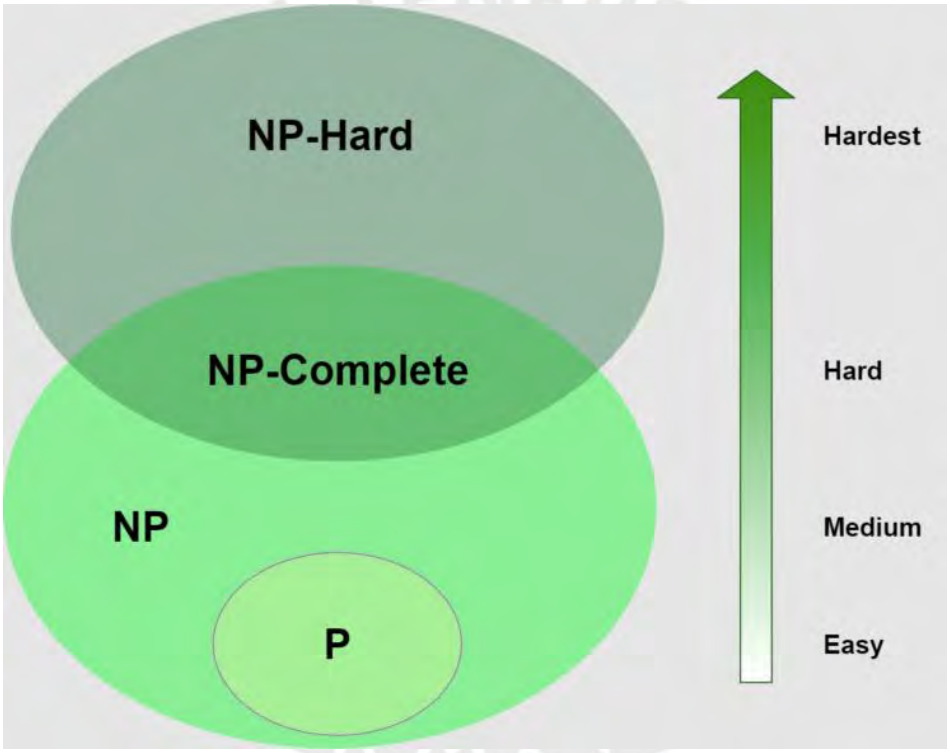

Figura 12. Relación entre clases de complejidad de problemas
Fuente: Baeldung (2022)

3.3.2. Problemas de Interés

A continuación, se muestran los tipos de problemas de optimización combinatoria empleados para modelar los escenarios de *Instance Placement* de

interés. Entre ellos se encuentran los problemas 0-1 de *Knapsack*, *Bin Packing* y *Graph Coloring*.

En el problema de 0-1 *Knapsack* ($\mathcal{NP}$-hard) se dispone de una mochila, la cual se desea llenar de objetos –cada uno tiene un valor y costo asociados– de manera que se obtenga el mayor valor total posible sin exceder la capacidad de la mochila. Este problema de puede ser visto como uno de *Integer Linear Programming* (ILP) y se formula de la siguiente manera: se tienen M objetos y una mochila de capacidad C, cada objeto i tiene un valor p_i y un costo w_i (Martello & Toth, 1990); seleccionar un conjunto de objetos que permitan:

$$\text{maximizar} \quad \sum_{i=1}^{M} p_i x_i$$

$$\text{sujeto a} \quad \sum_{i=1}^{M} w_i x_i \leq C$$

$$x_i = \{0,1\}, i \in \{1,2,\ldots,M\}$$

donde $x_i = 1$ si el objeto donde i es seleccionado o $x_i = 0$ en caso contrario.

Figura 13. Ejemplo del problema 0-1 de *Knapsack*
Fuente: Rana (2021)

El problema de *Bin Packing* ($\mathcal{NP}$-hard) consiste en asignar cada uno de los objetos a un contenedor de forma que se emplee la cantidad mínima de contenedores y no se exceda su capacidad (Martello & Toth, 1990). Este problema se formula de la

siguiente manera: dado un conjunto de N contenedores –cada contenedor j con capacidad C_j – y M objetos –cada objeto i con costo w_i –; entonces

$$\text{minimizar} \quad \sum_{j=1}^{N} y_j$$

$$\text{sujeto a} \quad \sum_{i=1}^{M} w_i x_{ij} \leq C_j y_j, j \in \{1,2,\ldots,N\}$$

$$\sum_{j=1}^{N} x_{ij} = 1, i \in \{1,2,\ldots,M\}$$

$$y_i = \{0,1\}, j \in \{1,2,\ldots,N\}$$

$$x_i = \{0,1\}, i \in \{1,2,\ldots,M\}$$

donde $y_i = 1$ si el contenedor j es utilizado o $y_i = 0$ en caso contrario, y $x_{ij} = 1$ si el objeto i es colocado en el contenedor j o $x_{ij} = 0$ en caso contrario.

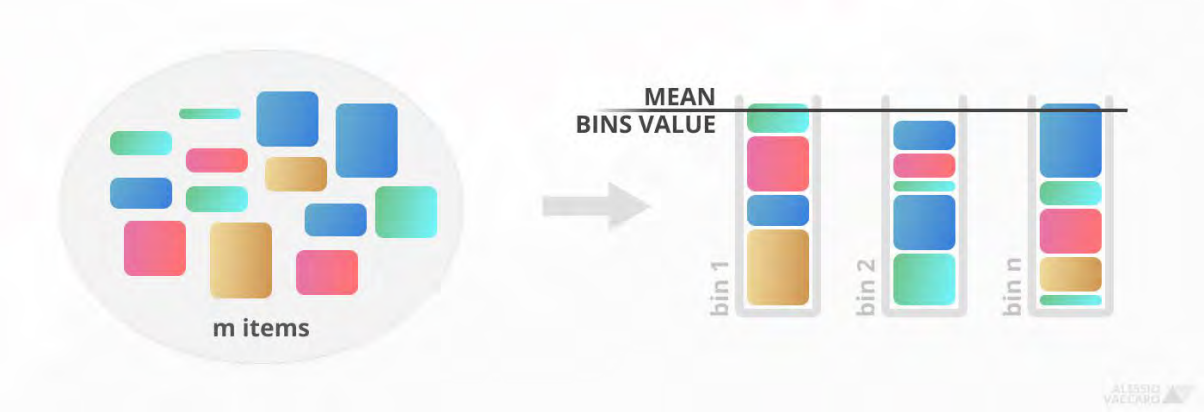

Figura 14. Ejemplo del problema *Bin Packing*

Fuente: Vaccaro (2020)

El problema de *Graph* o *Vertex Coloring* ($\mathcal{NP}$-hard) consiste en que, dado un grafo $G = (V, E)$, se debe asignar un color a cada vértice de G tal que dos vértices adyacentes reciban colores distintos y se minimice el número total de colores empleados (Hansen et al., 2009). El número mínimo de colores necesarios para colorear los vértices del grafo G se llama el número cromático $\chi(G)$. Este problema se formula de la siguiente manera: sean v y w dos vértices de V tal que $(v, w) \in E$ son

adyacentes, j el identificador de un color y $\overline{\chi}$ el límite superior en el número cromático de G; entonces

$$\text{minimizar} \quad \sum_{j=1}^{\overline{\chi}} y_j$$

$$\text{sujeto a} \quad \sum_{j=1}^{\overline{\chi}} x_{vj} = 1, \forall v \in V$$

$$x_{vj} + x_{wj} \leq y_j, \forall (v,w) \in E, j = \{1,2,\ldots,\overline{\chi}\}$$

$$x_{vj}, y_i = \{0,1\}, \forall v \in V, j = \{1,2,\ldots,\overline{\chi}\}$$

donde $y_j = 1$ si el color j es utilizado o $y_j = 0$ en caso contrario, y $x_{vj} = 1$ si al vértice v se le asigna el color j o $x_{vj} = 0$ en caso contrario.

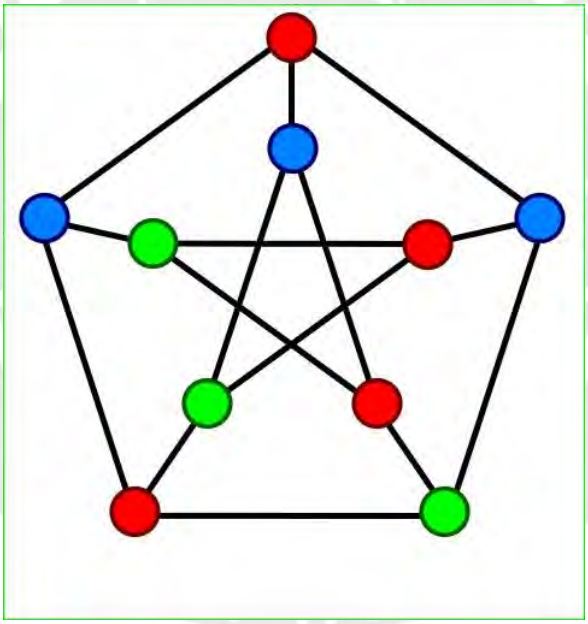

Figura 15. Ejemplo del problema *Graph Coloring*
Fuente: Parajuli (2021)

3.3.3. Métodos empleados en soluciones

Muchos de los problemas de optimización combinatoria relevantes son $\mathcal{NP}$-hard o $\mathcal{NP}$-complete, como los problemas de interés de la sección anterior, lo que

significa que por lo general no existe un algoritmo que los pueda resolver de forma eficiente (Hoos y Stutzle, 2004). En este contexto, la eficiencia de un algoritmo hace referencia a poder resolver el problema en tiempo polinomial en una máquina determinista. Sin embargo, existen formas de resolver el problema de manera práctica: encontrar una subclase del problema relevante a la aplicación que se pueda resolver eficientemente, usar un algoritmo de aproximación eficiente o usar métodos estocásticos. En el primer caso, puede existir una subclase o instancia del problema (no necesariamente $\mathcal{NP}$-hard) a la cual se reduce el problema original, por lo que puede existir un algoritmo eficiente que lo resuelva; en la situación que no se pueda reducir el problema a tal subclase, una opción es aceptar soluciones subóptimas. En el segundo caso, el uso de algoritmos de aproximación podría generar soluciones subóptimas (cercanas a la solución óptima) que se obtengan eficientemente. En el último caso, donde no exista un algoritmo de aproximación eficiente o se trata de un problema de decisión (donde no se puede aplicar el concepto de aproximación), se emplean algoritmos probabilísticos en vez de determinísticos.

Uno de los métodos estocásticos más empleados para resolver problemas combinatorios de gran complejidad son los algoritmos de tipo *Stochastic Local Search* (SLS). El proceso de *Local Search* inicia eligiendo una solución candidata inicial, y luego se pasa de forma iterativa de una solución candidata a otra solución candidata vecina, donde la decisión de búsqueda en cada paso está limitada a la información local. Dada la naturaleza estocástica de SLS, tanto la selección inicial como la decisión de búsqueda pueden ser aleatorias. Más aun, existe un tipo de SLS llamado *Population-based*, donde en vez de mantener solo una solución candidata en cada iteración del proceso de búsqueda, se mantienen una población de soluciones candidatas usualmente de tamaño fijo. Entre los métodos más conocidos de *Population-based* SLS se encuentran los algoritmos de *Ant-Colony Optmization* y los Algoritmos Evolutivos.

Ant-Colony Optimization (ACO) es un tipo de algoritmo que trata de imitar la forma en que las hormigas buscan comida y se comunican entre ellas para encontrar el camino más corto entre su nido y la comida. En ACO se tiene una población de hormigas artificiales que exploran el espacio de soluciones, donde cada hormiga construye una solución. En cada solución se selecciona el siguiente componente basado en una distribución de probabilidad, la cual es influenciada por la calidad de

los componentes (p. ej. la longitud del camino) y por el rastro de feromonas dejadas por otras hormigas Este rastro de feromonas representa una forma de comunicación entre hormigas, de modo que les permite seguir caminos que anteriormente han tenido éxito.

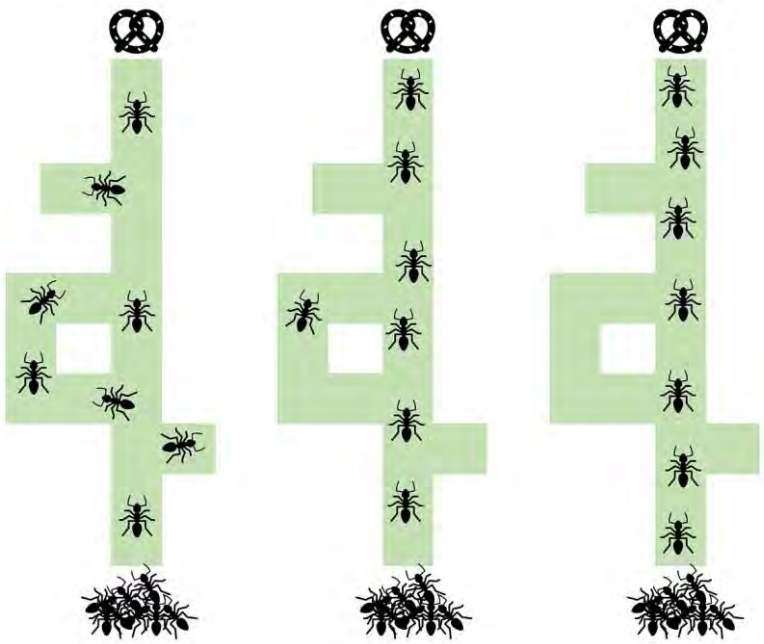

Figura 16. Representación de *Ant-Colony Optimization*
Fuente: de Harder (2023)

Por otro lado, los Algoritmos Evolutivos (EA) son una clase de algoritmos que está inspirada por los modelos de evolución natural de especies biológicas, donde por medio de los procesos de mutación, recombinación y selección del más apto, se desarrollan especies mejor adaptadas a la supervivencia en un determinado entorno. EA comienza con un conjunto de soluciones candidatas (población inicial), y luego de forma iterativa aplica las operaciones genéticas de selección (elección probabilística), mutación (pequeñas modificaciones aleatorias) y recombinación (combinación de información; el tipo más conocido llamado *crossover*). De esta manera, la población actual es parcial o completamente reemplazada por nuevas soluciones candidatas, formando nuevas poblaciones también llamadas generaciones. El tipo más conocido de EA para resolver problemas de optimización combinatoria es el Algoritmo Genético

(GA), donde las soluciones candidatas se representan mediante secuencias de bits de tamaño fijo.

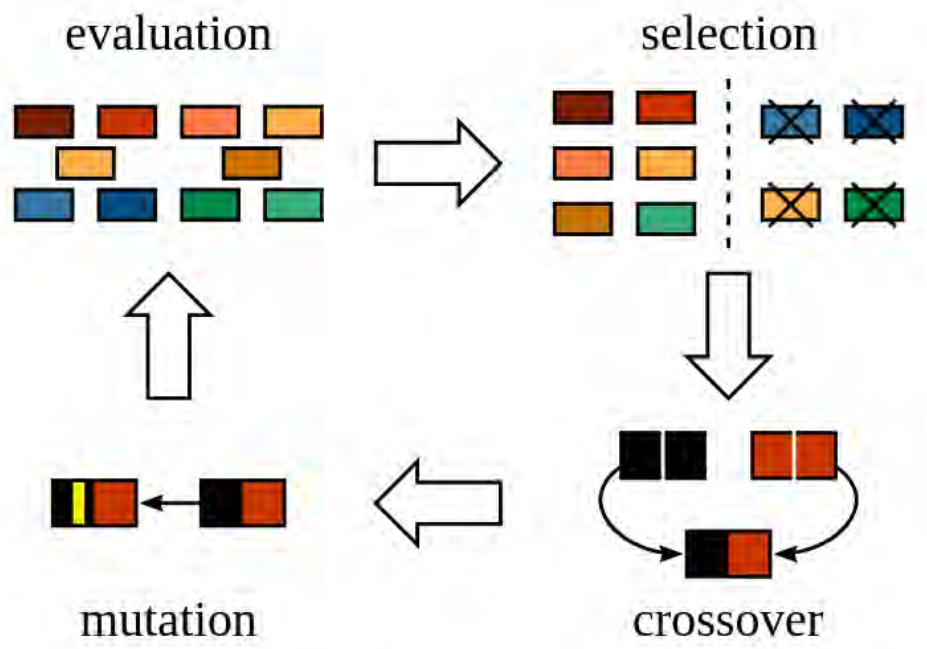

Figura 17. Operaciones genéticas empleadas en un Algoritmo Genético

Fuente: Cheng (2018)

SEGUNDA PARTE: DISEÑO METODOLÓGICO Y RESULTADOS

CAPÍTULO IV. DISEÑO E IMPLEMENTACIÓN

En el presente capítulo se presenta el diseño de la solución propuesta para el problema de *Instance Placement*. Para ello, primero se presenta el HAST, el orquestador de nube privada que la sección de Ingeniería de Telecomunicaciones emplea para crear los escenarios virtuales, tanto para los cursos de pregrado como para proyectos de investigación. Luego, se presenta tanto una taxonomía para los tipos de carga que se presentan en el HAST como el modelo propuesto para cada escenario. Finalmente, se muestra el diseño del algoritmo propuesto para solucionar el problema de *Instance Placement* con las distintas cargas de interés.

4.1. Hybrid Academic Scenario Testbed

El *Hybrid Academic Scenario Testbed* (HAST) es un orquestador de nube privada que permite desplegar escenarios o *slices* de red reales que requieran una alta performance (p. ej. clúster HPC) y permita obtener resultados de alta fidelidad (p. ej. *core* de red móvil con resultados reproducibles). En este contexto, un *slice* es un conjunto VMs que pertenecen a una aplicación o escenario de red determinado. Para ello, el HAST aprovecha tanto la escalabilidad de OpenStack (plataforma de nube privada sobre la cual está construida) como las optimizaciones en hardware para mejorar la performance. Adicionalmente existen escenarios donde no se puede emular la capa física (p. ej. WiFi o RAN móvil) o se desea evaluar el desempeño de equipos físicos; para dichos casos, el HAST permite integrar equipos físicos a la red virtual en capa 2 (modelo OSI) a 1/10 Gbps. Si bien la arquitectura del HAST presenta un módulo llamado *Scheduler* (Figura 18), actualmente se utiliza la asignación realizada por OpenStack, en el cual la unidad de asignación mínima son las VMs.

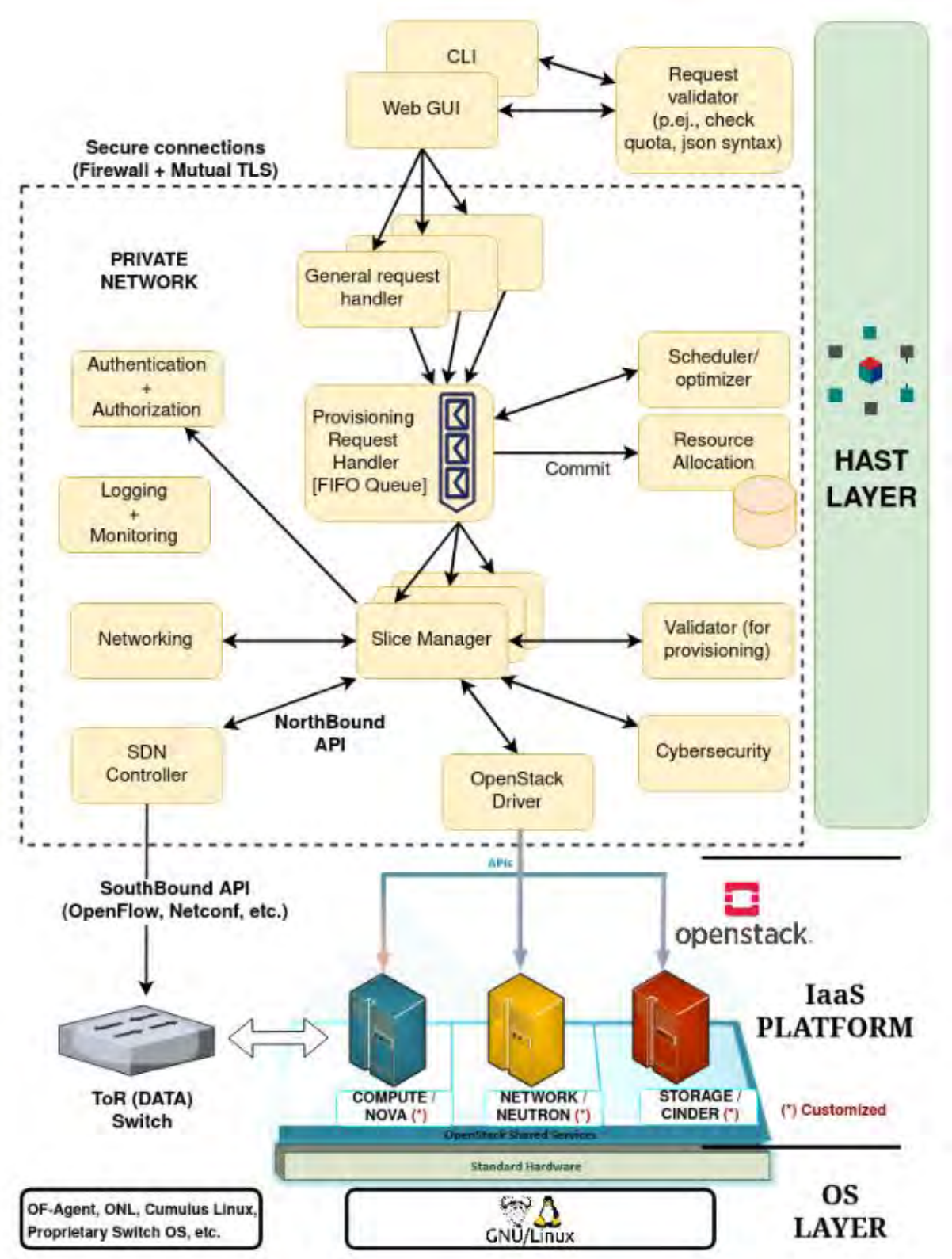

Figura 18. Arquitectura del HAST

Fuente: Elaboración propia

4.2. Escenarios de Interés

Tomando en consideración que el problema a resolver es cómo garantizar una alta performance y una alta fidelidad de los experimentos realizados cuando existen distintos tipos de cargas (otro término para referirse a los *slices*), se han identificado 3 escenarios de interés, basados en la utilización de recursos presente en los *datasets* de Shen et al. (2015). En el primer escenario se considera que solamente existen cargas que requieren acceder a los recursos de forma oportunista y se pueden encontrar en escenarios tradicionales de los entornos de *Cloud Computing*. Debido a ello, se les clasifica como cargas de baja prioridad.

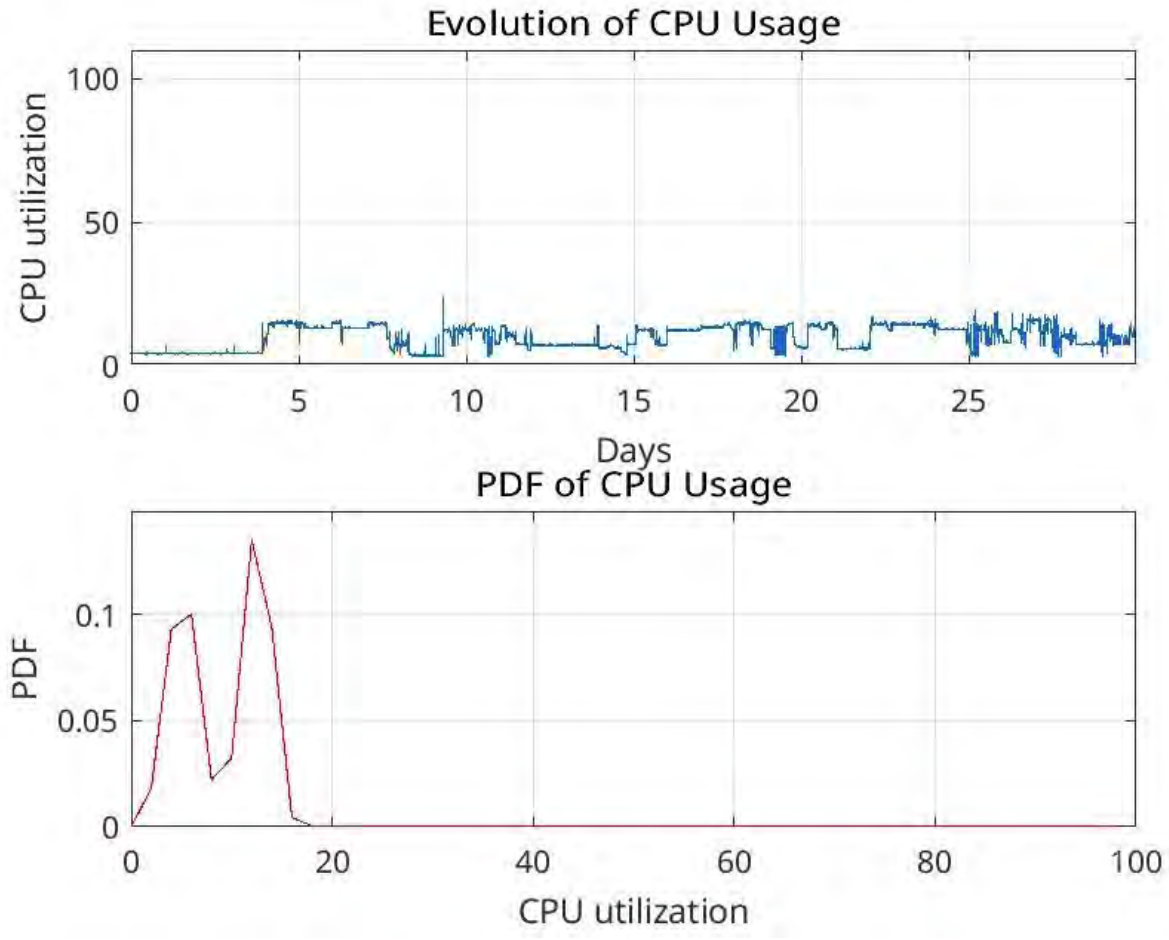

Figura 19. Uso del CPU y PDF de cargas de baja prioridad

Fuente: Elaboración propia

En el segundo escenario, las cargas de baja prioridad coexisten con otro tipo de cargas, las cuales utilizan por completo los recursos asignados. Estas cargas –clasificadas como de alta prioridad– son las que requieren una alta performance (p. ej. clústeres de HPC y Big Data) o alta fidelidad (p. ej. *core* de red móvil).

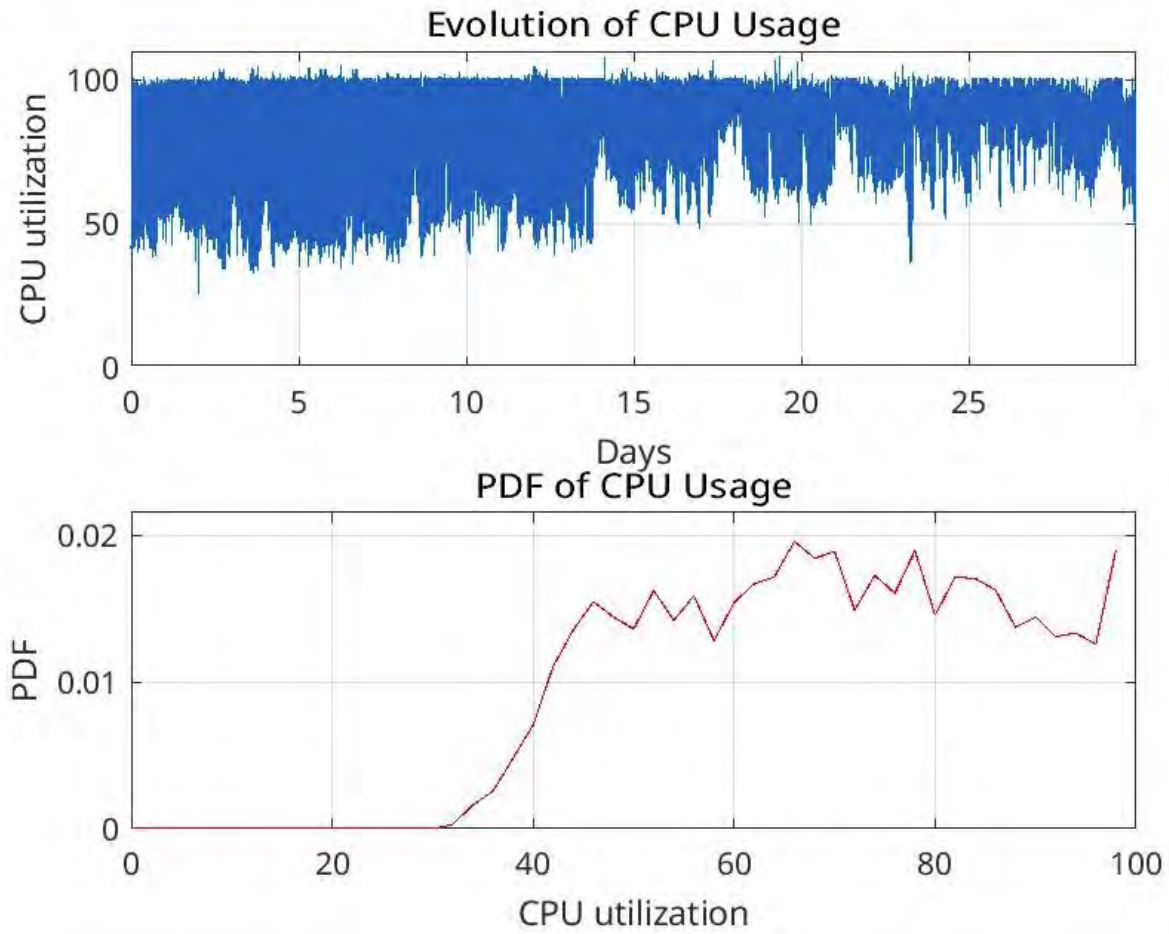

Figura 20. Uso del CPU y PDF de cargas de alta prioridad

Fuente: Elaboración propia

En el tercer escenario existen cargas que requieren una alta prioridad durante intervalos específicos de tiempo, y se vuelven de baja prioridad el resto del tiempo (p. ej. laboratorios de los cursos de pregrado, experimentos de tesis, etc.).

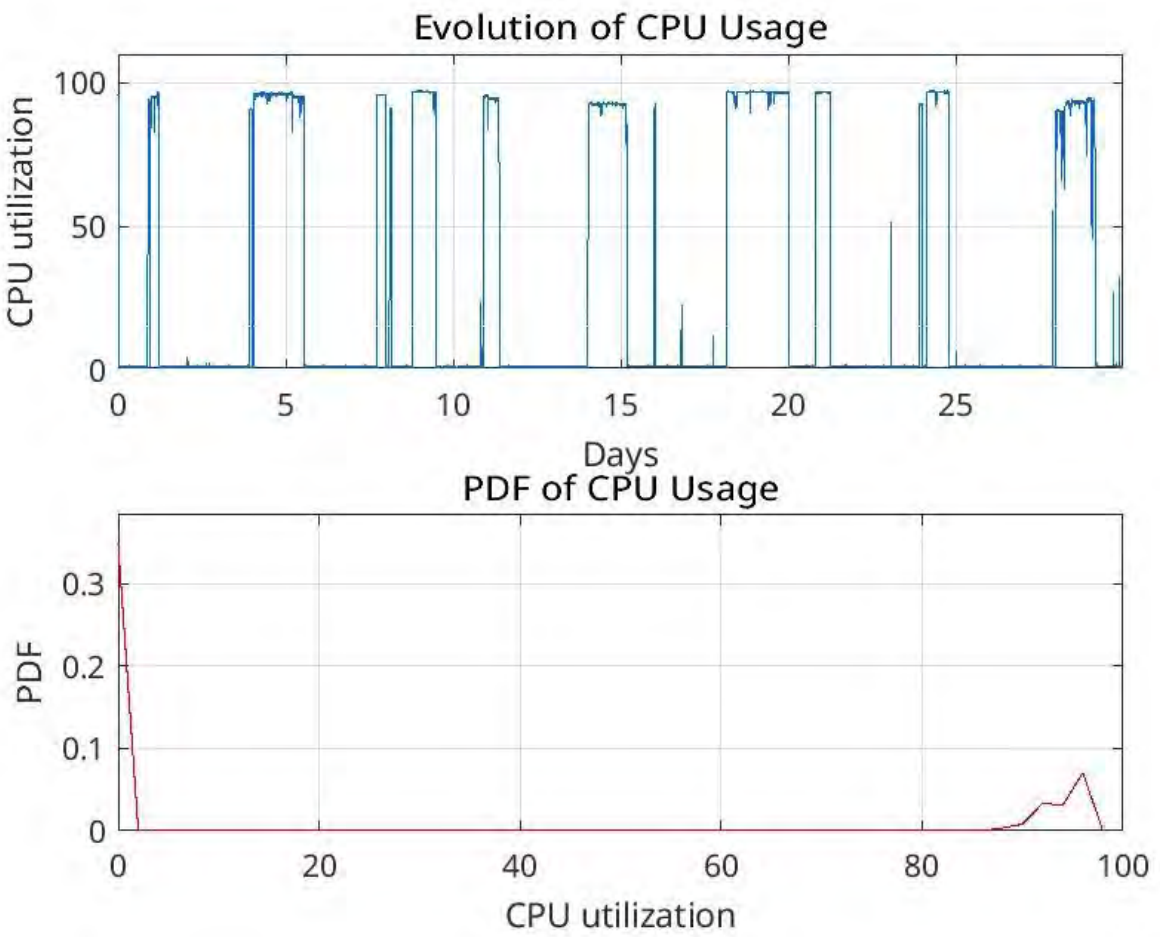

Figura 21. Uso del CPU y PDF de cargas de alta prioridad por intervalos

Fuente: Elaboración propia

Basado en los escenarios de interés anteriores, se definen 3 clases de cargas o *slices* basados en los escenarios descritos anteriormente: *Best Effort* (cargas con baja prioridad), *High Priority* (*slices* con alta prioridad permanente) y *Upgradable* (cargas que soportan reservaciones, las cuales tienen una alta prioridad durante el intervalo de reservación especificado). Las instancias de los *slices Best Effort* (BE) se caracterizan por tener un acceso oportunista a los recursos asignados, mientras que las instancias de tipo *High Priority* (HP) o *Upgradable* (UP) tienen acceso garantizado a los recursos.

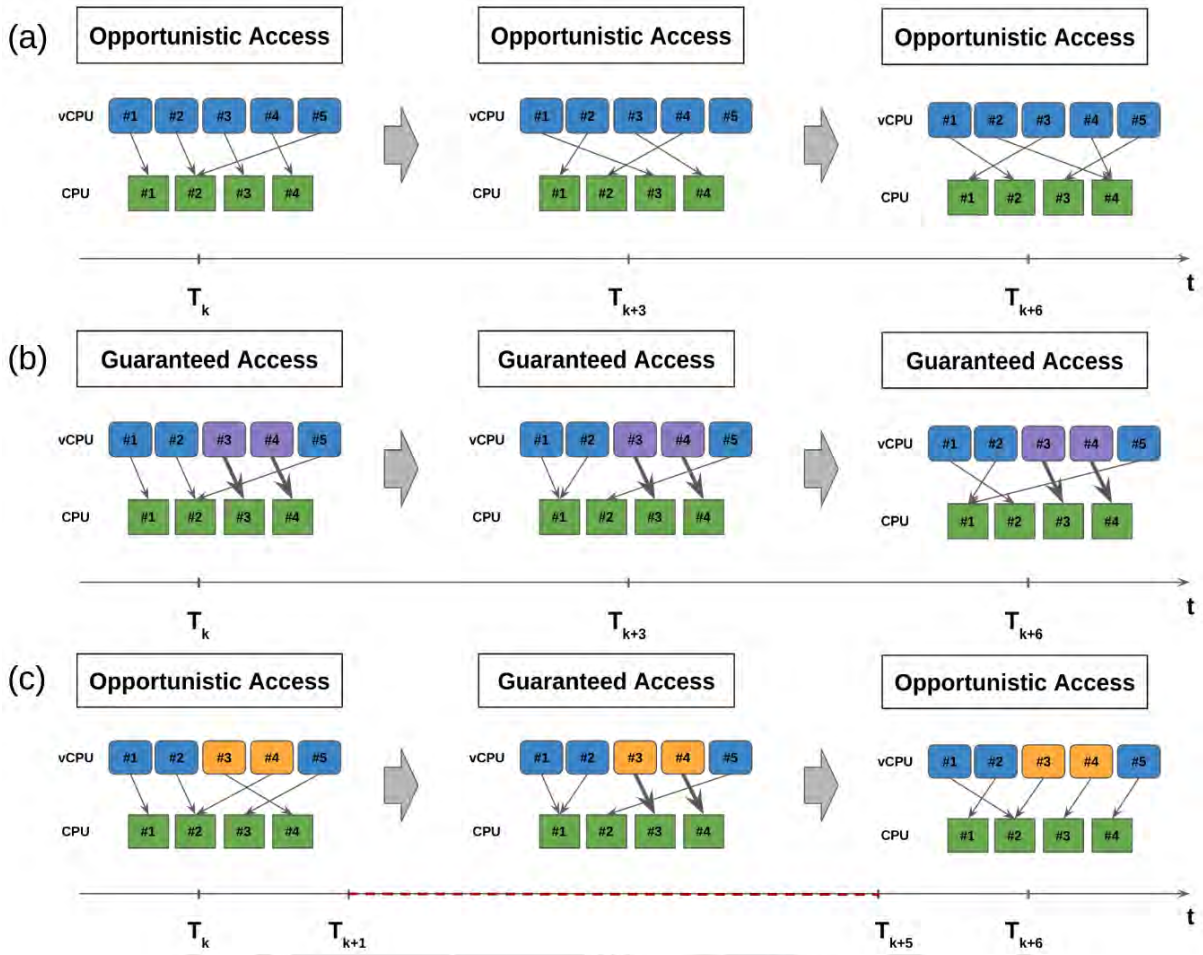

Figura 22. Acceso a recursos de cargas (a) BE, (b) HP y (c) UP

Fuente: Elaboración propia

4.3. Análisis y Planteamiento del Problema

Antes de plantear el problema de *Slice Placement*, primero se define el problema de VM o *Instance Placement*. Este se modela como una variación del problema 0/1 de *Knapsack* (Cacchiani et al., 2022a), donde la variable de decisión puede tomar únicamente los valores 0 o 1. A diferencia del problema inicial, existe más de un *knapsack* y cada uno tiene múltiples dimensiones, convirtiendo el problema en *Multiple Multidimensional Knapsack Problem* o MMdKP (Cacchiani et al., 2022b). Formalmente, se tienen M instancias a ser ubicadas en uno de los N servidores físicos (knapsacks), los cuales tienen una capacidad determinada para cada una de las D dimensiones. Adicionalmente, se asume que el problema es de tipo *offline*; es decir,

todos los parámetros del problema se conocen de antemano (cantidad de instancias, recursos que consume cada una, tiempos de llegada, entre otros). MMdKP es un problema de clase $\mathcal{NP}$-hard pues –al ser una variante del problema de *Knapsack*– es tan complejo como él.

Para poder analizar cada uno de los escenarios de interés, inicialmente se plantea el problema de optimización de tipo *Integer Linear Programming* (ILP) para un tipo de carga base llamado *Guaranteed Resources*, donde se garantiza el acceso a los recursos asignados; más adelante se introducirán algunas variaciones al problema base para modelar los problemas de los tipos de carga de interés, tanto de forma individual como de sus combinaciones. Por otro lado, en adelante se utilizará la misma nomenclatura que en la literatura de VMP, donde se emplean los términos máquina virtual (VM, para nuestro caso un tipo de instancia) y servidor físico (PM).

En la clase *Guaranteed Resources* (GR), la cantidad de recursos $w_i^k(t)$ que cada VM solicita es determinístico (su valor no varía en el tiempo, $w_i^k(t) = w_i^k$), por lo que se tiene el problema de optimización combinatoria mostrado a continuación:

maximizar
$$\sum_{i=1}^{M}\sum_{\mathrm{j}=1}^{N} p_i x_{ij} \tag{1}$$

sujeto a
$$\sum_{i=1}^{M} w_i^k x_{ij} \leq C_j^k \,, j = \{1,2,\dots,N\}, k = \{1,2,\dots,D\} \tag{2}$$

$$\sum_{j=1}^{N} x_{ij} \leq 1, i = \{1,2,\dots,M\} \tag{3}$$

$$x_{ij} = \{0,1\}, i = \{1,2,\dots,M\}, j = \{1,2,\dots,N\} \tag{4}$$

En la Tabla II se muestra la explicación de cada variable empleada. En este problema, el objetivo es maximizar el beneficio total de colocar las VMs en PMs, tal que no se exceda la capacidad C_j^k de cada PM y que las VMs solo pueden estar ubicados como máximo en un solo PM.

Tabla II. Significado de variables del modelo para cargas GR

Variable	Significado
i	VM i
j	PM j
k	dimensión k
M	cantidad total del VMs
N	cantidad total del PMs
D	cantidad total de dimensiones
p_i	beneficio de colocar la VM i en un PM
x_{ij}	variable de decisión, si la VM i se ubica en el PM j
w_i^k	peso de la VM i en la dimensión k
C_j^k	capacidad del PM j en la dimensión k

En la clase de cargas BE, se tiene un enfoque estocástico del problema inicial. La cantidad de recursos $w_i^k(t)$ que cada VM i solicita se modela como una variable aleatoria w_i^k con media $\overline{w_i^k}$ y varianza $Var\{w_i^k\}$. Utilizando el Teorema del Limite Central (w_i^k independientes), la combinación de las cargas de todas las VMs se aproxima a una variable aleatoria W_T^k con una Función de Distribución Acumulada (CDF) de una distribución normal.

$$W_T^k = \sum_{i=1}^{M} w_i^k \tag{5}$$

$$E\{W_T^k\} \approx \sum_{i=1}^{M} \overline{w_i^k} \tag{6}$$

$$Var\{W_T^k\} \approx \sum_{i=1}^{M} Var\{w_i^k\} \tag{7}$$

Entonces, se desea que la probabilidad de que la combinación de todas las cargas BE (W_T^k) exceda la capacidad del servidor (C_j^k) debe ser menor o igual al valor máximo permitido (ε):

$$P\left(W_T^k > C_j^k\right) \leq \varepsilon \tag{8}$$

Sea ω una variable aleatoria que representa los valores estandarizados de W_T^k; entonces la ecuación (8) se convierte en

$$P\left(\omega > \frac{C_j^k - E\{W_T^k\}}{\sqrt{Var\{W_T^k\}}}\right) \leq \varepsilon$$

Sea $\Phi(\omega)$ la Función de Distribución Acumulada (CDF) y $Q(\omega)$ la Función de Distribución Acumulada Complementaria (CCDF) de la distribución normal estándar $(1 - \Phi = Q)$; entonces

$$\Phi\left(\omega = \frac{C_j^k - E\{W_T^k\}}{\sqrt{Var\{W_T^k\}}}\right) \geq 1 - \varepsilon$$

$$1 - \Phi\left(\omega = \frac{C_j^k - E\{W_T^k\}}{\sqrt{Var\{W_T^k\}}}\right) \leq \varepsilon$$

$$Q\left(\omega = \frac{C_j^k - E\{W_T^k\}}{\sqrt{Var\{W_T^k\}}}\right) \leq \varepsilon \tag{9}$$

Sea Q^{-1} la función inversa de Q y $z = Q^{-1}(\varepsilon)$; entonces, la expresión (9) se convierte en

$$\frac{C_j^k - E\{W_T^k\}}{\sqrt{Var\{W_T^k\}}} \geq Q^{-1}(\varepsilon) = z$$

$$E\{W_T^k\} + z\sqrt{Var\{W_T^k\}} \leq C_j^k \tag{10}$$

lo que es equivalente a

$$\sum_{i=1}^{M} \overline{w_i^k}\, x_{ij} + z\sqrt{\sum_{i=1}^{M} Var\{w_i^k\}} \leq C_j^k \qquad (11)$$

Considerando lo anterior, la parte izquierda de la inecuación de (2) se reemplaza por (11), convirtiendo el problema de optimización en:

maximizar

$$\sum_{i=1}^{M}\sum_{j=1}^{N} p_i x_{ij} \qquad (12)$$

sujeto a

$$\sum_{i=1}^{M} \overline{w_i^k}\, x_{ij} + z\sqrt{\sum_{i=1}^{M} Var\{w_i^k\}} \leq C_j^k, \qquad (13)$$

$$j = \{1,2,\ldots,N\}, k = \{1,2,\ldots,D\}$$

$$\sum_{j=1}^{N} x_{ij} \leq 1, i = \{1,2,\ldots,M\} \qquad (14)$$

$$x_{ij} = \{0,1\}, i = \{1,2,\ldots,M\}, j = \{1,2,\ldots,N\} \qquad (15)$$

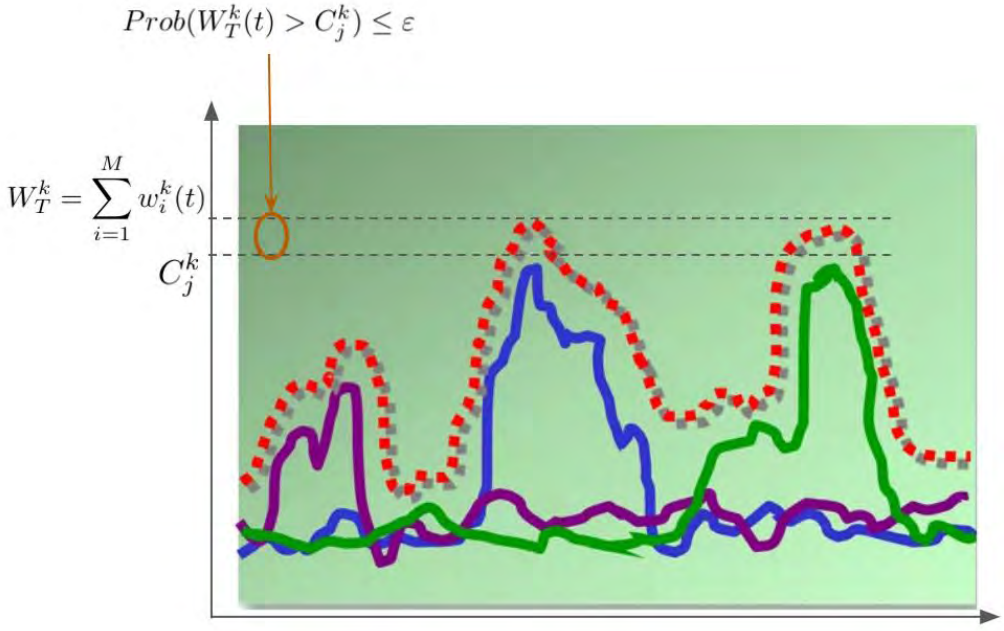

Figura 23. Ejemplo de carga *Best Effort*

Fuente: Egli (2015)

En la clase de cargas HP, el consumo de recursos $w_i^k(t)$ también varía con el tiempo, pero se garantiza que el recurso pueda utilizarse por completo cuando se requiera. Debido a ello, se toma en cuenta el valor máximo de la combinación de cargas

$$\max_t\{w_i^k(t)\} = w_{i,MAX}^k \tag{16}$$

Considerando lo anterior, se reemplaza la parte izquierda de la inecuación de (2) por (16); así, el problema de optimización se convierte en:

maximizar $$\sum_{i=1}^{M}\sum_{j=1}^{N} p_i x_{ij} \tag{17}$$

sujeto a $$\sum_{i=1}^{M} w_{i,MAX}^k \, x_{ij} \leq C_j^k, j = 1,2,\dots,N, k = \{1,2,\dots,D\} \tag{18}$$

$$\sum_{j=1}^{N} x_{ij} \leq 1, i = \{1,2,\dots,M\} \tag{19}$$

$$x_{ij} = \{0,1\}, i = \{1,2,\dots,M\}, j = \{1,2,\dots,N\} \tag{20}$$

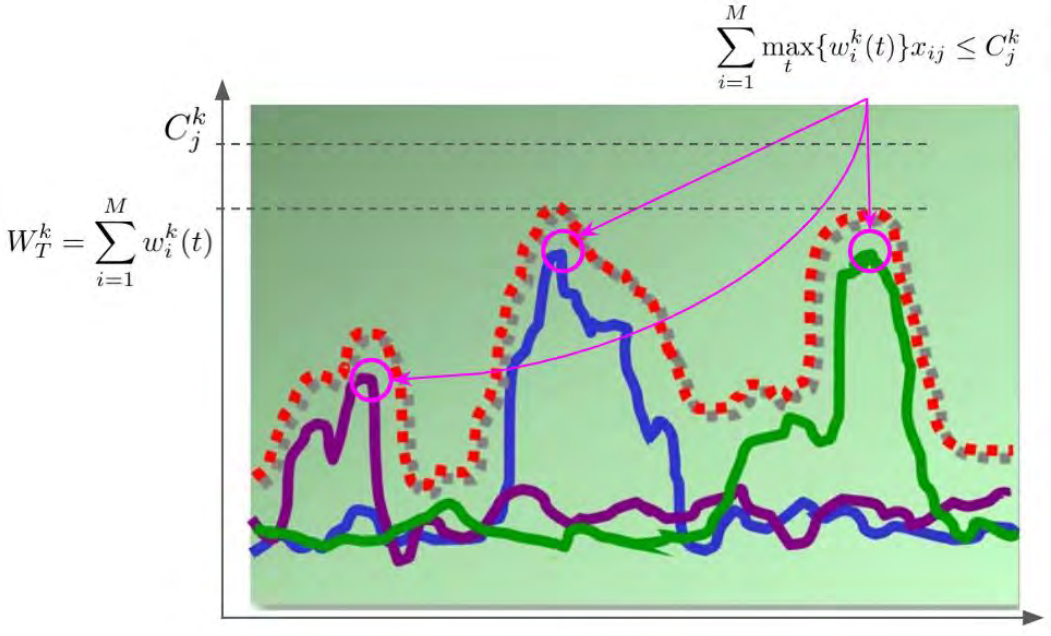

Figura 24. Ejemplo de carga *High Priority*

Fuente: Egli (2015)

En el caso de las cargas de clase UP, el consumo de recursos $w_i^k(t)$ también depende del tiempo. Sin embargo, debido a que tiene un comportamiento como una carga de HP por un intervalo de tiempo bien definido y un comportamiento BE fuera de él, se debe obtener el consumo de recursos máximo al combinar todas las cargas, y este no debe exceder la capacidad C_j^k del servidor; es decir

$$\max_t\{\sum_{i=1}^{M} w_i^k(t)x_{ij}\} \leq C_j^k \tag{21}$$

Entonces, se reformula el problema de optimización inicial reemplazando la parte izquierda de (2) por (21):

maximizar
$$\sum_{i=1}^{M}\sum_{j=1}^{N} p_i x_{ij} \tag{22}$$

sujeto a
$$\max_t\{\sum_{i=1}^{M} w_i^k(t)x_{ij}\} \leq C_j^k, \tag{23}$$
$$j = 1,2,\dots,N, k = \{1,2,\dots,D\}$$

$$\sum_{j=1}^{N} x_{ij} \leq 1, i = \{1,2,\dots,M\} \tag{24}$$

$$x_{ij} = \{0,1\}, i = \{1,2,\dots,M\}, j = \{1,2,\dots,N\} \tag{25}$$

En el caso que se tiene tanto cargas HP como UP, se modela el consumo de CPUs como si fuese el número cromático $\chi(G_j)$ de un grafo auxiliar $G_j = (V_j, E_j)$. El grafo G_j se construye de la siguiente manera:

1. Cada vértice representa un virtual CPU (vCPU) de una VM que pertenece a un *slice* de tipo HP o UP, ubicada en el PM j.
2. Un enlace conecta 2 vértices si y solo si existe un instante t en el cual los slices asociados a estos vCPUs tienen una alta prioridad; por ejemplo, 2 *slices* UP con tiempos de reservación que se traslapan, o un *slice* UP cuyo tiempo de reservación se traslapa con el tiempo en que esta creado un *slice* de tipo HP.

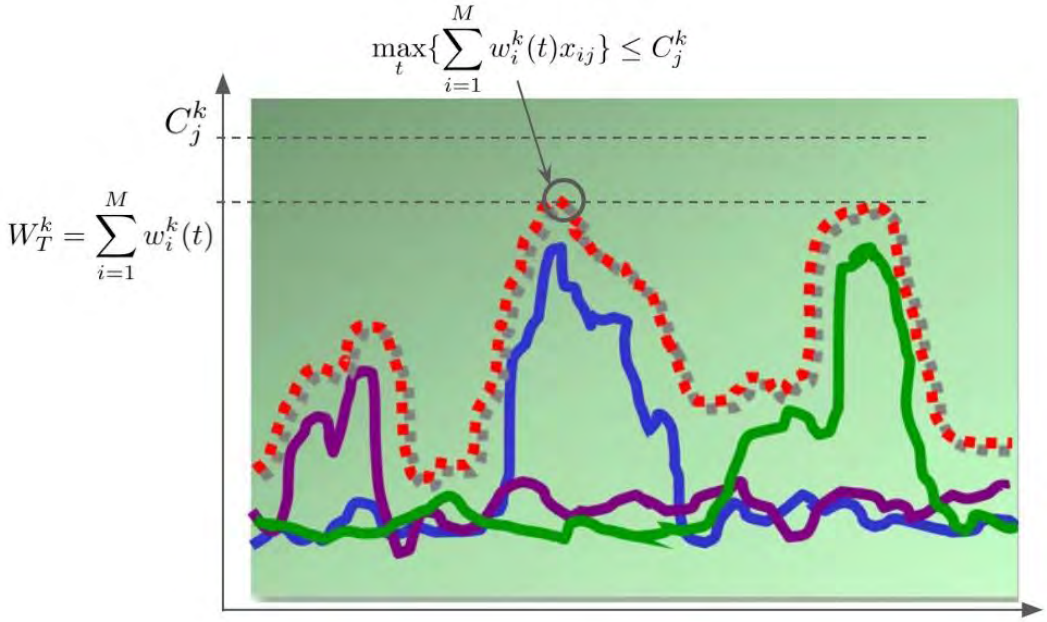

Figura 25. Ejemplo de carga *Upgradable*

Fuente: Egli (2015)

En el grafo resultante, los vértices asociados a la misma VM forma un *clique*. También los vértices asociados a VMs en el mismo *slice* HP o UP forman un *clique*. Adicionalmente, los vértices asociados a VMs de *slices* HP o UP que se traslapan forman un *clique*. Finalmente, el número de CPUs físicos requeridos en el servidor j para satisfacer todas las cargas HP y UP durante el tiempo en que están creados es igual a $\chi(G_j)$, el número cromático del grafo G_j. Entonces se puede reemplazar la parte izquierda de (2) por el número cromático. Con ello, el problema de optimización se convierte en:

maximizar $$\sum_{i=1}^{M}\sum_{j=1}^{N} p_i x_{ij} \quad (26)$$

sujeto a $$\chi(G_j) \leq C_j^k, j = 1,2,\dots,N, k = \{1,2,\dots,D\} \quad (27)$$

$$\sum_{j=1}^{N} x_{ij} \leq 1, i = \{1,2,\dots,M\} \quad (28)$$

$$x_{ij} = \{0,1\}, i = \{1,2,\dots,M\}, j = \{1,2,\dots,N\} \quad (29)$$

donde la variable de decisión $x_{ij} = 1$ si los vCPUs de la VM i son considerados como vértices de G_j; caso contrario ($x_{ij} = 0$), son ignorados.

Por último, en el caso que se tiene una combinación de cargas BE, HP y UP, se formula el problema de optimización reemplazando la parte izquierda de (2) por una combinación de (11), (16) y (21):

maximizar
$$\sum_{i=1}^{M}\sum_{\mathrm{j}=1}^{N} p_i x_{ij} \tag{30}$$

sujeto a
$$\sum_{i\in L_{BE}} \overline{w_i^k} x_{ij} + z\sqrt{\sum_{i\in L_{BE}} Var\{w_i^k\}} + \sum_{i\in L_{HP}} w_{i,MAX}^k x_{ij} + \max_t\{\sum_{L_{UP}} w_i^k(t) x_{ij}\} \le C_j^k, \tag{31}$$

$$j = 1,2,\dots,N, k = \{1,2,\dots,D\}, L = L_{BE} \cup L_{HP} \cup L_{UP}$$

$$\sum_{j=1}^{N} x_{ij} \le 1, i = \{1,2,\dots,M\} \tag{32}$$

$$x_{ij} = \{0,1\}, i = \{1,2,\dots,M\}, j = \{1,2,\dots,N\} \tag{33}$$

el cual es una combinación de los casos donde se tienen de forma independientes carga BE, HP y UP. Cabe resaltar que L es un conjunto que representa la VMs a ser ubicadas en los servidores, y está conformado por la unión de los subconjuntos L_{BE} (cargas *Best Effort*), L_{HP} (cargas *High Priority*) y L_{UP} (cargas *Upgradable*).

Por último, para pasar del problema de *VM Placement* a *Slice Placement*, se debe considerar que un *slice* se considera ubicado si y solo si todas las VMs que lo conforman se han ubicado en un servidor físico. Para reflejar este cambio, se debe modificar las funciones objetivo de cada caso para incluir la variable de decisión y_h, la cual indica si el *slice* h es admitido ($y_h = 1$) o no ($y_h = 0$). Es decir, las ecuaciones (1), (12), (17), (22), (26) y (30) se convierten en:

maximizar
$$\sum_{h=1}^{H} p_h y_h \tag{34}$$

donde p_h es el beneficio de colocar todas las VMs que pertenecen al slice h, así como y_h es una variable de decisión que indica si se colocan todas las VMs del *slice* en algún PM ($y_h = 1$ o no se coloca ninguna ($y_h = 0$). Esto se expresa con la siguiente restricción

$$\sum_{j=1}^{N} x_{ij} \geq y_h \, , i \in S_h, h = \{1,2, \ldots, H\} \qquad (35)$$

donde S_h es el *slice* al cual pertenece la VM i y H hace referencia al número total de *slices*.

4.4. Algoritmo Propuesto

Para resolver los problemas de *Instance Placement*, se emplea un algoritmo genético, el cual codifica los cromosomas como se muestra en la Figura 26. Para poder realizar los cálculos del *fitness* en cada generación del algoritmo genético, se representa el cromosoma en su forma matricial, como se muestra en la Figura 27. Esto se debe a que los pesos de cada VM (w_i^k), las capacidades de cada servidor C_j^k y el *profit* de cada VM (p_i) están representados como matrices.

Cromosoma (PM)	2	0	3	5	1	0	0	2	4	0
Posición (VM)	1	2	3	4	5	6	7	8	9	10

Figura 26. Codificación del cromosoma del Algoritmo Genético
Fuente: Elaboración propia

Para poder detallar el funcionamiento del algoritmo genético, en la Figura 28 se muestra su diagrama de bloques. A continuación, se detalla el funcionamiento de cada etapa:

PM / VM	1	2	3	4	5
1	0	1	0	0	0
2	0	0	0	0	0
3	0	0	1	0	0
4	0	0	0	0	1
5	1	0	0	0	0
6	0	0	0	0	0
7	0	0	0	0	0
8	0	1	0	0	0
9	0	0	0	1	0
10	0	0	0	0	0

Figura 27. Representación matricial del cromosoma

Fuente: Elaboración propia

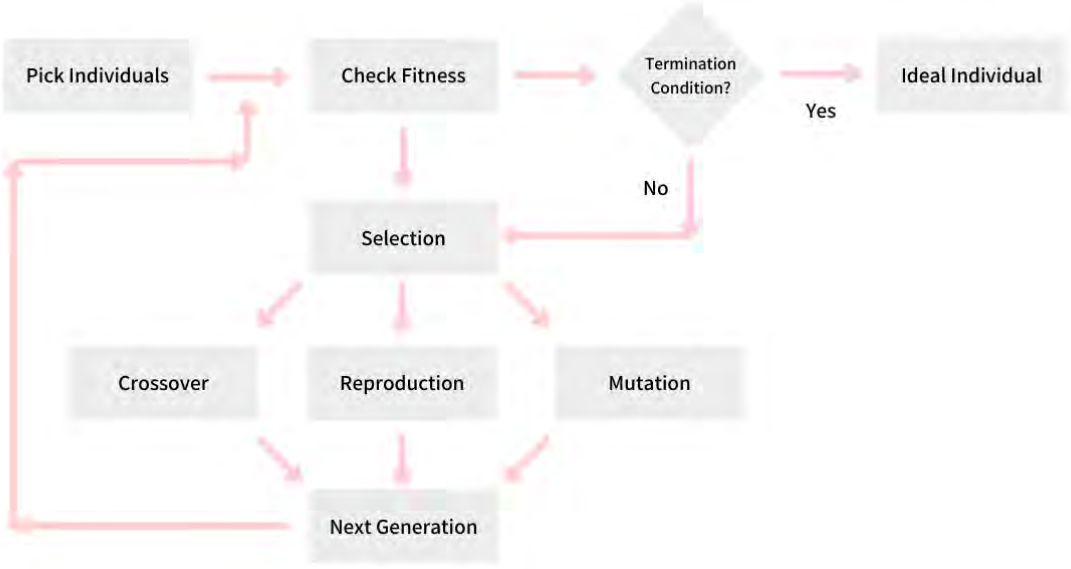

Figura 28. Etapas del Algoritmo Genético propuesto.

Fuente: Bhayani (2022)

1. *Pick Individuals*: se selecciona la población inicial, la cual está formada por 25 cromosomas con genes aleatorios. Estos cromosomas tienen el máximo *fitness* que cumpla con las restricciones del problema; en caso que no se cumpla alguna restricción, se quitan de forma aleatoria entre 2 y 5 genes de forma recurrente hasta cumplir todas las restricciones del problema. Se utilizó este método pues, al utilizar soluciones cercanas a máximos locales, se necesitaron menos generaciones para que el algoritmo genético obtuviera una solución *feasible*. Adicionalmente, a los elementos la población inicial también se les conoce como la generación 0.

2. *Check Fitness*: Se calcula el *fitness* de cada uno de los cromosomas de la generación actual, el cual es la suma de los beneficios de cada gen seleccionado. El beneficio de cada gen se definió como el cuadrado del número de *cores* de la VM que representa, considerando que las máquinas de *clusters* de alta performance suelen requerir 8 o más vCPUs, mientras que las VMs que se emplean en cargas típicas de Cloud utilizan entre 1 y 2 *cores*. Así, el beneficio de incluir VMs de aplicaciones de alta performance es mayor que el de las VMs de aplicaciones *Cloud* genéricas.

3. *Selection*: este proceso tiene como objetivo elegir 2 cromosomas de la generación actual para que sean los padres de los cromosomas de la siguiente generación. Para ello, se utilizó un *tournament selection* donde se seleccionan de forma aleatoria 4 cromosomas distintos y se agrupan en pares para competir entre sí; los 2 cromosomas que tengan el mayor *fitness* ganan el torneo y se convierten en los padres de la siguiente generación

4. *Reproduction:* el proceso genético de reproducción indica que los padres se reproducirán en hijos con los mismos cromosomas que ellos; de esta manera, se tienen 2 hijos idénticos a los padres.

5. *Crossover:* en el proceso genético de *crossover* los genes de los padres se mezclan para generar hijos que tengan genes de ambos padres. Para ello, se realizó un *k-point crossover* con $k = 4$, donde se eligieron 4 índices de genes aleatorios con los cuales se construyeron 5 intervalos; se combinan los intervalos pares de un padre con los intervalos impares del otro padre y viceversa, obteniendo 2 hijos con la combinación de genes de ambos padres.

6. *Mutation:* en el proceso genético de mutación se eligen algunos genes de forma aleatoria y se alteran: si el gen está presente, se quita del cromosoma; si el gen no está presente, se selecciona en el cromosoma (en algún PM).

7. *Next Generation:* para obtener la siguiente generación del algoritmo genético se realiza lo siguiente: primero se eligen a los 2 padres mediante el proceso de selección; luego se decide si los hijos se generan como resultado del proceso de reproducción (probabilidad de 0%) o de la combinación de los procesos de *crossover* (probabilidad de 100%) y mutación (probabilidad 5%); estos pasos se repiten hasta que la cantidad de cromosomas presentes en la siguiente generación sea igual al tamaño de la población, en este caso 25. Cabe resaltar que en caso algún cromosoma no cumpla con las restricciones (*unfeasible*), se le quitan genes de forma aleatoria hasta que las satisfaga; así, se pueden mantener genes de las soluciones *unfeasible* que pueden mejorar el *fitness*. Una vez se tenga la siguiente generación con el tamaño de población deseado, se procede a repetir las etapas de *Check Fitness*, *Selection*, *Reproduction*, *Crossover* y *Mutation* las veces indicadas por el valor máximo de generaciones (en este caso 40,000).

8. *Termination Condition*: al llegar al valor máximo de generaciones, el algoritmo devuelve la solución que haya obtenido el mejor fitness a lo largo de todas las generaciones (*Ideal Individual*).

4.5. Generación de Data de las Cargas

Como punto de partida de las pruebas a realizar, se deben modelas lar cargas que el algoritmo de *scheduling* deberá ubicar en los servidores físicos. Para ello, se utilizó la data presente en la base de datos del OpenStack instalado en el *rack* Hitachi, ubicado en el laboratorio 305 del pabellón V. Esta data presenta el histórico de VMs creadas en dicho entorno de nube privada, y consta de 5,556 registros de VMs desde agosto del 2021. A partir de ella, obtuvo el histograma del uso de recursos, como se muestra en la Figura 29. Para cada una de las clases de VMs (el término “clase” hace referencia al número de CPUs solicitados por cada VM), se obtuvo el histograma del uso de memoria RAM y disco de almacenamiento, como se observa en la Figura 30. En ambas figuras se observa que existen VMs cuyo disco solicitado es 0, debido a

que dichas máquinas virtuales no utilizan el almacenamiento local al servidor donde se ubican, sino el almacenamiento del *Storage Area Network* (SAN). En base a los histogramas anteriores, se obtuvo la función de masa de probabilidad (PMF) experimental tanto de las clases de VMs como del uso de RAM y disco de almacenamiento por cada clase.

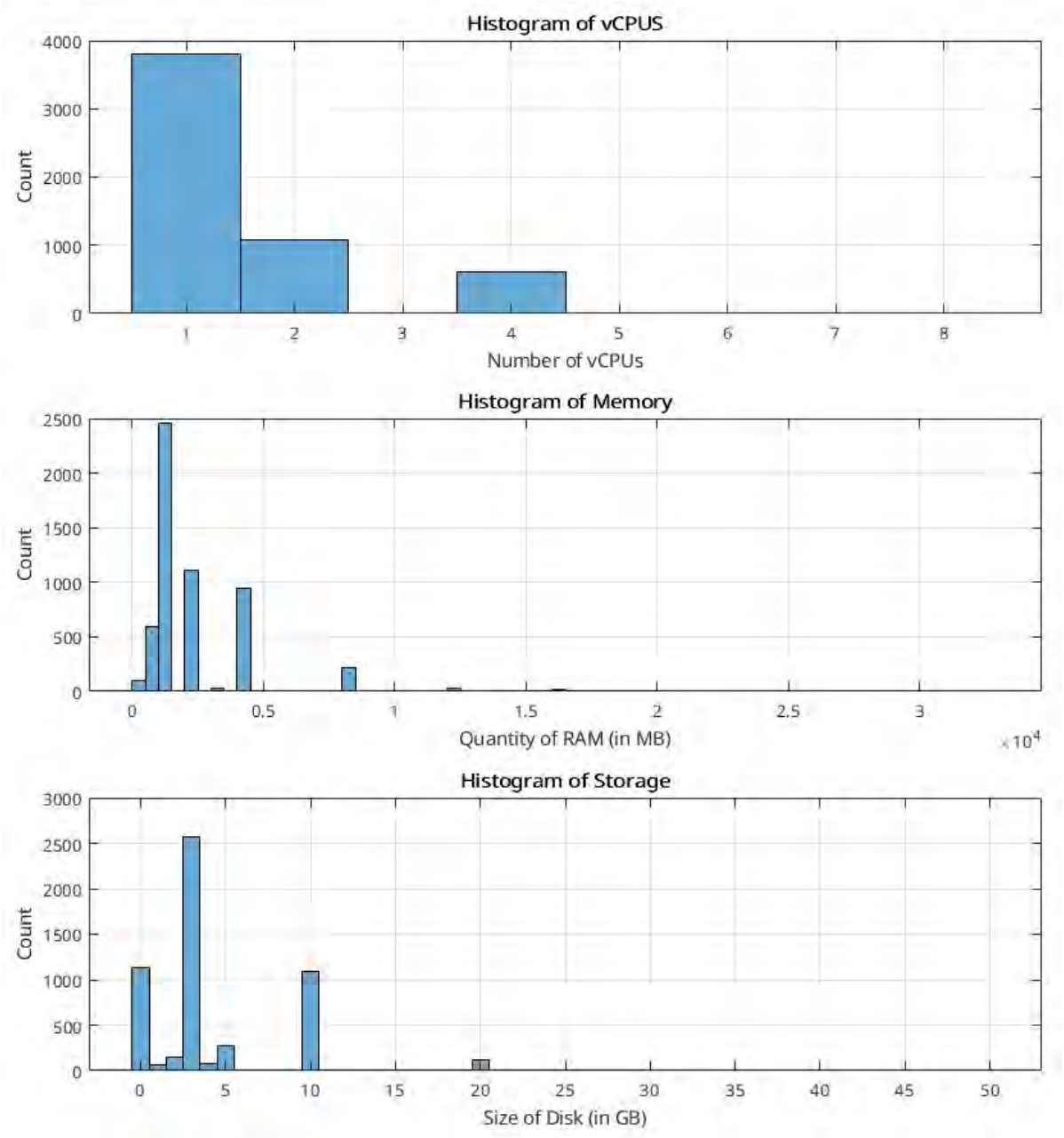

Figura 29. Histograma de recursos de VM.

Fuente: Elaboración propia

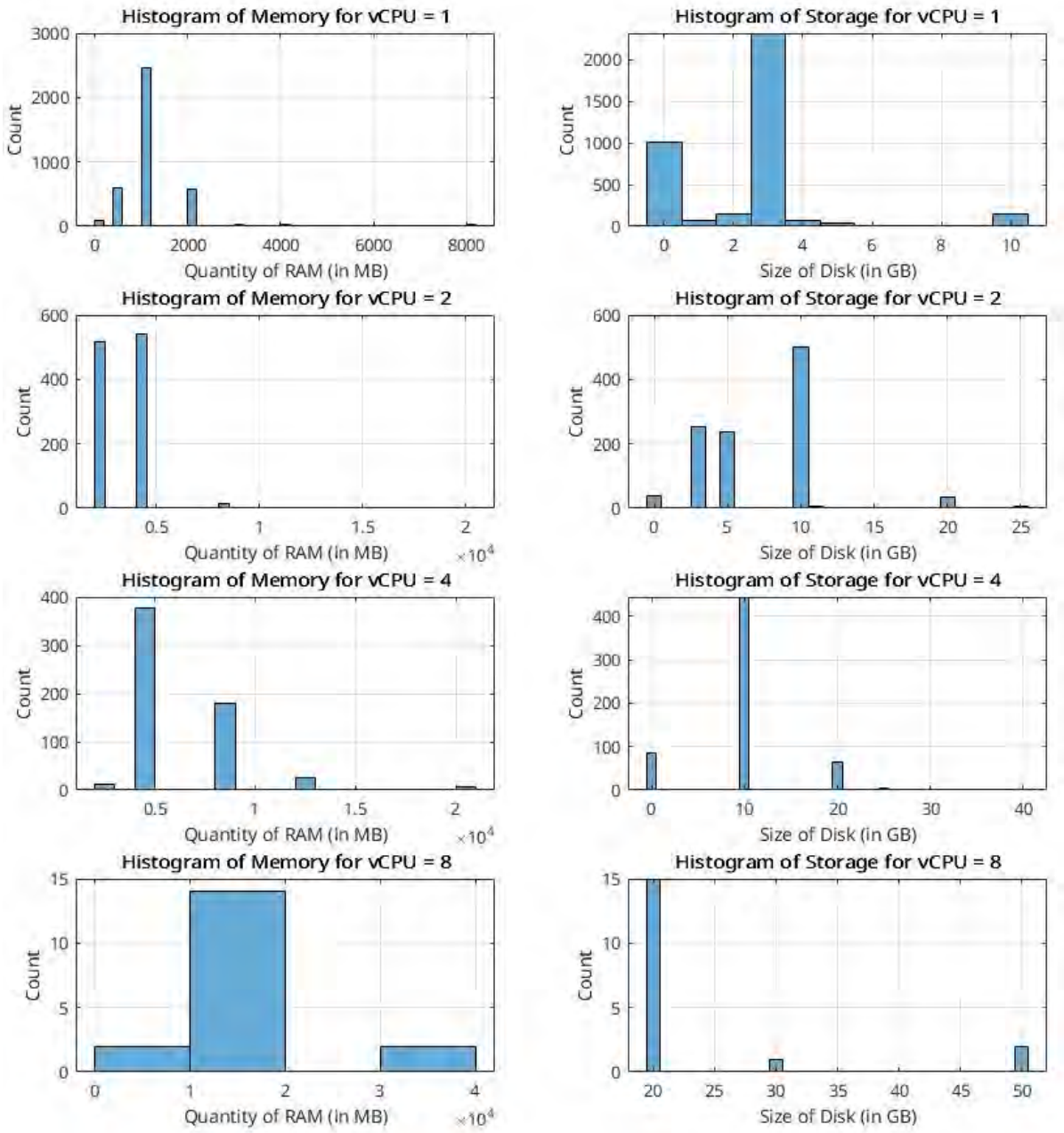

Figura 30. Histograma de RAM y disco de almacenamiento por clase de VM. Elaboración propia

CAPÍTULO V. ANÁLISIS Y DISCUSIÓN DE RESULTADOS

5.1. Comparación de scheduling de cargas *Best-Effort* con OpenStack

En la presente sección se presenta un análisis entre el *scheduling* propuesto para las cargas *Best-Effort* y el realizado por OpenStack, la cual tiene como objetivo comparar el número de instancias que se pueden tener con cada uno, así como entender las ventajas y desventajas de utilizarlos. Para ello, se considerará la ecuación 10 definida anteriormente para un valor de k que haga referencia a los *cores*.

Primero se considera que existe un solo tipo de carga; entonces, la ecuación 10 se convierte en

$$n_1\mu_1 + z\sqrt{n_1{\sigma_1}^2} \leq C_j \rightarrow n_1\mu_1 + z\sigma_1\sqrt{n_1} \leq C_j \quad (36)$$

Como $n_1$hace referencia al número de VMs, este puede tomar valores entre 0 y n_{max}, donde n_{max} hace referencia al valor máximo que puede tomar n_1. Para hallar n_{max}, se plantea lo siguiente:

$$n_1\mu_1 + z\sigma_1\sqrt{n_1} = C_j \quad (37)$$

Si $n_1 = x^2$, entonces

$$\mu_1 x^2 + z\sigma_1 x = C_j \rightarrow \mu_1 x^2 + z\sigma_1 x - C_j = 0 \quad (38)$$

Usando la fórmula general para resolver una ecuación cuadrática, se tiene que

$$x = \frac{-z\sigma_1 \pm \sqrt{(z\sigma_1)^2 + 4\mu_1 C_j}}{2\mu_1} \quad (39)$$

Considerando que $0 \leq z, \sigma_1$ y que x debe ser positivo, entonces

$$x = \frac{-z\sigma_1 + \sqrt{(z\sigma_1)^2 + 4\mu_1 C_j}}{2\mu_1} \quad (40)$$

Reemplazando x por $\sqrt{n_1}$

$$\sqrt{n_1} = \frac{-z\sigma_1 + \sqrt{(z\sigma_1)^2 + 4\mu_1 C_j}}{2\mu_1} \rightarrow n_1 = \left[\frac{-z\sigma_1 + \left((z\sigma_1)^2 + 4\mu_1 C_j\right)^{1/2}}{2\mu_1}\right]^2$$

$$n_1 = \frac{1}{\mu_1}\left[\frac{\left((z\sigma_1)^2 + 4\mu_1 C_j\right)^{1/2}}{2\sqrt{\mu_1}} - \frac{z\sigma_1}{2\sqrt{\mu_1}}\right]^2 \rightarrow n_1 = \frac{1}{\mu_1}\left[\left(\frac{(z\sigma_1)^2 + 4\mu_1 C_j}{4\mu_1}\right)^{1/2} - \frac{z\sigma_1}{2\sqrt{\mu_1}}\right]^2$$

$$n_1 = \frac{1}{\mu_1}\left[\left(\left(\frac{z\sigma_1}{2\sqrt{\mu_1}}\right)^2 + C_j\right)^{1/2} - \frac{z\sigma_1}{2\sqrt{\mu_1}}\right]^2 \rightarrow n_1 = \frac{C_j}{\mu_1}\left[\left(\left(\frac{z\sigma_1}{2\sqrt{\mu_1 C_j}}\right)^2 + 1\right)^{1/2} - \frac{z\sigma_1}{2\sqrt{\mu_1 C_j}}\right]^2$$

$$n_1 = \frac{C_j}{\mu_1}\left[\left(\left(\frac{z\sigma_1}{2\mu_1\sqrt{\frac{C_j}{\mu_1}}}\right)^2 + 1\right)^{1/2} - \frac{z\sigma_1}{2\mu_1\sqrt{\frac{C_j}{\mu_1}}}\right]^2$$

$$n_1 = \frac{C_j}{\mu_1}\left[\left(\left(\frac{\frac{z\sigma_1}{2\mu_1}}{\sqrt{\frac{C_j}{\mu_1}}}\right)^2 + 1\right)^{1/2} - \frac{\frac{z\sigma_1}{2\mu_1}}{\sqrt{\frac{C_j}{\mu_1}}}\right]^2 \quad (41)$$

Asumiendo que

$$\frac{z\sigma_1}{2\mu_1} \ll \sqrt{\frac{C_j}{\mu_1}} \quad (42)$$

Entonces

$$n_1 \approx \frac{C_j}{\mu_1} \quad (43)$$

Lo que se puede observar de la ecuación (43) es que para valores grandes de $\frac{C_j}{\mu_1}$ o para valores pequeños de $\frac{\sigma_1}{\mu_1}$, la varianza ${\sigma_1}^2$ no impacta en el número de VMs. Esto es coherente con lo visto en la configuración del proyecto Nova de OpenStack, el cual utiliza un factor de *overprovisioning* para realizar el *scheduling* de VMs. Por defecto utiliza un factor de 16 para los *cores*, con lo que –considerando la multiplexación estadística del consumo de recursos– por defecto el *scheduler* de OpenStack asume que cada *core* asignado es utilizado solo el 6.25% del tiempo; por ello, realiza una sobreasignación (en un factor de *overprovision*) de los recursos disponibles en los servidores. Sin embargo, si no se cumple la ecuación (42), el impacto de la varianza no es despreciable y, por consiguiente, realizar la

sobreasignación de recursos puede generar contención de recursos, incumpliendo el SLA. Esto se puede extender para el resto de recursos (p. ej. RAM; disco, entre otros).

Por otro lado, ahora se consideran 2 posibles tipos de cargas, como se muestra a continuación:

Tabla III. Características de 2 posibles tipos de cargas

Tipo de Carga	Número de Cores	Media del Consumo de Cores (μ)	Desviación estándar del consumo de cores (σ)
Tipo 1	1	0.1	0.3
Tipo 2	2	0.2	0.6

También se definen las variables n_1 y n_2 (ambos mayores o iguales a 0) para indicar el número de instancias de tipo 1 y 2, respectivamente. Entonces la ecuación 10 definida como

$$E\{W_T^k\} + z\sqrt{Var\{W_T^k\}} \leq C_j^k$$

se convierte en

$$n_1\mu_1 + n_2\mu_2 + z\sqrt{n_1{\sigma_1}^2 + n_2{\sigma_2}^2} \leq C_j \qquad (44)$$

donde

$$E\{W_T^k\} = n_1\mu_1 + n_2\mu_2$$

$$Var\{W_T^k\} = n_1{\sigma_1}^2 + n_2{\sigma_2}^2$$

$$0 \leq n_1 \wedge 0 \leq n_2$$

El siguiente paso es darle la forma de una ecuación cuadrática, para poder graficar la curva que determina el comportamiento entre n_1 y n_2. Desarrollando la ecuación 36 se tiene que

$$z\sqrt{n_1{\sigma_1}^2 + n_2{\sigma_2}^2} \leq C_j - (n_1\mu_1 + n_2\mu_2)$$

$$z^2(n_1{\sigma_1}^2 + n_2{\sigma_2}^2) \leq \left[C_j - (n_1\mu_1 + n_2\mu_2)\right]^2$$

$$z^2(n_1{\sigma_1}^2 + n_2{\sigma_2}^2) \leq {C_j}^2 - 2C_j(n_1\mu_1 + n_2\mu_2) + (n_1\mu_1 + n_2\mu_2)^2$$

$$z^2{\sigma_1}^2 n_1 + z^2{\sigma_2}^2 n_2 \leq {C_j}^2 - 2C_j\mu_1 n_1 - 2C_j\mu_2 n_2 + {\mu_1}^2{n_1}^2 + 2\mu_1\mu_2 n_1 n_2 + {\mu_2}^2{n_2}^2$$

$$0 \leq {\mu_1}^2{n_1}^2 + 2\mu_1\mu_2 n_1 n_2 + {\mu_2}^2{n_2}^2 - \left(2C_j\mu_1 + z^2{\sigma_1}^2\right)n_1 - \left(2C_j\mu_2 + z^2{\sigma_2}^2\right)n_2 + {C_j}^2 \quad (45)$$

Tomando como referencia la fórmula general de una ecuación cuadrática

$$ax^2 + 2bxy + cy^2 + ex + fy + h = 0$$

se tiene que

$$x = n_1 \qquad a = {\mu_1}^2 \qquad e = -\left(2C_j\mu_1 + z^2{\sigma_1}^2\right)$$

$$y = n_2 \qquad b = \mu_1\mu_2 \qquad f = -\left(2C_j\mu_2 + z^2{\sigma_2}^2\right)$$

$$c = {\mu_2}^2 \qquad h = {C_j}^2$$

Considerando un valor de $z = 3.2$ (para un valor de confiabilidad de 99.999) y $C_j = 24$ (número de *cores* en un servidor), se tiene una curva cuadrática con un ángulo de rotación $\theta = 63.38°$ respecto al eje vertical, como se muestra a continuación

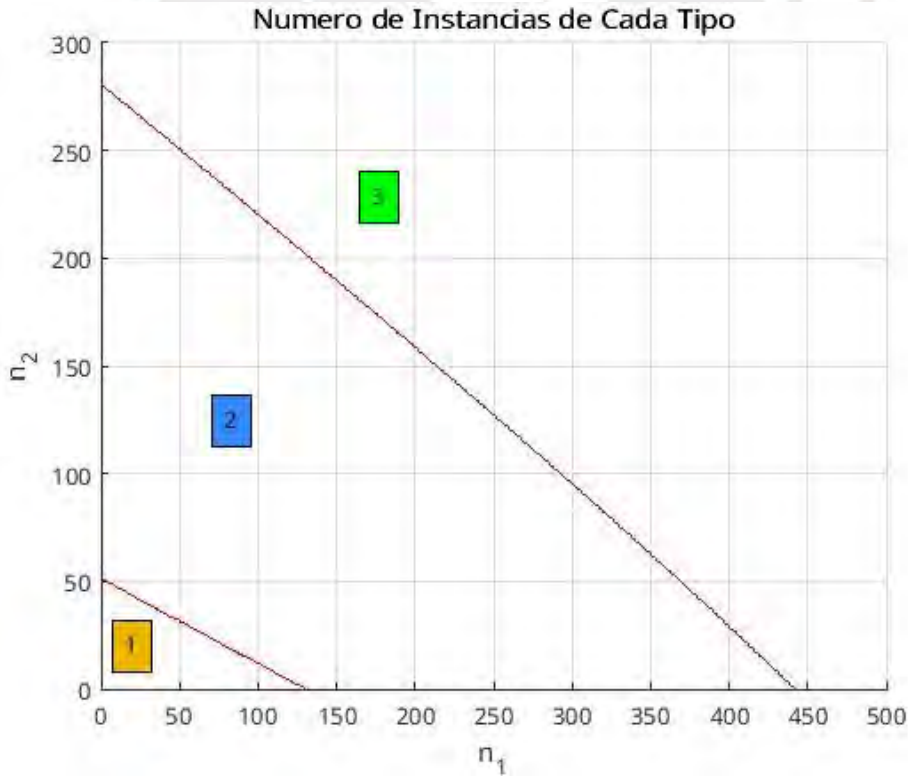

Figura 31. Curva del número de instancias de cada tipo

Fuente: Elaboración propia

En la figura anterior se distinguen 3 zonas de valores de n_1 y n_2; sin embargo, los valores de la zona 2 no cumplen con la inecuación (45) y los valores de la zona 3 no cumplen con la inecuación (44). Debido a ello, en la zona 1 se encuentran los valores de operación de n_1 y n_2.

Luego, se grafican las curvas de operación de n_1 y n_2 cuando se utiliza el *scheduler* de OpenStack para los valores de *overprovisioning* (OP) 1, 5 y 16, como se muestra en la Figura 32. En ella se observa la curva obtenida con *scheduler* propuesto (rojo) y con el *scheduler* de OpenStack para los distintos valores de OP. En particular, se observan 2 zonas de operación de interés: una donde OpenStack logra asignar más VMs que el *scheduler* propuesto, pero no se puede asegurar que se cumpla el SLA (zona A, donde curva de OpenStack está por encima de la curva del *scheduler* propuesto), y otra donde OpenStack asigna menos VMs que el *scheduler* propuesto, ocasionando que queden recursos subutilizados (zona B, donde curva de OpenStack está por debajo de la curva del *scheduler* propuesto).

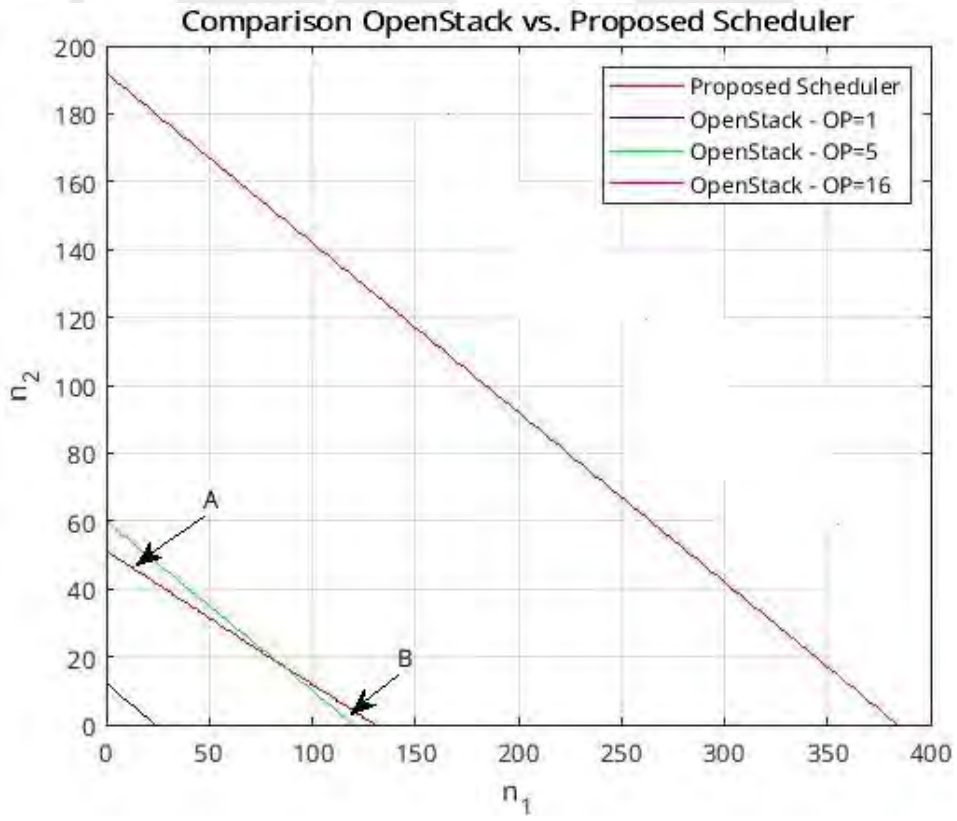

Figura 32. Comparación del número de VMs de cada tipo

Fuente: Elaboración propia

5.2. Resultados del algoritmo genético propuesto

Como punto de partida, primero se comparan los resultados obtenidos por el algoritmo genético con el obtenido por el *solver* CPLEX de IBM, con el objetivo de tener una referencia de qué tan cerca se encuentra la solución obtenida por el algoritmo genético de la solución óptima. Esto debido a que, como se observa en la Figura 33, el algoritmo genético logra encontrar un cromosoma con mejor *fitness* a medida que avancen las generaciones, pero no puede asegurar que se trate del máximo global.

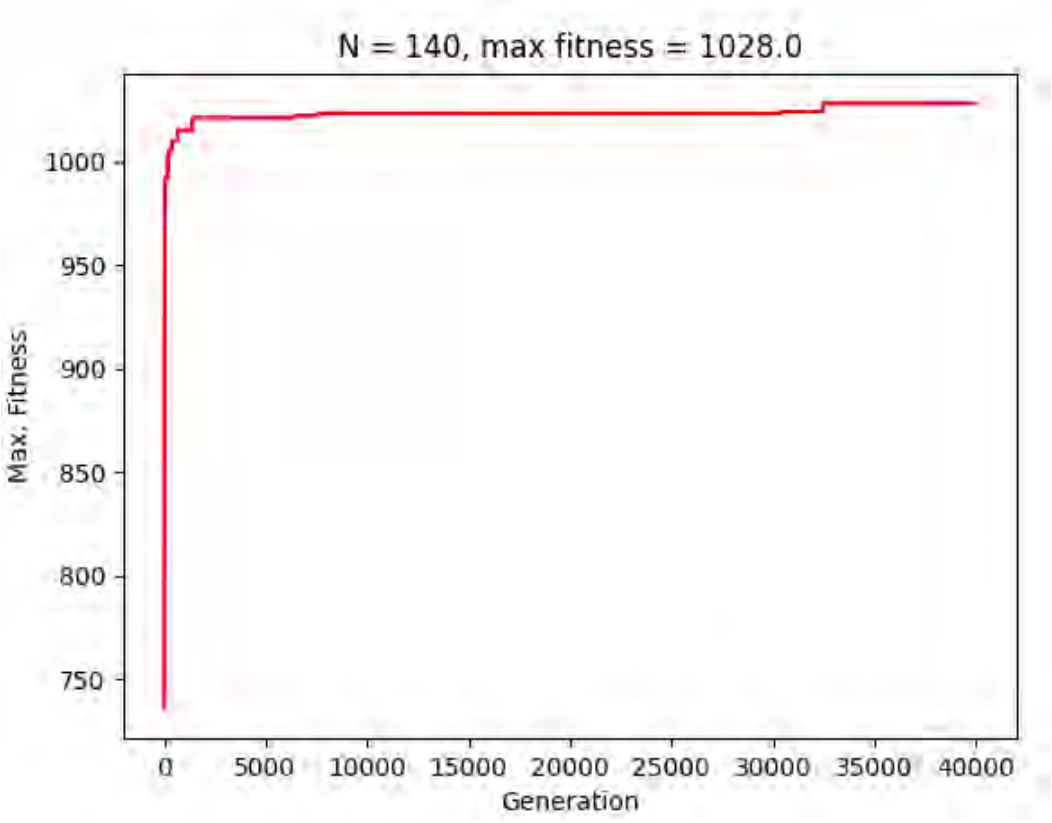

Figura 33. Evolución del *fitness* del Algoritmo Genético para $N = 140$

Fuente: Elaboración propia

Con la finalidad de evaluar la calidad (*quality*) del algoritmo genético, se comparan los resultados que obtiene con los del *solver* CPLEX para el caso de *Guaranteed Resources*. Para ello, con las PMFs obtenidas en la sección 4.5 se

generaron *datasets* o trazas del uso de recursos de VMs para distintos tamaños del problema: para cada tamaño $N = \{100,110,120,\ldots,200\}$ se generaron 6 instancias del problema. Luego, se definió la calidad del algoritmo de la siguiente manera:

$$quality = \frac{fitness_{OPTIMAL} - fitness_{GA}}{fitness_{OPTIMAL}}$$

donde $fitness_{OPTIMAL}$ es el valor de *fitness* óptimo obtenido con el *solver* CPLEX y $fitness_{GA}$ es el valor de *fitness* obtenido por el algoritmo genético desarrollado. En la Tabla IV se muestran los resultados para un tamaño de problema $N = 140$, donde se observa que para las instancias 1 y 3 se obtiene el valor óptimo (máximo global), mientras que para el resto se obtiene una solución subóptima (máximo local) pero muy cercana a la solución óptima.

Tabla IV. Resultados obtenidos para $N = 140$

Instancia del Problema	Fitness Óptimo (CPLEX)	Fitness Algoritmo Genético	Calidad
1	740	740	0.00%
2	1074	1128	4.79%
3	772	772	0.00%
4	880	890	1.12%
5	960	984	2.44%
6	1220	1304	6.44%

En la Tabla V se muestra la calidad media obtenida al promediar las calidades de las 6 instancias de cada tamaño de problema.

Tabla V. Resultados obtenidos para diferentes tamaños de problema

Tamaño del problema	Calidad media
100	100.00%
110	100.00%
120	99.53%
130	98.96%
140	97.54%
150	97.41%
160	96.29%
170	95.07%
180	94.12%
190	91.40%

200	91.49%

Se observa que a medida que se va aumentando el tamaño de problema, la calidad va decreciendo. Esto se debe a que, como la capacidad de los servidores es fija, la cantidad de máquinas virtuales que se rechazan va aumentando (bloqueo). Para un $N = 100$ se tiene un bloqueo entre 0% y 25%, mientras que para un $N = 200$ se tiene un bloqueo entre 57% y 74.5%. De lo anterior se puede identificar que a medida que aumenta el bloqueo, la calidad de la solución disminuye.

Por otro lado, se construyó un segundo modelo de algoritmo genético utilizando la librería PyGAD (Gad, 2023) para tener un punto de comparación del desempeño y calidad del algoritmo genético inicialmente propuesto. Este algoritmo presenta las características: indicadas en la Tabla VI.

Tabla VI. Configuración del algoritmo genético con PyGAD

Característica	**Descripción**
Codificación del cromosoma	Similar al indicado en la Figura 26
Población inicial	Genes aleatorios: se consideran cromosomas que no satisfagan las restricciones
Selección	*Steady-State Selection*: se utilizan algunos cromosomas con mejor *fitness* como padres de la siguiente generación, se descartan algunos cromosomas con peor *fitness*, y el resto de cromosomas se mantienen
Reproducción	implícito en *Steady-State Selection*
Crossover	Uniforme: cada gen de los padres tiene la misma probabilidad de ser elegido.
Mutación	*Swap*: se intercambian genes del mismo cromosoma (10%)
Elitismo	Sí. Se pasa el mejor cromosoma a la siguiente generación
Población	500
Número de padres	200
Generaciones	300

Con esta configuración se ejecutaron las simulaciones con la misma data de entrada de VMs y PMs (tamaño y número de instancias), obteniendo los resultados mostrados en la Tabla VII.

Tabla VII. Resultados obtenidos para diferentes tamaños de problema

Tamaño del problema	Calidad media de GA propuesto	Calidad Media de GA con PyGAD
100	100.00%	99.49%
110	100.00%	99.89%
120	99.53%	99.01%
130	98.96%	98.50%
140	97.54%	97.32%
150	97.41%	96.68%
160	96.29%	94.99%
170	95.07%	91.86%
180	94.12%	91.27%
190	91.40%	85.73%
200	91.49%	85.31%

Se observa que el algoritmo genético propuesto tiene mejor resultados que el segundo algoritmo, construido con PyGAD. Esto se puede deber a que el algoritmo genético utiliza una menor cantidad de generaciones debido al tiempo elevado de ejecución (más de 30 minutos). Sin embargo, eligió un número bajo de generaciones junto con una mayor población al momento de realizar la selección debido a que puede permitir obtener mejores resultados (Roeva et. al., 2015).

5.3. Comparación entre *Scheduler* de OpenStack y *Scheduler* Propuesto

Por último, se evalúa el funcionamiento del *scheduler* desarrollado en base del algoritmo genético propuesto con el *scheduler* de OpenStack. Para compararlos, se utiliza la métrica de *fitness* (utilidad) empleada en la sección anterior y un factor de *overprovisioning* de 5.

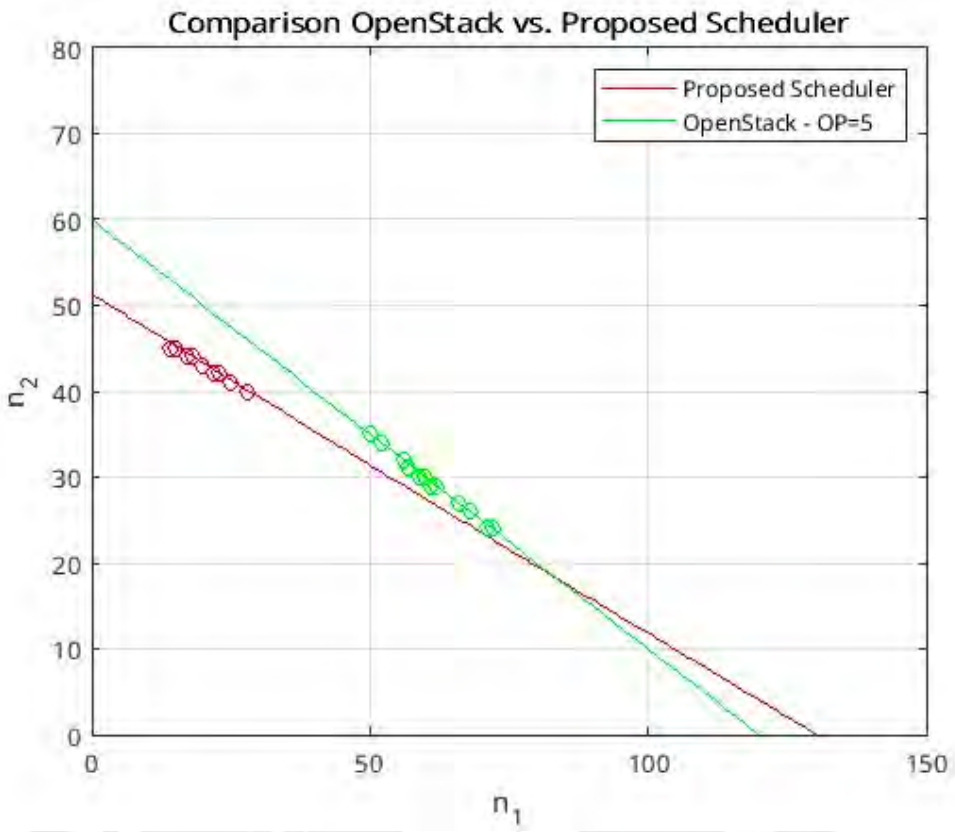

Figura 34. Algoritmo Propuesto vs. *Scheduler* de OpenStack
Fuente: Elaboración propia

Se consideraron los tipos de instancias de la Tabla III y que tienen la misma probabilidad de consumir un determinado número de *cores*. Por ejemplo, si se disponen 12 *cores*, las instancias de tipo 1 y 2 utilizan 6 *cores* cada una; más aún, se tienen 6 instancias de tipo 1 y 3 instancias de tipo 2 ya que las primeras requieren 1 *core* mientras que las segundas 2 *cores* (inversamente proporcional a la relación del número de *cores* de cada tipo). Por ello, la probabilidad de tener VMs de tipo 1 es el doble de la cantidad de instancias de tipo 2.

En la Figura 34 se indican como círculos de color verde (*scheduler* de OpenStack con factor de *overprovision* de 5) y de color rojo (*scheduler* propuesto) las cantidades de VMs de tipo 1 (eje horizontal) y de tipo 2 (eje vertical). Se observa que el *scheduler* de OpenStack, debido a que ubica las máquinas virtuales a medida que van llegando las solicitudes de creación, indica que se creen el doble de instancias de tipo 1 que de tipo 2 (coherente con la probabilidad de ocurrencia de cada tipo de instancia). No obstante, el *scheduler* propuesto basado en el algoritmo genético evalúa todas las posibles instancias a crear, y tiende a seleccionar las que brindan

mayor utilidad; en este caso, son las instancias con mayor número de *cores*, según lo se definió el beneficio de cada una. Por esta razón, se consideran más instancias de tipo 2 (mayor número de *cores*) que de tipo 1.

Conclusiones

A continuación, se presentan las conclusiones de obtenidas del presente trabajo de investigación:

- Se modelaron los escenarios de *Instance Placement* como problemas de Optimización Combinatoria para los 3 tipos de instancias de interés: *Best-Effort*, *High-Priority* y *Upgradable*.
- Se logró diseñar e implementar un algoritmo de *Instance Placement* para cargas de diferentes tipos (incluidas de alta performance), el cual se puede ejecutar en tiempo real.
- El *scheduler* de OpenStack no considera el efecto de la varianza en el uso de recursos de las VMs. Este efecto puede ser despreciado para valores grandes de $\frac{c_j^k}{\mu_1}$ (k hace referencia a los *cores*) o valores pequeños de $\frac{\sigma_1}{\mu_1}$; si ninguno de los 2 casos anteriores se cumple, puede que se incumpla el SLA.
- Para cargas de distinto tipo, el *scheduler* propuesto presenta una mayor utilidad (*fitness*) en comparación con el *scheduler* de OpenStack, ya que considera qué VMs son las que contribuyen de mejor manera a la utilidad.
- Para las cargas de tipo *Guaranteed Resources* (equivalente para cargas *High Priority* y *Upgradable* considerando que fuera del intervalo de reserva no consumen recursos), el algoritmo de *scheduling* propuesto logra obtener soluciones muy cercanas a las soluciones óptimas.

Bibliografía

The Linux Kernel documentation (s/f). *CFS Scheduler*. Kernel.org. https://docs.kernel.org/scheduler/sched-design-CFS.html

Lozi, J.-P., Lepers, B., Funston, J., Gaud, F., Quéma, V., & Fedorova, A. (2016). The Linux scheduler: A decade of wasted cores. *Proceedings of the Eleventh European Conference on Computer Systems*.

Masdari, M., Nabavi, S. S., & Ahmadi, V. (2016). An overview of virtual machine placement schemes in cloud computing. *Journal of Network and Computer Applications, 66*, 106-127. https://doi.org/10.1016/j.jnca.2016.01.011

Usmani, Z., & Singh, S. (2016). A survey of virtual machine placement techniques in a cloud data center. *Procedia Computer Science, 78*, 491-498. https://doi.org/10.1016/j.procs.2016.02.093

Bharathi, P. D., Prakash, P., & Kiran, M. V. K. (2017). Virtual machine placement strategies in cloud computing. *2017 Innovations in Power and Advanced Computing Technologies (i-PACT)*.

Luo, J., Fan, X., & Yin, L. (2019). Communication-aware and energy saving virtual machine allocation algorithm in data center. *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*.

Barán, B., & López-Pires, F. (2017). Resource allocation for cloud infrastructures: Taxonomies and research challenges. En *Research Advances in Cloud Computing* (pp. 263-289). Springer Singapore.

Madni, S. H. H., Latiff, M. S. A., Coulibaly, Y., & Abdulhamid, S. M. (2017). Recent advancements in resource allocation techniques for cloud computing environment: a systematic review. *Cluster Computing, 20*(3), 2489-2533. https://doi.org/10.1007/s10586-016-0684-4

Liu, L., & Qiu, Z. (2016). A survey on virtual machine scheduling in cloud computing. *2016 2nd IEEE International Conference on Computer and Communications (ICCC)*.

Singh, S., & Chana, I. (2016). A survey on resource scheduling in cloud computing: Issues and challenges. *Journal of Grid Computing, 14*(2), 217-264. https://doi.org/10.1007/s10723-015-9359-2

Talebian, H., Gani, A., Sookhak, M., Abdelatif, A. A., Yousafzai, A., Vasilakos, A. V., & Yu, F. R. (2020). Optimizing virtual machine placement in IaaS data centers: taxonomy, review and open issues. *Cluster Computing, 23*(2), 837–878. https://doi.org/10.1007/s10586-019-02954-w

Seyyedsalehi, S. M., & Khansari, M. (2022). Virtual machine placement optimization for big data applications in cloud computing. *IEEE access: practical innovations, open solutions*, 1-1. https://doi.org/10.1109/access.2022.3203057

Abohamama, A. S., & Hamouda, E. (2020). A hybrid energy–Aware virtual machine placement algorithm for cloud environments. *Expert Systems with Applications, 150*(113306), 113306. https://doi.org/10.1016/j.eswa.2020.113306

Mosa, A., & Sakellariou, R. (2019). Dynamic virtual machine placement considering CPU and memory resource requirements. *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*.

Wu, J., & Shen, H. (2017). Efficient algorithms for VM placement in cloud data center. En *Communications in Computer and Information Science* (pp. 353-365). Springer Singapore.

Xing, H., Zhu, J., Qu, R., Dai, P., Luo, S., & Iqbal, M. A. (2022). An ACO for energy-efficient and traffic-aware virtual machine placement in cloud computing. *Swarm and Evolutionary Computation, 68*(101012), 101012. https://doi.org/10.1016/j.swevo.2021.101012

Netto, M. A. S., Calheiros, R. N., Rodrigues, E. R., Cunha, R. L. F., & Buyya, R. (2019). HPC cloud for scientific and business applications: Taxonomy, vision, and research challenges. *ACM Computing Surveys, 51*(1), 1-29. https://doi.org/10.1145/3150224

Gupta, A., Kale, L. V., Milojicic, D., Faraboschi, P., & Balle, S. M. (2013). HPC-Aware VM Placement in Infrastructure Clouds*. 2013 IEEE International Conference on Cloud Engineering (IC2E)*.

Gupta, Abhishek, Faraboschi, P., Gioachin, F., Kale, L. V., Kaufmann, R., Lee, B.-S., March, V., Milojicic, D., & Suen, C. H. (2016). Evaluating and improving the performance and scheduling of HPC applications in cloud. *IEEE transactions on cloud computing, 4*(3), 307-321. https://doi.org/10.1109/tcc.2014.2339858

Melo Alves, M., Teylo, L., Frota, Y., & Drummond, L. M. A. (2018). An interference-aware virtual machine placement strategy for high performance computing applications in clouds. *2018 Symposium on High Performance Computing Systems (WSCAD)*.

Melo Alves, M., & Drummond, L. M. de A. (2017). A multivariate and quantitative model for predicting cross-application interference in virtual environments. *The Journal of systems and software, 128*, 150-163. https://doi.org/10.1016/j.jss.2017.04.001

Bhayani, A. (2022, 7 de marzo). *Genetically solving the age old Knapsack Problem*. Arpit Bhayani. https://arpitbhayani.me/blogs/genetic-knapsack/

Red Hat Customer Portal. (2021, 14 de junio). *Getting started with high performance computing (HPC) in Red Hat Enterprise Linux 8*. https://access.redhat.com/articles/4354751

Gamess, E., & Ortiz-Zuazaga, H. (2016). Evaluation of Point-to-Point Network Performance of HPC Clusters at the Level of UDP, TCP, and MPI. En *IV Simposio Científico y Tecnológico en Computación, Caracas, Venezuela*.

Center for High Throughput Computing (s.f.). *HTCondor's Power*. HTCondor Manual. https://htcondor.readthedocs.io/en/latest/overview/htcondors-power.html

Forshaw, M., Thomas, N., & McGough, A. S. (2014). Trace-driven simulation for energy consumption in high throughput computing systems. *2014 IEEE/ACM 18th International Symposium on Distributed Simulation and Real Time Applications*.

Oracle (s/f). *What is Big Data?* Oracle Cloud Infrastructure. https://www.oracle.com/big-data/what-is-big-data/

Alla, S. (2018). *Introduction to Hadoop*. En *Big Data Analytics with Hadoop 3: Build highly effective analytics solutions to gain valuable insight into your big data* (pp. 42-48). Packt Publishing.

Lockwood, G. (2015). *Conceptual Overview of Map-Reduce and Hadoop*. Glenn Lockwood. https://www.glennklockwood.com/data-intensive/hadoop/overview.html

OpenStack Team. (2020). NFV Performance Considerations. En *Red Hat OpenStack Platform 16.0 Network Functions Virtualization Product Guide* (pp. 12-14). https://access.redhat.com/documentation/en-us/red_hat_openstack_platform/16.0/pdf/network_functions_virtualization_product_guide/red_hat_openstack_platform-16.0-network_functions_virtualization_product_guide-en-us.pdf

Advanced Networking Research Lab (s/f). *Service Function Chaining*. Gini5. https://citelab.github.io/gini5/features/service-function-chaining/

Klech, J., Kurup, S., Doleželová, M. Svistunov, M., Bíba, R., Ryan, D., Tan, C., Brindley, L., & Young, A. (2020). Affinity. En *Red Hat Enterprise Linux for Real Time 7 Reference Guide* (pp. 19-21). https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_for_real_time/7/pdf/reference_guide/red_hat_enterprise_linux_for_real_time-7-reference_guide-en-us.pdf

Srivastava, A. (2021, 14 de julio*). 3 tips for Linux process performance improvement with priority and affinity*. Red Hat. https://www.redhat.com/sysadmin/tune-linux-tips

Carrigan, T. (2021, 8 de enero). *Linux commands: How to manipulate process priority*. Red Hat. https://www.redhat.com/sysadmin/manipulate-process-priority

Krzyzanowski, P. (2015, 18 de febrero). *Process Scheduling: Who gets to run next?* Paul Krzyzanowski. https://people.cs.rutgers.edu/~pxk/416/notes/07-scheduling.html

Radvan, S., Parker, D., Curran, C., & Holzer, J. M. (2014). PCI passthrough. En *Red Hat Enterprise Linux 5 Virtualization Guide* (pp. 190-193).

https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/5/pdf/virtualization/red_hat_enterprise_linux-5-virtualization-en-us.pdf

Jones, M. (2009, 13 de octubre). *Linux virtualization and PCI passthrough*. IBM Developer. https://developer.ibm.com/tutorials/l-pci-passthrough/

OpenStack Documentation. (2020, 28 de julio). SR-IOV. En *OpenStack Networking Guide*. https://docs.openstack.org/neutron/pike/admin/config-sriov.html

VMware Docs. (2013, 27 de septiembre). *Improve Networking Performance by using SR-IOV* [Archivo de Video]. Youtube. https://youtu.be/n0A6LPL6COE?si=HIo9CSRclJaV5OKv

Hoos, H. H., & Stutzle, T. (2004). Introduction. En D. Penrose (Ed.), *Stochastic Local Search: Foundations and Applications* (pp. 16-31). Morgan Kaufmann.

Baeldung. (2022, 25 de noviembre). *P, NP, NP-Complete and NP-Hard Problems in Computer Science*. Baeldung CS. https://www.baeldung.com/cs/p-np-np-complete-np-hard

Martello, S. & Toth, P. (1990). *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons.

Rana, Y. (2021, 20 de febrero). *Knapsack Problems*. Medium. https://medium.com/algobox/knapsack-problems-4ccee7c295ea

Vaccaro, A. (2020, 9 de noviembre). *4 Steps to Easily Allocate Resources with Python & Bin Packing*. Medium. https://towardsdatascience.com/4-steps-to-easily-allocate-resources-with-python-bin-packing-5933fb8e53a9

Hansen, P., Labbé, M., & Schindl, D. (2009). Set covering and packing formulations of graph coloring: Algorithms and first polyhedral results. *Discrete Optimization, 6*(2), 135–147. https://doi.org/10.1016/j.disopt.2008.10.004

Parajuli, A. (2021, 1 de agosto). *Graph Coloring and Applications*. Medium. https://medium.com/analytics-vidhya/graph-coloring-and-applications-2157912f505d

de Harder, H. (2023, 4 de septiembre). *Meta-heuristics explained: Ant colony optimization*. Towards Data Science. https://towardsdatascience.com/meta-heuristics-explained-ant-colony-optimization-d016fe925108

Cheng, J. (2018, 17 de octubre). *Evolutionary Optimization: A Review and Implementation of Several Algorithms*. Strong. https://www.strong.io/blog/evolutionary-optimization

Shen, S., van Beek, V., & Iosup, A. (2015). Statistical characterization of business-critical workloads hosted in cloud datacenters. *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*.

Cacchiani, V., Iori, M., Locatelli, A., & Martello, S. (2022a). Knapsack problems — An overview of recent advances. Part I: Single knapsack problems. *Computers & Operations Research, 143*(105692), 105692. https://doi.org/10.1016/j.cor.2021.105692

Cacchiani, V., Iori, M., Locatelli, A., & Martello, S. (2022b). Knapsack problems — An overview of recent advances. Part II: Multiple, multidimensional, and quadratic knapsack problems. *Computers & Operations Research, 143*(105693), 105693. https://doi.org/10.1016/j.cor.2021.105693

Egli, P. (2015). *Overview of Cloud Computing*. [Presentación de diapositivas]. Slideshare. https://www.slideshare.net/PeterREgli/overview-of-cloud-computing-26076530

Gad, A. F. (2023). *PyGAD: an intuitive genetic algorithm Python library*. Multimedia Tools and Applications, 83(20), 58029–58042. https://doi.org/10.1007/s11042-023-17167-y

Roeva, O., Fidanova, S., & Paprzycki, M. (2015). Population size influence on the genetic and ant algorithms performance in case of cultivation process modeling. *In Recent Advances in Computational Optimization* (pp. 107–120). Springer International Publishing