

Ray: un marco distribuido para aplicaciones emergentes de IA

Philipp Moritz *, Robert Nishihara *, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, Ion Stoica

Universidad de California, Berkeley

Resumen

La próxima generación de aplicaciones de IA interactuará continuamente con el medio ambiente y aprenderá de estas interacciones. Estas aplicaciones imponen requisitos de sistemas nuevos y exigentes, tanto en términos de rendimiento como de flexibilidad. En este artículo, consideramos estos requisitos y presentamos Ray, un sistema distribuido para abordarlos. Ray implementa una interfaz unificada que puede expresar cálculos tanto en paralelo de tareas como basados en actores, con el apoyo de un único motor de ejecución dinámica. Para cumplir con los requisitos de desempeño, Ray emplea un programador distribuido y un almacén distribuido y tolerante a fallas para administrar el estado de control del sistema. En nuestros experimentos, demostramos escalar más allá de 1.8 millones de tareas por segundo y un mejor rendimiento que los sistemas especializados existentes para varias aplicaciones desafiantes de aprendizaje por refuerzo.

y su uso en la predicción. Estos marcos a menudo aprovechan hardware especializado (por ejemplo, GPU y TPU), con el objetivo de reducir el tiempo de entrenamiento en un entorno por lotes. Los ejemplos incluyen TensorFlow [7], MXNet [18] y PyTorch [46].

Sin embargo, la promesa de la IA es mucho más amplia que el aprendizaje supervisado clásico. Las aplicaciones de IA emergentes deben operar cada vez más en entornos dinámicos, reaccionar a los cambios en el entorno y realizar secuencias de acciones para lograr objetivos a largo plazo [8, 43]. Deben apuntar no solo a explotar los datos recopilados, sino también a explorar el espacio de posibles acciones. Estos requisitos más amplios están naturalmente enmarcados dentro del paradigma de *aprendizaje reforzado* (RL). RL se ocupa de aprender a operar de forma continua en un entorno incierto basado en una retroalimentación limitada y retardada [56]. Los sistemas basados en RL ya han dado resultados notables, como el AlphaGo de Google venciendo a un campeón mundial humano [54], y están comenzando a encontrar su camino hacia los sistemas de diálogo, UAVs [42] y manipulación robótica [25, 60].

1. Introducción

Durante las últimas dos décadas, muchas organizaciones han estado recopilando, y con el objetivo de explotar, cantidades cada vez mayores de datos. Esto ha llevado al desarrollo de una gran cantidad de marcos para el análisis de datos distribuidos, que incluyen lotes [20, 64, 28], streaming [15, 39, 31] y gráficos [34, 35, 24] sistemas de procesamiento. El éxito de estos marcos ha hecho posible que las organizaciones analicen grandes conjuntos de datos como parte central de su estrategia comercial o científica, y ha marcado el comienzo de la era de los "Big Data".

Más recientemente, el alcance de las aplicaciones centradas en datos se ha expandido para abarcar técnicas más complejas de inteligencia artificial (IA) o aprendizaje automático (ML) [30]. El caso paradigmático es el de *aprendizaje supervisado*, donde los puntos de datos están acompañados de etiquetas, y donde la tecnología de caballo de batalla para mapear puntos de datos en etiquetas es proporcionada por redes neuronales profundas. La complejidad de estas redes profundas ha llevado a otra oleada de marcos que se centran en el entrenamiento de redes neuronales profundas.

El objetivo central de una aplicación de RL es aprender una política, un mapeo del estado del medio ambiente a una opción de acción, que produzca un desempeño efectivo a lo largo del tiempo, por ejemplo, ganar un juego o pilotar un dron. Encontrar políticas efectivas en aplicaciones a gran escala requiere tres capacidades principales. Primero, los métodos de RL a menudo se basan en *simulación* para evaluar políticas. Las simulaciones permiten explorar muchas opciones diferentes de secuencias de acción y aprender sobre las consecuencias a largo plazo de esas elecciones. En segundo lugar, al igual que sus contrapartes de aprendizaje supervisado, los algoritmos de RL deben funcionar *entrenamiento distribuido* para mejorar la política en base a datos generados a través de simulaciones o interacciones con el entorno físico. En tercer lugar, las políticas están destinadas a proporcionar soluciones para controlar los problemas, por lo que es necesario *atender* la política en escenarios interactivos de control de circuito cerrado y circuito abierto.

Estas características impulsan los requisitos de nuevos sistemas: un sistema para RL debe admitir *de grano fino* cálculos (por ejemplo, representar acciones en milisegundos al interactuar con el mundo real y realizar una gran cantidad de simulaciones

* contribución igual

ulaciones), debe apoyar *heterogeneidad* tanto en el tiempo (por ejemplo, una simulación puede tardar milisegundos u horas) como en el uso de recursos (por ejemplo, GPU para entrenamiento y CPU para simulaciones), y debe admitir *dinámica* de ejecución, ya que los resultados de simulaciones o interacciones con el entorno pueden cambiar los cálculos futuros. Por lo tanto, necesitamos un marco de cálculo dinámico que maneje millones de tareas heterogéneas por segundo en latencias de milisegundos.

Los marcos existentes que se han desarrollado para cargas de trabajo de Big Data o para cargas de trabajo de aprendizaje supervisado no satisfacen estos nuevos requisitos para RL. Los sistemas paralelos masivos síncronos como Map-Reduce [20], Apache Spark [64] y Dryad [28] no admiten simulación de grano fino ni servicio de políticas. Los sistemas de tareas paralelas como CIEL [40] y Dask [48] brindan poco apoyo para la capacitación y el servicio distribuidos. Lo mismo ocurre con los sistemas de transmisión por secuencias como Naiad [39] y Storm [31]. Los marcos distribuidos de aprendizaje profundo como TensorFlow [7] y MXNet [18] no son compatibles con la simulación y el servicio. Finalmente, los sistemas de servicio de modelos como TensorFlow Serving [6] y Clipper [19] no admiten ni entrenamiento ni simulación.

Si bien, en principio, se podría desarrollar una solución de extremo a extremo uniendo varios sistemas existentes (por ejemplo, Horovod [53] para entrenamiento distribuido, Clipper [19] para servir y CIEL [40] para simulación), en la práctica este enfoque es insostenible debido a la *acoplamiento apretado* de estos componentes dentro de las aplicaciones. Como resultado, los investigadores y profesionales de la actualidad crean sistemas únicos para aplicaciones de RL especializadas [58, 41, 54, 44, 49, 5]. Este enfoque impone una carga de ingeniería de sistemas masiva en el desarrollo de aplicaciones distribuidas al impulsar esencialmente los desafíos de los sistemas estándar como la programación, la tolerancia a fallas y el movimiento de datos en cada aplicación.

En este artículo, proponemos Ray, un marco de computación en clúster de propósito general que permite la simulación, el entrenamiento y el servicio para aplicaciones de RL. Los requisitos de estas cargas de trabajo van desde cálculos ligeros y sin estado, como para la simulación, hasta cálculos de larga duración y con estado, como para la formación. Para satisfacer estos requisitos, Ray implementa una interfaz unificada que puede expresar tanto *tarea paralela* y *actor basado* cálculos. *Tareas* Permita que Ray cargue simulaciones de manera eficiente y dinámica, procese grandes entradas y espacios de estado (p. ej., imágenes, video) y se recupere de fallas. A diferencia de, *actores* permitir que Ray admite de manera eficiente cálculos con estado, como el entrenamiento de modelos, y exponer el estado mutable compartido a los clientes (por ejemplo, un servidor de parámetros). Ray implementa el actor y las abstracciones de tareas sobre un único motor de ejecución dinámica que es altamente escalable y tolerante a fallas.

Para cumplir con los requisitos de rendimiento, Ray distribuye dos componentes que suelen estar centralizados en los marcos existentes [64, 28, 40]: (1) el programador de tareas y (2) un

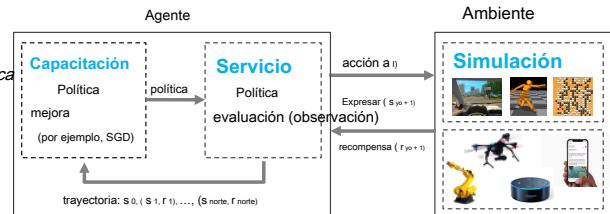


Figura 1: Ejemplo de un sistema RL.

almacén de metadatos que mantiene el linaje de computación y un directorio para objetos de datos. Esto le permite a Ray programar millones de tareas por segundo con latencias de milisegundos. Además, Ray proporciona tolerancia a fallas basada en el linaje para tareas y actores, y tolerancia a fallas basada en replicación para el almacén de metadatos.

Si bien Ray admite el servicio, la capacitación y la simulación en el contexto de las aplicaciones de RL, esto no significa que deba verse como un reemplazo de los sistemas que brindan soluciones para estas cargas de trabajo en otros contextos. En particular, Ray no pretende sustituir sistemas de servicio como Clipper [19] y TensorFlow Serving [6], ya que estos sistemas abordan un conjunto más amplio de desafíos en la implementación de modelos, incluida la gestión, las pruebas y la composición del modelo. . Del mismo modo, a pesar de su flexibilidad, Ray no sustituye a los marcos de datos paralelos genéricos, como Spark [64], ya que actualmente carece de la rica funcionalidad y API (p. Ej., Mitigación de rezagados, optimización de consultas) que estos marcos proporcionan.

Hacemos lo siguiente contribuciones:

- Diseñamos y construimos el primer marco distribuido que unifica el entrenamiento, la simulación y el servicio, componentes necesarios de las aplicaciones emergentes de RL.
- Para admitir estas cargas de trabajo, unificamos el actor y las abstracciones paralelas de tareas sobre un motor de ejecución de tareas dinámico.
- Para lograr escalabilidad y tolerancia a fallas, proponemos un principio de diseño de sistema en el que el estado de control se almacena en un almacén de metadatos fragmentados y todos los demás componentes del sistema no tienen estado.
- Para lograr la escalabilidad, proponemos una estrategia de programación distribuida de abajo hacia arriba.

2 Motivación y requisitos

Comenzamos considerando los componentes básicos de un sistema RL y completando los requisitos clave para Ray. Como se muestra en la Figura 1, en una configuración RL, un *agente* interactúa repetidamente con el *ambiente*. El objetivo del agente es aprender una política que maximice un *recompensa*. A *política* es

```

// evaluar política interactuando con env. (por ejemplo, simulador)
desenrollar( política, medio ambiente):
    trayectoria = []
    estado = medio ambiente. estado inicial()
    mientras no ambiente. has_terminated ()�:
        acción = política. calcular( Expresar) //Servicio
        estado, recompensa = medio ambiente. paso( acción) //Simulación
        trayectoria. adjuntar( estado, recompensa)
    regreso trayectoria

// mejorar la política de forma iterativa hasta que converja
train_policy ( ambiente):
    política = política_inicial ()
    mientras ( política no ha convergido):
        trayectorias = []
        por / desde 1 a k:
            // evaluar política generando k lanzamientos
            trayectorias. añadir (despliegue ( política, medio ambiente))
            // mejorar política
            política = política. actualizar( trayectorias) //Capacitación
    regreso política

```

Figura 2: pseudocódigo RL típico para aprender una política.

un mapeo del estado del medio ambiente a una elección de *acción*. Las de *recompensa* son específicas de la aplicación.

Para conocer una política, un agente generalmente emplea un proceso de dos pasos: (1) *evaluación de políticas* y 2) *mejora de la política*. Para evaluar la política, el agente interactúa con el entorno (por ejemplo, con una simulación del entorno) para generar *trayectorias*, donde una trayectoria consiste en una secuencia de tuplas (estado, recompensa) producidas por la política actual. Luego, el agente usa estas trayectorias para mejorar la política; es decir, actualizar la política en la dirección del gradiente que maximiza la recompensa. La Figura 2 muestra un ejemplo del pseudocódigo utilizado por un agente para aprender una política. Este pseudocódigo evalúa la política invocando desenrollar(*medio ambiente, política*) para generar trayectorias.

política de trenes () luego utiliza estas trayectorias para mejorar la política actual a través de *política. actualizar(trayectorias)*. Este proceso se repite hasta que la política converge.

Por lo tanto, un marco para las aplicaciones de RL debe proporcionar un soporte eficiente para *entrenamiento, servicio, y simulación* (Figura 1). A continuación, describimos brevemente estas cargas de trabajo.

Capacitación normalmente implica ejecutar el descenso de gradiente estocástico (SGD), a menudo en un entorno distribuido, para actualizar la política. El SGD distribuido generalmente se basa en un paso de agregación de reducción total o en un servidor de parámetros [32].

Servicio utiliza la política capacitada para realizar una acción basada en el estado actual del medio ambiente. Un sistema de servicio tiene como objetivo minimizar la latencia y maximizar el número de decisiones por segundo. Para escalar, la carga generalmente se equilibra en varios nodos que sirven a la política.

Finalmente, la mayoría de las aplicaciones RL existentes utilizan *simulaciones* para evaluar la política, los algoritmos RL actuales no son

muestra lo suficientemente eficiente como para depender únicamente de los datos obtenidos de las interacciones con el mundo físico. Estas simulaciones varían ampliamente en complejidad. Pueden tomar unos pocos ms (por ejemplo, simular un movimiento en una partida de ajedrez) a minutos (por ejemplo, simular un entorno realista para un automóvil autónomo).

En contraste con el aprendizaje supervisado, en el que la formación y el servicio pueden manejar por separado por diferentes sistemas, en RL *estas tres cargas de trabajo están estrechamente vinculadas en una sola aplicación*, con estrictos requisitos de latencia entre ellos. Actualmente, ningún marco soporta este acoplamiento de cargas de trabajo. En teoría, se podrían unir múltiples marcos especializados para proporcionar las capacidades generales, pero en la práctica, el movimiento de datos resultante y la latencia entre sistemas es prohibitivo en el contexto de RL. Como resultado, los investigadores y profesionales han estado construyendo sus propios sistemas únicos.

Este estado de cosas requiere el desarrollo de nuevos marcos distribuidos para RL que puedan respaldar de manera eficiente la capacitación, el servicio y la simulación. En particular, dicho marco debería satisfacer los siguientes requisitos:

Cálculos heterogéneos y detallados. La duración de un cálculo puede variar desde milisegundos (p. Ej., Realizar una acción) hasta horas (p. Ej., Entrenar una política compleja). Además, el entrenamiento a menudo requiere hardware heterogéneo (p. Ej., CPU, GPU o TPU).

Modelo de cálculo flexible. Las aplicaciones de RL requieren cálculos tanto sin estado como con estado. Los cálculos sin estado se pueden ejecutar en cualquier nodo del sistema, lo que facilita el equilibrio de carga y el movimiento del cálculo a los datos, si es necesario. Por lo tanto, los cálculos sin estado son una buena opción para la simulación de grano fino y el procesamiento de datos, como la extracción de características de imágenes o videos. Por el contrario, los cálculos con estado son una buena opción para implementar servidores de parámetros, realizar cálculos repetidos en datos respaldados por GPU o ejecutar simuladores de terceros que no exponen su estado.

Ejecución dinámica. Varios componentes de las aplicaciones de RL requieren ejecución dinámica, ya que el orden en el que terminan los cálculos no siempre se conoce de antemano (por ejemplo, el orden en que terminan las simulaciones), y los resultados de un cálculo pueden determinar cálculos futuros (por ejemplo, los resultados de una simulación determinarán si necesitamos realizar más simulaciones).

Hacemos dos comentarios finales. Primero, para lograr una alta utilización en grandes clústeres, dicho marco debe manejar *millones de tareas por segundo*. • En segundo lugar, dicho marco no está diseñado para implementar redes neuronales profundas o simuladores complejos desde cero. En cambio, debería permitir una integración perfecta con los simuladores existentes [13, 11, 59] y marcos de aprendizaje profundo [7, 18, 46, 29].

• Suponga tareas de un solo núcleo de 5 ms y un clúster de 200 nodos de 32 núcleos. Este clúster puede ejecutar $(1 \text{ s}/5 \text{ ms}) \times 32 \times 200 = 1,28 \text{ millones de tareas / seg}$.

Nombre	Descripción
<code>futuros = f.remote(argumentos)</code>	Ejecutar función <i>F</i> de forma remota. <code>f.remote()</code> puede tomar objetos o futuros como insumos y devolver uno o más futuros. Esto es sin bloqueo.
<code>objetos = ray.get(futuros)</code>	Devuelve los valores asociados con uno o más futuros. Esto está bloqueando. Devolver los futuros cuyas
<code>futuros_listos = ray.wait(futuros, k, tiempo_de_espera)</code>	tareas correspondientes se hayan completado tan pronto como <i>k</i> se ha completado o el tiempo de espera expira. Instanciar clase <i>Clase</i> como actor remoto, y le devolvemos un
<code>actor = Class.remote(argumentos)</code> <code>futuros = actor.método.remoto(argumentos)</code>	identificador. Llame a un método en el actor remoto y devuelva uno o más futuros. Ambos son sin bloqueo.

Tabla 1: API de Ray

3 Modelo de programación y computación

Ray implementa un modelo de cálculo de gráfico de tareas dinámico, es decir, modela una aplicación como un gráfico de tareas dependientes que evoluciona durante la ejecución. Además de este modelo, Ray proporciona un actor y una abstracción de programación paralela de tareas. Esta unificación diferencia a Ray de sistemas relacionados como CIEL, que solo proporciona una abstracción paralela de tareas, y de Orleans [14] o Akka [1], que principalmente proporcionan una abstracción de actor.

3.1 Modelo de programación

Tareas. A *tarea* representa la ejecución de una función remota en un trabajador apátrida. Cuando se invoca una función remota, *futuro* que representa el resultado de la tarea se devuelve inmediatamente. Los futuros se pueden recuperar usando `ray.get()` y se pasan como argumentos a otras funciones remotas sin esperar su resultado. Esto permite al usuario expresar paralelismo mientras captura dependencias de datos. La Tabla 1 muestra la API de Ray.

Las funciones remotas operan en objetos inmutables y se espera que sean *apátrida* y libre de efectos secundarios: sus salidas están determinadas únicamente por sus entradas. Esto implica idempotencia, que simplifica la tolerancia a fallas mediante la re-ejecución de funciones en caso de falla.

Actores. Un *actor* representa un cálculo con estado. Cada actor expone métodos que se pueden invocar de forma remota y se ejecutan en serie. La ejecución de un método es similar a una tarea, en que se ejecuta de forma remota y devuelve un futuro, pero difiere en que se ejecuta en un *con estado* trabajador. A *encargarse de*

a un actor se puede pasar a otros actores o tareas, lo que les permite invocar métodos sobre ese actor.

Tareas (sin estado)	Actores (con estado)
Equilibrio de carga de grano fino	Equilibrio de carga de grano grueso
Compatibilidad con la localidad del objeto Alta	Apoyo de localidad deficiente
sobrecarga para pequeñas actualizaciones	Baja sobrecarga para pequeñas actualizaciones
Manejo eficiente de fallas	Sobrecarga por puntos de control

Tabla 2: Compensaciones entre tareas y actores.

La Tabla 2 resume las propiedades de las tareas y los actores. Las tareas permiten un equilibrio de carga de grano fino mediante el aprovechamiento de la programación consciente de la carga en la granularidad de la tarea, la ubicación de los datos de entrada, ya que cada tarea se puede programar en el nodo que almacena sus entradas y la sobrecarga de recuperación baja, ya que no hay necesidad de punto de control y recuperar el estado intermedio. Por el contrario, los actores proporcionan actualizaciones de grano fino mucho más eficientes, ya que estas actualizaciones se realizan en el estado interno en lugar de externo, que normalmente requiere serialización y deserialización. Por ejemplo, los actores se pueden utilizar para implementar servidores de parámetros [32] y cálculos iterativos basados en GPU (por ejemplo, entrenamiento). Además, los actores se pueden utilizar para envolver simuladores de terceros y otros identificadores opacos que son difíciles de serializar.

Para satisfacer los requisitos de heterogeneidad y flexibilidad (Sección 2), aumentamos el API de tres formas. Primero, para manejar tareas concurrentes con duraciones heterogéneas, presentamos `ray.wait()`, que espera la primera *k* resultados disponibles, en lugar de esperar *todos* resultados como `ray.get()`. En segundo lugar, para manejar tareas con recursos heterogéneos, permitimos a los desarrolladores especificar los requisitos de recursos para que el programador de Ray pueda administrar los recursos de manera eficiente. En tercer lugar, para mejorar la flexibilidad, permitimos *funciones remotas anidadas*, lo que significa que las funciones remotas pueden invocar otras funciones remotas. Esto también es fundamental para lograr una alta escalabilidad (Sección 4), ya que permite que múltiples procesos invoquen funciones remotas de forma distribuida.

3.2 Modelo de cálculo

Ray emplea un modelo de cálculo de gráfico de tareas dinámico [21], en el que el sistema activa automáticamente la ejecución de las funciones remotas y los métodos de actor cuando sus entradas están disponibles. En esta sección, describimos cómo se construye el gráfico de cálculo (Figura 4) a partir de un programa de usuario (Figura 3). Este programa utiliza la API de la Tabla 1 para implementar el pseudocódigo de la Figura 2.

En primer lugar, ignorando a los actores, hay dos tipos de nodos en un gráfico de cálculo: objetos de datos e invocaciones o tareas de funciones remotas. También hay dos tipos de bordes: bordes de datos y bordes de control. Los bordes de datos capturan la

```

@ ray.remote
def create_policy():
    # Inicialice la política al azar.
    regreso política

@ ray.remote (num_gpus = 1 )
clase Simulador ( objeto ):
    def __en_eso__ ( uno mismo ):
        # Inicializar el entorno.
        uno mismo . env = Ambiente()
    def desenrollar ( uno mismo , política, núm_pasos):
        observaciones = []
        observación = uno mismo . env . estado actual()
        por _en abarcar (núm_pasos):
            acción = política (observación)
            observación = uno mismo . env . paso (acción)
            observaciones . añadir (observación)
        regreso observaciones

@ ray.remote (num_gpus = 2 )
def update_policy (política, * despliegues):
    # Actualice la política.
    regreso política

@ ray.remote
def tren_política ():
    # Cree una política.
    policy_id = create_policy . remoto()
    # Crea 10 actores.
    simuladores = [Simulador . remoto() por _en abarcar ( 10 )]
    # Realiza 100 pasos de entrenamiento.
    por _en abarcar ( 100 ):
        # Realice un lanzamiento en cada actor.
        rollout_ids = [s . desenrollar . remoto (policy_id)
                      por s en simuladores]
        # Actualice la política con los lanzamientos.
        policy_id =
            update_policy . remoto (policy_id, * rollout_ids)
    regreso rayo . obtener (policy_id)

```

Figura 3: Código Python que implementa el ejemplo de la Figura 2 en Ray. Tenga en cuenta que @ ray.remote indica funciones y actores remotos. Las invocaciones de funciones remotas y métodos de actor devuelven futuros, que se pueden pasar a funciones remotas posteriores o métodos de actor para codificar dependencias de tareas. Cada actor tiene un objeto de entorno self.env compartido entre todos sus métodos.

dependencias entre objetos de datos y tareas. Más precisamente, si el objeto de datos D es una salida de la tarea T , agregamos un borde de datos desde T a D . Del mismo modo, si D es una entrada para T , agregamos un borde de datos desde D a T . Los bordes de control capturan las dependencias de cálculo que resultan de las funciones motrices (Sección 3.1): si la tarea T_1 invoca tarea T_2 , luego agregamos un borde de control de T_1 a T_2 .

Las invocaciones al método de actor también se representan como nodos en el gráfico de cálculo. Son idénticas a las tareas con una diferencia clave. Para capturar la dependencia de estado a través de invocaciones de métodos posteriores en el mismo actor, agregamos un tercer tipo de borde: un borde con estado. Si el método

METRO se llama justo después del método *METRO* en el mismo actor, luego agregamos una ventaja con estado de *METRO* a *METRO*. Así, todos

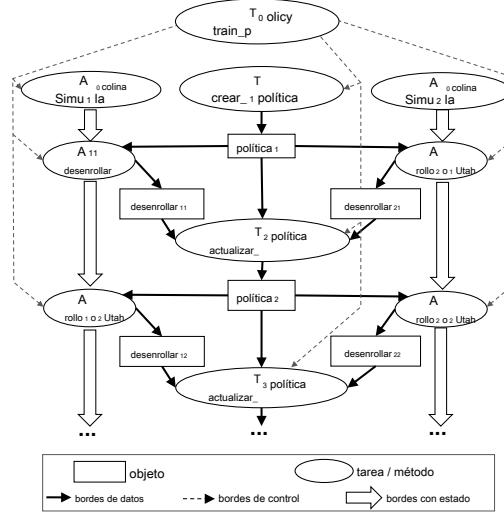


Figura 4: El gráfico de tareas correspondiente a una invocación de `train_policy.remote()` en la Figura 3. Las llamadas a funciones remotas y las llamadas al método actor corresponden a tareas en el gráfico de tareas. La figura muestra dos actores. Las invocaciones de método para cada actor (las tareas etiquetadas A_1 y A_2) tienen bordes con estado entre ellos, lo que indica que comparten el estado de actor mutable. Hay control. El control cambia de la política del tren a las tareas que invoca. Para entrenar varias políticas en paralelo, podríamos llamar a `train_policy.remote()` varias veces.

Los métodos invocados en el mismo objeto actor forman una cadena que está conectada por bordes con estado (Figura 4). Esta cadena captura el orden en el que se invocaron estos métodos.

Los bordes con estado nos ayudan a incorporar actores en un gráfico de tareas sin estado, ya que capturan la dependencia de datos implícita entre invocaciones de métodos sucesivos que comparten el estado interno de un actor. Los bordes con estado también nos permiten mantener el linaje. Como en otros sistemas de flujo de datos [64], rastreamos el linaje de los datos para permitir la reconstrucción. Al incluir explícitamente bordes con estado en el gráfico de linaje, podemos reconstruir fácilmente los datos perdidos, ya sean producidos por funciones remotas o métodos de actor (Sección 4.2.3).

4 Arquitectura

La arquitectura de Ray comprende (1) una capa de aplicación que implementa la API y (2) una capa de sistema que proporciona alta escalabilidad y tolerancia a fallas.

4.1 Capa de aplicación

La capa de aplicación consta de tres tipos de procesos:

- *Conductor*: Un proceso que ejecuta el programa de usuario.
- *Trabajador*: Un proceso sin estado que ejecuta tareas (funciones remotas) invocadas por un controlador u otro

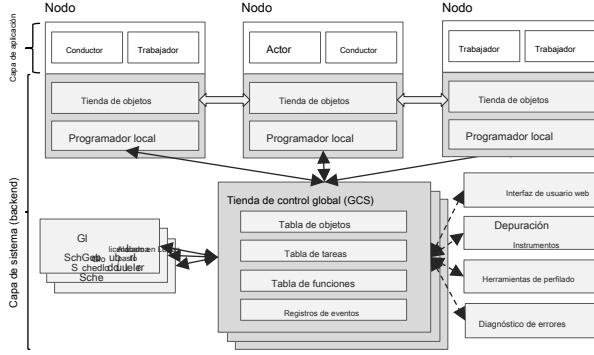


Figura 5: La arquitectura de Ray consta de dos partes: una *solicitud* capa y una *sistema* capa. La capa de aplicación implementa la API y el modelo de cálculo descrito en la Sección 3, la capa del sistema implementa la programación de tareas y la gestión de datos para satisfacer los requisitos de rendimiento y tolerancia a fallas.

trabajador. Los trabajadores son iniciados automáticamente y asignados por la capa del sistema. Cuando se declara una función remota, la función se publica automáticamente para todos los trabajadores. Un trabajador ejecuta tareas en serie, sin que se mantenga un estado local en todas las tareas.

- *Actor*: Un proceso con estado que ejecuta, cuando se invoca, solo los métodos que expone. A diferencia de un trabajador, un trabajador o un conductor crean una instancia explícita de un actor. Como los trabajadores, los actores ejecutan métodos en serie, excepto que cada método depende del estado resultante de la ejecución del método anterior.

4.2 Capa de sistema

La capa del sistema consta de tres componentes principales: un almacén de control global, un planificador distribuido y un almacén de objetos distribuidos. Todos los componentes son escalables horizontalmente y tolerantes a fallas.

4.2.1 Almacén de control global (GCS)

El almacén de control global (GCS) mantiene todo el estado de control del sistema y es una característica única de nuestro diseño. En esencia, GCS es una tienda de valor clave con funcionalidad de publicación. Usamos fragmentación para lograr escala y replicación en cadena por fragmento [61] para proporcionar tolerancia a fallas. La razón principal del GCS y su diseño es mantener la tolerancia a fallas y la baja latencia para un sistema que puede generar millones de tareas por segundo de forma dinámica.

La tolerancia a fallas en caso de falla del nodo requiere una solución para mantener la información de linaje. Las soluciones existentes basadas en el linaje [64, 63, 40, 28] se centran en el paralelismo de grano grueso y, por lo tanto, pueden utilizar un solo nodo (p. Ej., Maestro, conductor) para almacenar el linaje sin afectar el rendimiento. Sin embargo, este diseño no es escalable para una carga de trabajo dinámica y precisa como la simulación. Por lo tanto,

desacoplamos el almacenamiento de linaje duradero de los otros componentes del sistema, lo que permite que cada uno escale de forma independiente.

Mantener una latencia baja requiere minimizar los gastos generales en la programación de tareas, lo que implica elegir dónde ejecutar y, posteriormente, el despacho de tareas, lo que implica recuperar entradas remotas de otros nodos. Muchos sistemas de flujo de datos existentes [64, 40, 48] los acoplan almacenando ubicaciones y tamaños de objetos en un planificador centralizado, un diseño natural cuando el planificador no es un cuello de botella. Sin embargo, la escala y la granularidad que apunta Ray requieren mantener al programador centralizado fuera de la ruta crítica. Involucrar al planificador en cada transferencia de objetos es prohibitivamente caro para las primitivas importantes para el entrenamiento distribuido como allreduce, que es tanto de comunicación intensiva como sensible a la latencia. Por lo tanto, almacenamos los metadatos del objeto en el GCS en lugar de en el programador, desacoplando completamente el envío de tareas de la programación de tareas.

En resumen, el GCS simplifica significativamente el diseño general de Ray, ya que *permite que todos los componentes del sistema sean apátridas*. Esto no solo simplifica la compatibilidad con la tolerancia a fallas (es decir, en caso de falla, los componentes simplemente se reinician y leen el linaje del GCS), sino que también facilita escalar el almacén de objetos distribuidos y el programador de forma independiente, ya que todos los componentes comparten el estado necesario a través de GCS. Un beneficio adicional es el fácil desarrollo de herramientas de depuración, perfilado y visualización.

4.2.2 Programador distribuido ascendente

Como se discutió en la Sección 2, Ray necesita programar dinámicamente millones de tareas por segundo, tareas que pueden tomar tan solo unos pocos milisegundos. Ninguno de los programadores de clúster que conocemos cumple con estos requisitos. La mayoría de los marcos de computación en clúster, como Spark [64], CIEL [40] y Dryad [28] implementan un programador centralizado, que puede proporcionar localidad pero con latencias de decenas de ms. Los programadores distribuidos como robo de trabajo [12], Sparrow [45] y Canary [47] pueden lograr una gran escala, pero no consideran la localidad de datos [12] o asumen que las tareas pertenecen a trabajos independientes [45], o asumen se conoce el gráfico de cálculo [47].

Para satisfacer los requisitos anteriores, diseñamos un programador jerárquico de dos niveles que consta de un programador global y programadores locales por nodo. Para evitar sobrecargar el planificador global, las tareas creadas en un nodo se envían primero al planificador local del nodo. Un programador local programa tareas localmente a menos que el nodo esté sobrecargado (es decir, su cola de tareas local exceda un umbral predefinido), o no pueda satisfacer los requisitos de una tarea (p. Ej., Carece de GPU). Si un programador local decide no programar una tarea localmente, la reenvía al programador global. Dado que este programador intenta programar tareas localmente primero (es decir, en las hojas de la jerarquía de programación), lo llamamos un *programador ascendente*.

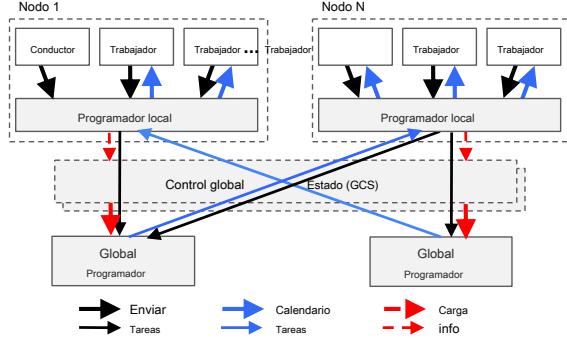


Figura 6: Programador distribuido ascendente. Las tareas se envían de abajo hacia arriba, desde los controladores y los trabajadores a un programador local y se reenvían al programador global solo si es necesario (Sección 4.2.2). El grosor de cada flecha es proporcional a su tasa de solicitud.

El programador global considera la carga de cada nodo y las limitaciones de la tarea para tomar decisiones de programación. Más precisamente, el planificador global identifica el conjunto de nodos que tienen suficientes recursos del tipo solicitado por la tarea, y de estos nodos selecciona el nodo que proporciona el menor *tiempo de espera estimado*. En un nodo dado, este tiempo es la suma de (i) el tiempo estimado que la tarea estará en cola en ese nodo (es decir, el tamaño de la cola de tareas multiplicado por la ejecución promedio de la tarea), y (ii) el tiempo de transferencia estimado de la tarea remota. entradas (es decir, el tamaño total de las entradas remotas dividido por el ancho de banda promedio). El programador global obtiene el tamaño de la cola en cada nodo y la disponibilidad de recursos del nodo a través de latidos, y la ubicación de las entradas de la tarea y sus tamaños de GCS. Además, el programador global calcula la ejecución de tareas promedio y el ancho de banda de transferencia promedio usando un promedio exponencial simple. Si el programador global se convierte en un cuello de botella, podemos crear más réplicas compartiendo la misma información a través de GCS. Esto hace que nuestra arquitectura de programador sea altamente escalable.

4.2.3 Almacén de objetos distribuidos en memoria

Para minimizar la latencia de las tareas, implementamos un sistema de almacenamiento distribuido en memoria para almacenar las entradas y salidas de cada tarea o cálculo sin estado. En cada nodo, implementamos el almacén de objetos a través de *memoria compartida*. Esto permite compartir datos sin copia entre tareas que se ejecutan en el mismo nodo. Como formato de datos, utilizamos Apache Arrow [2].

Si las entradas de una tarea no son locales, las entradas se replican en el almacén de objetos local antes de la ejecución. Además, una tarea escribe sus salidas en el almacén de objetos local. La replicación elimina el posible cuello de botella debido a los objetos de datos calientes y minimiza el tiempo de ejecución de la tarea, ya que una tarea solo lee / escribe datos desde / hacia la memoria local. Esto aumenta el rendimiento de las cargas de trabajo vinculadas a la computación, un perfil compartido por muchas aplicaciones de IA. Para una latencia baja, mantenemos los objetos completamente en la memoria y los desalojamos como

necesario para el disco utilizando una política LRU.

Al igual que con los marcos de computación en clúster existentes, como Spark [64] y Dryad [28], el almacenamiento de objetos se limita a *datos inmutables*. Esto evita la necesidad de protocolos de consistencia complejos (ya que los objetos no se actualizan) y simplifica el soporte para la tolerancia a fallas. En el caso de falla del nodo, Ray recupera cualquier objeto necesario mediante la re-ejecución del linaje. El linaje almacenado en el GCS rastrea tanto las tareas sin estado como los actores con estado durante la ejecución inicial; usamos el primero para reconstruir objetos en la tienda.

Para simplificar, nuestro almacén de objetos no admite objetos distribuidos, es decir, cada objeto encaja en un solo nodo. Los objetos distribuidos como grandes matrices o árboles pueden implementarse a nivel de aplicación como colecciones de futuros.

4.2.4 Implementación

Ray es un proyecto de código abierto activo ⁷desarrollado en la Universidad de California, Berkeley. Ray se integra completamente con el entorno Python y es fácil de instalar simplemente ejecutando pip install ray. La implementación comprende $\approx 40K$ líneas de código (LoC), 72% en C++ para la capa del sistema, 28% en Python para la capa de aplicación. El GCS utiliza un almacén de valores-clave de Redis [50] por fragmento, con operaciones de clave única. Las tablas GCS se fragmentan por ID de objeto y tarea para escalar, y cada fragmento se replica en cadena [61] para tolerancia a fallas. Implementamos los programadores locales y globales como procesos de un solo subproceso impulsados por eventos. Internamente, los programadores locales mantienen el estado en caché de los metadatos de los objetos locales, las tareas en espera de entradas y las tareas listas para enviar a un trabajador. Para transferir objetos grandes entre diferentes almacenes de objetos, distribuimos el objeto en varias conexiones TCP.

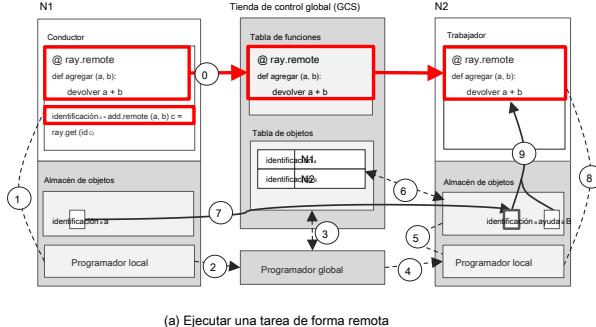
4.3 Uniendo todo

La Figura 7 ilustra cómo funciona Ray de un extremo a otro con un ejemplo simple que agrega dos objetos *a* y *B*, que pueden ser escalares o matrices, y devuelve el resultado *C*. La función remota agregar() se registra automáticamente con el GCS tras la inicialización y se distribuye a todos los trabajadores del sistema (paso 0 en la Figura 7a).

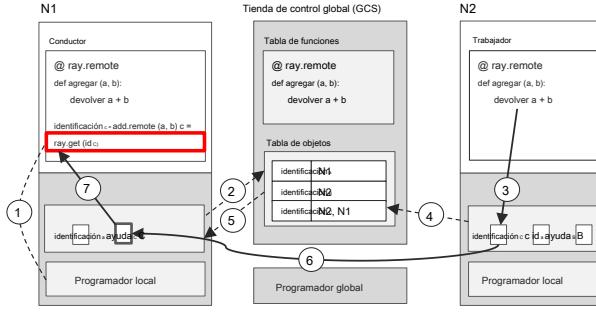
La figura 7a muestra las operaciones paso a paso activadas por un controlador que invoca add.remote(*a*, *b*), donde *a* y *B* se almacenan en nodos *norte 1* y *norte 2*, respectivamente. El conductor se somete agregar(*a*, *b*) al planificador local (paso 1), que lo reenvía a un planificador global (paso 2). ⁸A continuación, el planificador global busca las ubicaciones de agregar(*a*, *b*) en el GCS (paso 3) y decide programar la tarea en el nodo *norte 2*, que almacena el argumento *B* (paso 4). El planificador local en el nodo *norte 2* comprueba si el almacén de objetos local contiene agregar(*a*, *b*). Argumentos de (paso 5). Desde el

⁷ <https://github.com/ray-project/ray>

⁸ Tenga en cuenta que *norte* También podría decidir programar la tarea localmente.



(a) Ejecutar una tarea de forma remota



(b) Devolver el resultado de una tarea remota

Figura 7: Un ejemplo de un extremo a otro que agrega a y B y vuelve C . Las líneas continuas son operaciones del plano de datos y las líneas de puntos son operaciones del plano de control. (a) La función agregar() está registrado en el GCS por el nodo 1 (*norte 1*), invocado en *norte 1*, y ejecutado en *norte 2*. (b) *norte 1* obtiene agregar() resultado usando ray.get(). La entrada de la tabla de objetos para C se crea en el paso 4 y se actualiza en el paso 6 después de la ejecución de ray.get().

C se copia a *norte 1*.

la tienda local no tiene objeto a , mira hacia arriba a en el GCS (paso 6). Aprendiendo eso a se almacena en *norte 1*, *norte 1* El almacén de objetos de 2 lo replica localmente (paso 7). Como todos los argumentos de agregar() ahora se almacenan localmente, el programador local invoca agregar() en un trabajador local (paso 8), que accede a los argumentos a través de la memoria compartida (paso 9).

La figura 7b muestra las operaciones paso a paso desencadenadas por la ejecución de ray.get() a *norte 1*, y de agregar() a *norte 2*, respectivamente. Sobre ray.get *identificación C* invocación, el controlador comprueba el almacén de objetos local para el valor C , utilizando la futura *identificación C* devuelto por agregar() (paso 1). Dado que la tienda de objetos local no almacena C , busca su ubicación en el GCS. En este momento, no hay entrada para C , como C aún no se ha creado. Como resultado, *norte 1* El almacén de objetos de 1 registra una devolución de llamada con la tabla de objetos para que se active cuando C Se ha creado la entrada (paso 2). Mientras tanto, en *norte 2*, agregar() completa su ejecución, almacena el resultado C en la tienda de objetos local (paso 3), que a su vez agrega C Entrada de GCS (paso 4). Como resultado, el GCS activa una devolución de llamada a *norte 1* tienda de objetos con C Entrada de (paso 5). Próximo, *norte 1* réplicas C desde *norte 2* (paso 6) y vuelve C a ray.get() (paso 7), que finalmente completa la tarea.

Si bien este ejemplo involucra una gran cantidad de RPC,

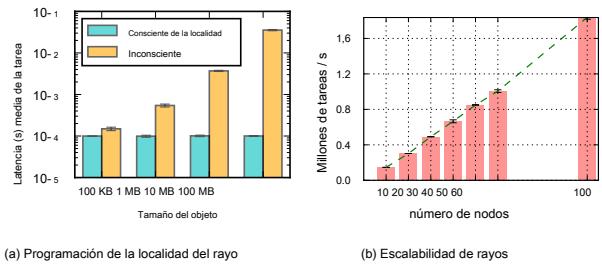


Figura 8: (a) Las tareas aprovechan la ubicación consciente de la localidad. Se programan 1000 tareas con una dependencia de objeto aleatoria en uno de dos nodos. Con la política de reconocimiento de la localidad, la latencia de la tarea sigue siendo independiente del tamaño de las entradas de la tarea en lugar de crecer en 1-2 órdenes de magnitud. (b) Escalabilidad casi lineal que aprovecha el GCS y el planificador distribuido ascendente. Ray alcanza 1 millón de tareas por segundo con 60 nodos. $X \in \{70, 80, 90\}$ omitido debido al costo.

en muchos casos, este número es mucho menor, ya que la mayoría de las tareas se programan localmente y los programadores globales y locales almacenan en caché las respuestas de GCS.

5 Evaluación

En nuestra evaluación, estudiamos las siguientes preguntas:

1. ¿Qué tan bien cumple Ray con los requisitos de latencia, escalabilidad y tolerancia a fallas enumerados en la Sección 2? (Sección 5.1)
2. ¿Qué gastos generales se imponen a las primitivas distribuidas (por ejemplo, allreduce) escritas con la API de Ray? (Sección 5.1)
3. En el contexto de las cargas de trabajo de RL, ¿cómo se compara Ray con los sistemas especializados para la formación, el servicio y la simulación? (Sección 5.2)
4. ¿Qué ventajas proporciona Ray para las aplicaciones de RL, en comparación con los sistemas personalizados? (Sección 5.3) Todos los experimentos se ejecutaron en Amazon Web Services. A menos que se indique lo contrario, usamos instancias de CPU m4.16xlarge e instancias de GPU p3.16xlarge.

5.1 Microbenchmarks

Ubicación de tareas consciente de la localidad. Balance de carga de grano fino
El posicionamiento con reconocimiento de localidad y ubicación son los principales beneficios de las tareas en Ray. Los actores, una vez colocados, no pueden trasladar sus cálculos a grandes objetos remotos, mientras que las tareas pueden hacerlo. En la Figura 8a, las tareas colocadas sin conocimiento de la localidad de datos (como es el caso de los métodos de actor), sufren de 1 a 2 órdenes de aumento de latencia de magnitud a tamaños de datos de entrada de 10 a 100 MB. Ray unifica tareas y actores a través del almacén de objetos compartidos, lo que permite a los desarrolladores utilizar tareas para, por ejemplo, un costoso posprocesamiento en la salida producida por actores de simulación.

Escalabilidad de un extremo a otro. Uno de los beneficios clave de

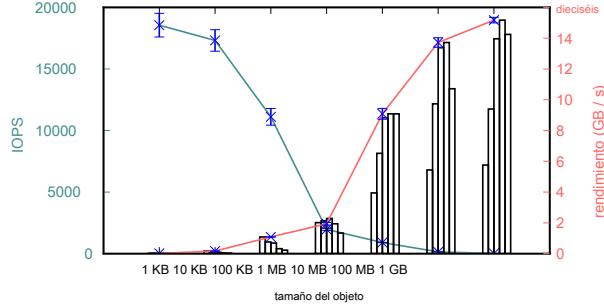
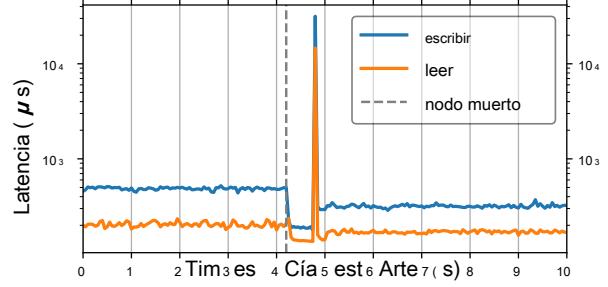


Figura 9: IOPS y rendimiento de escritura del almacén de objetos. Desde un solo cliente, el rendimiento supera los 15 GB / s (rojo) para objetos grandes y 18K IOPS (cian) para objetos pequeños en una instancia de 16 núcleos (m4.4xlarge). Utiliza 8 hilos para copiar objetos de más de 0,5 MB y 1 hilo para objetos pequeños. Los gráficos de barras informan el rendimiento con 1, 2, 4, 8, 16 hilos. Los resultados se promedian en 5 ejecuciones.

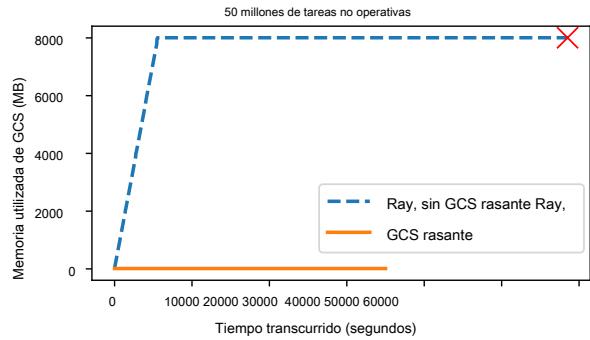
Global Control Store (GCS) y el programador distribuido de abajo hacia arriba es la capacidad de escalar horizontalmente el sistema para admitir un alto rendimiento de tareas de grano fino, mientras se mantiene la tolerancia a fallas y la programación de tareas de baja latencia. En la Figura 8b, evaluamos esta capacidad en una carga de trabajo vergonzosamente paralela de tareas vacías, aumentando el tamaño del grupo en el eje x. Observamos una linealidad casi perfecta en el aumento progresivo del rendimiento de las tareas. Ray supera el rendimiento de 1 millón de tareas por segundo en 60 nodos y continúa escalando linealmente más allá de 1,8 millones de tareas por segundo en 100 nodos. El punto de datos más a la derecha muestra que Ray puede procesar 100 millones de tareas en menos de un minuto (54 s), con una variabilidad mínima. Como se esperaba, aumentar la duración de la tarea reduce el rendimiento proporcionalmente a la duración media de la tarea, pero la escalabilidad general sigue siendo lineal. Si bien muchas cargas de trabajo realistas pueden exhibir una escalabilidad más limitada debido a las dependencias de los objetos y los límites inherentes al paralelismo de las aplicaciones, esto demuestra la escalabilidad de nuestra arquitectura general bajo una carga alta.

Rendimiento del almacén de objetos. Para evaluar el desempeño a partir del almacén de objetos (Sección 4.2.3), realizamos un seguimiento de dos métricas: IOPS (para objetos pequeños) y rendimiento de escritura (para objetos grandes). En la Figura 9, el rendimiento de escritura de un solo cliente supera los 15 GB / s a medida que aumenta el tamaño del objeto. Para objetos más grandes, memcpy domina el tiempo de creación de objetos. Para objetos más pequeños, los gastos generales principales están en la serialización e IPC entre el cliente y el almacén de objetos.

Tolerancia a fallos de GCS. Para mantener una latencia baja y, al mismo tiempo, proporcionar una coherencia sólida y tolerancia a fallas, creamos una capa de replicación en cadena ligera [61] sobre Redis. La Figura 10a simula la grabación de tareas de Ray y las tareas de lectura desde el GCS, donde las claves son 25 bytes y los valores son 512 bytes. El cliente envía las solicitudes lo más rápido que puede, teniendo como máximo una solicitud de vuelo a la vez. Las fallas se informan al maestro de la cadena desde el cliente (habiéndose recibido errores explícitos o tiempos de espera a pesar de los reintentos).



(a) Una línea de tiempo para las latencias de lectura y escritura de GCS como se ve desde un cliente que envía tareas. La cadena comienza con 2 réplicas. Activamos manualmente la reconfiguración de la siguiente manera. A $t \approx 4.2$, un miembro de la cadena muere; inmediatamente después, un nuevo miembro de la cadena se une, inicia la transferencia de estado y restaura la cadena a la replicación bidireccional. La latencia máxima observada por el cliente es inferior a 30 ms a pesar de las reconfiguraciones.



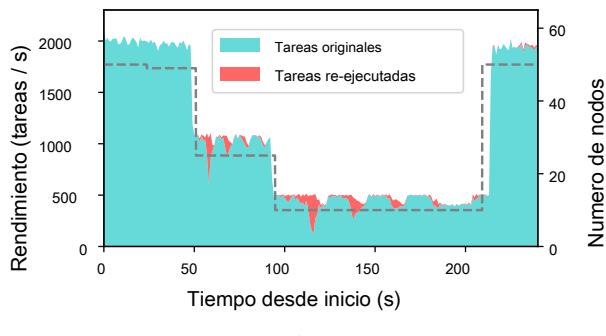
(b) Ray GCS mantiene una huella de memoria constante con la descarga de GCS. Sin la descarga de GCS, la huella de memoria alcanza una capacidad máxima y la carga de trabajo no se completa dentro de una duración predeterminada (indicated por la cruz roja).

Figura 10: Tolerancia a fallas y lavado de Ray GCS.

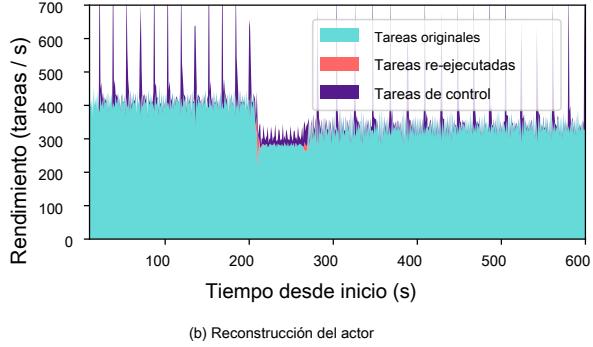
desde cualquier servidor de la cadena (habiéndose recibido errores explícitos). En general, las reconfiguraciones causaron un máximo observado por el cliente retraso de menos de 30 ms (esto incluye retrasos en la detección de fallas y en la recuperación).

Lavado de GCS. Ray está equipado para descargar periódicamente el contenido de GCS en el disco. En la Figura 10b, enviamos 50 millones de tareas vacías secuencialmente y monitoreamos el consumo de memoria de GCS. Como se esperaba, crece linealmente con el número de tareas rastreadas y finalmente alcanza la capacidad de memoria del sistema. En ese punto, el sistema se paraliza y la carga de trabajo no termina en un período de tiempo razonable. Con el lavado periódico de GCS, logramos dos objetivos. Primero, la huella de memoria se limita a un nivel configurable por el usuario (en el microbenchmark empleamos una estrategia agresiva en la que la memoria consumida se mantiene lo más baja posible). En segundo lugar, el mecanismo de descarga proporciona una forma natural de capturar el linaje en el disco para aplicaciones Ray de larga ejecución.

Recuperación de fallas de tareas. En la Figura 11a,



(a) Reconstrucción de tareas



(b) Reconstrucción del actor

Figura 11: Tolerancia a fallas de Ray. (a) Ray reconstruye las dependencias de tareas perdidas a medida que se eliminan los nodos (línea de puntos) y recupera el rendimiento original cuando se vuelven a agregar nodos. Cada tarea es de 100 ms y depende de un objeto generado por una tarea enviada previamente. (B) Los actores se reconstruyen desde su último punto de control. A $t = 200$, matamos 2 de los 10 nodos, lo que provoca que 400 de los 2000 actores del clúster se recuperen en los nodos restantes ($t = 200-270$).

Demuestre la capacidad de Ray para recuperarse de forma transparente de las fallas del nodo trabajador y escalar elásticamente, utilizando el almacenamiento de linaje GCS duradero. La carga de trabajo, que se ejecuta en instancias m4.xlarge, consta de cadenas lineales de tareas de 100 ms enviadas por el controlador. A medida que se eliminan los nodos (a los 25, 50, 100), los programadores locales reconstruyen los resultados anteriores en la cadena para continuar con la ejecución. En general *por nodo* el rendimiento permanece estable en todo momento.

Recuperarse de los fracasos de los actores. Al codificar actor llamadas a métodos como bordes con estado directamente en el gráfico de dependencia, podemos reutilizar el mismo mecanismo de reconstrucción de objetos que en la Figura 11a para proporcionar tolerancia transparente a fallas para *computación con estado*. Además, Ray aprovecha las funciones de punto de control definidas por el usuario para limitar el tiempo de reconstrucción de los actores (Figura 11b). Con una sobrecarga mínima, los puntos de control permiten que solo se vuelvan a ejecutar 500 métodos, frente a 10k de re-ejecuciones sin puntos de control. En el futuro, esperamos reducir aún más el tiempo de reconstrucción del actor, por ejemplo, permitiendo a los usuarios anotar métodos que no mutan de estado.

Allreduce. Allreduce es una comunicación distribuida

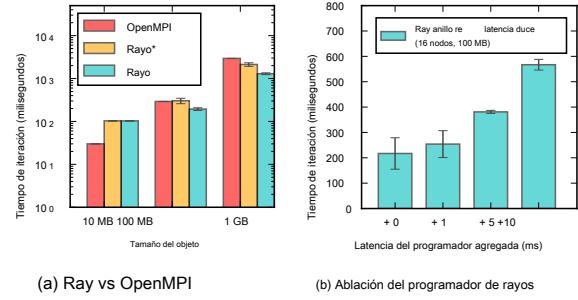


Figura 12: (a) Tiempo medio de ejecución de allreduce en nodos de 16 m4.16xl. Cada trabajador se ejecuta en un nodo distinto. Ray * restringe Ray a 1 hilo para enviar y 1 hilo para recibir. (b) La programación de baja latencia de Ray es fundamental para allreduce.

primitivo importante para muchas cargas de trabajo de aprendizaje automático. Aquí, evaluamos si Ray puede soportar de forma nativa una implementación de reducción total [57] con una sobrecarga lo suficientemente baja como para igualar las implementaciones existentes [53]. Descubrimos que Ray completa toda la reducción en 16 nodos en 100 MB de

~200ms y 1GB en ~ 1200ms, superando sorprendentemente a OpenMPI (v1.10), una popular implementación de MPI, por $1.5 \times$ y $2 \times$ respectivamente (Figura 12a). Atribuimos el rendimiento de Ray al uso de múltiples subprocesos para transferencias de red, aprovechando al máximo la conexión de 25 Gbps entre nodos en AWS, mientras que OpenMPI envía y recibe datos secuencialmente en un solo subproceso [22]. Para objetos más pequeños, OpenMPI supera a Ray al cambiar a un algoritmo de sobrecarga más baja, una optimización que planeamos implementar en el futuro.

El rendimiento del programador de Ray es fundamental para implementar primitivas como allreduce. En la Figura 12b, inyectamos retrasos en la ejecución de tareas artificiales y mostramos que el rendimiento cae casi $2 \times$ con solo unos pocos ms de latencia adicional. Los sistemas con programadores centralizados como Spark y CIEL suelen tener gastos generales del programador en decenas de milisegundos [62, 38], lo que hace que tales cargas de trabajo sean poco prácticas. Programador *rendimiento* También se convierte en un cuello de botella ya que el número de tareas requeridas por el anillo se reduce cuadráticamente con el número de participantes.

5.2 Bloques de construcción

Las aplicaciones de un extremo a otro (por ejemplo, AlphaGo [54]) requieren una estrecha combinación de capacitación, servicio y simulación. En esta sección, aislamos cada una de estas cargas de trabajo en una configuración que ilustra los requisitos de una aplicación RL típica. Debido a un modelo de programación flexible dirigido a RL y un sistema diseñado para admitir este modelo de programación, Ray iguala y a veces supera el rendimiento de los sistemas dedicados para estas cargas de trabajo individuales.

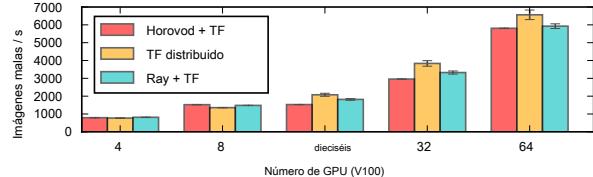


Figura 13: Imágenes por segundo alcanzadas al distribuir el entrenamiento de un modelo ResNet-101 TensorFlow (del punto de referencia oficial de TF). Todos los experimentos se ejecutaron en instancias p3.16xl conectadas por Ethernet de 25 Gbps, y los trabajadores asignaron 4 GPU por nodo como se hizo en Horovod [53]. Observamos algunas desviaciones de medición con respecto a lo informado anteriormente, probablemente debido a diferencias de hardware y mejoras recientes de rendimiento de TensorFlow. Usamos OpenMPI 3.0, TF 1.8 y NCCL2 para todas las ejecuciones.

5.2.1 Entrenamiento distribuido

Implementamos SGD síncrono paralelo a datos aprovechando la abstracción del actor Ray para representar réplicas de modelos. Los pesos de los modelos se sincronizan a través de allreduce (5.1) o del servidor de parámetros, ambos implementados sobre la API de Ray.

En la Figura 13, evaluamos el rendimiento de la implementación de SGD del servidor de parámetros (síncrono) Ray contra implementaciones de última generación [53], utilizando el mismo modelo de TensorFlow y el mismo generador de datos sintéticos para cada experimento. Solo comparamos con los sistemas basados en TensorFlow para medir con precisión la sobrecarga impuesta por Ray, en lugar de las diferencias entre los propios marcos de aprendizaje profundo. En cada iteración, los actores de réplica del modelo calculan gradientes en paralelo, envían los gradientes a un servidor de parámetros fragmentados y luego leen los gradientes sumados del servidor de parámetros para la siguiente iteración.

La Figura 13 muestra que Ray coincide con el rendimiento de Horovod y está dentro del 10% de TensorFlow distribuido (en distribuido replicado modo). Esto es debido a la capacidad de expresar las mismas optimizaciones a nivel de aplicación que se encuentran en estos sistemas especializados en la API de propósito general de Ray. Una optimización clave es la canalización del cálculo, la transferencia y la suma de gradientes dentro de una sola iteración. Para superponer el cálculo de la GPU con la transferencia de red, usamos un operador de TensorFlow personalizado para escribir tensores directamente en el almacenamiento de objetos de Ray.

5.2.2 Servicio

El servicio de modelos es un componente importante de las aplicaciones de un extremo a otro. Ray se centra principalmente en el *incrustado* envío de modelos a simuladores que se ejecutan dentro del mismo gráfico de tareas dinámicas (por ejemplo, dentro de una aplicación RL en Ray). En contraste, sistemas como Clipper [19] se enfocan en entregar predicciones a clientes externos.

En esta configuración, la baja latencia es fundamental para lograr una alta utilización. Para mostrar esto, en la Tabla 3 comparamos el

Sistema	Pequeña entrada	Entrada más grande
Clipper	4400 ± 15 estados / seg	6200 ± 21 estados / seg
Rayo	estados / seg	estados / seg

Tabla 3: Comparaciones de rendimiento para Clipper [19], un sistema de servicio dedicado, y Ray para dos cargas de trabajo de servicio integradas. Usamos una red residual y una pequeña red completamente conectada, tardando 10ms y 5ms en evaluar, respectivamente. Los clientes preguntan al servidor que cada uno envía estados de tamaño 4KB y 100KB respectivamente en lotes de 64.

rendimiento del servidor logrado usando un actor Ray para servir una política en lugar de usar el sistema Clipper de código abierto sobre REST. Aquí, los procesos de cliente y servidor se ubican en la misma máquina (una instancia p3.8xlarge). Este suele ser el caso de las aplicaciones RL, pero no de las cargas de trabajo de servicios web generales que tratan sistemas como Clipper. Debido a su serialización de baja sobrecarga y abstracciones de memoria compartida, Ray logra un rendimiento de orden de magnitud mayor para un modelo de política pequeño completamente conectado que toma una entrada grande y también es más rápido en un modelo de política de red residual más costoso, similar al usado en AlphaGo Zero, eso requiere una entrada más pequeña.

5.2.3 Simulación

Los simuladores utilizados en RL producen resultados con longitudes variables ("pasos de tiempo") que, debido al ciclo estrecho con el entrenamiento, deben usarse tan pronto como estén disponibles. Los requisitos de heterogeneidad y puntualidad de las tareas hacen que las simulaciones sean difíciles de soportar de manera eficiente en los sistemas de estilo BSP. Para demostrarlo, comparamos (1) una implementación de MPI que presenta 3 *norte* la simulación paralela se ejecuta en *norte* núcleos en 3 rondas, con una barrera global entre rondas §, a (2) un programa de Ray que emite los mismos 3 *norte* tareas mientras se recopilan simultáneamente los resultados de la simulación para el conductor. La tabla 4 muestra que ambos sistemas escalan bien, pero Ray logra hasta 1.8 × rendimiento. Esto motiva un modelo de programación que puede generar y recopilar dinámicamente los resultados de tareas de simulación de grano fino.

Sistema, modelo de programación	1 CPU	16 CPU	256 CPU
MPI, síncrono a granel	22.6 mil	208K	2,16 millones
Ray, tareas asíncronas	22.3 mil	290 mil	Los 4.03M

Tabla 4: Pasos de tiempo por segundo para el simulador Pendulum-v0 en OpenAI Gym [13]. Ray permite una mejor utilización cuando se ejecutan simulaciones heterogéneas a escala.

§ Tenga en cuenta que los expertos *lata* utilizamos las primitivas asíncronas de MPI para sortear las barreras, a expensas de una mayor complejidad del programa, no obstante, elegimos una implementación de este tipo para simular BSP.

5.3 Aplicaciones de RL

Sin un sistema que pueda acoplar estrechamente los pasos de entrenamiento, simulación y servicio, los algoritmos de aprendizaje por refuerzo hoy en día se implementan como soluciones únicas que hacen que sea difícil incorporar optimizaciones que, por ejemplo, requieren una estructura de cálculo diferente. o que utilizan arquitecturas diferentes. En consecuencia, con la implementación de dos aplicaciones de aprendizaje por refuerzo representativas en Ray, podemos igualar e incluso superar los sistemas personalizados construidos específicamente para estos algoritmos. La razón principal es la flexibilidad del modelo de programación de Ray, que puede expresar optimizaciones a nivel de aplicación que requerirían un esfuerzo de ingeniería sustancial para migrar a sistemas personalizados, pero son respaldados de forma transparente por el motor de ejecución de gráficos de tareas dinámicas de Ray.

5.3.1 Estrategias de evolución

Para evaluar Ray en cargas de trabajo de RL a gran escala, implementamos el algoritmo de estrategias de evolución (ES) y lo comparamos con la implementación de referencia [49], un sistema especialmente construido para este algoritmo que se basa en Redis para enviar mensajes. y bibliotecas de multiprocesamiento de bajo nivel para compartir datos. El algoritmo transmite periódicamente una nueva política a un grupo de trabajadores y agrega los resultados de aproximadamente 10000 tareas (cada una realiza de 10 a 1000 pasos de simulación).

Como se muestra en la Figura 14a, una implementación en Ray escala a 8192 núcleos. Duplicar los núcleos disponibles produce una aceleración del tiempo de finalización promedio de $1.6 \times$. Por el contrario, el sistema de propósito especial no se completa en 2048 núcleos, donde el trabajo en el sistema excede la capacidad de procesamiento del controlador de la aplicación. Para evitar este problema, la implementación de Ray utiliza un árbol de agregación de actores, alcanzando un tiempo medio de 3.7 minutos, más del doble de rápido que el mejor resultado publicado (10 minutos).

La paralelización inicial de una implementación en serie usando Ray requirió modificar solo 7 líneas de código. La mejora del rendimiento a través de la agregación jerárquica fue fácil de realizar con el soporte de Ray para tareas y actores anidados. Por el contrario, la implementación de referencia tenía varios cientos de líneas de código dedicadas a un protocolo para comunicar tareas y datos entre trabajadores, y requeriría más ingeniería para soportar optimizaciones como la agregación jerárquica.

5.3.2 Optimización de políticas próximas

Implementamos Proximal Policy Optimization (PPO) [51] en Ray y lo comparamos con una implementación de referencia altamente optimizada [5] que utiliza primitivas de comunicación OpenMPI. El algoritmo es una recopilación de dispersión asincrónica, donde se asignan nuevas tareas a los actores de simulación a medida que

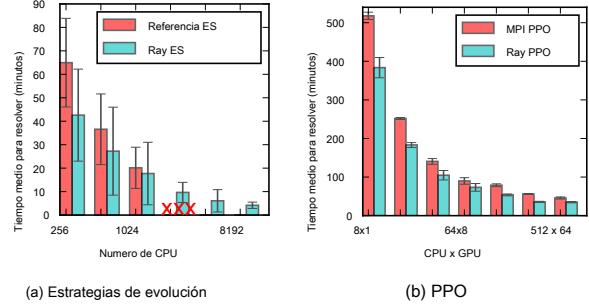


Figura 14: Tiempo para alcanzar una puntuación de 6000 en la tarea Humanoid-v1 [13]. (a) La implementación de Ray ES escala bien a 8192 núcleos y alcanza un tiempo medio de 3,7 minutos, más del doble de rápido que el mejor resultado publicado. El sistema de propósito especial no se ejecutó más allá de 1024 núcleos. ES es más rápido que PPO en este punto de referencia, pero muestra una mayor variación en el tiempo de ejecución. (B)

La implementación Ray PPO supera a una implementación MPI especializada [5] con menos GPU, a una fracción del costo. La implementación de MPI requirió 1 GPU por cada 8 CPU, mientras que la versión Ray requirió como máximo 8 GPU (y nunca más de 1 GPU por 8 CPU).

devuelva los despliegues al conductor. Las tareas se envían hasta que se recopilan 320000 pasos de simulación (cada tarea produce entre 10 y 1000 pasos). La actualización de la política realiza 20 pasos de SGD con un tamaño de lote de 32768. Los parámetros del modelo en este ejemplo son aproximadamente 350 KB. Estos experimentos se ejecutaron utilizando instancias p2.16xlarge (GPU) y m4.16xlarge (CPU alta).

Como se muestra en la Figura 14b, la implementación de Ray supera a la implementación de MPI optimizada en todos los experimentos, mientras utiliza una fracción de las GPU. La razón es que Ray es consciente de la heterogeneidad y permite al usuario utilizar arquitecturas asimétricas al expresar los requisitos de recursos en la granularidad de una tarea o actor. La implementación de Ray puede aprovechar la compatibilidad con múltiples GPU de proceso único de TensorFlow y puede anclar objetos en la memoria de la GPU cuando sea posible. Esta optimización no se puede trasladar fácilmente a MPI debido a la necesidad de recopilar implementaciones de forma asincrónica en un solo proceso de GPU. De hecho, [5] incluye dos implementaciones personalizadas de PPO, una que usa MPI para clústeres grandes y otra que está optimizada para GPU pero que está restringida a un solo nodo. Ray permite una implementación adecuada para ambos escenarios.

La capacidad de Ray para manejar la heterogeneidad de los recursos también redujo el costo de PPO en un factor de 4.5 [4], ya que las tareas de sólo CPU se pueden programar en instancias más baratas de alta CPU. En contraste, las aplicaciones MPI a menudo exhiben arquitecturas simétricas, en las que todos los procesos ejecutan el mismo código y requieren recursos idénticos, en este caso evitando el uso de máquinas solo con CPU para escalado horizontal. Además, la implementación de MPI requiere instancias bajo demanda ya que no maneja fallas de manera transparente. Asumiendo 4 × instancias puntuales más baratas, *La tolerancia a fallas de Ray y la programación consciente de los recursos reducen los costos en 18 ×*.

6 Trabajo relacionado

Gráficos de tareas dinámicos. Ray está estrechamente relacionado con CIEL [40] y Dask [48]. Los tres admiten gráficos de tareas dinámicas con tareas anidadas e implementan la abstracción de futuros. CIEL también proporciona tolerancia a fallas basada en el linaje, mientras que Dask, como Ray, se integra completamente con Python. Sin embargo, Ray se diferencia en dos aspectos que tienen importantes consecuencias en el rendimiento. Primero, Ray amplía el modelo de tareas con una abstracción de actor. Esto es necesario para un cálculo con estado e fiel en el entrenamiento y el servicio distribuidos, para mantener los datos del modelo junto con el cálculo. En segundo lugar, Ray emplea un planificador y un plan de control totalmente distribuidos y desacoplados, en lugar de depender de un único maestro que almacena todos los metadatos. Esto es crítico para soportar de manera eficiente primitivas como allreduce sin modificar el sistema. Al máximo rendimiento para 100 MB en 16 nodos, todos reduzca en Ray (Sección 5.1) envía 32 rondas de 16 tareas en 200 ms. Mientras tanto, Dask informa un rendimiento máximo del programador de 3000 tareas / s en 512 núcleos [3]. Con un planificador centralizado, cada ronda de reducción total ocurriría en un mínimo de ~5 ms de retraso en la programación, lo que se traduce en hasta 2 × peor tiempo de finalización (Figura 12b). Incluso con un programador descentralizado, acoplar la información del plano de control con el programador deja a este último en la ruta crítica para la transferencia de datos, lo que agrega un viaje de ida y vuelta adicional a cada ronda de reducción total.

Sistemas de flujo de datos. Los sistemas de flujo de datos populares, como MapReduce [20], Spark [65] y Dryad [28] tienen una adopción generalizada para cargas de trabajo de análisis y ML, pero su modelo de cálculo es demasiado restrictivo para una carga de trabajo de simulación dinámica y precisa. Spark y MapReduce implementan el modelo de ejecución BSP, que asume que las tareas dentro de la misma etapa realizan el mismo cálculo y toman aproximadamente la misma cantidad de tiempo. Dryad relaja esta restricción pero carece de soporte para gráficos de tareas dinámicos. Además, ninguno de estos sistemas proporciona una abstracción de actor, ni implementa un plan de control y planificador escalable distribuido. Por último, Naiad [39] es un sistema de flujo de datos que proporciona una escalabilidad mejorada para algunas cargas de trabajo, pero solo admite gráficos de tareas estáticos.

Marcos de aprendizaje automático. TensorFlow [7] y MXNet [18] apunta a cargas de trabajo de aprendizaje profundo y aprovecha de manera eficiente tanto las CPU como las GPU. Si bien logran un gran rendimiento para las cargas de trabajo de entrenamiento que consisten en DAG estáticos de operaciones de álgebra lineal, tienen soporte limitado para el cálculo más general requerido para acopiar estrechamente el entrenamiento con la simulación y el servicio integrado. TensorFlow Fold [33] proporciona cierto soporte para gráficos de tareas dinámicos, así como MXNet a través de sus API internas de C++, pero ninguno admite completamente la capacidad de modificar el DAG durante la ejecución en respuesta al progreso de la tarea, los tiempos de finalización de la tarea o fallas. TensorFlow y MXNet en principio logran generalidad al permitir que el programa

mer para simular primitivas de sincronización y transmisión de mensajes de bajo nivel, pero las dificultades y la experiencia del usuario en este caso son similares a las de MPI. OpenMPI [22] puede lograr un alto rendimiento, pero es relativamente difícil de programar ya que requiere una coordinación explícita para manejar gráficos de tareas heterogéneos y dinámicos. Además, obliga al programador a manejar explícitamente la tolerancia a fallas.

Sistemas de actores. Orleans [14] y Akka [1] son dos frameworks de actores muy adecuados para desarrollar sistemas concurrentes y de alta disponibilidad. Sin embargo, en comparación con Ray, brindan menos soporte para la recuperación de la pérdida de datos. Para recuperar *actores estatales*, el desarrollador de Orleans debe controlar explícitamente el estado del actor y las respuestas intermedias. *Actores apátridas* en Orleans se pueden replicar para escalamiento horizontal y, por lo tanto, podrían actuar como tareas, pero a diferencia de Ray, no tienen linaje. De manera similar, aunque Akka admite explícitamente el estado del actor persistente a través de fallas, no proporciona una tolerancia de falla eficiente para *computación sin estado* es decir, tareas. Para la entrega de mensajes, Orleans proporciona al menos una vez y Akka proporciona semántica como máximo una vez. Por el contrario, Ray proporciona tolerancia a fallas transparente y semántica de una sola vez, ya que cada llamada al método se registra en el GCS y tanto los argumentos como los resultados son inmutables. Descubrimos que, en la práctica, estas limitaciones no afectan el rendimiento de nuestras aplicaciones. Erlang [10] y C ++ Actor Framework [17], otros dos sistemas basados en actores, tienen un soporte igualmente limitado para la tolerancia a fallos.

Almacenamiento y programación de control global. El concepto de centralizar lógicamente el plano de control se ha propuesto previamente en redes definidas por software (SDN) [16], sistemas de archivos distribuidos (p. ej., GFS [23]), gestión de recursos (p. ej., Omega [52]) y tramas distribuidas funciona (por ejemplo, MapReduce [20], BOOM [9]), por nombrar algunos. Ray se inspira en estos esfuerzos pioneros, pero proporciona mejoras significativas. A diferencia de SDN, BOOM y GFS, Ray desacopla el almacenamiento de la información del plano de control (p. Ej., GCS) de la implementación lógica (p. Ej., Programadores). Esto permite que las capas de almacenamiento y de computación escalen de forma independiente, lo cual es clave para lograr nuestros objetivos de escalabilidad. Omega utiliza una arquitectura distribuida en la que los programadores se coordinan mediante un estado compartido globalmente. A esta arquitectura, Ray agrega programadores globales para equilibrar la carga entre los programadores locales y apunta a nivel ms,

Ray implementa un programador ascendente distribuido único que es escalable horizontalmente y puede manejar gráficos de tareas construidos dinámicamente. A diferencia de Ray, la mayoría de los sistemas informáticos de clúster existentes [20, 64, 40] utilizan una arquitectura de planificador centralizado. Si bien Sparrow [45] está descentralizado, sus programadores toman decisiones independientes, limitando las posibles políticas de programación, y todas las tareas de un trabajo son manejadas por el mismo programador global. Mesos [26] implementa un planificador jerárquico de dos niveles, pero su planificador de nivel superior gestiona marcos, no tareas individuales.

Canary [47] logra un rendimiento impresionante al hacer que cada instancia del programador maneje una parte del gráfico de tareas, pero no maneja gráficos de cálculo dinámico.

Cilk [12] es un lenguaje de programación paralelo cuyo planificador de robo de trabajo logra un equilibrio de carga demostrablemente eficiente para gráficos de tareas dinámicos. Sin embargo, sin un coordinador central como el programador global de Ray, este diseño completamente paralelo también es difícil de extender para admitir la ubicación de datos y la heterogeneidad de recursos en un entorno distribuido.

7 Discusión y experiencias

Construir Ray ha sido un largo viaje. Comenzó hace dos años con una biblioteca Spark para realizar simulaciones y entrenamientos distribuidos. Sin embargo, la relativa en flexibilidad del modelo BSP, la alta sobrecarga por tarea y la falta de abstracción del actor nos llevaron a desarrollar un nuevo sistema. Desde que lanzamos Ray hace aproximadamente un año, varios cientos de personas lo han usado y varias compañías lo están ejecutando en producción. Aquí discutimos nuestra experiencia desarrollando y usando Ray, y algunos comentarios de los primeros usuarios.

API. Al diseñar la API, hemos enfatizado el minimalismo. Inicialmente comenzamos con un básico *tarea* abstracción. Más tarde, agregamos el *Espere()* primitivo para acomodar despliegues con duraciones heterogéneas y el *actor* abstracción para acomodar simuladores de terceros y amortizar la sobrecarga de inicializaciones costosas. Si bien la API resultante es de nivel relativamente bajo, ha demostrado ser potente y fácil de usar. Ya hemos utilizado esta API para implementar muchos algoritmos RL de última generación además de Ray, incluidos A3C [36], PPO [51], DQN [37], ES [49], DDPG [55] y Ape-X [27]. En la mayoría de los casos, solo nos llevó unas pocas decenas de líneas de código portar estos algoritmos a Ray. Según los primeros comentarios de los usuarios, estamos considerando mejorar la API para incluir primitivas y bibliotecas de mayor nivel, que también podrían informar las decisiones de programación.

Limitaciones. Dada la generalidad de la carga de trabajo, las optimizaciones especializadas son difíciles. Por ejemplo, debemos tomar decisiones de programación sin un conocimiento completo del gráfico de cálculo. La programación de optimizaciones en Ray puede requerir perfiles de tiempo de ejecución más complejos. Además, almacenar el linaje para cada tarea requiere la implementación de políticas de recolección de basura para limitar los costos de almacenamiento en el GCS, una característica que estamos desarrollando activamente.

Tolerancia a fallos. A menudo se nos pregunta si la tolerancia a fallas es realmente necesaria para las aplicaciones de IA. Después de todo, debido a la naturaleza estadística de muchos algoritmos de IA, uno podría simplemente ignorar los despliegues fallidos. Según nuestra experiencia, nuestra respuesta es "sí". Primero, la capacidad de ignorar fallas hace que las aplicaciones sean mucho más fáciles de escribir y razonar. En segundo lugar, nuestra implementación particular de tolerancia a fallas mediante la reproducción determinista simplifica drásticamente la depuración, ya que nos permite reproducir fácilmente la mayoría de los errores. Esto es particularmente importante ya que, debido a su estocasticidad, la IA al-

goritmos son muy difíciles de depurar. En tercer lugar, la tolerancia a fallos ayuda a ahorrar dinero, ya que nos permite ejecutar recursos económicos como instancias puntuales en AWS. Por supuesto, esto tiene el precio de algunos gastos generales. Sin embargo, encontramos que esta sobrecarga es mínima para nuestras cargas de trabajo objetivo.

GCS y escalabilidad horizontal. La dramaturgia de GCS

Desarrollo y depuración de Ray simplificó los códigos. Nos permitió consultar todo el estado del sistema mientras depuramos el propio Ray, en lugar de tener que exponer manualmente el estado del componente interno. Además, el GCS también es el backend de nuestra herramienta de visualización de la línea de tiempo, que se utiliza para la depuración a nivel de aplicación.

El GCS también fue fundamental para la escalabilidad horizontal de Ray. En la Sección 5, pudimos escalar agregando más fragmentos cada vez que el GCS se convirtió en un cuello de botella. El GCS también permitió al programador global escalar simplemente agregando más réplicas. Debido a estas ventajas, creemos que la centralización del estado de control será un componente clave del diseño de los futuros sistemas distribuidos.

8 Conclusión

Ningún sistema de propósito general hoy en día puede soportar de manera eficiente el circuito cerrado de entrenamiento, servicio y simulación. Para expresar estos componentes básicos y satisfacer las demandas de las aplicaciones de IA emergentes, Ray unifica modelos de programación de actores y de tareas paralelas en un solo gráfico de tareas dinámico y emplea una arquitectura escalable habilitada por el almacén de control global y un programador distribuido de abajo hacia arriba. La flexibilidad de programación, el alto rendimiento y las bajas latencias logradas simultáneamente por esta arquitectura son particularmente importantes para las cargas de trabajo de inteligencia artificial emergentes, que producen tareas diversas en sus requisitos de recursos, duración y funcionalidad. Nuestra evaluación demuestra escalabilidad lineal de hasta 1.8 millones de tareas por segundo, tolerancia transparente a fallas, y mejoras sustanciales en el rendimiento en varias cargas de trabajo de RL contemporáneas. Por lo tanto, Ray proporciona una poderosa combinación de flexibilidad, rendimiento y facilidad de uso para el desarrollo de futuras aplicaciones de IA.

9 Agradecimientos

Esta investigación es apoyada en parte por NSF CISE Expeditions Award CCF-1730628 y obsequios de Alibaba, Amazon Web Services, Ant Financial, Arm, CapitalOne, Ericsson, Facebook, Google, Huawei, Intel, Microsoft, Scotiabank, Splunk y VMware, así como por la subvención de NSF DGE-1106400. Agradecemos a nuestros revisores anónimos ya nuestro pastor, Miguel Castro, por sus comentarios reflexivos, que ayudaron a mejorar la calidad de este artículo.

Referencias

- [1] Akka. <https://akka.io/>.
- [2] Apache Arrow. <https://arrow.apache.org/>.
- [3] Dask Benchmarks. <http://matthewrocklin.com/blog/trabajo/2017/07/03/escalado>.
- [4] Precios de las instancias EC2. <https://aws.amazon.com/ec2/precos/bajo-demanda/>.
- [5] Líneas de base de OpenAI: implementaciones de alta calidad de algoritmos de aprendizaje ment. <https://github.com/openai/líneas-de-base>.
- [6] Publicación de TensorFlow. <https://www.tensorflow.org/servicio/>.
- [7] A BADI, M., B ARHAM, P., C GALLINA, J., C GALLINA, Z., D AVIS, A., D EAN, J., D EVIN, M., G HEMAWAT, S., yo RVING, G., yo SARD, METRO, ET AL. TensorFlow: un sistema para el aprendizaje automático a gran escala. En *Actas del 12º Simposio USENIX sobre Diseño e Implementación de Sistemas Operativos (OSDI). Savannah, Georgia, Estados Unidos* (2016).
- [8] A GARWAL, A., B IRD, S., C OZOWICZ, M., H OANG, L., L ANG-VADO, J., L EE, S., L I, J., M ELAMED, HACER SHRI, G., R IBAS, O., S ES, S., Y SLIVKINS, A. Un servicio de decisiones de pruebas de múltiples mundos. *preimpresión de arXiv arXiv: 1606.03966* (2016).
- [9] A LVARO, P., C ONDIE, T., C EN CAMINO, N., E LMELEEGY, K., H ELLERSTEIN, JM, Y S OREJAS, R. BOOM Analytics: exploración de programación declarativa centrada en datos para la nube. En *Actas de la 5a conferencia europea sobre sistemas informáticos* (2010), ACM, pág. 223–236.
- [10] A RMSTRONG, J., V IRDING, R., W IKSTRÖM, C., Y W ILLIAMS, M. Programación concurrente en ERLANG.
- [11] B COMER, C., L EIHO, JZ, T EPLYASHIN, D., W ARD, T., W AINWRIGHT, M., K ÜTTLER, H., L EFRANCQ, A., G REEN, S., V ALDÉS, V., S ADIK, A., ET AL. DeepMind Lab. *preimpresión arXiv arXiv: 1612.03801* (2016).
- [12] B LUMOFE, RD, Y L EISERSON, CE Scheduling cálculos multiproceso por robo de trabajo. *J ACM* 46, 5 (septiembre 1999), 720–748.
- [13] B ROCKMAN, G., C HEUNG, V., P ETTERSSON, L., S SCHNEIDER, J., S CHULMAN, J., T ANG, J., Y Z AREMBA, W. Gimnasio OpenAI. *preimpresión arXiv arXiv: 1606.01540* (2016).
- [14] B YKOV, S., G ELLER, A., K LIOT, G., L ARUS, JR, P ANDYA, R., Y T HELIN, J. Orleans: Computación en la nube para todos. En *Actas del segundo simposio de ACM sobre computación en la nube* (2011), ACM, pág. dieciséis.
- [15] C ARBONE, EDUCACIÓN FÍSICA WEN, S., F ÓRA, G., H ARIDI, S., R ICHTER, S., Y T ZOUMAS, K. Gestión de estado en Apache Flink: Procesamiento de flujo distribuido con estado constante. *Proc. VLDB Endow.* 10, 12 (agosto de 2017), 1718–1729.
- [16] C ASADO, M., F REEDMAN, MJ, P ETTIT, J., L UO, J., M C K MI-PROPIO, NORTE., Y S HENKER, S. Ethane: Tomando el control de la empresa. *SIGCOMMComput. Comun. Apocalipsis* 37, 4 (agosto de 2007), 1–12.
- [17] C HAROUSSET, D., S CHMIDT, TC, H IESGEN, R., Y W ÄHLISCH, M. Actores nativos: una plataforma de software escalable para entornos distribuidos y heterogéneos. En *Actas del taller de 2013 sobre Programación basada en actores, agentes y control descentralizado* (2013), ACM, pág. 87–96.
- [18] C GALLINA, T., L I, M., L I, Y., L EN, M., W ANG, N., W ANG, M., X IAO, T., X U, B., Z COLGAR, C., Y Z COLGAR, Z. MXNet: biblioteca de aprendizaje automático flexible y eficiente para sistemas distribuidos heterogéneos. En *Taller NIPS sobre sistemas de aprendizaje automático (LearningSys'16)* (2016).
- [19] C RANKSHAW, D., W ANG, X., Z HOU, G., F RANKLIN, MJ, G ONZALEZ, JE, Y S TOICA, I. Clipper: un sistema de servicio de predicción en línea de baja latencia. En *XIV Simposio de USENIX sobre diseño e implementación de sistemas en red (NSDI 17)* (Boston, MA, 2017), Asociación USENIX, pág. 613–627.
- [20] D EAN, J., Y GRAMO HEMAWAT, S. MapReduce: procesamiento de datos simplificado en grandes conglomerados. *Comun. ACM* 51, 1 (enero de 2008), 107–113.
- [21] D ENNIS, JB, Y METRO ISUNAS, DP Una arquitectura preliminar para un procesador de flujo de datos básico. En *Actas del 2º Simposio Anual sobre Arquitectura de Computadores* (Nueva York, NY, EE. UU., 1975), ISCA '75, ACM, pág. 126–132.
- [22] G ABRIEL, E., F AGG, GE, B OSILCA, G., A NGSKUN, T., D EN-GARRA, JJ, S QUYES, JM, S AHAY, V., K AMBADUR, PAG., B ARRETT, B., L UMSDAINE, A., C UNA MANCHA, RH, D ANIEL, DJ, G RAHAM, RL, Y W OODALL, TS Open MPI: Objetivos, concepto y diseño de una implementación MPI de próxima generación. En *Actas de la XI Reunión del Grupo de Usuarios Europeos de PVM / MPI* (Budapest, Hungría, septiembre de 2004), pág. 97–104.
- [23] G HEMAWAT, S., G OBIOFF, H., Y L EUNG, S T. El sistema de archivos de Google. 29–43.
- [24] G ONZALEZ, JE, X EN, RS, D CRA, A., C RANKSHAW, D., F RANKLIN, MJ, Y S TOICA, I. GraphX: procesamiento de gráficos en un marco de flujo de datos distribuidos. En *Actas de la XI Conferencia de USENIX sobre diseño e implementación de sistemas operativos* (Berkeley, CA, EE.UU., 2014), OSDI'14, Asociación USENIX, pág. 599–613.
- [25] G U*, S., H OLLY*, E., L ILLICRAP, T., Y L EVINE, S. Aprendizaje de refuerzo profundo para la manipulación robótica con actualizaciones asincrónicas fuera de la política. En *Conferencia Internacional IEEE sobre Robótica y Automatización (ICRA 2017)* (2017).
- [26] H INDMAN, B., K ONWINSKI, A., Z AHARIA, M., G HODSI, A., J JOSEPH, AD, K ATZ, R., S HENKER, S., Y S TOICA, I. Mesos: una plataforma para compartir recursos de forma precisa en el centro de datos. En *Actas de la 8a Conferencia de USENIX sobre diseño e implementación de sistemas en red* (Berkeley, CA, Estados Unidos, 2011), NSDI'11, Asociación USENIX, pág. 295–308.
- [27] H ORGANO, D., Q UAN, J., B UDDE, D., B ARTH-METRO ARON, G., HESSEL, METRO., CAMIONETA H ASSLET, H., Y S ILVER, D. Reproducción distribuida de experiencias priorizadas. *Conferencia internacional sobre representaciones de aprendizaje* (2018).
- [28] YO SARD, M., B UDIU, MI U, Y., B IRRELL, A., Y F ETTERLY, D. Dryad: programas paralelos de datos distribuidos a partir de bloques de construcción secuenciales. En *Actas de la 2a Conferencia Europea ACM SIGOPS / EuroSys sobre Sistemas Informáticos 2007* (Nueva York, NY, EE.UU., 2007), EuroSys '07, ACM, pág. 59–72.
- [29] J IA, Y., S HELHAMER, E., D ONAHUE, J., K ARAYEV, S., L ONG, J., G IRSCHICK, R., G UADARRAMA, S., Y D ARRELL, T. Caffe: Arquitectura convolucional para una rápida incorporación de funciones. *arXiv preimpresión arXiv: 1408.5093* (2014).
- [30] J ORDAN, MI, Y METRO ITCELL, TM Machine learning: tendencias, perspectivas y prospectos. *Ciencia* 349, 6245 (2015), 255–260.

- [31] L EIBIUSKY, J., E ISBRUCH, GRAMO., Y S IMONASSI, D. *Consiguiendo Comenzó con Storm*. O'Reilly Media, Inc., 2012.
- [32] L I, M., A NDERSEN, DG, P ARCA, JW, S MOLA, AJ, A HMED, A., J OSIFOVSKI, V., L ONG, J., S HEKITA, EJ, Y S U, POR. Escalar el aprendizaje automático distribuido con el servidor de parámetros. En *Actas de la 11a Conferencia de USENIX sobre diseño e implementación de sistemas operativos* (Berkeley, CA, Estados Unidos, 2014), OSDI'14, págs. 583–598.
- [33] L OK, M., H ERRESHOFF, M., H UTCHINS, D., Y norte ORVIG, P. Aprendizaje profundo con gráficos de cálculo dinámico. *preimpresión arXiv arXiv: 1702.02181* (2017).
- [34] L AY, Y., G ONZALEZ, J., K YROLA, A., B ICKSON, D., GRAMO UESTRIN, C., Y H ELLERSTEIN, J. GraphLab: un nuevo marco para el aprendizaje automático paralelo. En *Actas de la 26^a Conferencia sobre Incertidumbre en Inteligencia Artificial* (Arlington, Virginia, Estados Unidos, 2010), UAI'10, págs. 340–349.
- [35] M ALEWICZ, G., A USTERN, MH, B IK, AJ, D EHNE, JC, H ORN, ILLINOIS EISER, NORTE., Y C ZAJKOWSKI, G. Pregeł: un sistema para el procesamiento de gráficos a gran escala. En *Actas de la Conferencia Internacional ACM SIGMOD de 2010 sobre Gestión de Datos* (Nueva York, NY, EE. UU., 2010), SIGMOD '10, ACM, págs. 135–146.
- [36] M NIH, V., B ADIA, AP, M IRZA, M., G RAVES, A., L ILLICRIP, T P, H ARLEY, T., S ILVER, D., Y K AVUKCUOGLU, K. Métodos asincrónicos para el aprendizaje por refuerzo profundo. En *Conferencia internacional sobre aprendizaje automático* (2016).
- [37] M NIH, V., K AVUKCUOGLU, K., S ILVER, DR USU, AUTOMÓVIL CLUB BRITÁNICO, V ENESS, J., B ELLEMARE, MG, G RAVES, A., R IEDMILLER, M., F IDJELAND, AK, O STROVSKI, GRAMO., ET AL. Control a nivel humano a través del aprendizaje por refuerzo profundo. *Naturaleza* 518, 7540 (2015), 529–533.
- [38] M URRAY, D. *Un motor de ejecución distribuido que admite datos Flujos de control dependiente*. Universidad de Cambridge, 2012.
- [39] M URRAY, DG, M C S HERRY, F., yo SAACS, R., yo SARD, M., B ARHAM, PAG., Y A BADI, M. Náyade: Un sistema de flujo de datos oportuno. En *Actas del vigésimo cuarto simposio de ACM sobre principios de sistemas operativos* (Nueva York, NY, EE. UU., 2013), SOSP '13, ACM, págs. 439–455.
- [40] M URRAY, DG, S CHWARZKOPF, SRA MOWTON, C., S MITH, S., M ADHAVAPEDDY, A., Y H Y, S. CIEL: Un motor de ejecución universal para la computación de flujo de datos distribuidos. En *Actas de la 8a Conferencia de USENIX sobre diseño e implementación de sistemas en red* (Berkeley, CA, EE. UU., 2011), NSDI'11, Asociación USENIX, págs. 113–126.
- [41] N AIRE, COMO RINIVASAN, P., B FALTA, S., A LCICEK, C., F EARON, R., M ARIA, AD, P ANNEERSHELVAM, V., S ULEY- HOMBRE, M., B COMER, C., P ETERSEN, S., L HUEVO, S., M NIH, V., K AVUKCUOGLU, K., Y S ILVER, D. Métodos masivamente paralelos para el aprendizaje por refuerzo profundo, 2015.
- [42] N GRAMO, A., C OATES, A., D IEL, M., G ANAPATHI, V., S CHULTE, J., T SE, B., B ERGER, MI., Y L IANG, E. Vuelo autónomo en helicóptero invertido mediante aprendizaje por refuerzo. *Robótica Experimental IX* (2006), 363–372.
- [43] N ISHIHARA, R., M ORITZ, P., W ANG, S T UMANOV, A., P AUL, W., S CHLEIER- S MITH, J., L IAW, R., N IKNAMI, M., J ORDAN, MI, Y S TOICA, I. Aprendizaje automático en tiempo real: las piezas que faltan. En *Taller sobre temas candentes en sistemas operativos* (2017).
- [44] O LÁPIZ AI. Bot de OpenAI Dota 2 1v1. <https://openai.com/> el internacional!, 2017.
- [45] O USTERHOUT, K., W ENDELL, P., Z AHARIA, METRO., Y S TOICA, I. Sparrow: programación distribuida de baja latencia. En *Actas del vigésimo cuarto simposio de ACM sobre principios de sistemas operativos* (Nueva York, NY, EE. UU., 2013), SOSP '13, ACM, págs. 69–84.
- [46] P ASZKE, A., G ROSS, S., C HINTALA, S., C HANAN, G., Y ANG, E., D mi V ITO, Z., L EN, Z., D ESMAISON, A., A NTIGA, L., Y L ERER, A. Diferenciación automática en PyTorch.
- [47] Q U, H., M ASHAYEKI, O., T EREI, D., Y L EVIS, P. Canary: una arquitectura de programación para computación en la nube de alto rendimiento. *preimpresión de arXiv arXiv: 1602.01412* (2016).
- [48] R OCKLIN, M. Dask: Cálculo paralelo con algoritmos bloqueados y programación de tareas. En *Actas de la 14th Python in Science Conference* (2015), K. Huff y J. Bergstra, Eds., Págs. 130 – 136.
- [49] S ALIMANS, T., H OH J., C GALLINA, X., Y S UTSKEVER, I. Las estrategias de evolución como alternativa escalable al aprendizaje por refuerzo. *preimpresión de arXiv arXiv: 1703.03864* (2017).
- [50] S ANFILIPPO, S. Redis: una estructura de datos en memoria de código abierto Tienda. <https://redis.io/>, 2009.
- [51] S CHULMAN, J., W OLSKI, F., D HARIWAL, P., R ADFORD, A., Y K LIMOV, O. Algoritmos de optimización de políticas próximas. *preimpresión de arXiv arXiv: 1707.06347* (2017).
- [52] S CHWARZKOPF, M., K ONWINSKI, A., A BD- mi L- METRO ALEK, METRO., Y W ILKES, J. Omega: programadores flexibles y escalables para grandes clústeres informáticos. En *Actas de la 8a Conferencia europea de ACM sobre sistemas informáticos* (Nueva York, NY, EE.UU., 2013), EuroSys '13, ACM, págs. 351–364.
- [53] S ERGEEV, A., Y D EL B ADEMÁS, M. Horovod: aprendizaje profundo distribuido rápido y fácil en tensor fl ujo. *preimpresión arXiv arXiv: 1802.05799* (2018).
- [54] S ILVER, D., H UANG, SOY ADDISON, CJ, G UEZ, A., S IFRE, L., V UN D ES D RIESSCHE, G., S CHRITTWIESER, J., A NTONOGLOU, I., P ANNEERSHELVAM, V., L ANCTOT, METRO., ET AL. Dominar el juego de Go con redes neuronales profundas y búsqueda de árboles. *Naturaleza* 529, 7587 (2016), 484–489.
- [55] S ILVER, D., L ALGUNA VEZ, G., H EESS, N., D EGRIS, T., W IERSTRAS, D., Y R IEDMILLER, M. Algoritmos de gradiente de política determinista. En *ICML* (2014).
- [56] S ILVER, D., H UANG, SOY ADDISON, CJ, G UEZ, A., S IFRE, L., V UN D ES D RIESSCHE, G., S CHRITTWIESER, J., A NTONOGLOU, I., P ANNEERSHELVAM, V., L ANCTOT, METRO., ET AL. Una introducción. Prensa del MIT Cambridge, 1998.
- [57] T HAKUR, R., R ABENSEIFNER, R., Y GRAMO ROPP, W. Optimización de las operaciones de comunicación colectiva en MPICH. *Revista internacional de aplicaciones informáticas de alto rendimiento* 19, 1 (2005), 49–66.
- [58] T IAN, Y., G ONG, Q., S COLGAR, W., W U, Y., Y Z ITNICK, CL ELF: una plataforma de investigación extensa, liviana y flexible para juegos de estrategia en tiempo real. *Avances en los sistemas de procesamiento de información neuronal (NIPS)* (2017).
- [59] T ODOROV, E., E REZ, T., Y T ASSA, Y. Mujoco: Un motor de física para el control basado en modelos. En *Robots y sistemas inteligentes (IROS), Conferencia internacional IEEE / RSJ de 2012 sobre* (2012), IEEE, págs. 5026–5033 .

[60] VUN D ES BERGIO, J., MILLER, DAKOTA DEL SUR UCKWORTH, D., HU, H., WUN, A., FU, X.-Y., GOLDBERG, K., YABBEEL, P. Realización sobrehumana de tareas quirúrgicas por robots utilizando el aprendizaje iterativo de demostraciones guiadas por humanos. En *Robótica y automatización (ICRA), Conferencia Internacional IEEE 2010 sobre* (2010), IEEE, págs. 2074-2081.

[61] CAMIONETA RENESSE, R., Y SCHNEIDER, Replicación de FB Chain para soportar un alto rendimiento y disponibilidad. En *Actas de la Sexta Conferencia sobre Simposio sobre Diseño e Implementación de Sistemas Operativos - Volumen 6* (Berkeley, CA, EE. UU., 2004), OSDI'04, Asociación USENIX.

[62] ENKATARAMAN, S., PANDA, A., OUSTERHOUT, KG HODSI, A., ARMBRUST, SEÑOR ECHT, B., FRANKLIN, METRO., Y S TOICA, I. Drizzle: procesamiento de flujo rápido y adaptable a escala. En *Actas del vigésimo sexto simposio de ACM sobre principios de sistemas operativos* (2017), SOSP '17, ACM.

[63] WHITE, T. *Hadoop: la guía definitiva*. O'Reilly Media, Inc., 2012.

[64] AHARIA, Z., HOWDHURY, M., COMO, D., CRAVEN, D., SOYAN, J., METRO, C., CAULEY, M., FRANKLIN, M.J., HENKER, S., YASA, ICA, I. Conjuntos de datos distribuidos resistentes: una abstracción tolerante a fallas para la computación en clúster en memoria. En *Actas de la novena conferencia de USENIX sobre diseño e implementación de sistemas en red* (2012), Asociación USENIX, págs. 2-2.

[65] AHARIA, Z., XEN, RS., ENDELL, P., COMO, D., ARMBRUST, M., CRAVEN, D., SOYAN, J., ENKATARAMAN, S., FRANKLIN, M.J., HODSI, A., GONZALEZ, J., HENKER, S., Y S TOICA, I. Apache Spark: un motor unificado para el procesamiento de big data. *Comun. ACM* 59, 11 (octubre de 2016), 56–65.