Departamento de Enxeñaría de Computadores.
Facultad de Informática de A Coruña
UNIVERSIDADE DA CORUÑA

TRABALLO DE FIN DE MÁSTER
MÁSTER INTERUNIVERSITARIO EN
COMPUTACIÓN DE ALTAS PRESTACIÓNS

Análise Distribuída de Datos Sobre a Marcha para Simulacións Numéricas baseado en Ray

Estudante: Xico Fernández Lozano
Director/a/es/as: Emilio José Padrón González

A Coruña, 20 de xuño de 2023.

A toda miña familia,

que sempre consegue sobrepoñerse a calquera adversidade.

Agradecementos

Primeiro de todo, quero agradecerlle a miña familia o apoio e a confianza recibidos en calquera decisión que tome. Moitas veces non ten que ser doado.

Grazas a Emilio polo trato, pola oportunidade, por confiar en min para este traballo e por estar dispoñible en calquera intre para axudarme.

Grazas a Bruno pola xentileza no día a día e por contribuír á miña formación dunha maneira máis que sobresaínte.

Grazas a Julien por plantexarme cuestións que me axudaron a avanzar no proxecto e nunca se me ocurrirían. Grazas tamén a todo o equipo da *Maison de la Simulation* por brindarme acceso a Ruche.

Grazas a Amal pola amabilidade, dispoñibilidade e axuda prestada para avanzar neste proxecto.

Resumo

O presente traballo trata de atopar unha solución baseada en Ray (*framework* de programación paralela distribuída de Pyrhon) para analizar sobre a marcha os datos producidos por unha simulación paralelizada con MPI pensada para ser executada nun contorno HPC. Un dos obxectivos do traballo é obter comparacións de rendemento e eficiencia entre as ferramentas Ray e Dask dentro do ámbito de análise de datos.

Abstract

This work aims to find a solution based on Ray (a distributed parallel programming framework for Python) to analyze on-the-fly the data produced by an MPI-parallelized simulation designed to be executed in an HPC environment. One of the objectives of the work is to obtain performance and efficiency comparisons between the Ray and Dask tools within the scope of data analysis.

Palabras chave:

- Análise de datos
- Computación de Altas Prestacións
- Programación paralela distribuída
- In situ
- In transit
- Big Data
- Ray

Keywords:

- Data analytics
- High Performance Computing
- Distributed parallel programming
- In situ
- In transit
- Big Data
- Ray

Índice xeral

[1] Entrada/Saída

Índice de figuras

Índice de táboas

Capítulo 1

Introdución

Hoxe en día, gran parte do código paralelizado e executado en supercomputadores está baseado en MPI (Message Passing Interface), unha das ferramentas máis utilizadas pola comunidade. Este tipo de códigos paralelos poden ser executados a grande escala, producindo elevadas cantidades de datos que seguen unha tendencia incremental ligada ao aumento da capacidade computacional do sistema. A análise efectiva e eficiente deses datos presenta novos desafíos e obstáculos.

Neste traballo propoñemos unha solución para a análise distribuída e sobre a marcha (en tempo real, a medida que se obtén a información) dos datos producidos por simulacións numéricas paralelizadas con MPI, buscando aliviar o atoamento que supón utilizar o subsistema de entrada/saída para escribir toda a información xerada para a súa posterior análise.

1.1 Contexto e Motivación

Na era actual da información, a capacidade para xerar grandes volumes de datos é cada vez máis accesible grazas aos avances no ámbito da computación de altas prestacións, informática de alto rendemento ou HPC (High Performance Computing). Os sistemas con este tipo de características permiten a execución de aplicacións altamente complexas e a realización de simulacións a gran escala que non sería posible executalas nun sistema convencional.

A día de hoxe, o fenómeno coñecido como *big data*, que trata de representar cantidades masivas de datos impropias dun sistema convencional, volveuse unha realidade innegable neste contexto. As simulacións HPC adoitan ser programas iterativos que evolucionan ao longo do tempo e poden chegar a producir varios TB de información por hora. Os datos froito deste tipo de programas, en moitos casos, necesitan ser procesados ou analizados para extraer información relevante agochada detrás da súa interpretación. Cos métodos tradicionais, os datos xerados por este tipo de aplicacións deben ser escritos nun disco para ser lidos posteriormente, este método coñécese como postprocesamento ou procesamento *post-hoc*. Hoxe en día, as

distintas librerías de programación paralela distribuída preparadas para traballar con grandes conxuntos de datos facilitan e incrementan o rendemento e a eficiencia do proceso de análise dos datos producidos por este tipo de simulacións.

O presente traballo céntrase en abordar a necesidade de realizar análises sobre a marcha ou en tempo real dos datos xerados por simulacións numéricas complexas e paralelizadas con MPI, **de agora en adiante chamaremos simulación ao programa que actúa como fonte dos datos de análise**. Como xa vimos, a estratexia de análise *post-hoc* presenta limitacións en canto a capacidade de resposta e rendemento, xa que a xestión da E/S (Entrada e Saída) para mover os ficheiros a disco supón un atoamento no sistema debido á latencia natural deste tipo de operacións. Por isto, a análise realizarase en tempo real en lugar de gardar os datos xerados nun ficheiro para a súa posterior interpretación, evitando así o retardo producido polo acceso a disco dende a simulación.

Existen dous enfoques para realizar este tipo de análises sobre a marcha, *in situ* ou *in transit*. A análise *in situ* refírese á técnica na que se realiza a análise dos datos nos procesadores fisicamente próximos ao lugar onde se producen os datos ou nos propios procesadores utilizados pola simulación, desta maneira, redúcense os tempos de comunicación e conségue-se executar as tarefas de análise o máis rapidamente posible. Normalmente, estas simulacións son executadas en diferentes nodos de computación, polo que sería necesario dispoñer de hardware para realizar a análise *in situ* en cada un dos nodos utilizados para a simulación. Por outra banda, a estratexia *in transit* baséase en transmitir os datos xerados en tempo real dende o sistema no que son producidos a un sistema de análise centralizado situado en nodos de computación externos á simulación, desta maneira, ao dispoñer de máis recursos, pódense realizar tarefas de procesamento intensivas. Existe unha terceira técnica que consiste en combinar as dúas anteriores, dando lugar a unha estratexia híbrida. [1]

Neste traballo implementarase unha solución que baseará as tarefas de análise no *framework* de programación paralela distribuída Ray[2], en Python. Estudarase como integrar este tipo de estratexias de procesamento (*in situ*, *in transit*...) no marco de traballo de Ray para levar a cabo o análise de datos producidos por simulacións pesadas e paralelizadas con MPI nunha contorna HPC, aproveitando os recursos dispoñibles neste tipo de sistemas. Examina-ranse os desafíos inherentes á manipulación de grandes cantidades de datos e investigaranse técnicas eficientes para o seu procesamento en tempo real. Ademais avaliarase o rendemento do sistema en diferentes casos de uso.

Os principais motivos que nos levan á realización deste traballo están directamente relacionado co descrito nos parágrafos anteriores, falamos dun incremento expoñencial a nivel de manexo de datos. Isto débese, en gran parte, á accesibilidade a recursos sofisticados grazas a conceptos coma a computación na nube, onde non é necesario dispoñer do hardware físico necesario para executar unha simulación complexa que produza moitos datos. Por iso, o

alcance e a utilidade que pode chegar a ter este traballo na comunidade científica convírtese nunha das principais motivacións.

Nesta situación, os sistemas computación de altas prestacións desempeñan un papel fundamental xa que dispoñen de capacidade para executar este tipo de programas a gran escala con tempos razoables e ofrecendo novas posibilidades que eviten almacenar os datos en disco realizar unha análise *post-hoc*. Evitar o atoamento producido polo subsistema de entrada e saída, no momento de analizar os datos producidos, tamén será un desafío á hora de implementar unha solución para este obstáculo propio do análise *post-hoc*. Ademais, a análise de datos en tempo real, permite tomar decisións rápidas de maneira áxil e dinámica baseándonos en parámetros da simulación en curso como poden ser a cantidade de datos gardados en memoria ou a carga computacional nun determinado instante.

A día de hoxe existen outras solucións para a análise de datos *in situ* como pode ser a ferramenta DEISA (Dask-Enabled In Situ Analytics)[3] baseada na librería Dask, como o seu propio nome indica. DEISA apóiase principalmente no concepto de Dask Array, unha funcionalidade que permite dividir un *array* de grandes dimensións en *chunks* para o seu procesamento distribuído e obter o mesmo resultado que ao utilizar un *array* convencional nun menor tempo, este concepto de Dask Array axuda a deseñar unha visión global e sinxela para o usuario á hora de introducir o código de análise. DEISA está instrumentado con PDI (PDI Data Interface)[4], unha interface capaz de executar, de maneira síncrona e dende un código MPI en C, C++ ou Fortran, un *script* de Python conservando diferentes datos do programa orixinal. Esta ferramenta (PDI) é a base para aproximar os datos producidos por unha simulación paralelizada con MPI deseñada nunha linguaxe de baixo nivel (tipicamente utilizadas por motivos de rendemento), ás distintas librerías de programación paralela distribuída de Python para executar as correspondentes tarefas de análise de maneira asíncrona ao longo das iteracións nas que se divide a simulación.

No noso caso, como xa avanzamos, faremos uso do *framework* de programación paralela distribuída Ray, en Python, aproveitando a súa flexibilidade e a gran cantidade de posiblidades que ofrece, especialmente, traballando nunha contorna HPC. A diferencia de Dask, Ray posúe un gran desenvolvemento arredor do concepto de actores ou Ray Actors, elementos con persistencia de datos ao longo do tempo, moi interesante para preservar os datos a analizar durante as iteracións. Por outro lado, a xestión de memoria de Ray baséase en Plasma[5], un tipo de memoria compartida de alto rendemento desenvolta por Apache. A idea de obter maior eficiencia con Ray que con Dask e poder comparar o comportamento das dúas librerías en termos de comunicacións e computacións neste escenario forma tamén parte das motivacións deste proxecto.

En xeral, as motivacións deste traballo xorden da necesidade de facer fronte á explosión de datos xerados por simulacións cada vez máis complexas e sofisticadas. Os recursos dun

sistema HPC dannos a posibilidade de poder desenvolver unha solución a grande escala que execute tarefas de análise de maneira asíncrona sobre os datos xerados pola simulación[6]. Estas motivacións impulsan a investigación, o desenvolvemento e a implementación dunha nova ferramenta baseada no *framework* de programación paralela distribuída Ray para a análise de datos en HPC.

1.2 Obxectivos

O principal obxectivo do traballo consiste en obter unha solución baseada en Ray para o problema descrito na sección anterior (1.1). Trataremos de partir da solución existente, DEISA, para obter unha ferramenta cun funcionamento similar que permita ao usuario introducir o código correspondente para realizar as tarefas de análise sobre os datos producidos na simulación. É importante ofrecer ao usuario flexibilidade en canto ás tarefas de análise a executar e permitirlle modificar os cómputos de análise ao longo das iteracións recibidas dende a simulación podendo así decidir que se executa en cada momento. Ademais, intentaremos aproveitar funcionalidades de Ray como os Ray Actors ou a memoria compartida Plasma para obter un maior rendemento que con DEISA.

De maneira xeral, o obxectivo será obter unha solución pensada para ser executada nunha contorna HPC, e baseada en Ray, para a análise de datos producidos por unha simulación paralelizada con MPI. Utilizaremos enfoques de análise de datos sobre a marcha (*in situ* ou *in transit*) que eviten o atoamento producido por gardar os datos que se queren analizar mediante operacións de entrada e saída cunha técnica de análise de datos *post-hoc*. Unha vez obtida unha solución válida, o seu rendemento será comparado co de DEISA, ferramenta co mesmo propósito pero baseada en Dask. Para os experimentos utilizaranse distintos códigos de análise e modificaranse variables de configuración como o número de procesos MPI que se executan na simulación, o nivel de paralelismo no sistema de análise, a cantidade de datos xerados por proceso MPI en cada iteración e o número de iteracións. Os obxectivos destes experimentos serán obter conclusións analizando o comportamento de Ray e Dask fronte a situacións similares e decidir cal delas obtén un mellor rendemento realizando análise de datos en tempo real nunha contorna HPC.

1.3 Estrutura da Memoria

Para obter unha vista global do mesmo, nesta sección, organizaremos o presente traballo nos seguintes capítulos:

- **Introdución**: O presente capítulo. Trata o contexto, a motivación, os obxectivos e a estrutura da memoria.

- **Estado da arte**: Capítulo para expoñer o traballo existente relacionado con este proxecto e analizar o actual estado da arte.
- **Fundamentos tecnolóxicos**: Amosar as ferramentas utilizadas para a realización do proxecto e o papel que desempeñaron tanto a nivel de hardware como de software.
- **Arquitectura**: Capítulo que explica o deseño, implementación e funcionamento en xeral da solución proposta para este proxecto.
- **Experimentos**: Datos recollidos das probas realizadas e conclusións ao respecto.
- **Conclusións**: Visión final do traballo realizado e futuras liñas de traballo.

Capítulo 2

Estado da Arte

Neste capítulo analízase o estado da arte no marco de traballo no que se encadra este proxecto. No capítulo trátanse as diferentes estratexias de análise e exemplos prácticos na realidade actual, ferramentas software utilizadas pola comunidade que permitan realizar as tarefas de análise de maneira asíncrona e amósanse diferentes opcións para o tratamento dos datos xerados pola simulación amosando tamén algunhas das ferramentas con máis renome.

Finalmente, o capítulo profundiza no deseño, arquitectura e implementación de DEISA. Este último punto considérase fundamental para entender a base deste proxecto xa que DEISA é a ferramenta que se toma como referencia e punto de partida para elaborar a nosa solución baseada en Ray.

2.1 Análise de Datos Sobre a Marcha

A análise de datos en tempo real ofrécenos unha vantaxe competitiva moi significativa á hora de toma de decisións con pouca marxe temporal, optimización de cómputos e detectar inmediatamente patróns nos datos obtidos dende a simulación. Para este tipo de análises pódense utilizar as estratexias *in transit* (en nodos de computación adicados e externos á simulación, implicando movemento de datos), *in situ* (nos mesmos nodos de cómputo que a simulación) ou utilizar un enfoque híbrido que combina ámbalas dúas metodoloxías. Este tipo de técnicas de análise sobre a marcha aportan gran cantidade de vantaxes en canto a rendemento e toma decisións en tempo real en función aos datos que foron xerados en intres anteriores.

Como avanzamos sobre a técnica *in situ*, consiste na análise de datos na ubicación actual evitando a latencia da súa transferencia a un sistema de análise centralizado. Este obxectivo implica uso de recursos no mesmo lugar onde se xeran os datos. Un exemplo deste caso podería ser un sistema embebido que necesite obter resultados sobre os datos de entrada de maneira inmediata, como por exemplo, un vehículo autónomo, onde os diferentes sensores do

automóbil encárganse de recoller datos que son analizados a bordo e de maneira inmediata sen necesidade de transferilos a un sistema externo. Este enfoque tamén é moi utilizado para obter unha visualización interactiva dos datos xerados, aspecto no que se enfocan moitos dos traballos realizados pola comunidade [7] [8].

Por outra banda, se nos referimos ao concepto *in transit*, falamos da técnica consistente en analizar os datos de maneira remota, é dicir, sería necesario transferilos dende o seu lugar de orixe a un sistema de análise de datos centralizado. Nesta estratexia son de vital importancia conceptos como o ancho de banda ou a latencia da rede utilizada para as comunicacións, unha boa solución para este aspecto podría ser Infiniband[9], unha rede de alto rendemento e baixa latencia utilizada especialmente en contornas HPC; está claro que a distancia entre a fonte de datos e o sistema de análise será un condicionante á hora de elexir a tecnoloxía para as comunicacións. Como exemplo para esta estratexia podemos tomar a tecnoloxía IoT ou Internet das Cousas, onde sistemas convencionais, por exemplo un electrodoméstico, que recolle grandes cantidades de datos e para poder analizalos necesita transferilos a un sistema externo debido a limitacións nos recursos hardware.

Combinando estes dous enfoques conseguimos unha aproximación híbrida na que se adoita realizar operacións de compresión sobre os datos a analizar na ubicación onde son producidos (*in situ*) para axilizar a súa transferencia de cara a un posterior análise remoto (*in transit*) como veñen realizando algúns dos proxectos relacionados[1]. Poderiamos poñer como exemplo unha fábrica con diferentes liñas de produción situadas en diferentes habitáculos, xerándose unha cantidade masiva de datos en cada liña de produción; nesta situación sería interesante facer unha análise *in situ* en cada liña de produción facendo un proceso de selección sobre os datos e reducir a latencia para unha posterior transferencia que permita realizar unha análise *in transit* centralizada cos datos recibidos de cada liña de produción.

2.2 Ferramentas para a Computación Paralela Distribuída

Para desenvolver unha solución adicada á análise de datos nunha contorna HPC, como no noso caso, existen distintas ferramentas externas e marcos de traballo (ou *frameworks*) que poden ser utilizados durante o proceso de desenvolvemento do software. Unha boa elección para as ferramentas a utilizar será clave para obter o rendemento desexado. Neste ámbito adoitan utilizarse ferramentas de programación paralela distribuída, permitindo un procesamento asíncrono, eficiente e escalable para grandes volumes de datos. Algunhas das ferramentas máis destacadas a día de hoxe para este propósito son as seguintes:

- **Modelo MapReduce.**:
  - **Apache Hadoop**: *Framework* de código aberto que permite o procesamento dis-

tribuído de grandes volumes de datos nun ecosistema HPC. Está baseado no modelo MapReduce, consistente en dividir un problema en subproblemas para repartir a carga de traballo de forma paralela. Hadoop ofrece un sistema de arquivos distribuído chamado HDFS moi interesante para a análise de datos nun sistema HPC.[10]

- **Apache Spark**: Sistema de procesamento distribuído en memoria que permite un análise rápido e escalable de grandes volumes de datos, da mesma maneira que Hadoop. Baséase nun deseño modular e ofrece un amplo conxunto de librerías que o fan unha posible alternativa para realizar análise de datos de maneira distribuída. En xeral, baséase no mesmo concepto que Hadoop pero engadindo funcionalidades que aportan flexibilidade e melloran o rendemento.[11]
- **Apache Flink**: *Framework* sistema de procesamento de datos en tempo real e a gran escala de código aberto. Baséase no método de *pipelining*. Útil para solucións con procesos lentos e pesados que manexen cantidades masivas de datos[12]. Existen solucións baseadas en Apache Flink interesantes para o análise de grandes cantidades de datos coas que pode asumirse un procesamento lento[13].

- **Librerías de Python de programación paralela distribuída.**:
  - **Parsl**: Librería de programación paralela para Python que permite describir funcións executadas en paralelo e inxectar dependencias para crear fluxos de traballo ou execución. Parsl permite crear programas paralelos compostos por funcións de Python de maneira sinxela. Unha das súas propiedades é a adaptación do mesmo código a distintos contornos podendo executarse en calquera tipo de hardware dende un computador persoal ata un supercomputador.[14]
  - **Dask**: Librería de Python que permite escalar a contornas HPC as ferramentas tradicionais utilizadas para a análise de datos como NumPy, Pandas ou Scikit-Learn. Dask ofrece coleccións procesables de forma paralela como Dask Arrays ou Dask Dataframe. [15]
  - **Ray**: *Framework* de desenvolvemento software en Python que facilita a construción e execución de aplicacións distribuídas que requiran alto rendemento, escalabilidade e tolerancia a fallos. Ray baséase no concepto de tarefa remota asíncrona, utiliza un sistema de memoria compartida e de alta velocidade baseado na memoria Plasma. Dispón dunha API para xestionar os obxectos gardados en memoria e a invocación e control de tarefas remotas. Ray ofrece tamén o concepto de Ray Actor, capaz de executar tarefas podendo preservar datos ao longo do tempo. A diferencia de Dask, Ray permite a execución de tarefas aniñadas, permitindo que sexa unha tarefa a que invoque novas tarefas.[2]

Ademais de utilizar ferramentas que traballen a máis alto nivel poden utilizarse novas estratexias como pode ser conseguir unha granularidade fina de tarefas con MPI, existen solucións baseadas na librería TBB de Intel[6] e en OpenMP[16].

A medio camiño entre a granularidade fina de tarefas (análise con MPI, OpenMP, TBB...) e as enfocadas no *big data* poden encaixar algunhas das ferramentas mencionadas anteriormente. Algunhas delas son Dask, ferramenta na que se basa DEISA, e Ray, *framework* utilizado para elaborar a solución que define o presente proxecto.

2.2.1 Dask

Dask é unha librería de computación paralela e distribuída deseñada *ad-hoc* para o análise de datos en Python. Unha das principais características de Dask é a capacidade para manexar conxuntos de datos masivos que sobrepasarían os límites da memoria. Unha das súas propiedades fundamentais é a utilización de Dask Arrays, unha estructura de datos que representa un conxunto de *arrays* NumPy (librería de cálculo de Python baseada en OpenMP). Os Dask Arrays divídense en bloques ou *chunks* para habilitar o seu procesamento paralelo axudándose dun grafo de tarefas obtido a partir da función introducida polo usuario.

Dask tamén utiliza un sistema de planificación ou *scheduler* para coordinar e distribuír a execución das tarefas de computación entre os diferentes *workers*, encargados precisamente de albergar a carga computacional. O *scheduler* divide as tarefas que se desexan executar nun grafo de tarefas para maximizar a eficiencia da paralelización do código. Os *workers* son procesos que manexan varios fíos de execución capaces de executar tarefas en paralelo. Dask ofrece escalabilidade coa posibilidade de engadir *workers* segundo sexa necesario, permitindo procesar volumes masivos de datos de maneira distribuída e eficiente.

2.2.2 Ray

Por outro lado, Ray é un *framework* de código aberto enfocado á programación paralela distribuída mantendo escalabilidade e eficiencia. O modelo de memoria de Ray baséase na memoria Plasma, unha memoria compartida de alta velocidade que permite o almacenamento de grandes volumes de datos dunha maneira relativamente áxil[17]. Cada dato almacenado na memoria plasma é considerado un Ray Object, no momento que un Ray Object é almacenado, Ray xera unha referencia única para o Ray Object que permitirá a súa identificación e facilitará a comunicación entre procesos. A xestión da memoria plasma e o manexo das referencias a obxectos é clave se queremos obter un bo rendemento á hora de elaborar unha solución no campo da análise de datos.

Outros conceptos claves en Ray son as Ray Tasks e os Ray Actors. As Ray Tasks son unidades de traballo independentes que poden ser executadas en paralelo, de maneira remota e asíncrona. Por outra parte, os Ray Actors, son unidades similares ás tarefas pero capaces

de manter o seu propio estado podendo acceder a el e executar os seus métodos de maneira concurrente, sendo similar a unha clase típica das linguaxes orientadas a obxectos, pero ofrecendo concurrencia. Nun sistema distribuído como un cluster HPC, isto permite manter un estado común para as múltiples tarefas de análise de maneira eficiente.

Ray tamén traballa o concepto de Ray Cluster pensado para situacións nas que se traballa con diferentes máquinas, ideal para integrar solucións baseadas nesta ferramenta nun cluster HPC. Ray ofrece posibilidades como autoescalabilidade ou a despregadura do Ray Cluster na nube. Da mesma maneira que nos clusters convencionais, nos Ray Cluster cada máquina representa un nodo e ten asignada unha dirección IP, o elemento principal desta rede de computadores é o nodo cabeceira ou *head node*, que é o encargado de xestionar procesos externos á computación de tarefas como a monitorización de recursos dispoñibles no cluster ou a rede de heartbeat. O nodo cabeceira tamén pode executar tarefas e actuar como *worker*.

Ademais, Ray implementa unha funcionalidade denominada Ray Client, capaz de establecer unha conexión externa ao cluster coa que se permite enviar código dende un proceso externo ao Ray Cluster coa intención ser executado utilizando as funcionalidades de Ray e os recursos do cluster.

O *scheduler* en Ray tamén é importante xa que é o compoñente encargado de asignar as tarefas e actores aos diferentes nodos do Ray Cluster en función dos recursos dispoñibles. Cando se solicita a execución dunha tarefa, o cliente, envía unha solicitude ao *scheduler* global situado no nodo cabeceira como vimos anteriormente. O *scheduler* global verifica se nese momento hai algún nodo cos recursos necesarios dispoñibles para executar a tarefa ou actor. Se hai máis dun nodo dispoñible aplícase a estratexia de *scheduling* por defecto ou introducida polo usuario (localidade, *spread*[1]). A estratexia por defecto elixe un nodo aleatorio entre os dispoñibles con menor carga computacional, favorecendo a localidade e o balanceo de carga. No caso de que non houbese nodos dispoñibles, a tarefa engadiríase a unha cola de execución. Unha vez decidido en que nodo se executará a tarefa, o *scheduler* global envía a tarefa a outro *scheduler* local situado no nodo elexido, este *scheduler* local asignaralle á tarefa ou ao actor un proceso *worker* para a súa futura execución, se o *worker* non está dispoñible nese momento a tarefa ou actor introducirase na cola relativa ao *scheduler* local. Unha vez remata a execución da tarefa ou actor solicitado, o *scheduler* local notifícao ao *scheduler* global e libéranse os recursos reservados para a súa execución. Este sistema de *schedulers* fai transparente para o usuario a distribución de tarefas nun Ray Cluster.

Tanto Dask como Ray traballan con valores futuros ou referencias para representar futuros resultados de tarefas que están sendo executadas ou pendentes de execución, é dicir, resultados indeterminados. Os dous proporcionan abstraccións para facilitar unha análise de datos eficiente. Ámbalas dúas ferramentas aproveitan a capacidade de diferentes unidades

[1]Elexir o nodo baseándose no percorrido dunha lista cos nodos dispoñibles.

de procesamento para traballar cunha cantidade masiva de datos. Non obstante, posúen diferentes enfoques e características específicas, como xa vimos. Dentro das posibilidades de desenvolvemento con Ray, existe unha técnica coñecida como Dask on Ray[18] que combina as posibilidades e flexibilidade de Ray co manexo de grandes estruturas de datos de Dask, a idea consiste en utilizar un *scheduler* propio de Dask nun sistema despregado con Ray.

En resumo, Dask e Ray son ferramentas adicadas á programación paralela distribuída que ofrecen capas de abstracción para facilitar a análise de datos. Dask céntrase na análise distribuida de datos utilizando estruturas de datos como os Dask Arrays e un sistema de planificación eficiente cun grafo de tarefas, mentras que Ray ofrece características avanzadas como a memoria Plasma, os Ray Actors e a integración nun cluster de altas prestacións co Ray Cluster e un sistema de *scheduling* distribuído. Ambos *frameworks* son opcións sólidas para abordar o desafío de procesar grandes volumes de datos en contornas distribuídas.

2.3 Xestión da E/S[2]

O presente traballo céntrase nunha solución para análise en tempo real dos datos producidos por unha simulación paralelizada con MPI, concretamente. Por iso, imos centrarnos nas alternativas que nos permitirán transmitir os datos xerados na simulación a unha contorna Ray. A xestión eficiente da E/S de datos é fundamental, como xa vimos, para minimizar os atoamentos e acadar un rendemento óptimo. Neste contexto, existen múltiples ferramentas e enfoques para abordar os desafíos que supón esta situación en programas MPI Estas son algunhas das máis relevantes:

- **ADIOS (Adaptable I/O System)**: Ferramenta de E/S de alto rendemento deseñada especificamente para aplicacións HPC. Proporciona unha interface flexible e escalable para a xestión de datos xerados en contornas MPI. Esta ferramenta permite aos usuarios definir de maneira eficiente os patróns de acceso e comunicación dos datos, adaptándose ás características específicas dunha aplicación e dun sistema de almacenamento concretos. ADIOS tamén ofrece soporte para unha amplia gama de formatos de arquivo e mecanismos de compresión para optimizar o uso de espazo de almacenamento e o rendemento da E/S.[19]

- **MPI I/O**: Extensión do propio MPI que proporciona funcións e mecanismos para a xestión da E/S en aplicacións paralelas. Permite aos procesos MPI acceder directamente a arquivos compartidos e distribuír as operacións de lectura e escritura de maneira eficiente. MPI I/O ofrece unha abstracción de alto nivel para realizar operacións de E/S colectivas e permite a implementación de estratexias avanzadas de E/S, como escritura

[2]Entrada/Saída

ou lectura en paralelo dun arquivo. Esta ferramenta é ampliamente utilizada en aplicacións MPI para a análise de datos en contornas HPC debido a que é unha solución *ad-hoc* para MPI.

- **PDI (PDI Data Interface)**: Interface de E/S deseñado para abordar os desafíos da xestión de datos entre dous formatos diferentes. PDI proporciona unha interface común e simplificada para a transferencia de datos entre compoñentes e sistemas de almacenamento en contornas MPI. Permite aos desenvolvedores definir e xestionar de maneira eficiente os fluxos de datos entre as diferentes etapas do procesamento, optimizando a transferencia e o acceso aos datos xerados. PDI ofrece flexibilidade na configuración a través dun ficheiro YAML no que tamén se poden indicar os *plugins* a utilizar, para o noso caso, ao buscar unha solución con Ray, interésannos o *plugin* MPI e o *plugin* Pycall, que permiten executar código Python a partir dun evento PDI indicado no código MPI, dando ademais a posibilidade de expoñer datos entre MPI en C e Python.[4]

Ademais destas ferramentas, existen outras complementarias que facilitan o tratamento dos datos xerados por unha simulación de alto rendemento. Diferentes formatos de datos como HDF5 (Hierarchical Data Format 5) ou NetCDF (Network Common Data Form) útiles no proceso de análise de datos; o *middleware* Damaris[8], que habilita análises *in-situ* e visualización interactiva sobre os datos tratados, e Lustre, un sistema de ficheiros paralelo de alto rendemento moi común en clusters HPC.

En resumo, a xestión eficiente da E/S nun programa MPI para a posterior análise dos datos xerados nunha contorna HPC é esencial para maximizar o rendemento e evitar os atoamentos producidos pola latencia natural destas operacións. Para iso, podemos apoiarnos en diferentes ferramentas que ofrecen abstraccións e mecanismos capaces de optimizar a transferencia, acceso e almacenamento dos datos xerados nunha aplicación MPI.

2.4 DEISA

Unha das principais motivacións e obxectivos á hora de realizar este traballo é tomar como punto de partida a ferramenta DEISA[3]. Como adiantamos anteriormente, esta ferramenta propón unha solución para a análise en tempo real dos datos xerados por unha simulación paralelizada con MPI mediante un sistema de análise baseado en Dask Arrays. O paso de datos dende a simulación a Dask está instrumentado con PDI, baseándose no *plugin* Pycall especialmente, de feito, DEISA está dispoñible para ser instalada como un *plugin* de PDI. DEISA está pensado para ser executado nun cluster HPC no que uns nodos terán o papel de *workers* e encargaranse de executar as tarefas de análise con Dask, outros nodos estarán reservados para a simulación MPI, e un nodo será reservado para o *scheduler* e o cliente de maneira conxunta.

Unha das intencións de DEISA é proporcionar unha solución flexible na que o usuario poida introducir calquera tipo de cálculo analítico sobre os datos da simulación antes de coñecelos. Para iso, DEISA crea unha visión global dos datos de saída da simulación definida nunha estrutura de datos chamada Deisa Arrays. Os Deisa Arrays están deseñados sobre Dask Arrays, e ofrecen a posibilidade de deseñar as tarefas de análise incluso antes de que a simulación comece a producir datos. A primeira dimensión dun Deisa Array será a dimensión temporal da simulación, por exemplo, mediante o seguinte código estaríamos sumando as dúas primeiras iteracións dos datos xerados pola simulación coas dúas últimas:

```
def miña_tarefa(deisa_array):
  return deisa_array[:2] + deisa_array[-2:]
```

Código 2.1: Exemplo de acceso aos Deisa Arrays

En DEISA, a tarefa de análise deseñada polo usuario xera un grafo de tarefas (visto na sección 2.2.1). Cabe destacar que DEISA soamente utiliza os datos seleccionados polo usuario, por exemplo, se a simulación se encargase de xerar matrices (dúas dimensións) ao longo das iteracións e a tarefa de análise accede unicamente á diagonal principal, só se terán en conta eses datos á hora de realizar a análise, o resto serán descartados alixeirando a carga de memoria, a carga de traballo no *scheduler* de Dask e producindo transferencias de menor latencia. Isto é xestionado mediante un elemento denominado contrato, no que o cliente DEISA decide os datos que son estrictamente necesarios. Desta maneira, o grafo de tarefas será máis sinxelo e o rendemento das tarefas de análise verase aumentado. Unha vez os datos están dispoñibles nos nodos de análise ou *workers*, séguese a execución das tarefas de análise seguindo as pautas establecidas polo grafo de tarefas froito das tarefas introducidas polo usuario.

Os seguintes elementos son claves para entender o funcionamento de DEISA:

- **DEISA Adapter**: É obtido polo cliente para recuperar os metadatos relativos á simulación, como tamaño dos datos en cada iteración, número de iteración, tipo de datos...É fundamental para que o usuario poida definir o código de análise de maneira flexible. Establece unha comunicación entre o cliente de análise e o DEISA Bridge adaptando os metadatos para a súa correcta interpretación dentro do cliente.

- **DEISA Bridge**: Xestiona a comunicación entre os datos expostos con PDI dende a simulación e os nodos de análise. A transferencia realízase mediante o método *scatter*, propio da librería de Dask. Soamente son transmitidos os datos indicados no contrato se foi validado polas dúas partes (o DEISA Bridge e o Cliente), para ser almacenados nos *workers*. Unha vez están dispoñibles, comeza a execución das tarefas necesarias baseándonos en Dask Arrays. Na Figura 2.1 podemos ver a arquitectura e o funcionamento de DEISA.

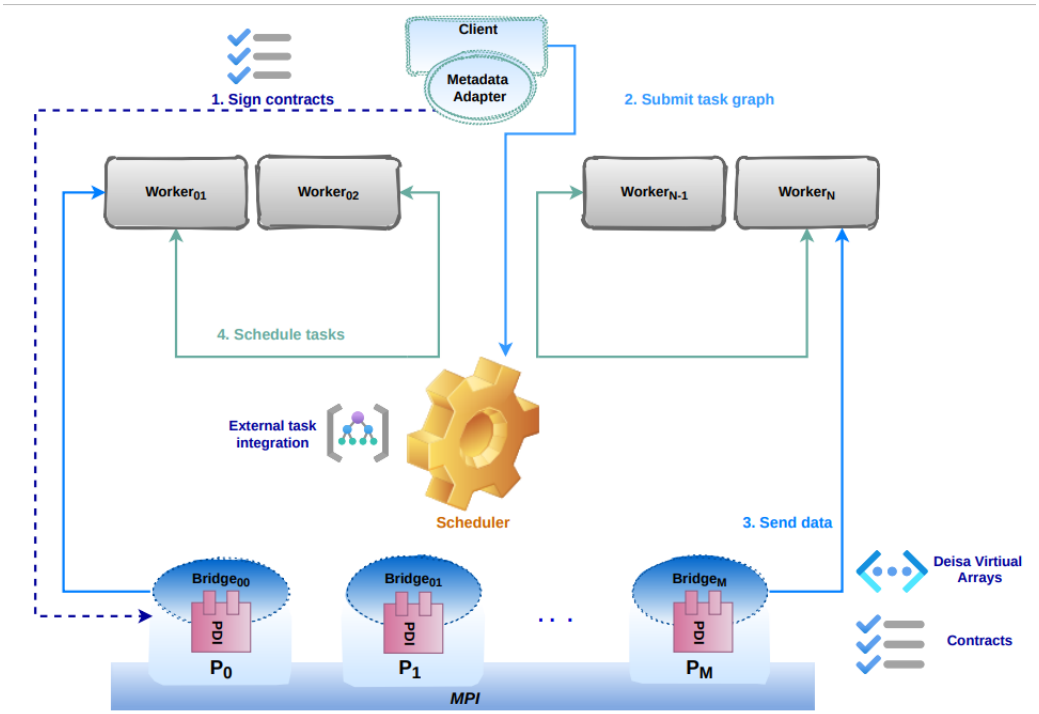

Figura 2.1: Arquitectura de DEISA (extraída da tese doutoral de Amal Gueroudji).

Capítulo 3

Fundamentos Tecnolóxicos

O presente capítulo está adicado a describir as ferramentas, tanto hardware como software, utilizadas para o desenvolvemento e implementación da aplicación resultante do presente proxecto, de agora en adiante REISA (Ray-Enabled In Situ Analytics).

3.1 MPI (Message Passing Interface)

REISA soamente se centra en analizar os datos xerados por unha simulación MPI. Elexi-mos MPI por ser unha das solucións máis utilizadas a día de hoxe para paralelizar código e polas súas capacidades nunha contorna HPC. MPI permite a comunicación entre procesos que poidan estar executándose en nodos diferentes, soporta diferentes linguaxes de programación como C, C++ e Fortran (linguaxes de baixo nivel capaces de aportar un bo rendemento), unha API moi variada con instruccións punto a punto ou colectivas e a capacidade de poder crear diferentes grupos de procesos ou comunicadores de maneira relativamente sinxela. Ademais, tendo en conta aumento da utilización da estratexia de programación híbrida (combinar MPI con outro tipo de software de paralelización, tipicamente OpenMP), amplíanse as posibilidades de que REISA poida traballar con códigos desenvoltos mediante esta técnica.

En resumo, a maioría dos programas pensados para ser executados nunha contorna HPC adoitan ser paralelizados con MPI en linguaxes de baixo nivel por cuestións de eficiencia. Con isto, a nosa solución permite que un programa que esté paralelizado con MPI total ou parcialmente (con enfoque híbrido), sexa válido para realizar unha análise dos datos xerados en tempo real.

Durante o desenvolvemento de REISA traballouse concretamente con OpenMPI v4.1.1 habilitado para GCC v11.2.0.

3.2 PDI (PDI Dara Interface) v. 1.6.0

Partindo de que a simulación que actuará como fonte de datos está paralelizada con MPI, necesitamos unha interface que xestione o paso de datos entre a simulación e o software de análise, neste caso Ray, en Python. Para afrontar este contratempo, utilizando o mesmo enfoque que DEISA, utilizamos PDI (visto na Sección 2.3) para xestionar as comunicacións entre a simulación e o software de análise. Esta ferramenta funciona mediante un sistema de eventos no que o código cliente, no noso caso a simulación, notifica a PDI cando son producidos ao longo do código e de que tipo de evento se trata , como exposición de datos ou soamente marcas temporais. PDI reacciona a estes eventos seguindo un *specification tree* definido polo usuario nun ficheiro YAML onde se especifica o comportamento da interface en canto ao manexo de datos expostos pola simulación, *plugins* utilizados... Está dispoñible un exemplo do *specification tree* no Código 3.1.

PDI ten soporte para C, C++ e Fortran, como dixemos anteriormente, as linguaxes máis utilizadas en ecosistemas HPC debido ao rendemento que poden chegar a ofrecer. PDI ofrece a posibilidade de cargar diferentes *plugins* en función do caso de uso, nós centrarémonos nos plugins MPI e Pycall. O plugin MPI habilita o uso de PDI para un programa paralelizado con MPI e o plugin Pycall é capaz de executar, de maneira síncrona, un fragmento de código Python introducido polo usuario. Con estes dous plugins, conseguimos a comunicación entre a simulación paralelizada en MPI e o software de análise en Python, neste caso, Ray.

No Código 3.1 podemos ver un exemplo do *specification tree* para o plugin Pycall reaccionando ao evento *test* no que se imprime o valor recibido como *$a* nun *array* de lonxitude 3.

Código 3.1: Exemplo do *specification tree* de PDI.

```
data: {a: {type: array, subtype: int, size: 3}}
plugins:
  mpi:
  pycall:
    on_event:
      test:
        with: { a_python: $a }
        exec:
          print('Recibido $a =',a_python)
          a_python[1]=7
          print('Cambiado $a =',a_python)
```

3.3 Ray v.2.4.0

Ray será a peza fundamental de REISA, o noso sistema de análise en tempo real pensado para unha contorna HPC. Como introducimos na Sección 2.2.2, Ray é un *framework* de programación paralela distribuída que encaixa perfectamente no caso de uso no que se centra este traballo. Está baseado na execución de tarefas asíncronas, podendo ser executadas na mesma máquina dende a que son invocadas ou nunha máquina adicada, perfecto para unha contorna HPC no que se dispoñen de varios nodos e poder realizar unha análise *in transit*. Ray ofrece a posibilidade de executar funcións remotas; chamamos función remota a unha tarefa executada de maneira asíncrona dentro dunha instancia de Ray independentemente de ser executada na mesma máquina na que foi invocada ou nunha máquina externa.

Para inicializar Ray existen dúas maneiras, dende a liña de comandos con `ray start` ou dende dentro dun script Python con `ray.init()`.[20]

O método `ray start` utilízase cando se quere iniciar un Ray Cluster con `ray start --head` ou cando se quere engadir un nodo a un cluster existente con:
`ray start --address <HEAD_NODE_IP>:<PORT>`, o porto por defecto é 6379.

Por outro lado, con `ray.init()` podemos iniciar Ray de diferentes maneiras para o poder utilizar as súas funcionalidades no resto do código. Para autodetectar un cluster existente no nodo local ou, no seu defecto, iniciar un dende cero, é suficiente con executar `ray.init()`, Se queremos conectarnos de maneira explícita a un cluster local existente executaremos `ray.init(address="auto")`, por outra banda, para conectarnos a un cluster remoto existente podemos facelo como clientes sen iniciar Ray na nosa máquina local da seguinte maneira: `ray.init(address="ray://<HEAD_NODE_IP>:<RAY_CLIENT_SERVER_PORT>")`, o porto por defecto é 10001. Para deter unha instancia Ray levantada con `ray.init()` será necesario executar `ray.shutdown()`. Para facer a operación análoga para `ray start`, utilizaremos `ray stop`. De seguido, analizaremos as principais compoñentes de Ray que desempeñan un rol clave en REISA e profundizaremos un pouco máis no seu funcionamento respecto ao da sección 2.2.2.

Cabe destacar, que se establecemos a variable de contorno RAY_ADDRESS co formato <HEAD _NODE_IP>:<PORT> poderemos conectarnos automaticamente ao cluster seleccionado sen necesidade de especificala como parámetro. É de importancia sinalar que dende Python non é posible inicializar un nodo e conectalo a un Ray Cluster existente, soamente se poden inciar clusters locais ou conectarnos a eles.

3.3.1 Ray Object

En Ray, as tarefas e os actores adoitan traballar con obxectos. Estes obxectos tamén son coñecidos como obxectos remotos xa que poden estar almacenados en calquera lugar dun Ray Cluster. Para referenciar a estes obxectos utilízase un tipo de dato chamado ObjectRef que se

comporta coma un punteiro. Os Ray Objects están almacenados na memoria compartida de alto rendemento baseada en Plasma adicada ao almacenamento de obxectos, cada nodo do Ray Cluster ten a súa propia memoria de obxectos. Os Ray Objects son inmutables e poden estar distribuídos en múltiples nodos do cluster independentemente do lugar onde foi almacenado por primeira vez e do nodo que mantén a referencia. [21]

Unha referencia a un obxecto é un punteiro ou un identificador que pode ser utilizado para referirnos a un dato determinado sen coñecer o seu valor. As referencias teñen un comportamento similar aos valores futuros ou valores sen determinar. Poden ser creadas de dúas maneiras:

- Devoltas por chamadas a funcións remotas, sexan sobre un actor ou unha tarefa.
- Devoltas tras unha operación de `ray.put(<datos>)` utilizada para almacenar datos na memoria de obxectos.

Para obter os datos reais reais apuntados por unha referencia utilizaremos a función `ray.get(<referencia>)`, que tamén pode traballar con listas de referencias. Este concepto será de vital importancia para entender o fluxo de datos e a comunicación entre nodos no noso sistema de análise de datos en tempo real baseado en Ray.

As referencias que son introducidas directamente como parámetros dunha tarefa son deserializadas automaticamente, é dicir, a función recibe os datos reais directamente sen necesidade de realizar `ray.get(<ref>)`, isto xera unha dependencia na que, ata que os datos que hai detrás da referencia non estén dispoñibles (por exemplo, se a referencia representa o resultado dunha tarefa con execución en curso), non se executará a tarefa que utiliza esa referencia como parámetro. Para evitar esta situación e poder traballar con referencias en lugar de con datos, cómpre introducilas nunha lista e así eliminar a deserialización automática, nesta caso sería necesario utilizar a operación `ray.get(<lista_refs[0]>)` ou simplemente `ray.get(<lista_refs>)`.[22]

3.3.2 Ray Cluster

Xa introducimos os Ray Cluster de maneira breve na Sección 3.3.2, pero aquí expoñeremos o concepto con máis detalle.

Un cluster, na informática, é un conxunto de máquinas interconectadas entre elas por unha rede de comunicacións. Este concepto é moi utilizado no mundo HPC xa que, combinando os recursos de diferentes máquinas, pódese acadar un sistema de altas prestacións en conxunto de maneira máis sinxela.

Ray permítenos que as súas instancias inicializadas en máquinas diferentes poidan traballar en conxunto grazas ao concepto de Ray Cluster. Desta maneira, pódense especificar recursos específicos nun nodo, asignarlle un rol determinado ou transferir datos entre eles

dentro das posibilidades de Ray[23]. Este aspecto é interesante xa que, no noso caso, os nodos de simulación MPI desempeñarán un papel diferente aos nodos de análise. Ademais, REISA está pensado para ser executado nun cluster HPC.

Existen dous tipos de nodos definidos por Ray nun Ray Cluster. O nodo cabeceira, ou *head node*, e o resto de nodos. A única diferencia entre o nodo cabeceira e o resto de nodos é que, por defecto, o nodo cabeceira alberga procesos de administración e xestión de recursos de todo o Ray Cluster así como GCS (Global Control Store), o *scheduler* global ou o servidor de Ray Client. O *head node* pode executar tarefas en función dos seus recursos dispoñibles da mesma maneira que o resto de nodos. Recordemos que Ray utiliza unha arquitectura de *schedulers* distribuída na que o *scheduler* global está situado no *head node* e distribúe a carga aos *schedulers* locais dos distintos nodos en función da dispoñibilidade. Para que un nodo poida executar tarefas ou actores utilizará procesos *worker*, que serán creados e destruídos dende un proceso *driver*, no momento que o scheduler local detecta un *worker* do cluster libre para executar unha tarefa ou actor en espera, será asignada ou asignado ao *worker* para proceder á súa execución.

Para inicializar un cluster pódese facer de dúas maneiras: inicializando cada un dos nodos de maneira manual con `ray start` cos parámetros necesarios, esta alternativa é moi cómoda se estamos utilizando Slurm (ferramenta tipicamente utilizada para executar programas nun cluster HPC)[24], ou mediante liña de comandos coa instrucción `ray up cluster.yml` na que se especifica a información do cluster nun ficheiro YAML, este último método está pensado para despregar un Ray Cluster con proveedores de clusters na nube como AWS ou Azure.

Recordemos que os actores e as tarefas en Ray consumen recursos, por defecto, cada actor e cada tarefa consumirá unha CPU. Ray permite asignar recursos personalizados a cada nodo do cluster (recursos virtuais/personalizados, GPUs...) e establecer o número de CPUs dos que dispón o cluster para executar tarefas ou actores en paralelo.

Este sería un exemplo dun cluster de tres nodo obtido co comando `ray status --address <HEAD_NODE_IP>:<PORT>`. Como podemos ver, no cluster atopamos os recursos personalizados "actor", "head", "compute" e "transit", ademais de amosar os CPUs e a memoria dispoñible de tódolos nodos en conxunto:

Código 3.2: Exemplo de Ray Cluster

```
======== Autoscaler status: 2023-06-09 15:56:14.915105 ========
Node status
---------------------------------------------------------------
Healthy:
 1 node_edb9e8dca4422d549b92a4246d240f6b3280f68e202050258fb4acb0
 1 node_2f55ae6f0106422fb6c76ed7227610a778eee38edd1982f72e7c460d
 1 node_b8a5a85d2c34e97d94868c9ac8f692629ed2072623f281138e390370
Pending:
```

```
 (no pending nodes)
Recent failures:
 (no failures)

Resources
---------------------------------------------------------------
Usage:
 0.0/81.0 CPU
 0.0/1.0 actor
 0.0/40.0 compute
 0.0/1.0 head
 0B/326.70GiB memory
 0B/210.01GiB object_store_memory
 0.0/1.0 transit

Demands:
 (no resource demands)
```

3.3.3 Ray Actor

Despois dunha breve introdución na Sección 2.2.2, podemos adiantar que os actores en Ray desempeñan un papel clave para deseño dunha solución para un software de análise de datos en tempo real, xa que son capaces de manter un estado ao longo do tempo[25], é dicir, preservar os datos xerados pola simulación en iteracións anteriores.

Para crear un actor en Ray utilízase unha clase de Python engadindo na parte superior o decorador `@ray.remote`. Vexamos un exemplo cun programa completo:

```python
import ray

ray.init(address="auto")

@ray.remote
class ActorDeExemplo:
    def __init__(self):
        self.count = 0

    def engadir(self):
        self.count = count + 1
        return self.count

actor = ActorDeExemplo.remote() # Crear o actor
refs = [actor.engadir.remote()\
```

```
    for _ in range(3)] # Executar os seus métodos
print(ray.get(refs)) # Imprimirá [1, 2, 3]

ray.shutdown()
```

Código 3.3: Exemplo de Ray Actor

Os actores en Ray son síncronos por defecto, para a nosa situación, na que un actor pode ter bastante carga de traballo poderemos utilizar actores multifío ou *threaded* actors. Cabe destacar que os actores deste tipo non garanten que a orden de execución dos métodos sexa a mesma que a de invocación. Para crear un actor deste tipo é suficiente con engadir o parámetro max_concurrency no decorador: `@ray.remote(max_concurrency=8)`. Ou nas opcións do actor ao invocalo: `actor = ActorDeExemplo.options(max_concurrency=8).remote)`. Existe tamén o concepto de actores asíncronos baseados na librería *asyncio* pero non traballaremos con eles para implementar REISA. Os *threaded* actors combinan a flexibilidade dun actor en canto a manexo de datos coa eficiencia das tarefas executadas de maneira independente.

Outra característica dos actores é que poden ser nomeados no momento de crealos para que procesos externos, dentro do mesmo cluster, poidan ter acceso ao propio actor. Isto é interesante dende o punto de vista do cliente de análise, que terá que acceder dalgunha maneira aos datos producidos pola simulación, este tipo de actores son unha boa solución para este problema.

Por outra banda, os actores tamén poden ser independentes ao proceso que os creou, isto é interesante para facer independente á simulación do software de análise, xa que o actor seguiría en execución despois de ser creado pola propia simulación a través do plugin Pycall de PDI e seguiría mantendo os seus datos. Este tipo de actores son coñecidos como *detached*.

No seguinte exemplo vemos a creación dun actor named e detached, supoñemos que Ray está inicializado anteriormente no código e o actor definido:

```
    actor = MeuActor.options(name="nome", namespace="exemplo",\
        lifetime="detached").remote()
```

3.3.4 Ray Client

Da mesma maneira que en DEISA, en REISA necesitaremos un compoñente que sexa o encargado de recibir as tarefas de análise introducidas polo usuario e comunicarse co Ray Cluster para levar a cabo as tarefas desexadas sobre os datos correspondentes. A este elemento denominarémolo cliente de análise. Como vimos na Sección 3.3, Ray pódese inicializar de diferentes maneiras, unha delas consiste en conectarse a un cluster (local ou remoto) existente.

Este rol de conexión é coñecido como Ray Client, que permite ao desenvolvedor conectarse a un Ray Cluster para executar as tarefas ou actores desexados cos recursos do cluster.[26]

Como vimos anteriormente, esta conexión pode establecerse da seguinte maneira:

```
ray.init(ray://address="ray://192.168.1.10:10001")
```

Esta sería unha conexión como Ray Client a un Ray Cluster cuxo nodo cabeceira ten a dirección IP 192.168.1.10 e utiliza o porto por defeco, 10001, para o servidor de Ray Client. Poderiamos usar este tipo de conexión para cada un dos procesos MPI pero se traballamos con moitos procesos xeraríanse demasiadas conexións cliente producindo un atoamento no servidor de Ray Client e producindo erros na execución. Por isto, optamos pola solución de utilizar `ray start` para cada nodo da simulación.

3.4 Hardware Utilizado

Para deseñar e implementar REISA utilizaremos principalmente o supercomputador Ruche do Mesocentre da universidade Paris Scalay (tendo como alternativa, en caso de contratempos, o Finisterrae III do Centro de Supercomputación de Galicia). A primeira opción é utilizar Ruche xa que foi un dos sistemas nos que se realizaron as algunhas das probas con DEISA, ferramenta que tomamos como referencia.

Ruche utiliza CentOS 7.9.2009 como sistema operativo e dispón dunha tecnoloxía de rede de comunicacións OPA (Omni-Path Architecture) de 100 Gbit/s de baixa latencia.

Algúns dos recursos de Ruche son os seguintes[27]:

Computacións Paralelas de Memoria Distribuída:

Táboa 3.1: Recursos de CPU en Ruche.

Nodos	CPUs	Memoria
216	2 x Intel Xeon Gold 6230 20C @ 2.1GHz Cascade Lake	192 GB

Cabe destacar que soamente están dispoñibles 144 dos 216 nodos totais de Ruche por motivos de aforro enerxético. Con todos os nodos dispoñibles alcánzanse 7680 núcleos con 516 TFLOPS usando servidores ThinkSystem SD530.[27]

Computacións Paralelas de Memoria Compartida:

Táboa 3.2: Recursos de memoria en Ruche.

Nodos	CPUs	Memoria	Disco
14	4 x Intel Xeon Gold 6230 20C @ 2.1GHz Cascade Lake	1.5 TB	870GB

Actualmente, soamente 10 dos 14 nodos están dispoñibles. Cos 14 nodos chégase a 1120 núcleos con 75 TFLOPS utilizando servidores de tipo ThinkSystem SR850P.[27]

3.5 Outras Ferramentas

Outras ferramentas que foron utilizadas no desenvolvemento do proxecto foron:

- **Git** como control de versións para gardar os avances sobre o desenvolvemento nun repositorio.
- **Visual Studio Code** como editor de código para escribir os programas necesarios para a nosa aplicación. En Visual Studio Code foron de vital importancia dúas das súas extensións: a de SSH para manternos conectados ao supercomputador e poder modificar o código e transferir arquivos de maneira sinxela e directa, e a extensión de Git, para facilitar as operacións relativas ao control de versións.

- Para traballar con REISA é necesario instalar a **librería netifaces** de python para poder conectar cada proceso MPI ao Ray Cluster a través da interface de alto rendemento, *ib0* no caso de Ruche.

Capítulo 4

Arquitectura

No presente capítulo trataremos a arquitectura e o funcionamento de REISA. Inicialización, compoñentes, comunicacións entre eles, fluxo de datos e xestión da memoria.

Recordemos que REISA está baseado en Ray e está pensado para ser executado nun cluster HPC, por isto, como é lóxico, faremos uso da funcionalidade de Ray Cluster. No noso caso teremos tres tipos de nodos:

- **Nodo cabeceira**: Será o encargado de albergar procesos de administración de Ray e a conexión co cliente de análise, que pode ser executado no propio nodo cabeceira ou nun nodo externo ao Ray Cluster. Este nodo podería actuar como nodo de análise pero decidiuse utilizar este esquema para que o nodo encargado da administración do cluster non se saturase.

- **Nodos de simulación**: Neles execútase a simulación, mediante PDI, os datos de cada iteración son recibidos nun *script* de Python que os gardará na memoria Plasma e enviará a referencia resultante a un actor (*named*, para poder ser obtido dende o cliente de análise, e *detached*, para que sexa independente da simulación) que se estará executando no mesmo nodo, é dicir, cada nodo de simulación terá un actor executándose de maneira local que servirá de almacenamento para as referencias dos datos da simulación. Mentras estas referencias estén ao alcance do programa (i.e. nunha variable dentro do actor), o recolector de lixo de Ray non eliminará os obxectos da memoria Plasma.

- **Nodos de análise**: Os seus recursos estarán adicados exclusivamente realizar as tarefas de análise *in transit*. Recibirán as referencias que están nos actores dos nodos de simulación segundo se vaian solicitando tarefas de análise e obterán os seus datos correspondentes coa operación `ray.get(<ref>)`. O resultado das tarefas será enviado ao cliente de análise.

Na Figura 4.1, sen ter en conta as comunicacións, podemos ver unha visión global dos compoñentes de REISA. Neste caso, utilízanse 4 procesos MPI (rectángulos verdes) repartidos en 2

nodos de simulación, podemos ver que cada nodo ten a súa propia memoria Plasma (en rosa), onde se gardarán os datos da simulación, e o seu propio actor (en amarelo), onde se almacenarán os punteiros dos devanditos datos para que o recolector de memoria de Ray non elimine os obxectos.[28] Para a análise *in transit*, neste exemplo, utilizaremos soamente un nodo.

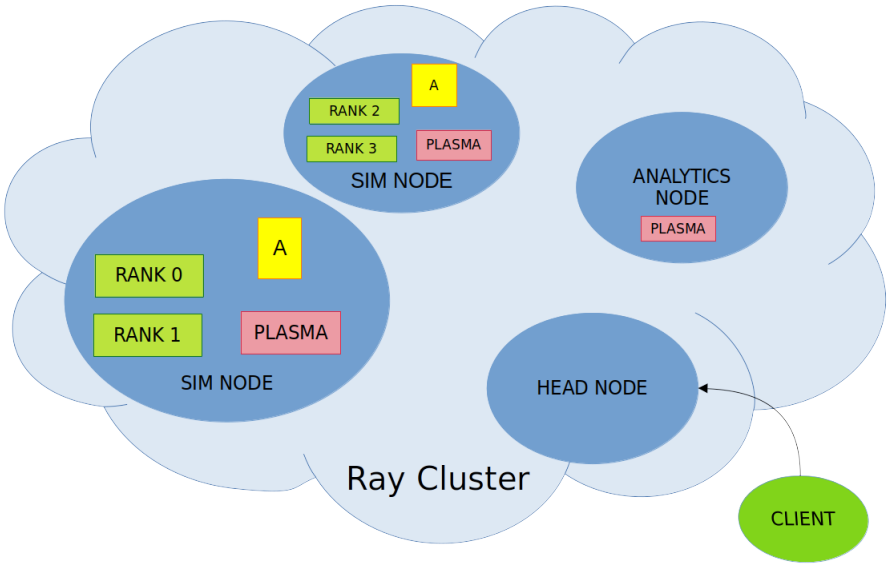

Figura 4.1: Arquitectura de REISA

En canto ao código de REISA, serán necesarios, como mínimo, un ficheiro binario dunha simulación que utilice de maneira correcta os eventos PDI respecto a REISA, un ficheiro YAML que defina o comportamento de PDI (ver Anexo 6.2.5), un *script* Python coas tarefas definidas polo cliente, o código de REISA en *reisa.py* (ver Anexo 6.2.5) que será importado polo cliente para conseguir transparencia, un ficheiro YAML que defina a configuración da simulación e do análise utilizado para inicializar REISA e un *script* de Bash utilizado para iniciar o Ray Cluster con Slurm nun cluster HPC (ver Anexo 6.2.5).

4.1 Despregadura do Ray Cluster

Para despregar este cluster utilizaremos un *script* de Slurm no que se iniciará Ray en cada nodo de maneira manual con `ray start`.

No seguinte *script* de exemplo veremos a maneira de facelo, de seguido, comentaremos algún aspecto relacionado co fragmento de código:

Código 4.1: Despregadura do Ray Cluster[28]

```
1
```

```
    [...]

export RAY_ADDRESS=$head_node_ip:$port

# NODO CABECEIRA
srun --nodes=1 --ntasks=1 --relative=0 \
    --cpus-per-task=$CPUS_PER_WORKER \
    ray start --head \
    --node-ip-address="$head_node_ip" \
    --port=$port \
    --redis-password "$REDIS_PASSWORD" \
    --include-dashboard False \
    --num-cpus $CPUS_PER_WORKER \
    --block \
    --resources='{"compute": 0, "head":1}' \
    --system-config='{"local_fs_capacity_threshold":0.999}' &

# NODOS DE ANÁLISE
for ((i = 1; i <= WORKER_NUM; i++)); do
    node_i=${NODES_ARRAY[$i]}
    srun --nodes=1 --ntasks=1 --relative=$i \
        --cpus-per-task=$CPUS_PER_WORKER --mem=128G \
        ray start --address $RAY_ADDRESS \
        --redis-password "$REDIS_PASSWORD" \
        --num-cpus $CPUS_PER_WORKER \
        --block -\
        -resources="{\"compute\": ${CPUS_PER_WORKER},\
            \"transit\": 1}" \
        --object-store-memory=$((72*10**9)) &
done

# NODOS DE SIMULACIÓN
for ((; i < $SLURM_JOB_NUM_NODES; i++)); do
    node_i=${NODES_ARRAY[$i]}
    srun  --nodes=1 --ntasks=1 --relative=$i \
    --cpus-per-task=$(($MPI_PER_NODE+2)) --mem=128G \
      ray start --address $RAY_ADDRESS \
      --block \
      --num-cpus=$((1+$IN_SITU_RESOURCES)) \
      --resources="{\"actor\": 1, \
        \"compute\": ${IN_SITU_RESOURCES}}"\
      --object-store-memory $((95*10**9)) &
done

    [...]
```

O código 4.1 é un fragmento dun *script* que levanta un Ray Cluster mediante Slurm para poder executar REISA. Vemos que as variables non están definidas, pero para o caso actual é indiferente.

Cada nodo é inicializado cunha instrucción de tipo `srun` e executado en segundo plano co indicador `&`, ademais, utilizamos o parámetro `--relative` para indicar o desprazamento na lista de nodos que nos adxucicou Slurm, e decidir en cal deles se executará a instrucción tras `srun`. O primeiro nodo será o nodo cabeceira.

Identificamos tres bloques no *script* para poder inicializar cada tipo de nodo. Os parámetros de `ray start` no primeiro bloque son:

- **–node-ip-address:** a dirección do nodo actual.
- **–port** : o porto de GCS (Global Control Store) (6379 por defecto).
- **–redis-password**: unha contraseña para a base de datos de Redis (encargada de procesos de adminsitración) definida por nós (isto é opcional).
- **–include-dashboard**: un parámetro para incluír o dashboard de Ray que nos ofrece unha interface de usuario cómoda para controlar o comportamento do cluster, descartamos o uso do dashboard para reducir carga no cluster.
- **–num-cpus**: o número de CPUs que terá o nodo de Ray.
- **–block**: bloquea o comando de maneira indefinida ata que se deteña a instancia de Ray manualmente.
- **–resources**: os recursos personalizados que terá o cluster (neste caso indícanse 0 recursos de ”compute” para que non se executen tarefas de análise nel).
- **–system-config**: configuración sobre un umbral relativo a un warning de uso de memoria.

Nos nodos de análise repítense a maioría dos parámetros utilizados para iniciar o nodo cabeceira. Os únicos diferentes son **–address** que indica a dirección e o porto do nodo cabeceira e **-object-store-memory** que define a cantidade de memoria reservada para a memoria de obxectos vista na sección 3.3.1.

Cabe destacar que é importante utilizar a interface de alta velocidade que ofrece un cluster HPC (*ib0* no caso de Ruche) especificando as direccións IPs correctas nos parámetros de `ray start`, se non, Ray probablemente utilice a rede de administración do cluster (tipicamente Ethernet) para as comunicacións entre nodos, isto non ten por que ser determinante en tódolos casos, pero a gran escala é necesario utilizar unha rede de alto rendemento para obter os resultados esperados. No noso caso, en Ruche, a dirección IP do nodo cabeceira obtémola do seguinte modo:

```
head_node_ip=$(srun -N 1 -n 1 --relative=0 \
    echo $(ip -f inet addr show ib0 | \
    sed -En -e 's/.*inet ([0-9.]+).*/\1/p') &)
```

Os procesos externos ao Ray Cluster son a simulación e o cliente, o cliente será, evidentemente, un proceso Python para poder conectarse ao Ray Cluster.

4.2 Fluxo de Datos

Unha vez vistos os compoñentes que forman parte de REISA e como inicialos veremos as comunicacións entre eles para realizar o proceso de análise de datos. Iremos por orde cronolóxica.

Para que REISA poida ler os parámetros da configuración utilizada apoiarémonos nun ficheiro YAML como o seguinte (non confundir co YAML de PDI):

Código 4.2: Ficheiro YAML de configuración

```
MaxtimeSteps: 12
  #Iteracións da simulación
cpus_per_worker: 40
global_size:
  #Tamaño total dos datos por iteración
  height: 32768
  width: 16384
mpi_per_node: 16
  #Procesos por nodo
parallelism:
  #Distribución dos procesos
  height: 4
  width: 4
workers: 1
  #Número de nodos de análise
```

Existen dous procesos independentes e paralelos nesta arquitectura, o cliente e a simulación. Veremos o que sucede para cada un deles.

4.2.1 Ray dende a Simulación

Unha vez Ray está iniciado nos nodos de simulación veremos o que sucede dende o punto de vista da fonte dos datos de análise.

O primeiro paso é xerar un evento PDI que lea os metadatos necesarios para executar REISA, estes son: o número de iteracións, o rango de cada proceso MPI e o número de procesos por nodo de simulación. A execución de REISA dende o punto de vista da simulación baséase en tres eventos xestionados dende o *plugin* Pycall de PDI, este código executarase dentro do ficheiro YAML que establece o *specification tree* de PDI que vimos na Sección 3.2:

- **init**: evento no que cada proceso se une á instancia Ray levantada no momento de creación do cluster con `ray.init()`, cabe destacar que é unha precondición que Ray esté iniciado en cada nodo de simulación. Neste proceso defínense os actores e tarefas necesarias, un proceso por cada nodo encárgase de crear o actor necesario para REISA e o resto dos procesos solicitan unha instancia súa para poder usalo máis adiante. Estes actores terán un método de inicialización, un método para indicar que o actor xa foi creado, un método para comunicarse coa simulación e outro para comunicarse co cliente de análise. Os actores dos nodos de simulación serán a ponte entre a simulación e o cliente. Este sería un exemplo de como invocar o presente evento PDI dende unha simulación en C:

```
    PDI_multi_expose("init",
        "rank", rank, PDI_OUT,
        "dsize", dsize,  PDI_OUT,
        "timestep", &ii, PDI_OUT,
        "MaxtimeSteps", &generations, PDI_OUT,
        "mpi_per_node", &mpi_per_node, PDI_OUT,
        NULL);
```

- **Available**: este evento xestiona o que sucede en cada iteración, recibe o número da iteración actual e os datos xerados na simulación para ser utilizados no *script* Python. Dentro do *script*, comprobará se hai memoria dispoñible para gardar o obxecto, nese caso, gardará o obxecto, se non quedarase agardando ata que as tarefas de análise lanzadas polo cliente liberen memoria. Gardará os datos recibidos con `ray.put(data)` e enviará a referencia ao actor. Vemos a alto nivel o funcionamento dentro de Pycall:

```
Available:
      #Recibimos iteracióna actual e datos.
      with: { i: $timestep, data: $local_t}
      exec: |
        comprobar_memoria()
        d = ray.put(data, _owner=actor)
        actor.add_value.remote(rank, [d], i)
```

Fixémonos nalgúns detalles. Vemos que na operación de `ray.put(data)` indícase un parámetro adicional que serve para establecer ao actor como propietario do Ray Object. Desta maneira, os datos seguirán na memoria plasma aínda que a simulación remate xa que o propietario, que é o actor cun ciclo de vida independente á simulación, seguirá vivo. Por outro lado, os datos son introducidos nunha lista para que non sexan deserializados de maneira automática ao executar o método `add_value` do actor e poder traballar soamente con referencias. Dende o código da simulación simulación, a invocación deste evento sería semellante ao seguinte código:

```
PDI_multi_expose("Available",
      "timestep",          &ii, PDI_OUT,
      "local_t", current, PDI_OUT,
       NULL);
```

Vemos que os nomes *"Available"*, *"timestep"* e *"local_t"* coinciden co fragmento de código anterior.

- **finish**: evento para pechar as instancias de Ray abertas por cada proceso mediante a operación de `ray.shutdown()`. Dende a simulación veríase desta maneira:

```
PDI_event("finish");
```

4.2.2 Ray dende o Cliente de Análise

Unha vez coñecido o que sucede no lado da simulación, vexamos agora as comunicacións dende o punto de vista do cliente.

Unha das principais diferencias entre REISA e DEISA é que en REISA deben definirse dous tipos de iteracións. As iteracións nas que a simulación xera datos, e as iteracións sobre as que se queren realizar tarefas de análise. Un exemplo práctico é unha simulación que xera unha cantidade de datos por proceso MPI pequena pero utilízanse moitos procesos; dende o lado da análise poderiase optar por reducir o *overhead* ou sobrecarga agrupando os datos producidos pola simulación, por exemplo, cada 10 iteracións dando lugar a un menor número de tarefas que requerirán un tempo de execución asumible ao traballar cunha menor cantidade de datos. Trátase de elexir entre evitar atoamento no scheduler pola fina granularidade das tarefas lanzadas ou maximizar o paralelismo da análise executando un maior número de tarefas.

Debido a que en Ray non dispoñemos dos Dask Arrays (ver Sección 2.2.1), non podemos ofrecer esa visión global característica de DEISA e dos Deisa Arrays (ver Sección 2.4). Por iso, ofrécese ao usuario a posibilidade de definir dous tipos de tarefas de análise. Unha executada por cada proceso MPI en cada iteración de análise (maior nivel de paralelismo) e outra executada en cada iteración de análise tipicamente para recolectar os resultados dos procesos da mesma iteración. Véxase a imaxe 4.2 para entender este enfoque a alto nivel.

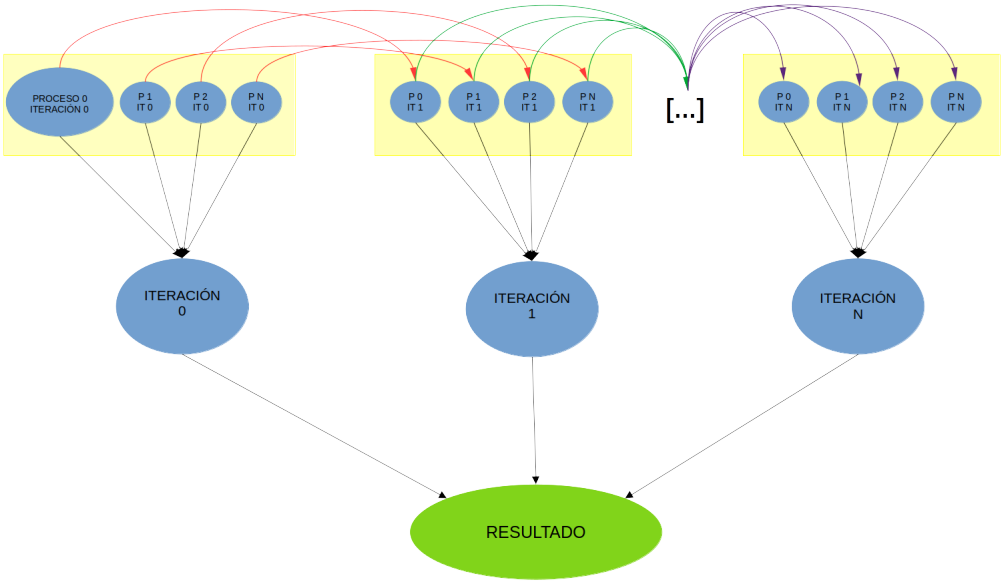

Figura 4.2: Arquitectura das tarefas de análise en REISA.

Apoiándonos na Figura 4.2, na que cada círculo azul representa unha tarefa en REISA, podemos ver que dentro da mesma iteración, os resultados das tarefas a nivel de proceso (nivel superior) son enviados como entrada á tarefa a nivel de iteración (nivel medio). Ademais, as tarefas a nivel de proceso poden acceder aos datos das iteracións anteriores sempre que estén dispoñibles en memoria como representan as catro frechas superiores (unha frecha por proceso, no exemplo da Figura 4.2) que viaxan dun rectángulo amarelo a outro; isto é grazas a que o actor ubicado no nodo de simulación garda as referencias dos datos xerados polos procesos MPI que se están executando nese nodo provenientes de iteracións anteriores, é dicir, se fose necesario, á iteración N chegarían os datos de toda a simulación. En canto á tarefa de nivel de iteración, recibe como parámetros o número da iteración actual e a saída do nivel superior, como indican as frechas descendentes dende os rectángulos. Adicionalmente, o usuario pode definir unha función e introducila como parámetro coa palabra clave *global_func*, esta tarefa executarase sobre os resultados obtidos do nivel de iteración antes de rematar a análise.

REISA está pensado para ser utilizado de maneira sinxela dende o punto de vista do código cliente. Elaboraremos un código cliente que lance unha tarefa de análise encargada de reducir mediante a suma os valores de cada unha das iteracións e obter un valor por iteración.

Imos amosar o código de exemplo para un posterior análise exhaustivo:

```python
import numpy as np
from  reisa import Reisa # Obrigatorio
import os

# Obter a dirección da rede de alta velocidade
address = os.environ.get("RAY_ADDRESS").split(":")[0]
# Conectarnos a REISA
handler = Reisa("config.yml", address)
# Gardar as iteracións da simulación
max = handler.iterations

# Tarefa de análise a nivel de proceso
def process_func(rank: int, i: int, queue):
    # Obter os datos da iteración desexada
    datos = np.array(queue[i])
    # Sumalos
    return np.sum(datos)

# Tarefa de análise a nivel de iteración
def iter_func(i: int, current_results):
    return np.sum(current_results[:])

# Elexir as iteracións que serán analizadas
iterations = [i for i in range(0, max)]

# Lanzar a análise
result = handler.get_result(process_func,\
    iter_func,\
    global_func=None,\
    selected_iters=iterations,\
    kept_iters=1)

# Escribir os resultados
with open("results.log", "a") as f:
    f.write("\nResults: "+str(result)+".\n")

handler.shutdown()
```

Código 4.3: Código cliente de REISA.

Analizaremos, comezando pola parte superior, o código 4.3 facendo referencia ao número de liña para contextualizar:

Na liña 2 importamos a clase Reisa do módulo reisa que dará transparencia con Ray ao usuario.

Na liña 8 conectámonos con REISA introducindo o ficheiro de configuración indicado no código 4.2 e a dirección IP do nodo cabeceira do Ray Cluster. O que sucede detŕas desta instrucción é a lectura do ficheiro e a conexión co Ray Cluster como Ray Client.

Na liña 13 defínese a tarefa que se executa a nivel de proceso, o nivel superior da imaxe 4.2. É unha precondición que esta tarefa reciba como argumentos o rango do proceso que xerou os datos, a iteración á que pertencen os datos e a cola onde o actor almacena as referencias aos datos da iteración actual e das anteriores. Na liña 15 obtemos os datos que o proceso MPI de rango `rank` almacenou na memoria plasma durante a iteración `i` utilizando `ray.get()` de maneira transparente. É importante sinalar que tanto para o argumento *queue* como para o argumento *current_results* cómpre utilizar o operando "`[]`" para obter os valores reais dos obxectos. Se se van a utilizar varios elementos recoméndase utilizar un *slice* como no caso da liña 21. Se neste caso executásemos `datos=np.array(queue[:])` estariamos recollendo todos os datos xerados polo proceso de rango `rank` dende o comezo da simulación ata a iteración `i`.

Na liña 20 deseñamos a función a nivel de iteración, volve a ser obrigatorio que os argumentos sexan a iteración e os resultados do nivel de procesos da iteración actual. Este proceso de redución é análogo ao fluxo de datos da Figura 4.2 sen ter en conta as flechas superiores xa que non necesitamos os datos das iteracións anteriores.

Na liña 27 lanzamos as tarefas de análise mediante unha operación bloqueante, é necesario introducir as tarefas dos dous niveis, unha lista coas iteracións nas que se quere realizar a análise (por defecto todas as iteracións) e é altamente recomendado introducir o número de iteracións que se deben preservar na memoria dos nodos de simulación (`kept_iters`), neste caso, 1, xa que soamente estamos accedendo aos datos da iteración actual.

Dentro desta instrución da liña 27, REISA invoca tódalas tarefas do nivel de iteración, cada tarefa deste tipo, de maneira aniñada, envía as peticións a tódolos actores para que executen as tarefas a nivel de proceso correspondentes á iteración correspondente, unha vez se obtén o resultado de tódolos actores (i.e. de tódalas tarefas a nivel de proceso) realízanse os cómputos correspondentes á tarefa de iteración e devólvese o resultado da iteración ao cliente; finalmente, se se indicou como parámetro `global_func`, execútase a función introducida no propio parámetro sobre os datos resultado de cada iteración de análise, esta idea está representada na Figura 4.2). Cabe destacar que a concurrencia das tarefas do nivel de iteración está limitada xa que cada unha delas desencadeará a execución de máis tarefas; con este límite garantimos que non se produzan bloqueos por mor de falta de recursos para as tarefas a nivel de proceso. Ver o Anexo 6.2.5 para máis detalle.

Finalmente, nas liñas 32 e 33 escríbense os resultados obtidos das tarefas de análise e na liña 35 péchase a conexión con REISA de maneira segura. Os resultados son devoltos como

un dicionario de Python no que se amosa o resultado para cada iteración sempre e cando non sexa nulo.

Sabemos que en REISA é necesario executar `ray start` en cada un dos nodos de simulación. Desta maneira poderemos aproveitar estas instancias de Ray para executar tarefas de análise *in situ* dentro dos propios nodos de simulación. Isto aumentará o rendemento especialmente con grandes cantidades de datos que nos permitan reducir a latencia ligada á transferencia. Desta maneira, REISA ofrece unha estratexia híbrida combinando análises *in situ* e *in transit*. Cabe destacar que soamente se executarán *in situ* as tarefas do nivel de proceso (ver Figura 4.2) onde a cantidade de datos manexados adoita ser maior, unha vez se reduzan os datos útiles na tarefa de análise, o seguinte nivel será executado en nodos exclusivos de análise. O número máximo de tarefas paralelas en cada nodo de simulación pódese especificar mediante a utilización de recursos como vimos no Código 4.1.

4.2.3 Ray Timeline

Se activamos o *profiling* con `export RAY_PROFILING=1` , coa axuda da ferramenta Ray Timeline[29] podemos ver con detalle a execución e o comportamento das tarefas e actores.

No exemplo da Figura 4.3, podemos ver como 16 actores situados no nodo 10.7.0.3 executan 10 iteracións e realizan as operacións de `ray.put()` de forma paralela e sincronizada en cada un dos drivers de cada proceso. Podemos observar como todas as tarefas a nivel de proceso se executan *in situ*, no nodo 10.7.0.3 e as tarefas a nivel de iteración no nodo de *in transit* análise 10.7.0.2. Ademais, o nodo 10.7.0.3 amosa a conexión co cliente e as comunicacións co Ray Cluster.

Figura 4.3: Timeline de REISA.

Capítulo 5

Experimentos Realizados

Neste capítulo recompílanse as probas e experimentos realizados para medir o rendemento de REISA e comparalo con DEISA. Ademais, mediremos o impacto de habilitar a execución *in situ* con diferentes niveis de recursos. A cantidade de experimentos que se poden levar a cabo con este tipo de aplicacións é inmensamente grande debido á variedade de situacións que se poden dar para realizar unha análise de datos en tempo real e ás múltiples variables que afectan á configuración do noso escenario.

Sabemos que a análise *post-hoc* é máis lenta que a análise en tempo real xa que o primeiro enfoque depende da completitude da simulación. No presente capítulo non se realizarán medicións con análise *post-hoc*. Para obter unha referencia sobre a diferencia entre o análise *post-hoc* e o análise de datos en tempo real pódese consultar a tese doutoral de Amal sobre DEISA. [3]

Nas seguintes probas teremos diferentes cofiguracións, tanto para a simulación (en número de procesos MPI, datos xerados, procesos MPI por nodo e iteracións) como para a parte de análise (nodos utilizados, paralelismo dispoñible e software de análise). Para cada configuración executaranse dúas probas de análise distintas: unha derivada temporal para reproducir un dos experimentos xa realizados con DEISA[3] e unha redución mediante a suma para cada iteración de datos xerada pola simulación, como a do Código 4.3.

Plantexouse tamén replicar o experimento de IPCA utilizado na tese de DEISA pero este algoritmo en concreto depende de funcións de alto nivel que encaixan perfectamente coa filosofía dos Deisa Arrays (ver Sección 2.4). No caso de REISA, estas funcións deberían ser implementadas en detalle especificando os cómputos a nivel de proceso e a nivel de iteración (ver Figura 4.1) eliminando a posibilidade de utilizar estas funcións de abstracción. Finalmente non se tivo en conta esta opción xa que a cantidade de experimentos realizados coa derivada e coa redución foron considerados suficientes como para obter unhas conclusións válidas sobre o rendemento de REISA e a súa comparación con DEISA.

Cabe destacar que, para realizar a medición de tempos desactivouse calquera tipo de *pro-*

filing ou monitorización en Ray para evitar posibles latencias, isto ten como consecuencia unha incertidume no tempo de análise. Coa medición utilizada o tempo de análise non pode ser moi inferior ao tempo total da simulación xa que non poderemos detectar a magnitude na que as tarefas de análise son máis rápidas que a simulación, soamente saberemos que se os tempos de simulación e de análise son moi próximos é probable que as tarefas de análise sexan máis áxiles. Os tempos de análise de REISA e de DEISA mediranse dende que se invoca á primeira tarefa de análise ata que se obteñen os resultados, incluíndo as esperas pola simulación no caso de que as tarefas de análise pendentes rematen antes de que haxa novos datos dispoñibles.

Existe un repositorio GitHub no que se poden atopar as execucións dos experimentos realizados para garantir a súa reproducibilidade.[30]

5.1 REISA vs. DEISA

Sabemos que Dask utiliza varios procesos *worker* por nodo da mesma maneira que Ray, e cada *worker* terá un número de fíos de execución ou *threads* dispoñibles para realizar cálculos. Para aproximar DEISA e REISA o máximo posible decidiuse utilizar un *thread* por *worker* con Dask, xa que REISA non se aproveita desta situación, por exemplo, ao utilizar a librería *numpy* para cálculos matemáticos en Python. Ademais, para REISA **deshabilitouse a execución *in situ* facendo que as dúas ferramentas estén obrigadas a realizar as tarefas de análise *in transit*** nos nodos de análise adicados.

Os experientos realizados dividiranse na derivada e na redución. Para cada un deles executaremos distintas configuracións MPI con distintos nodos de análise, podendo utilizar ata 30 *workers* en cada nodo de análise para REISA e para DEISA. Sabemos que os nodos de Ruche ofrecen 40 procesadores lóxicos[27]. Debido a erros de execución puntuais reservando os 40 procesadores en cada nodo de análise decidiuse reducir a cantidade máxima de *workers* por nodo a 30, tanto para Dask como para Ray coa finalidade de obter unha execución limpa en tódolos experimentos.

Os experimentos de comparación entre Dask e Ray poden dividirse en dúas aplicacións, a derivada (12 iteracións) e a redución (25 iteracións). Para cada unha delas utilizaranse distintas configuracións en canto a cantidade de procesos MPI, datos xerados por proceso MPI en cada iteración, e cantidade de nodos de análise, que como recordemos, utilizaremos ata 30 *workers* por nodo de análise. O código da simulación é o mesmo que o utilizado para os experimentos de DEISA. Para cada unha destas configuracións realizáronse 3 execucións para obter os seus tempos promedios.

De seguido, amosaremos distintas táboas nas que se agrupan distintas configuracións. Cada táboa conterá 4 subtáboas apiladas en vertical na seguinte orde descendente: paráme-

tros das configuracións, tempos con DEISA **en segundos**, tempos con REISA **en segundos** e aceleracións ou *speedups* indicando a magnitude na que REISA é máis rápida que DEISA. Amósanse os tempos de análise, simulación neta, bloqueo en PDI e total da simulación ou simulación bruta (PDI máis simulación sen PDI).

5.1.1 Derivada

256 MiB por proceso MPI

Táboa 5.1: Experimentos 1.

CONFIGURACIÓN DA EXECUCIÓN						
PROGRAMA[1]	**PROCESOS MPI**[2]	**NODOS SIM.**[3]	**MiB / PROCESO**[4]	**TOTAL GiB**[5]	**ITER.**[6]	**NODOS ANL. (total *workers*)**[7]
Derivada	4	2	256	1	12	2 (60)
Derivada	8	4	256	2	12	4 (120)
Derivada	16	8	256	4	12	8 (240)
DEISA						
ANL.[8]	**SIM.**[9]	**SIM. MED.**[10]	**PDI LOCK**[11]	**PDI LOCK MED.**[12]	**TOTAL SIM.**[13]	**TOTAL SIM. MED.**[14]
54,02	42,46	3,54	14,67	1,22	57,13	4,76
54,44	42,58	3,55	15,08	1,26	57,66	4,81
57,02	43,08	3,59	17,41	1,45	60,48	5,04
REISA						
ANL.	**SIM.**	**SIM. MED.**	**PDI LOCK**	**PDI LOCK MED.**	**TOTAL SIM.**	**TOTAL SIM. MED.**
48,11	42,77	3,56	9,90	0,83	52,67	4,39
56,46	43,82	3,65	10,59	0,88	54,41	4,53

[1] Aplicación utilizada para o análise.
[2] Número total de procesos MPI utilizados na simulación.
[3] Nodos nos que se executou a simulación.
[4] MiB de datos xerados por procesoo MPI en cada iteración
[5] GiB de datos xerados en total en cada iteración.
[6] Número de iteracións da simulación.
[7] Nodos de análise utilizados e *workers* totais
[8] Tempos de análise (seg.)
[9] Tempos de simulación neta (sen PDI) (seg.)
[10] Tempos de simulación neta por iteración (seg.)
[11] Espera total causada por PDI (seg.)
[12] Espera total causada por PDI en cada iteración (seg.)
[13] Tempos de simulación bruta (con PDI) (seg.)
[14] Tempos de simulación bruta por iteración (seg.)

64,95	44,82	3,73	10,83	0,90	55,64	4,64
SPEEDUPS[15]						
ANL.	**SIM.**	**SIM. MED.**	**PDI LOCK**	**PDI LOCK MED.**	**TOTAL SIM.**	**TOTAL SIM. MED.**
1,12	0,99	0,99	1,48	1,48	1,08	1,08
0,96	0,97	0,97	1,42	1,42	1,06	1,06
0,88	0,96	0,96	1,61	1,61	1,09	1,09

512 MiB por proceso MPI

Táboa 5.2: Experimentos 2.

CONFIGURACIÓN DA EXECUCIÓN						
PROGRAMA	**PROCESOS MPI**	**NODOS SIM.**	**MiB / PROCESO**	**TOTAL GiB**	**ITER.**	**NODOS ANL. (total *workers*)**
Derivada	4	2	512	2	12	2 (60)
Derivada	8	4	512	4	12	4 (120)
Derivada	16	8	512	8	12	8 (240)
DEISA						
ANL.	**SIM.**	**SIM. MED.**	**PDI LOCK**	**PDI LOCK MED.**	**TOTAL SIM.**	**TOTAL SIM. MED.**
100,94	85,54	7,13	21,40	1,78	106,94	8,91
98,92	84,46	7,04	20,51	1,71	104,96	8,75
102,88	86,97	7,25	22,32	1,86	109,30	9,11
REISA						
ANL.	**SIM.**	**SIM. MED.**	**PDI LOCK**	**PDI LOCK MED.**	**TOTAL SIM.**	**TOTAL SIM. MED.**
97,30	84,59	7,05	18,37	1,53	102,96	8,58
111,71	86,79	7,23	19,66	1,64	106,45	8,87
138,64	88,74	7,39	19,68	1,64	108,42	9,03
SPEEDUPS						
ANL.	**SIM.**	**SIM. MED.**	**PDI LOCK**	**PDI LOCK MED.**	**TOTAL SIM.**	**TOTAL SIM. MED.**
1,04	1,01	1,01	1,16	1,16	1,04	1,04
0,89	0,97	0,97	1,04	1,04	0,99	0,99
0,74	0,98	0,98	1,13	1,13	1,01	1,01

[15] Magnitude na que REISA é máis rápida que DEISA.

128 procesos MPI

Táboa 5.3: Experimentos 3.

CONFIGURACIÓN DA EXECUCIÓN						
PROGRAMA	**PROCESOS MPI**	**NODOS SIM.**	**MiB / PROCESO**	**TOTAL GiB**	**ITER.**	**NODOS ANL. (total *workers*)**
Derivada	128	4	1	0,125	12	2 (60)
Derivada	128	4	128	16	12	2 (60)
DEISA						
ANL.	**SIM.**	**SIM. MED.**	**PDI LOCK**	**PDI LOCK MED.**	**TOTAL SIM.**	**TOTAL SIM. MED.**
25,62	1,17	0,10	29,45	2,45	30,62	2,55
71,33	29,35	2,45	48,46	4,04	77,81	6,48
REISA						
ANL.	**SIM.**	**SIM. MED.**	**PDI LOCK**	**PDI LOCK MED.**	**TOTAL SIM.**	**TOTAL SIM. MED.**
16,03	6,78	0,56	9,02	0,75	15,80	1,32
161,30	21,92	1,83	17,48	1,46	39,40	3,28
SPEEDUPS						
ANL.	**SIM.**	**SIM. MED.**	**PDI LOCK**	**PDI LOCK MED.**	**TOTAL SIM.**	**TOTAL SIM. MED.**
1,60	0,17	0,17	3,26	3,26	1,94	1,94
0,44	1,34	1,34	2,77	2,77	1,97	1,97

512 procesos MPI

Táboa 5.4: Experimentos 4.

CONFIGURACIÓN DA EXECUCIÓN						
PROGRAMA	**PROCESOS MPI**	**NODOS SIM.**	**MiB / PROCESO**	**TOTAL GiB**	**ITER.**	**NODOS ANL. (total *workers*)**
Derivada	512	16	1	0,5	12	8 (240)
Derivada	512	16	128	64	12	8 (240)
DEISA						
ANL.	**SIM.**	**SIM. MED.**	**PDI LOCK**	**PDI LOCK MED.**	**TOTAL SIM.**	**TOTAL SIM. MED.**
85,58	2,18	0,18	95,99	8,00	98,16	8,18
121,30	36,79	3,07	105,73	8,81	142,52	11,88

REISA						
ANL.	SIM.	SIM. MED.	PDI LOCK	PDI LOCK MED.	TOTAL SIM.	TOTAL SIM. MED.
20,24	9,56	0,80	9,64	0,80	19,20	1,60
295,02	38,05	3,17	22,86	1,91	60,91	5,08
SPEEDUPS						
ANL.	SIM.	SIM. MED.	PDI LOCK	PDI LOCK MED.	TOTAL SIM.	TOTAL SIM. MED.
4,23	0,23	0,23	9,96	9,96	5,11	5,11
0,41	0,97	0,97	4,62	4,62	2,34	2,34

5.1.2 Redución

256 MiB por proceso MPI

Táboa 5.5: Experimentos 5.

CONFIGURACIÓN DA EXECUCIÓN						
PROGRAMA	PROCESOS MPI	NODOS SIM.	MiB / PROCESO	TOTAL GiB	ITER.	NODOS ANL. (total *workers*)
Redución	4	2	256	1	25	2 (60)
Redución	8	4	256	2	25	4 (120)
Redución	16	8	256	4	25	8 (240)
DEISA						
ANL.	SIM.	SIM. MED.	PDI LOCK	PDI LOCK MED.	TOTAL SIM.	TOTAL SIM. MED.
100,11	89,11	3,56	17,08	0,68	106,19	4,25
98,69	88,24	3,53	17,00	0,68	105,24	4,21
100,22	88,93	3,56	18,87	0,75	107,80	4,31
REISA						
ANL.	SIM.	SIM. MED.	PDI LOCK	PDI LOCK MED.	TOTAL SIM.	TOTAL SIM. MED.
101,13	88,65	3,55	17,65	0,71	106,31	4,25
102,37	89,59	3,58	17,59	0,70	107,18	4,29
101,76	91,23	3,65	17,64	0,71	108,87	4,35
SPEEDUPS						
ANL.	SIM.	SIM. MED.	PDI LOCK	PDI LOCK MED.	TOTAL SIM.	TOTAL SIM. MED.
0,99	1,01	1,01	0,97	0,97	1,00	1,00

0,96	0,98	0,98	0,97	0,97	0,98	0,98
0,98	0,97	0,97	1,07	1,07	0,99	0,99

512 MiB por proceso MPI

Táboa 5.6: Experimentos 6.

CONFIGURACIÓN DA EXECUCIÓN						
PROGRAMA	**PROCESOS MPI**	**NODOS SIM.**	**MiB / PROCESO**	**TOTAL GiB**	**ITER.**	**NODOS ANL. (total *workers*)**
Redución	4	2	512	2	25	2 (60)
Redución	8	4	512	4	25	4 (120)
Redución	16	8	512	8	25	8 (240)
DEISA						
ANL.	**SIM.**	**SIM. MED.**	**PDI LOCK**	**PDI LOCK MED.**	**TOTAL SIM.**	**TOTAL SIM. MED.**
195,89	175,02	7,00	30,35	1,21	205,37	8,21
196,38	176,49	7,06	29,85	1,19	206,34	8,25
200,54	182,35	7,29	29,19	1,17	211,54	8,46
REISA						
ANL.	**SIM.**	**SIM. MED.**	**PDI LOCK**	**PDI LOCK MED.**	**TOTAL SIM.**	**TOTAL SIM. MED.**
196,40	176,38	7,06	29,81	1,19	206,19	8,25
201,36	178,41	7,14	30,27	1,21	208,68	8,35
202,52	179,30	7,17	30,77	1,23	210,07	8,40
SPEEDUPS						
ANL.	**SIM.**	**SIM. MED.**	**PDI LOCK**	**PDI LOCK MED.**	**TOTAL SIM.**	**TOTAL SIM. MED.**
1,00	0,99	0,99	1,02	1,02	1,00	1,00
0,98	0,99	0,99	0,99	0,99	0,99	0,99
0,99	1,02	1,02	0,95	0,95	1,01	1,01

128 procesos MPI

Táboa 5.7: Experimentos 7.

CONFIGURACIÓN DA EXECUCIÓN

PROGRAMA	PROCESOS MPI	NODOS SIM.	MiB / PROCESO	TOTAL GiB	ITER.	NODOS ANL. (total *workers*)
Redución	128	4	1	0,125	25	2 (60)
Redución	128	4	128	16	25	2 (60)
DEISA						
ANL.	**SIM.**	**SIM. MED.**	**PDI LOCK**	**PDI LOCK MED.**	**TOTAL SIM.**	**TOTAL SIM. MED.**
65,52	1,82	0,07	17,26	0,69	19,07	0,76
102,16	60,24	2,41	58,73	2,35	118,98	4,76
REISA						
ANL.	**SIM.**	**SIM. MED.**	**PDI LOCK**	**PDI LOCK MED.**	**TOTAL SIM.**	**TOTAL SIM. MED.**
31,61	16,48	0,66	13,14	0,53	29,62	1,18
104,39	73,66	2,95	26,13	1,05	99,79	3,99
SPEEDUPS						
ANL.	**SIM.**	**SIM. MED.**	**PDI LOCK**	**PDI LOCK MED.**	**TOTAL SIM.**	**TOTAL SIM. MED.**
2,07	0,11	0,11	1,31	1,31	0,64	0,64
0,98	0,82	0,82	2,25	2,25	1,19	1,19

512 procesos MPI

Táboa 5.8: Experimentos 8.

CONFIGURACIÓN DA EXECUCIÓN						
PROGRAMA	**PROCESOS MPI**	**NODOS SIM.**	**MiB / PROCESO**	**TOTAL GiB**	**ITER.**	**NODOS ANL. (total *workers*)**
Redución	512	16	1	1	25	8 (240)
Redución	512	16	128	64	25	8 (240)
DEISA						
ANL.	**SIM.**	**SIM. MED.**	**PDI LOCK**	**PDI LOCK MED.**	**TOTAL SIM.**	**TOTAL SIM. MED.**
239,86	7,25	0,29	90,95	3,64	98,21	3,93
237,03	68,01	2,72	126,49	5,06	194,50	7,78
REISA						
ANL.	**SIM.**	**SIM. MED.**	**PDI LOCK**	**PDI LOCK MED.**	**TOTAL SIM.**	**TOTAL SIM. MED.**
45,64	19,53	0,78	25,74	1,03	45,27	1,81

139,30	90,99	3,64	46,71	1,87	137,70	5,51
SPEEDUPS						
ANL.	**SIM.**	**SIM. MED.**	**PDI LOCK**	**PDI LOCK MED.**	**TOTAL SIM.**	**TOTAL SIM. MED.**
5,26	0,37	0,37	3,53	3,53	2,17	2,17
1,70	0,75	0,75	2,71	2,71	1,41	1,41

5.2 *In Situ* vs. *In Transit*

5.2.1 Derivada

128 procesos MPI con 128 MiB por iteración

Táboa 5.9: Experimentos 9.

CONFIGURACIÓN DA EXECUCIÓN						
PROGRAMA	**PROCESOS MPI**	**NODOS SIM. (total *workers in situ*)**	**MiB / PROCESO**	**TOTAL GiB**	**ITER.**	**NODOS ANL. (total *workers*)**
Derivada	128	4	128	16	12	2 (60)
Derivada	128	4 (8)	128	16	12	2 (60+8)
Derivada	128	4 (16)	128	16	12	2 (60+16)
DEISA						
ANL.	**SIM.**	**SIM. MED.**	**PDI LOCK**	**PDI LOCK MED.**	**TOTAL SIM.**	**TOTAL SIM. MED.**
71,33	29,35	2,45	48,46	4,04	77,81	6,48
REISA						
ANL.	**SIM.**	**SIM. MED.**	**PDI LOCK**	**PDI LOCK MED.**	**TOTAL SIM.**	**TOTAL SIM. MED.**
161,30	21,92	1,83	17,48	1,46	39,40	3,28
99,55	23,21	1,93	16,02	1,34	39,23	3,27
61,32	23,21	1,93	17,83	1,49	41,04	3,42
SPEEDUPS						
vs. orixinal REISA				**vs. DEISA**		
1,00				0,44		
1,62				0,72		
2,63				1,16		

512 procesos MPI con 128 MiB por iteración

Táboa 5.10: Experimentos 10.

CONFIGURACIÓN DA EXECUCIÓN						
PROGRAMA	**PROCESOS MPI**	**NODOS SIM. (total *in situ workers*)**	**MiB / PROCESO**	**TOTAL GiB**	**ITER.**	**NODOS ANL. (total *workers*)**
Derivada	512	16	128	64	12	8 (240)
Derivada	512	16 (32)	128	64	12	8 (240+32)
Derivada	512	16 (64)	128	64	12	8 (240+64)
DEISA						
ANL.	**SIM.**	**SIM. MED.**	**PDI LOCK**	**PDI LOCK MED.**	**TOTAL SIM.**	**TOTAL SIM. MED.**
121,30	36,79	3,07	105,73	8,81	142,52	11,88
REISA						
ANL.	**SIM.**	**SIM. MED.**	**PDI LOCK**	**PDI LOCK MED.**	**TOTAL SIM.**	**TOTAL SIM. MED.**
295,02	38,05	3,17	22,86	1,91	60,91	5,08
144,77	35,64	2,97	24,25	2,02	59,90	4,99
87,10	36,76	3,06	24,18	2,02	60,94	5,08
SPEEDUPS						
vs. orixinal REISA				**vs. DEISA**		
1,00				0,41		
2,04				0,84		
3,39				1,39		

5.2.2 Redución

128 procesos MPI con 128 MiB por iteración

Táboa 5.11: Experimentos 11.

CONFIGURACIÓN DA EXECUCIÓN						
PROGRAMA	**PROCESOS MPI**	**NODOS SIM. (total *in situ workers*)**	**MiB / PROCESO**	**TOTAL GiB**	**ITER.**	**NODOS ANL. (total *workers*)**
Redución	128	4	128	16	25	2 (60)
Redución	128	4 (8)	128	16	25	2 (60+8)
Redución	128	4 (4)	128	16	25	2 (60+16)
DEISA						
ANL.	**SIM.**	**SIM. MED.**	**PDI LOCK**	**PDI LOCK MED.**	**TOTAL SIM.**	**TOTAL SIM. MED.**

102,16	60,24	2,41	58,73	2,35	118,98	4,76
REISA						
ANL.	**SIM.**	**SIM. MED.**	**PDI LOCK**	**PDI LOCK MED.**	**TOTAL SIM.**	**TOTAL SIM. MED.**
104,39	73,66	2,95	26,13	1,05	99,79	3,99
93,71	62,08	2,48	31,36	1,25	93,44	3,74
93,74	64,56	2,58	27,97	1,12	92,53	3,70
SPEEDUPS						
vs. orixinal REISA				**vs. DEISA**		
1,00				0,98		
1,11				1,09		
1,11				1,09		

512 procesos MPI con 128 MiB por iteración

Táboa 5.12: Experimentos 12.

CONFIGURACIÓN DA EXECUCIÓN						
PROGRAMA	**PROCESOS MPI**	**NODOS SIM. (total *in situ workers*)**	**MiB / PROCESO**	**TOTAL GiB**	**ITER.**	**NODOS ANL. (total *workers*)**
Redución	512	16	128	64	25	8 (240)
Redución	512	16 (32)	128	64	25	8 (240+32)
Redución	512	16 (64)	128	64	25	8 (240+64)
DEISA						
ANL.	**SIM.**	**SIM. MED.**	**PDI LOCK**	**PDI LOCK MED.**	**TOTAL SIM.**	**TOTAL SIM. MED.**
237,03	68,01	2,72	126,49	5,06	194,50	7,78
REISA						
ANL.	**SIM.**	**SIM. MED.**	**PDI LOCK**	**PDI LOCK MED.**	**TOTAL SIM.**	**TOTAL SIM. MED.**
139,30	90,99	3,64	46,71	1,87	137,70	5,51
109,88	77,12	3,08	32,00	1,28	109,12	4,36
105,90	71,72	2,87	34,11	1,36	105,83	4,23
SPEEDUPS						
vs. orixinal REISA				**vs. DEISA**		
1,00				1,70		
1,27				2,16		

1,32	2,24

5.3 Interpretación dos Resultados

5.3.1 REISA vs. DEISA

Comecemos extraendo datos das execucións que pretenden comparar REISA con DEISA ou Dask con Ray, dende a Táboa 5.1 ata a Táboa 5.8, recordemos que todas estas análises foron realizadas *in transit*.

Tempos de análise.

En moitos casos podemos ver que os tempos de análise son moi parellos tanto en REISA como en DEISA, véxase a Táboa 5.6 na que a maioría dos *speedups* son próximos a 1.

Por outro lado, hai algúns casos especiais, podemos observar que algúns dos tempos de análise están moi distanciados entre as dúas ferramentas, véxase a Táboa 5.4 con *speedups* de 4,23 e 0,41 (moi alto e moi baixo) que nos fai pensar que REISA xestiona mellor execucións con poucos datos e DEISA traballa mellor con cantidades grandes. Este argumento perde forza cando observamos a Táboa 5.8 na que a única diferencia coa táboa anterior é o tipo de aplicación (no anterior caso derivada e neste caso redución) e os *speedups* pasan de 4,23 e 0,41 a 5,26 e 1,70. Se obtemos unha conclusión tendo en conta toda esta información podemos deducir que DEISA traballa mellor coa derivada (aplicación con dependencias temporais), REISA traballa mellor coa redución e REISA tamén xestiona moito mellor as cantidades pequenas de datos (*speedups* de 4,23 e 5,26), como prediciamos anteriormente.

Por outro lado, os peores rexistros de REISA son as execucións coa derivada, con moitos procesos e moitos datos por proceso (ver Táboas 5.3 e Táboa 5.4) onde DEISA é 2,27 e 2,44 veces máis rápido que REISA no análise. As maiores diferencias neste sentido. Chama a atención que na mesma táboa onde se atopa o peor rexistro de REISA con respecto a DEISA (Táboa 5.4) tamén se atopa o seu segundo *speedup* máis alto (4,23) se utilizamos 1 MiB por proceso en cada iteración.

Ademais, en tódolos casos do experimento (dende a Táboa 5.1 ata a Táboa 5.8) sempre se cumpre que, a medida que aumenta o tamaño global de datos xerados en cada iteración, o *speedup* diminúe. Isto lévanos a pensar que o tempo de transferencia dende os nodos de simulación ata os nodos de análise en Ray é maior que o de Dask ou que REISA é menos eficiente con grandes masas de datos que DEISA. Sería interesante comprobar o funcionamento de ambas ferramentas coa mesma cantidade de datos global pero variando o número de procesos MPI.

Se calculamos a media de tódolos *speedups* obtemos que REISA é 1,41 veces máis rápido

que DEISA nos tempos de análise obtidos con certa incertidume. Profundizando neste *speedup* podemos realizar unha media de tódalas execucións dende a Táboa 5.1 ata a Táboa 5.8 e comparar os tempos de análise medio de REISA e DEISA co tempo de simulación netos promedio. Con estes cálculos obtemos que a media dos tempos de simulación de tódalas execucións, sen pasar por PDI, é de 73,6 segundos, con DEISA son necesarios 115,4 segundos de media para realizar a análise (1,61 segundos de análise por cada segundo de simulación) e con REISA cómpren 111,8 segundos de promedio (1,56 segundos de análise para cada segundo de simulación).

Tempos de simulación.

Os tempos de simulación non son tan determinantes coma os tempos de análise pero siguen tendo importancia xa que, para execucións deste calibre, calquera perda de rendemento na simulación pode implicar tamén un retardo no análise. Se obtemos a media de tódolos *speedups* dende a Táboa 5.1 ata a Táboa 5.8 obtemos que o tempo de bloqueo dentro de PDI (recordemos que é síncrono) é 2,21 veces menor en REISA que en DEISA. Isto débese a que a transferencia dos datos en DEISA ()operación *scatter* de Dask) faise dende dentro de PDI, formando parte da simulación, e a transferencia de datos en REISA realízase dende as funcións chamadas polo cliente (*ray.get()*), de maneira dinámica, sen interferir na simulación. Vemos que isto ten un impacto positivo para REISA nas execucións con menor cantidade de datos pero, a medida que aumentan os datos, o rendemento de DEISA pode volverse superior.

5.3.2 *In Situ* vs. *In Transit*

Recordemos que neste apartado engadimos, para 4 das configuracións anteriores, dúas execucións habilitando a análise *in situ*, dando un lugar a 8 execucións novas. Unha das probas *in situ* consiste en permitir un máximo de dúas tarefas paralelas por nodo de simulación e a outra catro tarefas por nodo de simulación como máximo, dobrando os recursos anteriores.

Neste apartado soamente nos interesan os tempos de análise xa que serán as únicas diferencias con respecto aos experimentos anteriores. Os resultados relativos a estas execucións atópanse na Táboa 5.9, Táboa 5.10, Táboa 5.11 e Táboa 5.12.

Comezaremos analizando a aceleración con respecto á execución de REISA soamente *in transit*. A simmple vista podemos ver que a execución *in situ* gaña moita máis aceleración coa derivada que coa redución. Isto débese a que, co enfoque de REISA, na derivada teñen que obterse 5 iteracións cada vez que se executa unha das tarefas de nivel de proceso mentras que coa redución soamente se traen os datos da iteración actual (5 veces menos datos). Coa execución *in situ* habilitada, o impacto producido pola derivada ao ter que enviar 5 iteracións de datos aos nodos de análise para cada proceso, en cada iteración, vese mitigado. O máximo *speedup* obtido coa derivada é de 3,39 mentras que o da redución é de 1,32.

Por outro lado, se aumentamos os nodos de análise, procesos MPI e datos globais por iteración, o rendemento das execucións *in situ* vese incrementado. Isto podémolo ver comparando a Táboa 5.9 coa Táboa 5.10 ou a Táboa 5.11 coa Táboa 5.12.

En canto á comparación con DEISA, na derivada, é necesario habilitar 4 tarefas paralelas por nodo de simulación para mellorar os seus tempos (ver Táboa 5.9 e Táboa 5.10), nos dous casos case se dobra o *speedup* con 2 recursos por nodo e case se triplica con 4 recursos por nodo. Coa redución os tempos son parellos coas execucións orixinais entón verase aumentado nunha magnitude similar á indicada anteriormente, observamos que os *speedups* non melloran (Táboa 5.11 e Táboa 5.12) xa que o tempo de análise xa é parello ao tempo de simulación total e non pode chegar a ser inferior como explicamos na introdución deste capítulo, pola maneira na que se toman as medicións. Para poder ver como se comportan realmente as tarefas e as transferencias de datos en cada nodo podemos activar a monitorización coa ferramenta *ray timeline*.

Capítulo 6

Conclusións

Derradeiro capítulo da memoria no que se expoñen as conclusións obtidas froito do traballo realizado. Tamén se amosarán as posibles liñas futuras de traballo que darían continuidade a este proxecto.

6.1 REISA vs. DEISA

Podemos comezar esta sección recordando que DEISA e REISA teñen o mesmo propósito pero con enfoques lixeiramente diferentes debido á natureza dos seus compoñentes. Os Dask Arrays ofrecen unha visión global da simulación que alixeira o tráfico de datos mentras que Ray ofrece unha persistencia de datos máis sinxela ao longo do tempo grazas seus actores.

En REISA, a creatividade do usuario á hora de desenvolver os distintos niveis de tarefas de análise pode ter un impacto moi significativo nos tempos de análise sendo unha arma de dobre fío, mentras que en DEISA, chega con deseñar unha función a alto nivel que especifique as tarefas de análise. Isto tamén se debe a que REISA é unha versión preliminar e menos madura do que é DEISA. Probablemente, con máis horas de desenvolvemento ambas solucións tendan a converxer neste aspecto proporcionando enfoques cada vez máis similares.

REISA ofrece a posibilidade de executar parte das tarefas de análise *in situ* para mitigar un dos seus puntos débiles, a transferencia de datos en aplicacións como a derivada, na que a análise depende das iteracións anteriores. Vimos que explotando esta posibilidade pódese superar o tempo de DEISA partindo con moita inferioridade nas execucións *in transit*.

Por outra banda, DEISA ofrece unha versión máis estable como aplicación que DEISA debido á diferencia de traballo adicado a desenvolver cada unha das ferramentas ata o día de hoxe. REISA, por agora, é unha versión preliminar realizada coa finalidade de comparar Ray con Dask. Isto leva a pensar que DEISA probablemente soporte unha maior escalabilidade e maior tolerancia a fallos con respecto a REISA.

Como conclusión xeral, vimos que, para as probas realizadas neste traballo Ray ofrece un

maior rendemento promedio que Dask para a análise de datos distribuída nas mesmas condicións. O máis interesante disto, é que o rendemento obtido con Ray pode chegar a incrementarse aínda máis de maneira considerable se se acada unha visión similar á que proporciona DEISA cos Deisa Arrays e reducindo o tráfico de datos.

6.2 Liñas Futuras de Traballo

Actualmente, REISA trátase dunha versión preliminar por cuestións de tempo de desenvolvemento sobre a propia aplicación. Na presente sección falaremos das posibles melloras que se poderían aplicar sobre a versión actual de REISA coa finalidade de acadar unha ferramenta estable e con maior usabilidade.

6.2.1 Dask on Ray

Falamos de que REISA podería mellorar o rendemento de maneira considerable se se adquire a visión global de DEISA cos Deisa Arrays baseada en Dask Arrays. Un camiño para acadar esta funcionalidade podería ser traballar con Dask on Ray[18] onde as funcionalidades de Dask están dispoñibles dentro das infraestruturas de Ray, é dicir, combinaríamos os Dask Arrays e o grafo de tarefas de Dask, co Ray Cluster, cos Ray Actors, coa memoria Plasma, etc. Ademais, non se partiría dende cero, coñecendo o deseño dos Deisa Arrays. En resumo, Dask on Ray pode ser un enfoque híbrido entre DEISA e REISA no que se poden aproveitar as vantaxes de cada un dos *framweorks* e sortear os seus puntos febles.

6.2.2 Flexibilidade con PDI

Unha das principais liñas de traballo que deberían incrementar a usabilidade deste proxecto é a integrción de REISA como *plugin* de PDI, desta maneira a instalación dende o punto de vista do usuario sería máis sinxela e, ademais, podería ser o punto de partida para que, dende PDI, se envíe un coxunto de variables para a análise en lugar de soamente unha variable coa que traballa actualmente REISA.

6.2.3 Facilitar a despregadura do cluster

Por outro lado, un dos procesos máis tediosos en REISA é definir o Ray Cluster explícitamente en cada un dos nodos dentro dun *script* Slurm. Por iso, facilitaría o trbaallo ao usuario incluír un ficheiro YAML no que se especifique a información do Ray Cluster de REISA e poder levantalo con `ray up cluster.yml` en nodos dun cluster local.

6.2.4 Heurísticas

Para aunmentar o rendemento de REISA sería interesante engadir heurísticas que tratasen de recoñecer o tipo de datos que vai xerar a simulación e o tipo de análise a executar sobre eles. Desta maneira, o propio software de REISA poderá decidir a granularidade das tarefas de análise e a cantidade de datos que deben persistir en memoria sen que o usuario teña que especificar o número de iteracións que se deben manter nos actores ao longo do tempo.

6.2.5 Un só actor

Outro enfoque sobre REISA que sería interesante comparar coa versión actual sería o de utilizar un actor global en lugar dun actor por cada nodo da simulación, a idea sería que este actor almacenase todas as referencias e soamente se comunicase co cliente en dúas ocasións: ao recibir as tarefas de análise e ao enviar os resultados finais.

Actualmente, o actor debe bloquear a petición do cliente ata que os datos da simulación están dispoñibles en cada iteración solicitada dende o cliente, xa que o cliente é o encargado de recompilar as respostas de tódolos actores implicados na simulación. En cambio, esta versión ofrecería a posibilidade de executar as tarefas tan axiña como chegan os datos sen ningunha sincronización nin tempo de espera, esta situación tamén facilitaría, presuntamente, o proceso de liberación de memoria.

En resumo, sería interesante comprobar se, con este enfoque, a latencia das comunicacións e o tráfico de datos se ve reducido ou e se mellora o rendemento ou se, por outro lado, o actor sería un colo de botella e producindo un atoamento nas tarefas de análise.

Apéndices

Ficheiro YAML Utilizado para Configurar PDI en REISA

```yaml
pdi:
  logging:
    level: "warn"

  metadata:
    pcoord_1d: int
    pcoord: { type: array, subtype: int, size: 2 }
    psize: { type: array, subtype: int, size: 2 }
    dsize: { type: array, subtype: int, size: 2 }
    MaxtimeSteps: int
    timestep: int
    mpi_per_node: int

  data:
    local_t:
      type: array
      subtype: double
      size: ['$dsize[0]', '$dsize[1]']
      subsize: ['$dsize[0] - 2', '$dsize[1] - 2']
      start: [1, 1]

  plugins:
    mpi:
    pycall:
      on_event:
        init:
          with: {rank: $pcoord_1d, iterations: $MaxtimeSteps, \
            mpi_per_node: $mpi_per_node}
          exec: |
            from ray._private.internal_api import free
            from datetime import datetime
            from reisa import RayList
            import numpy as np
            import netifaces
            import logging
            import time
            import ray
            import sys
            import os
            import gc

            def eprint(*args, **kwargs):
                print(*args, file=sys.stderr, **kwargs)

            # Evento para inicializar Ray e crear os actores.
            mpi_per_node = int(mpi_per_node)
            ib = netifaces.ifaddresses('ib0')[netifaces.AF_INET][0]\
              ['addr'] # Dirección Infiniband
            concurrency=8 # Concurrencia do actor
```

```
cg = int(concurrency*3/4)
threshold = 0.5 # Umbral de memoria utilizada
ray.init(address="auto", namespace="mpi", \
  logging_level=logging.ERROR, _node_ip_address=ib)

# Obter o estado inicial de memoria
initial_status=\
  ray._private.state.\
  state._available_resources_per_node()

# Definimos o actor coas opcións correspondentes
@ray.remote (max_restarts=-1, max_task_retries=-1, \
  resources={"actor":1}, \
  concurrency_groups={"store": cg, \
  "exec": cg, "free": 1}, \
    scheduling_strategy=\
      ray.util.scheduling_strategies.NodeAffinitySchedulingStrategy(
        # Dar preferencia a nodo actual in situ
        node_id=ray.get_runtime_context().get_node_id(),
        soft=False,
    ))
class ProcessActor:
    def __init__(self):
        self.data_queues = [] # Datos da simulación
        self.offset = int(rank) # Rango do creador
        self.control = \ # Control de datos almacenados
                [0 for _ in range(iterations)]
        self.node_id =\
        ray.get_runtime_context().get_node_id()
        for i in range(mpi_per_node):
            self.data_queues.append(\
              [None for _ in range(iterations)])
        return

    # Método para liberar memoria
    @ray.method(concurrency_group="store")
    def free_mem(self, dep, i):
        ray.wait(dep, num_returns=len(dep))
        refs = [self.data_queues[mpid][i] for mpid in range(mpi_per_node)]
        free(refs, local_only=True)
        for mpid in range(mpi_per_node):
            self.data_queues[mpid][i] = None
        del refs
        gc.collect()
        return

    def ready(self):
        return

    # Método para almacenar unha referencia de datos
    @ray.method(concurrency_group="store")
    def add_value(self, mpid, result, i):
        self.data_queues[mpid%mpi_per_node][i]=result[0]
        self.control[i] = self.control[i]+1
        gc.collect()
        return

    # Método invocado polo cliente en cada iteración
    @ray.method(concurrency_group="exec")
    def trigger(self, process_task: ray.remote_function.RemoteFunction, i: int):
        tasks = RayList()

        # Asegurarnos de que recibimos os datos
        while np.sum(self.control[:i+1]) < (mpi_per_node*(i+1)):
            pass

        tasks = [process_task.options(
                    scheduling_strategy=ray.util.scheduling_strategies.NodeAffinitySchedulingStrategy(
                        node_id=self.node_id,
                        soft=True,
                    )
                ).remote(mpid+self.offset, i, RayList(self.data_queues[mpid][:i+1])) \
```

```
                            for mpid in range(mpi_per_node)]

                    gc.collect()
                    return tasks

            put_time = 0
            actor = None
            actorname = "ranktor"+str(rank - (rank%mpi_per_node))

            if rank%mpi_per_node == 0: # Un proceso crea o actor
                actor = ProcessActor.options(\
                  max_concurrency=concurrency, name=actorname,\
                  namespace="mpi", lifetime="detached").remote()
            else: # Agardar polo actor do mesmo nodo
                timeout = 10
                start = time.time()
                error = True
                while error:
                    try:
                        actor = ray.get_actor(actorname, namespace="mpi")
                        error = False
                    except Exception as e:
                        actor = None
                        elapsed_time = time.time() - start
                        if elapsed_time >= timeout:
                            raise Exception("Cannot get the Ray actors. Simulation may break.")

            ray.get(actor.ready.remote())

        Available:
          with: { i: $timestep, data: $local_t}
          exec: |
            # Evento executado durante cada iteración da simulación.
            flag = True
            while(flag): # Comprobar estado de memoria
                status=ray._private.state.state._available_resources_per_node()
                for node in status:
                    osm_used=initial_status[node]['object_store_memory']-status[node]['object_store_memory']
                    m_used=initial_status[node]['memory']-status[node]['memory']
                    osm_limit=initial_status[node]['object_store_memory']*threshold
                    m_limit=initial_status[node]['memory']*threshold

                    if m_used > m_limit or osm_used > osm_limit:
                        flag = True
                        break
                    else:
                        flag = False
                gc.collect()

            # Medir o tempo de ray.put()
            error = True
            start = time.time()
            while error:
                try:
                    d = ray.put(data, _owner=actor)
                    error = False
                except Exception as e:
                    error = True
            put_time = put_time + (time.time()-start)
            del data

            # Gardar a referencia no actor
            actor.add_value.remote(rank, [d], i)

        finish:
          with: {rank: $pcoord_1d, iterations: $MaxtimeSteps, mpi_per_node: $mpi_per_node}
          exec: |
            ray.timeline(filename="timeline-client.json")
            ray.shutdown()
```

Ficheiro reisa.py

```
from datetime import datetime
from math import ceil
import itertools
import yaml
import time
import ray
import sys
import gc
import os

def eprint(*args, **kwargs):
    print(*args, file=sys.stderr, **kwargs)


# Clase que proporciona transparencia con Ray ao usuario
class RayList(list):
    def __getitem__(self, index): # Usaremos o operador "[]" para deserializar os obxectos
        item = super().__getitem__(index)
        if isinstance(index, slice):
            return ray.get(RayList(item))
        else:
            return ray.get(item)

class Reisa:
    def __init__(self, file, address):
        self.iterations = 0
        self.mpi_per_node = 0
        self.mpi = 0
        self.datasize = 0
        self.workers = 0
        self.actors = list()

        # Inifializar Ray
        if os.environ.get("REISA_DIR"):
            ray.init("ray://"+address+":10001", runtime_env={"working_dir": os.environ.get("REISA_DIR")})
        else:
            ray.init("ray://"+address+":10001", runtime_env={"working_dir": os.environ.get("PWD")})

        # Obter a configuración do ficheiro YAML introducido
        with open(file, "r") as stream:
            try:
                data = yaml.safe_load(stream)
                self.iterations = data["MaxtimeSteps"]
                self.mpi_per_node = data["mpi_per_node"]
                self.mpi = data["parallelism"]["height"] * data["parallelism"]["width"]
                self.workers = data["workers"]
                self.datasize = data["global_size"]["height"] * data["global_size"]["width"]
            except yaml.YAMLError as e:
                eprint(e)

        return

    # Obter os resultados da análise
    def get_result(self, process_func, iter_func, global_func=None,\
      selected_iters=None, kept_iters=None, timeline=False):

        # Máximas tarefas de análise en paralelo
        max_tasks = ray.available_resources()['compute']
        results = list()
        actors = self.get_actors()

        if selected_iters is None:
            selected_iters = [i for i in range(self.iterations)]
        if kept_iters is None:
            kept_iters = self.iterations

        # Definir como remota a tarefa a nivel de proceso deseñada polo usuario
        @ray.remote(max_retries=-1, resources={"compute":1}, scheduling_strategy="DEFAULT")
```

```
69          def process_task(rank: int, i: int, queue):
70              return process_func(rank, i, queue)
71
72          iter_ratio=1/(ceil(max_tasks/self.mpi)*2)
73          if iter_ratio>1:
74              iter_ratio=1
75
76          # Definir como remota a tarefa a nivel de iteración deseñada polo usuario
77          @ray.remote(max_retries=-1, resources={"compute":1, "transit":iter_ratio},\
78          scheduling_strategy="DEFAULT")
79          def iter_task(i: int, actors):
80              # Pedir aos actores que executen as tarefas para cada proceso
81              current_results = [actor.trigger.remote(process_task, i) for j, actor in enumerate(actors)]
82              current_results = ray.get(current_results)
83
84              # Librear memoria
85              if i >= kept_iters-1:
86                  [actor.free_mem.remote(current_results[j], i-kept_iters+1) for j, actor in enumerate(actors)]
87              # Aplicar as operacións introducidas polo usuario
88              return iter_func(i, RayList(itertools.chain.from_iterable(current_results)))
89
90          # Lanzar a análise
91          results = [iter_task.remote(i, actors) for i in selected_iters]
92
93          ray.wait(results, num_returns=len(results)) # Agardar os resultados
94
95          # Devolver a saída solicitada
96          if global_func:
97              return global_func(RayList(results))
98          else:
99              tmp = ray.get(results)
100             output = {} # Output dictionary
101
102             for i, _ in enumerate(selected_iters):
103                 if tmp[i] is not None:
104                     output[selected_iters[i]] = tmp[i]
105
106             if timeline:
107                 ray.timeline(filename="timeline-client.json")
108
109             return output
110
111     # Obter tódolos actores
112     def get_actors(self):
113         timeout = 60
114         start_time = time.time()
115         error = True
116         self.actors = list()
117         while error:
118             try:
119                 for rank in range(0, self.mpi, self.mpi_per_node):
120                     self.actors.append(ray.get_actor("ranktor"+str(rank), namespace="mpi"))
121                 error = False
122             except Exception as e:
123                 self.actors=list()
124                 end_time = time.time()
125                 elapsed_time = end_time - start_time
126                 if elapsed_time >= timeout:
127                     raise Exception("Cannot get the Ray actors. Client is exiting")
128             time.sleep(1)
129
130         return self.actors
131
132     # Pechar a conexión
133     def shutdown(self):
134         if self.actors:
135             for actor in self.actors:
136                 ray.kill(actor)
137             ray.shutdown()
```

Script de Slurm para despregar REISA

```
#!/bin/bash
#SBATCH --time=00:20:00
#SBATCH -o reisa.log
#SBATCH --error reisa.log
#SBATCH --mem-per-cpu=4G
#SBATCH --wait-all-nodes=1
#SBATCH --exclusive
##############################################################################################
start=$(date +%s%N)
echo -e "Slurm job started at $(date +%d/%m/%Y_%X)\n"

# Variables de contorno
unset RAY_ADDRESS;
export RAY_record_ref_creation_sites=0
export RAY_SCHEDULER_EVENTS=0
export OMP_NUM_THREADS=1 # Para evitar erros por falta de recursos
export RAY_memory_monitor_refresh_ms=500
export RAY_memory_usage_threshold=0.99
export RAY_verbose_spill_logs=0
export REISA_DIR=$PWD

REDIS_PASSWORD=$(uuidgen)
MPI_TASKS=$(($SLURM_NTASKS - $SLURM_NNODES - 1))
MPI_PER_NODE=4
CPUS_PER_WORKER=30
NUM_SIM_NODES=4
WORKER_NUM=$(($SLURM_JOB_NUM_NODES - 1 - $NUM_SIM_NODES))
IN_SITU_RESOURCES=0

# Obter os nodos reservados
NODES=$(scontrol show hostnames "$SLURM_JOB_NODELIST")
NODES_ARRAY=($NODES)
SIMARRAY=(${NODES_ARRAY[@]:$WORKER_NUM+1:$NUM_SIM_NODES});
for nodo in ${SIMARRAY[@]}; do
  SIM_NODE_LIST+="$nodo,"
done
SIM_NODE_LIST=${SIM_NODE_LIST%,}
echo -e "Initing Ray (1 head node + $WORKER_NUM analytics nodes + $NUM_SIM_NODES simulation\
    nodes) on nodes: \n\t${NODES_ARRAY[@]}"

# Información do nodo cabeceira
head_node=${NODES_ARRAY[0]}
    head_node_ip=$(srun -N 1 -n 1 --relative=0 \
      echo $(/sbin/ip -f inet addr show ib0 | sed -En -e 's/.*inet ([0-9.]+).*/\1/p') &)
ulimit -n 16384 >> reisa.log
port=6379
echo -e "Head node: $head_node_ip:$port\n"
export RAY_ADDRESS=$head_node_ip:$port

# Iniciar o nodo cabeceira
srun --nodes=1 --ntasks=1 --relative=0 --cpus-per-task=$CPUS_PER_WORKER \
    ray start --head --node-ip-address="$head_node_ip" --port=$port \
    --redis-password "$REDIS_PASSWORD" --include-dashboard False\
    --num-cpus $CPUS_PER_WORKER --block --resources='{"compute": 0}' \
    --system-config='{"local_fs_capacity_threshold":0.999}' 1>/dev/null 2>&1 &

cnt=0
k=0
max=10
# Agardar polo nodo cabeceira
while [ $cnt -lt 1 ] && [ $k -lt $max ]; do
    sleep 5
    cnt=$(ray status --address=$RAY_ADDRESS 2>/dev/null | grep -c node_)
    k=$((k+1))
done

# Iniciar Ray en nodos de análise, o recurso transit controla a cantidade de tarefas de nivel de iteración.
for ((i = 1; i <= WORKER_NUM; i++)); do
```

```
    node_i=${NODES_ARRAY[$i]}
    srun --nodes=1 --ntasks=1 --relative=$i --cpus-per-task=$CPUS_PER_WORKER --mem=128G \
        ray start --address $RAY_ADDRESS --redis-password "$REDIS_PASSWORD" \
        --num-cpus $CPUS_PER_WORKER --block \
        --resources="{\"compute\": ${CPUS_PER_WORKER}, \"transit\": 1}" \
        --object-store-memory=$((72*10**9)) 1>/dev/null 2>&1 &
done


# Iniciar Ray en nodos de simulación, habilitar execución in situ se se solicita
for (( ; i < $SLURM_JOB_NUM_NODES; i++)); do
    node_i=${NODES_ARRAY[$i]}
    srun  --nodes=1 --ntasks=1 --relative=$i --cpus-per-task=$((1+$IN_SITU_RESOURCES)) --mem=128G \
        ray start --address $RAY_ADDRESS --redis-password "$REDIS_PASSWORD" \
        --num-cpus=$((1+$IN_SITU_RESOURCES)) --block \
        --resources="{\"actor\": 1, \"compute\": ${IN_SITU_RESOURCES}}" \
        --object-store-memory $((96*10**9))  1>/dev/null 2>&1 &
done

cnt=0
k=0
max=10
# Agardar por tódolos nodos
while [ $cnt -lt $SLURM_JOB_NUM_NODES ] && [ $k -lt $max ]; do
    sleep 10
    cnt=$(ray status --address=$RAY_ADDRESS 2>/dev/null | grep -c node_)
    k=$((k+1))
done
end=$(date +%s%N)

# Iniciar o cliente no nodo cabeceira
srun --oversubscribe --overcommit --nodes=1 --ntasks=1 --relative=0 -c 1\
    `which python` client.py &
client=$!

# Imprimir estado do cluster
ray status --address=$RAY_ADDRESS

# Lanzar a simulación
pdirun srun --oversubscribe --overcommit -N $NUM_SIM_NODES --ntasks-per-node=$MPI_PER_NODE\
    -n $MPI_TASKS --nodelist=$SIM_NODE_LIST --cpus-per-task=1\
        ./simulation $SLURM_JOB_ID &
sim=$!

# Tempo de iniciación
elapsed=$((end-start))
elapsed=$(bc <<< "scale=5; $elapsed/1000000000")
sleep 1
printf "\n%-21s%s\n" "RAY_DEPLOY_TIME:" "$elapsed"

# Agardar polos programas
wait $client
wait $sim

echo -e "\nSlurm job finished at $(date +%d/%m/%Y_%X)"
```

Apéndice A

Glosario de acrónimos

HPC *High Performance Computing.*

DEISA *Dask-Enabled In Situ Analytics.*

REISA *Ray-Enabled In Situ Analytics.*

E/S *Entrada/Saída.*

PDI *PDI Data Interface.*

MPI *Message Passing Interface.*

SSH *Secure Shell*

Apéndice B

Glosario de terminos

Big Data Conxuntos de datos masivos ou complexos que normalmente requiren un software especial para o seu tratamento.

Framework Abstracción que proporciona unha forma estándar de construír e implementar aplicacións e é un entorno de software universal e reutilizable.

Cluster Conxunto de máquinas interconectadas entre elas por unha rede de comunicacións moi utilizado no mundo HPC xa que, combinando os recursos de diferentes máquinas, pódese acadar un sistema de altas prestacións en conxunto de maneira máis sinxela.

Scheduler Elemento encargado de administrar a execución de diferentes procesos ou tarefas en función a diferentes variables.

Worker Proceso exclusivamente encargado de realizar traballos relacionados con tarefas de procesamento.

Speedup Aceleración con respecto a un valor anterior.

Bibliografía

[1] J. C. Bennett, H. Abbasi, P.-T. Bremer, R. Grout, A. Gyulassy, T. Jin, S. Klasky, H. Kolla, M. Parashar, V. Pascucci, P. Pebay, D. Thompson, H. Yu, F. Zhang, and J. Chen, "Combining in-situ and in-transit processing to enable extreme-scale scientific analysis," in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012, pp. 1–9.

[2] Ray Team, "Ray documentation," https://docs.ray.io/en/latest/index.html, Data de acceso: 15/02/2023.

[3] A. Gueroudji, "Distributed task-based in situ data analytics for high-performance simulations," Ph.D. dissertation, Université Grenoble Alpes, 2023.

[4] PDI Development Team, "Pdi documentation," https://pdi.dev/master/index.html, Data de acceso: 15/02/2023.

[5] Apache Arrow, "Plasma: In-memory object store," https://arrow.apache.org/blog/2017/08/08/plasma-in-memory-object-store/, Data de acceso: 24/02/2023.

[6] E. Dirand, "Integration of High-Performance Task-Based In Situ for Molecular Dynamics on Exascale Computers," Theses, Université Grenoble Alpes, Nov. 2018. [Online]. Available: https://hal.science/tel-01949170

[7] K.-L. Ma, C. Wang, H. Yu, and A. Tikhonova, "In-situ processing and visualization for ultrascale simulations," *Journal of Physics: Conference Series*, vol. 78, no. 1, p. 012043, jul 2007. [Online]. Available: https://dx.doi.org/10.1088/1742-6596/78/1/012043

[8] M. Dorier, "Addressing the Challenges of I/O Variability in Post-Petascale HPC Simulations," Theses, Ecole Normale Supérieure de Rennes, Dec. 2014. [Online]. Available: https://theses.hal.science/tel-01099105

[9] InfiniBand Trade Association, "Infiniband roadmap," https://www.infinibandta.org/infiniband-roadmap/, Data de acceso: 05/02/2023.

[10] Apache Software Foundation, "Apache hadoop," https://hadoop.apache.org/, Data de acceso: 06/06/2023.

[11] ——, "Apache spark," https://spark.apache.org/, Data de acceso: 18/06/2023.

[12] ——, "Apache flink," https://flink.apache.org/, Data de acceso: 06/06/2023.

[13] H. C. Zanúz, B. Raffin, Mures, and E. J. Padrón, "In-transit molecular dynamics analysis with apache flink," in *Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization, ISAV@SC 2018, Dallas, TX, USA, November 11, 2018*, M. Wolf, K. Moreland, and E. W. Bethel, Eds. ACM, 2018, pp. 25–32. [Online]. Available: https://doi.org/10.1145/3281464.3281469

[14] Parsl Project, "Parsl," https://parsl-project.org/, Data de acceso: 18/06/2023.

[15] Dask Development Team, "Dask - parallel computing with task scheduling," https://www.dask.org/, Data de acceso: 15/02/2023.

[16] F. Zheng, H. Yu, C. Hantas, M. Wolf, G. Eisenhauer, K. Schwan, H. Abbasi, and S. Klasky, "Goldrush: Resource efficient in situ scientific data analytics using fine-grained interference aware execution," in *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–12.

[17] Ray Team, "Plasma: In-memory object store," https://ray-project.github.io/2017/08/08/plasma-in-memory-object-store.html, Data de acceso: 16/02/2023.

[18] ——, "Dask on ray," https://docs.ray.io/en/latest/ray-more-libs/dask-on-ray.html, Data de acceso: 30/03/2023.

[19] Oak Ridge National Laboratory, "Adios: Adaptive input/output system," https://csmd.ornl.gov/adios, Data de acceso: 06/06/2023.

[20] Ray Team, "Starting ray," https://docs.ray.io/en/latest/ray-core/starting-ray.html, Data de acceso: 12/03/2023.

[21] ——, "Ray objects," https://docs.ray.io/en/latest/ray-core/objects.html, Data de acceso: 16/02/2023.

[22] ——, "Ray v2 architecture," https://docs.google.com/document/d/1tBw9A4j62ruI5omIJbMxly-la5w4q_TjyJgJL_jN2fI/preview#, Data de acceso: 22/03/2023.

[23] ——, "Getting started with ray clusters," https://docs.ray.io/en/latest/cluster/key-concepts.html, Data de acceso: 16/02/2023.

[24] SchedMD LLC, "Slurm workload manager documentation," https://slurm.schedmd.com/documentation.html, Data de acceso: 06/06/2023.

[25] Ray Team, "Ray actors," https://docs.ray.io/en/latest/ray-core/actors.html, Data de acceso: 16/02/2023.

[26] ——, "Ray client," https://docs.ray.io/en/latest/cluster/running-applications/job-submission/ray-client.html, Data de acceso: 12/03/2023.

[27] Mesocentre CentraleSupélec, "Cluster overview," https://mesocentre.pages.centralesupelec.fr/user_doc/ruche/01_cluster_overview/, Data de acceso: 05/02/2023.

[28] X. Fernández, "Reisa," https://github.com/xicko7/reisa, Data de acceso: 06/06/2023.

[29] Ray Team, "Ray timeline," https://docs.ray.io/en/latest/ray-core/api/doc/ray.timeline.html#ray-timeline, Data de acceso: 23/05/2023.

[30] X. Fernández, "Reisa tests," https://github.com/xicko7/reisa_tests, Data de acceso: 10/06/2023.