Miguel Robledo Kusz

Desenvolupament d'una plataforma de computació voluntària per executar aplicacions (científiques) al navegador i/o Node.js

TREBALL DE FI DE GRAU

dirigit pel Dr. Marc Sánchez Artigas

Grau d'Enginyeria Informàtica

UNIVERSITAT ROVIRA I VIRGILI

Tarragona

2025

Agraïments

Vull expressar el meu sincer agraïment al professor German Telmo Eizaguirre. Segurament aquest projecte no hauria estat possible sense la seva ajuda i orientació en totes i cadascuna de les fases del treball.

També agraeixo al Dr. Marc Sánchez Artigas per haver dirigit la tesi.

Resum

Aquest projecte presenta PyEdgeCompute, un sistema per a l'execució distribuïda de càrregues de treball paral·leles d'anàlisi de dades en dispositius voluntaris. Habitualment, les càrregues d'anàlisi de dades s'executen o bé a clústers propietaris o bé a desplegaments al cloud. No obstant això, el cost d'aquest recursos ha motivat la cerca de solucions que aprofitin capacitat computacional dispersa i heterogènia dels dispositius voluntaris i a l'Edge. PyEdgeCompute proposa un framework per a executar l'anàlisi de dades directament a l'Edge, gestionant automàticament les dependències i l'intercanvi de dades entre les diferents etapes del càlcul.

PyEdgeCompute aprofita WebAssembly (WASM) per garantir la compatibilitat amb una àmplia varietat de dispositius, incloent navegadors web i entorns lleugers, sense necessitat d'instal·lacions complexes. El sistema permet que el client simplement defineixi les tasques, automatitzant la distribució, execució i coordinació dels càlculs.

Durant el desenvolupament s'ha dissenyat una arquitectura flexible que gestiona el control del flux de dades i l'assignació intel·ligent de tasques segons la disponibilitat i capacitat dels voluntaris. Els primers resultats obtinguts mostren una execució eficient i escalable en entorns heterogenis, obrint la porta a l'ús generalitzat de la computació voluntària en àmbits de data analytics complexos.

Aquest projecte contribueix a democratitzar l'accés a recursos de computació distribuïts, reduint la dependència de plataformes cloud costoses i infraestructures dedicades, i planteja un model sostenible per a l'aprofitament de recursos computacionals infrautilitzats.

Resumen

Este proyecto presenta PyEdgeCompute, un sistema para la ejecución distribuida de cargas de trabajo de análisis de datos en dispositivos voluntarios. Habitualmente, las cargas de análisis de datos se ejecutan o bien en clusters propietarios o bien en despliegues en el cloud. Sin embargo, el coste de estos recursos ha impulsado la búsqueda de soluciones que aprovechen la capacidad computacional dispersa y heterogénea de los dispositivos voluntarios y en la frontera. PyEdgeCompute propone un framework para ejecutar el análisis de datos directamente en el Edge, gestionando automáticamente las dependencias y el intercambio de datos entre las diferentes etapas del cálculo.

PyEdgeCompute utiliza WebAssembly (WASM) para asegurar la compatibilidad con una amplia variedad de dispositivos, incluyendo navegadores web y entornos ligeros, sin necesidad de instalaciones complejas. El sistema permite que el cliente simplemente proporcione un DAG que define las tareas y sus dependencias, automatizando la distribución, ejecución y coordinación de los cálculos.

Durante el desarrollo se ha diseñado una arquitectura flexible que gestiona el control del flujo de datos y la asignación inteligente de tareas según la disponibilidad y capacidad de los voluntarios. Los primeros resultados obtenidos muestran una ejecución eficiente y escalable en entornos heterogéneos, abriendo la puerta al uso generalizado de la

computación voluntaria en ámbitos de análisis de datos complejos.

Este proyecto contribuye a democratizar el acceso a recursos de computación distribuidos, reduciendo la dependencia de plataformas cloud costosas e infraestructuras dedicadas, y plantea un modelo sostenible para el aprovechamiento de recursos computacionales infrautilizados.

Abstract

This project introduces PyEdgeCompute, a system for the distributed execution of parallel data analytics workloads on volunteer devices. Usually, data analytics workloads are executed in proprietary clusters or cloud deployments. However, the cost of such resources incentives the search of solutions leveraging the disperse and heterogeneous computational capabilities of volunteer and edge devices. PyEdgeCompute presents a framework to execute data analytics directly in the Edge, automatically managing dependencies and data exchanges between computation stages.

PyEdgeCompute utilizes WebAssembly (WASM) to ensure compatibility across a wide range of devices, including web browsers and lightweight environments, without requiring complex installations. The system allows clients to simply provide a DAG defining tasks and dependencies, automating task distribution, execution, and coordination.

A flexible architecture has been designed to manage data flow control and intelligent task assignment based on volunteer availability and capacity. Preliminary results demonstrate efficient and scalable execution in heterogeneous environments, paving the way for the generalization of volunteer computing in complex data analytics domains.

This project contributes to democratizing access to distributed computing resources, reducing reliance on costly cloud platforms and dedicated infrastructures, and proposes a sustainable model for leveraging underutilized computational resources.

Índex

1. Introducció ... 1
1.1. Computació distribuïda per a data analytics ... 1
1.2. Computació voluntària ... 4
1.3. Limitacions actuals de la computació voluntària ... 5
1.4. PyEdgeCompute ... 5
2. Descripció general del projecte ... 7
2.1. Entorn ... 7
2.2. Necessitats ... 7
2.3. Previsions d'ús ... 8
2.4. Tipus de DAGs suportats ... 8
3. Disseny ... 9
3.1. Arquitectura de PyEdgeCompute ... 9
3.2. Compatibilitat dels workers amb dispositius voluntaris ... 10
3.3. Intercanvi de dades ... 10
3.4. Client ... 11
3.5. Orquestrador ... 13
3.6. Worker ... 15
3.7. Programació de tasques a PyEdgeCompute ... 16
4. Implementació ... 18
4.1. Client ... 18
4.2. Orquestrador ... 21
4.2.1. Main ... 21
4.2.2. Estructures de dades ... 23
4.2.3. Gestió de la comunicació amb clients i workers ... 30
4.2.4. Assignació i gestió de tasques ... 40
4.3. Worker ... 46
4.3.1. Main ... 46
4.3.2. PyodideRuntime ... 47
4.3.3. TasksExecutor ... 48
4.3.4. Serialització/Deserialització de resultats ... 49
4.3.5. Comunicació amb l'orquestrador ... 50
4.3.6. Interacció amb el Storage ... 51
4.4. Docker (proves locals) ... 53
4.5. Depuració i descobriment d'errors ... 54
5. Avaluació ... 56

5.1. Definició de funcions de rendiment....56
5.2. Experiments patró MapReduce Terasort....58
5.3. Experiments patró MapReduce Wordcount....64
5.4. Gestió del paral·lelisme i rendiment....71
5.5. Anàlisi d'overhead pyodide....74
6. Limitacions i següents passos....76
6.1. No Multitenancy....76
6.2. Error accés S3....76
6.3. Gestió de memòria....76
6.4. Avaluació a dispositius voluntaris reals....76
6.5. Implementació de càrregues més complexes....76
6.6. Millora de la interfície del client....77
6.7. Seguretat i control d'accés....77
6.8. Gestió de workers stagglers....77
6.9. Refactorització del codi....77
6.10. In memory cache....77
6.11. Assignació intel·ligent de tasques....78
6.12. Sistemes de recompensa....78
7. Aplicació de principis ètics i de responsabilitat social....79
8. Conclusions....80
9. Referències....81
10. Annexos....1
10.1. Codi font del projecte....1
10.2. Detecció i optimització d'un coll d'ampolla a l'orquestrador....1
10.3. Solució als problemes de rendiment detectats amb configuracions elevades....12
10.4. Detecció de problema amb execució Multitenancy....14

Índex de figures

Figura 1. DAG que representa l'execució d'un job seguint l'estil de MapReduce, P. Wangsom et al., 2017.. 2

Figura 2. Diagrama d'un treball terasort, Nowicki, 2020.. 3

Figura 3. Diagrama de l'arquitectura típica de volunteer computing, Chorazyk et al., 2017. 9

Figura 4. Exemple de fitxer de configuració per part del client (config.json)..................... 12

Figura 5: Constructor i gestió de registre de la classe ClientRegistry................................ 15

Figura 6: Definició d'una tasca mapper wordcount.. 17

Figura 7: Exemple de control d'errors de paràmetres al client... 19

Figura 8: Objecte javascript de configuració al client.. 19

Figura 9: Implementació funció sendTaskWithRetry(task, httpUrl).................................. 20

Figura 10: Implementació de l'endpoint registerWorker a l'orquestrador........................... 22

Figura 11: Afegir tasca al registre de clients de l'orquestrador... 24

Figura 12: Obtenir subtasca (MapReduce) a l'orquestrador.. 24

Figura 13: Funció remove(tasks) TaskQueue a l'orquestrador.. 26

Figura 14: Classes TaskNode i TaskQueue a l'orquestrador... 26

Figura 15: Obtenció i eliminació de la pròxima tasca de la cua... 27

Figura 16: Classe i constructor WorkerRegistry a l'orquestrador....................................... 28

Figura 17: Afegir worker al registre de workers de l'orquestrador..................................... 29

Figura 18: Enviament del resultat d'una tasca al client... 31

Figura 19: Assignació de tasca a un worker a l'orquestrador.. 32

Figura 20: Gestió de la comunicació amb els clients a l'orquestrador................................ 33

Figura 21: Comprovació d'error de subtasca a l'orquestrador... 35

Figura 22: Creació d'una tasca MapReduce a l'orquestrador.. 35

Figura 23: Codi per l'execució de cada mapper.. 36

Figura 24: Finalització de la fase Map i preparació de la fase Reduce a l'orquestrador...... 37

Figura 25: Codi per l'execució de cada reducer.. 37

Figura 26: Enviament del resultat d'una tasca al client per part de l'orquestrador.............. 39

Figura 27: Funció per reservar i enviar tasca al worker.. 42

Figura 28: Codi main del worker.. 46

Figura 29: Càrrega i inicialització del runtime de Pyodide al worker................................ 47

Figura 30: Injecció dels mòduls de Python a l'entorn d'execució de Pyodide..................... 48

Figura 31: Comprovació bucket local o S3 al worker... 49

Figura 32: Creació i execució del codi final usant Pyodide al worker................................ 50

Figura 33: Recepció i execució d'una tasca al worker.. 51

Figura 34: Càlcul d'offset i obtenció d'objecte parcial per part d'un mapper...................... 52

Figura 35: Obtenció dels resultats dels mappers per part d'un reducer................................52
Figura 36: Funció deserialize_partitions definida al Pyodide Runtime...............................53
Figura 37: Fitxer Docker compose de l'orquestrador..54
Figura 38: Fragment del fitxer terasort-240m...59
Figura 39: Wall-clock time amb Terasort, augmentant nombre de workers.........................59
Figura 40: Max-stage time amb Terasort, augmentant nombre de workers.........................60
Figura 41: Max-stage time / Wall-clock time amb Terasort, augmentant el nombre de workers.. 61
Figura 42: Speedup amb Terasort, augmentant el nombre de workers................................62
Figura 43: Eficiència amb Terasort, augmentant el nombre de workers..............................62
Figura 44: Comparació temps CPU total vs temps E/S total amb Terasort, augmentant nombre de workers.. 63
Figura 45: Diagrama d'un job MapReduce Wordcount (Pereira, Óscar, 2014)....................64
Figura 46: Fragment de text concatenated_gutenberg_books.txt..65
Figura 47: Wall-clock time amb Wordcount, augmentant nombre de workers....................66
Figura 48: Max-stage time amb Wordcount, augmentant nombre de workers....................67
Figura 49: Max-stage time / Wall-clock time amb Wordcount, augmentant el nombre de workers.. 67
Figura 50: Speedup amb Wordcount, augmentant nombre de workers...............................68
Figura 51: Eficiència amb Wordcount, augmentant nombre de workers............................69
Figura 52: Comparació temps CPU total vs temps E/S total amb Wordcount, augmentant nombre de workers.. 70
Figura 53: Wall-clock time amb Terasort, nombre de workers fixat a 5.............................71
Figura 54: Max-stage time amb Terasort, nombre de workers fixat a 5..............................72
Figura 55: Max-stage time / Wall-clock time amb Terasort, nombre de workers fixat a 5.. 72
Figura 56: Gràfica de comparació de l’overhead de Pyodide amb Wordcount....................74
Figura 57: Gràfica de comparació de l’overhead de Pyodide amb Terasort........................75
Figura 58: Wall-clock time amb Terasort, augmentant nombre de workers (annex).............1
Figura 59: Wall-clock time amb Wordcount, augmentant nombre de workers (annex).........2
Figura 60: Max-stage time amb Wordcount, augmentant nombre de workers (annex).........3
Figura 61: Max-stage time amb Terasort, augmentant nombre de workers (annex)..............3
Figura 62: Fitxer mapreducewordcount amb 5 mappers i 1 reducer (5 workers)..................4
Figura 63: Fitxer mapreducewordcount amb 10 mappers i 1 reducer (5 workers, primers 5 mappers).. 5
Figura 64: Fitxer mapreducewordcount amb 10 mappers i 1 reducer (5 workers, últims 5 mappers).. 6
Figura 65: Fitxer mapreducewordcount amb 10 mappers i 1 reducer, retornant workerId (5 workers, primers 5 mappers)... 8

Figura 66: Fitxer mapreducewordcount amb 10 mappers i 1 reducer, retornant workerId (5 workers, últims 5 mappers)....................9
Figura 67: Funció per reservar i enviar tasca al worker (ACTUALITZADA)....................11
Figura 68: Wall-clock time amb Terasort (annex)....................12
Figura 69: Max-stage time amb Terasort (annex)....................12
Figura 70: Max-stage time / Wall-clock time amb Terasort (annex)....................13
Figura 71: Wall-clock time amb Terasort, augmentant el nombre de workers i multitenancy. 14
Figura 72: Max-stage time amb Terasort, augmentant el nombre de workers i multitenancy. 15
Figura 73: Max-stage time / Wall-clock time amb Terasort, augmentant el nombre de workers i multitenancy....................15

1 Introducció

L'anàlisi de grans volums de dades s'ha convertit en un pilar fonamental en diversos àmbits, com ara la medicina, l'astronomia, l'economia o l'enginyeria. La capacitat per processar i extreure informació rellevant d'aquestes dades permet comprendre millor fenòmens complexos, optimitzar processos i prendre decisions informades que milloren la qualitat de vida i l'eficiència en diferents sectors.

Durant les últimes dues dècades, la quantitat de dades generades ha experimentat un creixement gairebé exponencial. Segons l'estudi Chen et al., 2017 [1], gran part d'aquestes dades generades suposen un repte important per ser processades, no només per la qualitat variable de les dades, sinó també per limitacions en la capacitat computacional i d'emmagatzematge disponibles. Aquesta realitat planteja importants reptes per a la gestió eficient i efectiva de la informació.

Per fer front a aquesta problemàtica, han sorgit tècniques de computació distribuïda, basades en clústers de computació o computació al núvol. L'àmbit que engloba aquest processament massiu de dades s'anomena data analytics.

1.1 Computació distribuïda per a data analytics

L'anàlisi de dades (o *data analytics)* consisteix en el procés d'explorar, netejar, transformar i modelar grans volums d'informació per extreure'n coneixement útil o prendre decisions informades. Aquest procés es representa habitualment com una càrrega de treball composta per un conjunt d'operacions ordenades i dependents, que poden ser modelades com un Graf Acíclic Dirigit *(*DAG). En aquest graf dirigit sense cicles, els nodes representen les operacions o transformacions de dades, mentre que les arestes indiquen les dependències entre aquestes. Aquest model permet definir clarament l'ordre d'execució i optimitzar el processament, especialment per a dades massives i complexes.

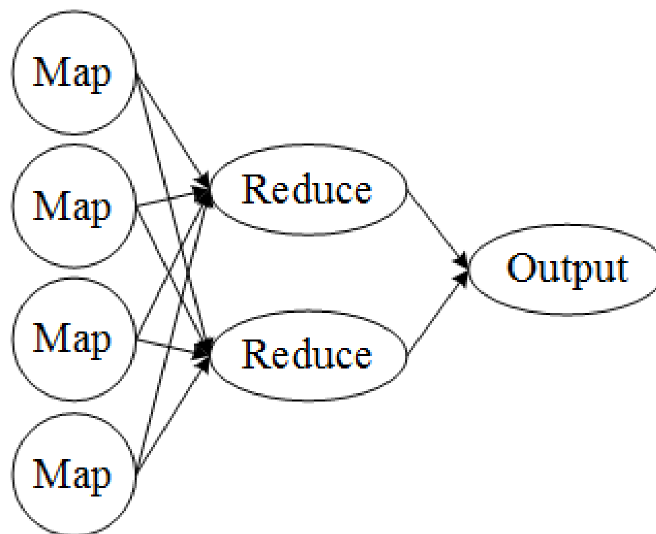

Figura 1. DAG que representa l'execució d'un job seguint l'estil de MapReduce, P. Wangsom et al., 2017 [2].

Quan les dades d'entrada són molt grans, les càrregues de treball d'anàlisi no es poden executar eficientment en un únic equip o servidor. Per això, s'usen entorns distribuïts, com ara *clusters* propietaris o plataformes al *cloud*, que permeten paral·lelitzar l'execució de les diferents operacions. En aquest context, les operacions del DAG es divideixen en *stages*, que són conjunts d'operacions executables en paral·lel. Cada unitat d'execució dins d'una stage es denomina *tasca* (task). Aquesta paral·lelització facilita l'escalabilitat i redueix el temps total d'execució.

Un exemple clàssic d'aquesta arquitectura és el Terasasort, que implementa un sort distribuït, una càrrega de treball que ordena grans volums de dades i que es representa mitjançant un DAG (Figura 1) format per dos stages principals:

- **Estadi 1:** lectura i partició de dades (map phase)

- **Estadi 2:** barreja i ordenació final (reduce phase)

Aquest DAG es pot representar gràficament, mostrant com les tasques de cada stage s'executen en paral·lel i com depenen entre elles.

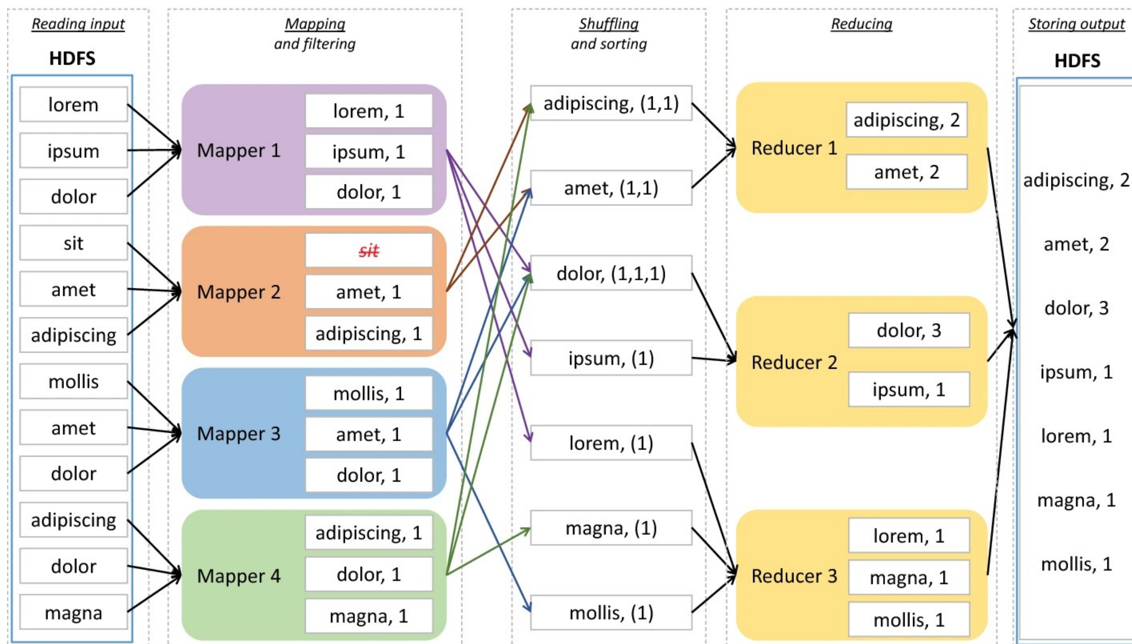

Figura 2. Diagrama d'un treball terasort, Nowicki, 2020 [3].

Per a l'execució de DAGs distribuïts com aquest, existeixen diversos frameworks establerts que simplifiquen la gestió de tasques i l'escalabilitat del processament de dades, encara que no suporten l'ús de dispositius voluntaris. Entre els més destacats hi ha **Apache Spark**, **Dask** i **Hadoop MapReduce** (Zaharia et al., 2016 [4]; Rocklin, 2015 [5]; White, 2012 [6]), que permeten implementar pipelines paral·leles de manera eficient i gestionar grans volums de dades amb consistència i tolerància a fallades.

El problema del data analytics distribuït

Les càrregues de treball d'anàlisi de dades a gran escala requereixen una infraestructura computacional robusta per processar grans volums de dades de manera eficient. L'accés a aquesta infraestructura es pot obtenir de diverses maneres. Una opció és utilitzar un clúster físic de computació, com el supercomputador MareNostrum, que sovint implica requisits d'accés estrictes i llistes d'espera que poden limitar la disponibilitat per a l'execució freqüent o puntual de tasques d'anàlisi. També es pot tractar d'adquirir un clúster privat, però suposaria uns costos econòmics i tècnics elevats.

Una altra alternativa és provisionar recursos virtualitzats a través de proveïdors d'Infraestructura com a Servei (IaaS), llogant instàncies de computació sota demanda. A AWS, això es pot fer utilitzant serveis com **Amazon EC2** (Elastic Compute Cloud) per desplegar instàncies de servidors virtuals escalables segons les necessitats de càrrega de treball, o **Amazon EBS** (Elastic Block Store) per a emmagatzematge persistent d'alta disponibilitat. Tot i que aquesta opció ofereix escalabilitat i flexibilitat, pot ser costosa i requereix un coneixement avançat per configurar i gestionar la infraestructura subjacent,

incloent la configuració de xarxes, seguretat i monitoratge amb **Amazon VPC** i **CloudWatch**.

Finalment, les solucions de Plataforma com a Servei (PaaS) ofereixen plataformes d'anàlisi gestionades, on la complexitat del manteniment de la infraestructura queda abstraïda. A AWS, exemples clars inclouen **Amazon EMR** (Elastic MapReduce) per al processament massiu de dades amb Hadoop o Spark gestionats, i **AWS Glue**, que ofereix serveis ETL (Extract, Transform, Load) completament gestionats per preparar dades per a l'anàlisi. No obstant això, aquestes solucions poden tenir un cost elevat i limitar les possibilitats de personalització. Tant les opcions IaaS com PaaS presenten reptes en la gestió dels costos, la complexitat en la configuració i la càrrega operativa.

1.2 Computació voluntària

Una possible solució als reptes relacionats amb la necessitat d'una infraestructura potent per executar tasques d'anàlisi de dades a gran escala és la computació voluntària (també anomenada computació col·laborativa). Aquesta modalitat consisteix a aprofitar la capacitat de càlcul inactiva de dispositius de consum (telèfons mòbils, tauletes, ordinadors personals) distribuïts a través d'Internet, aportada per voluntaris que cedeixen una part dels seus recursos computacionals de manera altruista. Aquests dispositius, sovint subtilitzats en termes de capacitat computacional, representen una oportunitat per augmentar el rendiment sense incrementar significativament el consum energètic ni l'impacte ambiental, cosa que es tradueix en una millora de l'eficiència computacional global. D'aquesta manera, es crea una xarxa distribuïda que permet executar càlculs complexos sense necessitat d'invertir en costosa infraestructura dedicada.

Històricament, un dels projectes pioners en aquest camp va ser **SETI@HOME**, iniciat l'any 1999 per la Universitat de Berkeley. Aquest projecte buscava senyals de vida extraterrestre processant dades de radiotelescopis a través dels ordinadors de voluntaris a tot el món. SETI@HOME va popularitzar el concepte de computació voluntària i va demostrar la seva viabilitat per abordar problemes que requereixen una gran capacitat de càlcul.

Actualment, existeixen diversos projectes que utilitzen la computació voluntària per a diferents finalitats. Per exemple, **WISDOM (Wide In Silico Docking On Malaria)** és un projecte enfocat en la recerca de nous medicaments contra la malària mitjançant el processament distribuït de simulacions moleculars. Altres projectes coneguts són **Folding@home**, que estudia el plegament de proteïnes per comprendre malalties com l'Alzheimer i el Parkinson, i **BOINC (Berkeley Open Infrastructure for Network Computing)** P. Anderson et al. 2004, [14], una plataforma que permet executar diversos projectes científics basats en computació voluntària de manera simultània.

La computació voluntària és especialment valuosa per a comunitats científiques i d'investigació que no disposen de recursos suficients per a clústers o infraestructures en el núvol, i contribueix a la democratització de l'accés a la capacitat de càlcul a gran escala.

1.3 Limitacions actuals de la computació voluntària

Els sistemes de computació voluntària actuals estan principalment optimitzats per executar càrregues de treball amb un alt grau de paral·lelisme sense dependències entre tasques, conegudes com a tasques "embarassingly parallel". Això facilita la distribució dels càlculs entre els voluntaris, ja que no requereix una gestió complexa de dependències ni d'intercanvi intens de dades entre les diferents unitats de càlcul.

Tanmateix, els fluxos de treball d'anàlisi de dades basats en DAGs impliquen dependències explícites entre stages i necessiten una coordinació i comunicació entre tasques que la majoria de plataformes de computació voluntària (com les mencionades a l'apartat anterior) encara no gestionen de manera nativa o eficient. Tot i que hi ha investigacions i prototips que exploren aquestes capacitats, com ara la plataforma SDDF per a computació voluntària en química quàntica (Ghukasyan et al., 2025 [7]), SLINC per a projectes distribuïts flexibles (Lavoie & Hendren, 2019 [8]) o entorns de ciència ciutadana basats en jocs que modelen tasques com a DAGs (González & Pérez, 2021 [9]), la implementació robusta d'aquesta funcionalitat en entorns voluntaris és encara limitada i representa un repte important per ampliar l'abast de la computació voluntària en aplicacions d'anàlisi de dades complexes.

1.4 PyEdgeCompute

En aquest treball proposem **PyEdgeCompute**, un sistema per a l'execució distribuïda de DAGs paral·lels de processos d'anàlisi de dades en dispositius voluntaris. PyEdgeCompute gestiona de manera automàtica les dependències i els intercanvis de dades entre les diferents etapes del DAG, de manera que el client només ha de proporcionar el treball, i el sistema s'encarrega de la seva execució eficient i segura.

Amb l'objectiu de garantir la compatibilitat amb un ampli ventall de dispositius, PyEdgeCompute aprofita **WebAssembly (WASM)**, un llenguatge compilat que pot executar-se tant en navegadors web com en diversos entorns d'execució lleugers, fent ús de node o deno (entorns d'execució de javascript), assegurant portabilitat i un baix consum de recursos. Aquesta elecció permet que dispositius heterogenis, des d'ordinadors personals fins a dispositius mòbils o encastats, puguin contribuir al càlcul.

En aquest context, cal destacar l'existència de prototipus de recerca com **Wasimoff** (A. Semjonov et al., 2024 [10]), que també utilitzen WebAssembly per facilitar l'*offloading* de tasques en l'*edge*. Tot i això, aquests enfocaments es centren principalment en l'execució aïllada de tasques i no donen suport nadiu a l'accés a dades externes ni a la

gestió de dependències entre etapes. En contraposició, **PyEdgeCompute** integra els avantatges de la portabilitat i l'eficiència de WASM amb la capacitat de gestionar de manera automàtica DAGs i fluxos de treball complexos, combinant així el millor dels dos mons.

Les contribucions principals d'aquest treball són:

- **Contribució 1:** Desenvolupament d'un sistema de computació distribuïda que permet la gestió automàtica de dependències i intercanvis de dades en DAGs paral·lels en entorns voluntaris, millorant l'eficiència i escalabilitat de l'execució.

- **Contribució 2:** Integració de WebAssembly per garantir la compatibilitat multiplataforma, permetent l'ús de recursos computacionals heterogenis sense necessitat de configuracions específiques per a cada dispositiu ni instal·lacions. A més això ens permet executar tasques en Python usant Pyodide, lo que ens permet executar tasques d'anàlisi de dades complexes amb Pandas i Numpy.

- **Contribució 3:** Proposta d'una interfície d'usuari simplificada que només requereix la definició del DAG per part del client, ocultant la complexitat de la distribució i coordinació de les tasques.

2 Descripció general del projecte

Aquest projecte és de caràcter acadèmic i consisteix en el desenvolupament d'un prototip funcional amb el que ja es poden executar les tasques d'anàlisi de dades de manera distribuïda en paral·lel. Encara hi ha diversos aspectes que caldria millorar per transformar-lo en un sistema apte per a producció, aquests es mencionen a l'apartat Limitacions i següents passos.

2.1 Entorn

Durant les primeres fases del desenvolupament, aquest projecte s'ha implementat en un entorn local utilitzant **MinIO**, que proporciona tant una **libreria per interactuar amb el servei** com un **contenidor Docker per allotjar-lo**. El servei simula el comportament dels **buckets d'AWS S3**, oferint funcions similars a l'API de S3, de manera que les funcions de la llibreria poden provar-se com si estiguessin treballant amb un entorn S3 real. Tot i això, l'entorn local no és l'ideal per realitzar proves que mesurin el rendiment real del sistema, ja que es necessita un nombre elevat d'unitats de processament per garantir que cada procés Node s'executi realment en paral·lel sobre diferents nuclis.

Aquest comportament de paral·lelisme real s'ha aconseguit amb l'entorn al núvol, on s'ha fet ús de la plataforma **Amazon Web Services (AWS)**, amb els serveis **AWS S3** per a l'emmagatzematge de dades i **AWS EC2** per al desplegament de nodes distribuïts sota demanda, aprofitant els múltiples nuclis i la infraestructura escalable proporcionada per Amazon (Amazon Web Services, 2023a [11]; Amazon Web Services, 2023b [12]).

Els tipus de instàncies usades han sigut les t3.medium (2 vCPU i 4 GB RAM) per tal d'evitar problemes de rendiment i memòria.

Aquest desplegament permet mesurar amb precisió el rendiment en condicions pròpies d'un entorn distribuït real, on múltiples nodes executen tasques simultàniament.

En un entorn real les instàncies EC2 serien usuaris voluntaris i les latències de comunicació serien majors.

2.2 Necessitats

Per al desenvolupament d'aquest projecte ha estat necessari, com s'ha esmentat anteriorment, disposar d'un ordinador personal per dur a terme les primeres proves de funcionament. Posteriorment, s'ha requerit accés a un compte d'AWS per poder realitzar el desplegament i els experiments en un entorn que s'apropa més a les condicions reals. Aquest compte ha estat facilitat per l'equip d'investigació de la URV CloudLab, que ha donat suport a la realització d'aquestes proves.

2.3 Previsions d'ús

Aquest projecte s'enfoca principalment com a projecte de recerca, amb l'objectiu principal d'incloure el prototip desenvolupat en un article científic. Tot i que no es descarta la possibilitat de continuar el desenvolupament i portar el sistema a un estat més madur en el futur, aquesta no és la finalitat immediata d'aquest treball.

2.4 Tipus de DAGs suportats

L'actual framework només suporta, de moment, treballs que segueixen el patró de computació distribuïda **MapReduce,** que suposa dos *stages* (map i reduce). Concretament, es troben disponibles la implementació de l'algorisme **WordCount**, que realitza un comptatge d'elements en fitxers de text, i la implementació de l'algorisme **TeraSort**, utilitzat per a l'ordenació distribuïda de grans volums de dades.
La idea del projecte seria estendre el framework per suportar altres tipus de DAGs més complexos, amb múltiples e*stadis* i dependències més elaborades, ampliant així l'abast de l'entorn distribuït més enllà del patró MapReduce bàsic.

3 Disseny

3.1 Arquitectura de PyEdgeCompute

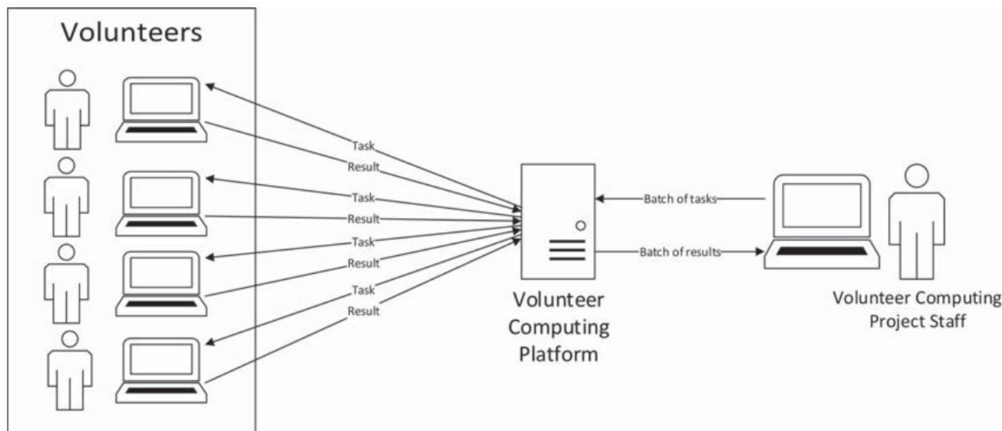

Figura 3. Diagrama de l'arquitectura típica de volunteer computing, Chorazyk et al., 2017 [13].

En aquest cas, al framework **PyEdgeCompute** hem utilitzat els mateixos components habituals en qualsevol arquitectura de tipus *volunteer computing*, però amb una nomenclatura pròpia i més familiar amb frameworks de data analytics establerts.

- **Worker**: com es pot veure a l'extrem esquerre de la Figura 3, aquest component representa l'usuari voluntari que es connecta al nostre sistema *orquestrador* per cedir, de manera altruista, els seus recursos de computació.

- **Orquestrador**: situat a la part central de la Figura 3, aquest component correspon al codi que s'executa en un servidor gestionat pels desenvolupadors o mantenidors del projecte. És el component principal de l'arquitectura i s'encarrega d'acceptar peticions d'execució de tasques per part dels clients, assignar-les als *workers* disponibles i mantenir tota la lògica de gestió necessària.

- **Client**: com es mostra a l'extrem dret de la Figura 3, el client equival al *Volunteer Computing Project Staff*. Aquest component es connecta a l'orquestrador quan vol enviar un lot (*batch*) de tasques i espera fins que aquest gestiona l'execució de totes elles abans de rebre el resultat final.

3.2 Compatibilitat dels workers amb dispositius voluntaris

Els *workers* executaran tasques definides en **Python** per part del client, ja que aquest és, avui dia, el llenguatge de programació de referència per a científics, enginyers i analistes de dades, gràcies al seu ampli ecosistema de llibreries optimitzades per a aquest tipus de treballs.

Per tal de permetre l'execució de Python en els *workers*, hem utilitzat **Pyodide**, una distribució de Python compilada a WebAssembly que permet executar codi Python directament en el navegador o en qualsevol entorn compatible amb WebAssembly, sense necessitat d'instal·lar Python al sistema. Pyodide ofereix una gran portabilitat, funciona en pràcticament qualsevol entorn web, i presenta un rendiment raonablement alt en comparació amb altres solucions interpretades al navegador. A més, inclou moltes de les llibreries més utilitzades de l'ecosistema Python, cosa que en facilita la integració en aplicacions científiques o d'anàlisi de dades.

En el nostre *framework*, hem fet ús, entre d'altres, de **Pandas** i **NumPy**. Pandas és una llibreria especialitzada en el tractament i la manipulació de dades en múltiples formats i estructures, mentre que NumPy ofereix implementacions altament optimitzades per a operacions d'àlgebra lineal, vectors i matrius.

Pel que fa a la implementació, el *worker*, igual que el client i l'orquestrador, s'ha desenvolupat amb **Node.js**. Aquesta elecció es deu al fet que Node.js proporciona un entorn d'execució ubicu i lleuger, amb un model d'aïllament a nivell de procés del sistema operatiu que contribueix a una execució més segura del codi del client. L'ús de Node.js no només és segur i portàtil, sinó també còmode i ràpid: l'usuari voluntari no ha de fer cap configuració manual. Pot, per exemple, connectar-se a un servidor web que li enviï automàticament el codi i les dependències necessàries (en format JSON, paquets JavaScript o mòduls de Pyodide), que el navegador descarregarà i inicialitzarà de manera transparent durant l'execució.

Finalment, Node.js pot oferir rendiments molt bons gràcies a les optimitzacions del motor **V8**, com **JIT compilation** i **inline caching**, que recompilen i optimitzen automàticament el codi que s'executa repetidament, millorant el rendiment amb el temps. La seva arquitectura d'**E/S asíncrona** basada en **libuv** gestiona múltiples operacions de manera concurrent sense crear un thread per a cada petició, evitant així l'overhead de creació, destrucció i canvi de context de threads, cosa que resulta especialment eficient en dispositius amb pocs nuclis.

A més, l'execució del codi del client en els **workers** es realitza mitjançant **Pyodide**, l'intèrpret de Python en WebAssembly (WASM), permetent assolir nivells de rendiment comparables als que s'obtindrien executant aquest codi en natiu. Aquesta combinació d'eficiència, portabilitat, facilitat de desplegament i un ampli ecosistema de llibreries fa que Node.js sigui una tecnologia idònia per a projectes d'edge computing lleuger.

3.3 Intercanvi de dades

Per a la gestió i persistència de les dades del sistema, durant el desenvolupament utilitzem **MinIO** en local, un servei d'**object storage** que simula els buckets i l'API d'**AWS S3**, mentre que per a les proves en entorn real fem servir **Amazon S3**. Aquest tipus de solució sempre està disponible (high availability) , és altament escalable i segueix un model serverless, la qual cosa significa que no requereix una configuració complexa ni manteniment continu, alhora que garanteix persistència i tolerància a errors. Com utilitzem dispositius voluntaris i no sabem el nombre ni el moment en el que es connectaran, seria complex saber quan i amb quina mesura desplegar un sistema in-memory de millor rendiment.

En el nostre flux de treball, el **client** defineix un *bucket* específic que conté els *inputs* necessaris per a cada tasca que volem executar. A partir d'aquí, l'**orquestrador** gestiona la distribució d'aquestes dades als *workers*. Un cop els *workers* completen la fase de *map* i *reduce*, escriuen els resultats en un *bucket* propi de l'orquestrador.

Aquesta arquitectura basada en *object storage* no només permet una gestió eficient i centralitzada de les dades, sinó que també afavoreix la flexibilitat del sistema i la seva capacitat per operar de manera fiable en entorns distribuïts o amb alta concurrència, però també presenta algunes limitacions com veurem a l'apartat Limitacions i següents passos.

3.4 Client

El client es defineix de manera que, per a cada execució, és necessari passar una sèrie de paràmetres obligatoris.

Format general de la comanda:

- node <ruta_fitxer_main> --config <ruta_fitxer_config.json> --orch <adreça_http>

A continuació es mostra un exemple concret:

- node client/src/main.mjs --config client/configs/mapreduce_wordcount.json --orch http://ec2-16-16-92-7.eu-north-1.compute.amazonaws.com:3000

L'adreça HTTP ha de començar obligatòriament amb http o https i incloure un port al

final.
El fitxer JSON de configuració defineix els detalls de l’execució.

```json
{
  "type": "mapreduceterasort",
  "code": [
    "client/codes/map_terasort.py",
    "client/codes/reduce_terasort.py"
  ],
  "args": [
    ["s3://clientbucketforstoringinputs/terasort-20m", 6, 5]
  ]
}
```

Figura 4. Exemple de fitxer de configuració per part del client (config.json).

`type`

Indica el tipus de tasca. Pot ser mapreduceterasort, mapreducewordcount o qualsevol altre valor personalitzat (nom aleatori). Les tasques que no son mapreduce són tasques simples, que el framework suporta per simples raons de retrocompatibilitat amb les primeres versions del sistema.

`code`

És una llista amb les rutes dels fitxers Python que implementen la lògica de la tasca.

Per a tasques mapreduce*, aquest array ha de tenir exactament dos elements: el primer corresponent al fitxer **map** i el segon al fitxer **reduce**. Si no es passen ambdós fitxers, l’orquestrador retornarà un error. Si es passen més de dos fitxers, només s’agafaran els dos primers.

En cas de les tasques simples, només poden definir un element, si en defineixen més, l’orquestrador només escollirà el primer element.

`args`

És una llista que conté els arguments d’entrada per a cada tasca.

Per a tasques mapreduce*, cada element és un array amb tres elements: la ruta de l’input (bucket + fitxer), el nombre de mappers i el nombre de reducers.

Exemple:

["s3://clientbucketforstoringinputs/terasort-20m", 6, 5]

Per a tasques que no són mapreduce, cada element és simplement un string amb la ruta d'entrada (bucket + fitxer) de la tasca.

Exemple:

["s3://clientbucketforstoringinputs/example1.txt"]

Nota:
Si falta algun dels camps obligatoris o algun paràmetre no compleix el format esperat, es retornarà un missatge d'error i no iniciarà l'execució.

3.5 Orquestrador

L'orquestrador s'inicia amb la següent comanda:

- node src/main.mjs

Un cop en execució, aixeca un servidor HTTP utilitzant la llibreria **Express.js** i defineix dos *endpoints* seguint el model d'una API REST:

- **Registre de client i tasques associades**.

POST /register_task

Paràmetre	**Tipus**	**Requerit**	**Descripció**
code	array de strings	Sí	Codi de les tasques a executar.
args	array de strings	Sí	Arguments per a cada tasca.
type	string	Sí	Tipus de tasca (per exemple,"mapreduce wordcount").

Descripció
Registra un client i les seves tasques. Retorna un `client_id` únic i dispatcha cada tasca als workers disponibles.

Resposta d'exemple (JSON)


```json
{
  "message": "Client registered successfully",
  "client_id": "550e8400-e29b-41d4-a716-446655440001"
}
```


- **Registre de nous workers**.

POST /register_worker			
Paràmetre	**Tipus**	**Requerit**	**Descripció**
numWorkers	integer	No	Nombre de processadors que té el worker.

Descripció
Registra un nou worker al sistema. Retorna un `worker_id` únic generat per identificar-lo.

Resposta d'exemple (JSON)


```json
{
  "message": "Worker registered successfully",
  "worker_id": "550e8400-e29b-41d4-a716-446655440000"
}
```


El servidor escolta peticions entrants al **port 3000** (actualment definit de manera fixa al codi). Quan un client o worker crida l'*endpoint* corresponent, l'orquestrador executa la lògica de registre i genera un identificador únic (UUIDv4, 128 bits). En cas d'èxit, aquest ID es retorna com a resposta; en cas d'error, es retorna un missatge d'error sense cap identificador.

Tant el client com el worker utilitzen aquest identificador per establir una connexió amb el **servidor WebSocket** gestionat per l'orquestrador. A través d'aquesta connexió persistent es duu a terme tota la comunicació posterior entre:

- Orquestrador ↔ Worker

- Orquestrador ↔ Client

```
class ClientRegistry {
  constructor() {
    if (ClientRegistry.instance) return ClientRegistry.instance;
    this.clients = new Map(); // clientId → { ws, numTasks, tasks: Map }
    ClientRegistry.instance = this;
  }

  registerClient(clientId, ws, numTasks) {
    this.clients.set(clientId, {
      ws,
      numTasks,
      numPendingTasks: numTasks,
      tasks: new Map(),
    });
  }
```

Figura 5: Constructor i gestió de registre de la classe ClientRegistry

Disseny de la comunicació

Aquest disseny respon a dos objectius principals:

- **Validació prèvia:** l'orquestrador pot verificar la informació del client i del worker abans de permetre la connexió WebSocket.

- **Separació de responsabilitats** (*Separation of Concerns*): la API REST només s'utilitza per a la petició inicial de registre, mentre que la comunicació bidireccional i en temps real es realitza exclusivament via WebSockets. Això evita haver d'aixecar un servidor HTTP als clients o als workers.

En l'implementació actual:

- L'orquestrador envia tasques als workers mitjançant WebSockets.

- Els workers retornen els resultats al mateix canal de comunicació.

- El client rep els resultats finals de les seves tasques.

En versions futures es podria afegir funcionalitat addicional, com ara **missatges de seguiment de progrés** o *polling* periòdic per part del client quan no s'han rebut resultats durant un període determinat.

Avantatges dels WebSockets a Node.js

- **Bidireccionalitat**: permeten enviar i rebre dades en temps real en ambdues direccions.

- **Eficiència**: mantenen una connexió persistent, evitant el cost de reobrir

connexions HTTP.

- **Robustesa**: suport per implementar *heartbeats*, detecció d'errors i reconeixement de connexions caigudes.

Tot i que el disseny actual de l'orquestrador basat en registre via HTTP i comunicació posterior mitjançant WebSockets és funcional i senzill, s'han considerat altres alternatives

que podrien aportar beneficis addicionals en entorns més grans o amb requisits més estrictes. Per exemple, l'ús d'un *message broker* com **RabbitMQ** o **Kafka** permetria desacoblar la comunicació entre l'orquestrador i els *workers*, millorar l'escalabilitat horitzontal i garantir la persistència de missatges, encara que aquesta persistència de missatges no és realment necessària a la implementació actual.

Una altra opció seria substituir els WebSockets per **gRPC amb *streams* bidireccionals**, que ofereix una comunicació binària més eficient, una API fortament tipada mitjançant Protobuf i una gestió d'errors més robusta.

Tot i aquests avantatges potencials, s'ha optat per mantenir el disseny actual amb WebSockets i HTTP per simplificar el desenvolupament i reduir la complexitat de desplegament, prioritzant la rapidesa d'implementació i la facilitat de manteniment en aquesta fase del projecte.

En resum, aquest enfocament garanteix **separació de responsabilitats**, **comunicació bidireccional eficient** i **alta fiabilitat** entre components distribuïts.

3.6 Worker

Es pot iniciar l'execució del worker amb la següent comanda:

- node src/main.mjs --orch http://ec2-13-48-55-235.eu-north-1.compute.amazonaws.com:3000 --storage s3://orchestratorfororchestratingworkers

El paràmetre --orch ja s'ha explicat prèviament en parlar del client, i aquí segueix exactament la mateixa lògica: indicar la direcció HTTP(s) i el port de l'orquestrador al qual el worker s'ha de connectar.

El paràmetre --storage especifica el bucket de l’orquestrador, que és on s’emmagatzemaran els resultats. Aquest paràmetre és essencial dins del nostre framework, ja que permet que els reducers puguin accedir als resultats generats pels mappers durant l’execució de les tasques.

Quan el worker s’inicia, carrega Pyodide i comprova la presència de les llibreries *pandas* i *numpy*, instal·lant-les automàticament si encara no es troben disponibles. A continuació, realitza una crida a l’endpoint corresponent de l’orquestrador (descrit a l’apartat anterior) per tal de registrar-se. Un cop registrat correctament, queda disponible per rebre i executar tasques que l’orquestrador li assigni.

3.7 Programació de tasques a PyEdgeCompute

El framework PyEdgeCompute defineix una sèrie de funcions que es poden usar per part del client per tal de facilitar algunes tasques. En aquest cas proporciona funcions bàsiques per a serialitzar i deserialitzar els resultats de les execucions de les tasques i d’altres necessàries per a les tasques Terasort per poder portar a terme el particionament i ordenació. A continuació podem veure la llista de funcions definides:

Funcions definides al framework PyEdgeCompute
pyedgecompute.serialize_partition = serialize_partition
pyedgecompute.deserialize_partitions = deserialize_partitions
pyedgecompute.deserialize_input_string = deserialize_input_string
pyedgecompute.deserialize_input_terasort = deserialize_input_terasort
pyedgecompute.partition_data = partition_data
pyedgecompute.concat_partitions = concat_partitions
pyedgecompute.sort_dataframe = sort_dataframe

El client pot fer ús d'aquestes funcions durant la programació de les seves tasques. Només ha d'indicar el mòdul des d'on es volen importar (en aquest cas, pyedgecompute) i importar-les. Una vegada fet això ja pot usar-les de manera normal al seu codi, depenent de la lògica que vulgui implementar. [1]

1 Durant la programació, és normal que l'interprèt del client senyali les importacions amb un avís, ja que l’entorn de python configurat al client no té aquest mòdul definit ni les funcions, sinó que està tot definit a l'entorn dels workers. Quan s'enviï aquest codi al *worker* i l’executi al seu entorn, el codi sí que tindrà accés a aquests mòduls i funcions. És una qüestió d’entorn d’execució.

```python
from pyedgecompute import deserialize_input_string, serialize_partition
from collections import Counter

def task(bytes):
    text = deserialize_input_string(bytes)
    words = text.split()
    counter = Counter(words)
    return serialize_partition(counter)
```

Figura 6: Definició d'una tasca mapper wordcount.

`serialize_partition(result)`

- **Signatura:** `serialize_partition(result: Any) -> str`
- **Descripció:** Serialitza un objecte de Python (per exemple un `DataFrame`) amb `pickle` i el codifica en base64 per transmetre'l fàcilment.

`deserialize_partitions(b64_list)`

- **Signatura:** `deserialize_partitions(b64_list: List[str]) -> List[Any]`
- **Descripció:** Deserialitza una llista de partitions codificades en base64 i retorna els objectes Python originals.

`deserialize_input_string(bytes_string)`

- **Signatura:** `deserialize_input_string(bytes_string: bytes) -> str`
- **Descripció:** Converteix un `bytes` en una cadena de text (`str`).

`deserialize_input_terasort(data)`

- **Signatura:** `deserialize_input_terasort(data: bytes) -> pd.DataFrame`
- **Descripció:** Processa dades del format Terasort, extrau claus i valors de cada línia i els retorna com un `DataFrame` amb dues columnes.

`get_partition(line, num_partitions)`

- **Signatura:** `get_partition(line: str, num_partitions: int) -> int`
- **Descripció:** Calcula a quina partició correspon una línia de text segons els primers 8 caràcters, utilitzant un rang ASCII normalitzat.

`partition_data(data, num_partitions)`

- **Signatura:** `partition_data(data: pd.DataFrame, num_partitions: int) -> Dict[int, pd.DataFrame]`
- **Descripció:** Divideix un `DataFrame` en `num_partitions` partitions segons la clau de la columna `"0"`. Retorna un diccionari amb les partitions.

`concat_partitions(partition_list)`

- **Signatura:** `concat_partitions(partition_list: List[pd.DataFrame]) -> pd.DataFrame`
- **Descripció:** Concatena una llista de `DataFrame` en un sol `DataFrame`.

`sort_dataframe(df)`

- **Signatura:** `sort_dataframe(df: pd.DataFrame) -> pd.DataFrame`
- **Descripció:** Ordena un `DataFrame` per la columna `"0"` i reinicia els índexs.

4 Implementació

Aquest apartat consisteix en els detalls tècnics i codi usat per aconseguir les funcionalitats del sistema. L'enllaç al repositori amb el codi del projecte es pot trobar a Annexos.

4.1 Client

El client consisteix en 3 directoris principals:

- `codes`: conté els codis de mapreduce wordcount i terasort, tant el codi dels mappers com dels reducers.
- `configs`: conté diferents configuracions que s'han usat durant l'execució d'experiments.
- `src`: conté el codi font del programa

El mòdul main.mjs conté el codi principal, que fa ús dels demés mòduls. Aquest mòdul principal s'encarrega d'obtenir els arguments que passem durant l'execució de la comanda del client com hem vist a l'apartat anterior Disseny i a més envia les tasques a l'orquestrador (amb possiblitat de reintentar) i per últim es connecta en cas d'enviament exitós per tal de començar a rebre els resultats.

Totes les funcions que s'encarreguen de portar a terme l'obtenció dels arguments de l'execució de la comanda de la línia de comandes estan definides a utils.mjs i configLoader.mjs. En aquests fitxers ja es realitza un control d'errors i format bastant rigurós, però de totes formes també es realitzarà un mínim control a l'orquestrador (això va en la linia de metodologies de desenvolupament web, on normalment se sanititza l'entrada del client, tant al mateix client, per tal que la interacció sigui més inmediata i evitar comunicació de xarxa innecessària, com també al servidor, per tal d'evitar que es puguin introduir comandes no esperades). A continuació s'adjunta un exemple d'aquest control d'errors al client.

```
export async function loadConfig(filePath) {
  if (!(await fileExists(filePath))) {
    throw new Error(`Config file "${filePath}" does not exist.`);
  }

  const raw = await fs.readFile(filePath, "utf-8");
  const config = JSON.parse(raw);

  if (!config.type || !config.code || !config.args) {
    throw new Error(`Config must include 'type', 'code', and 'args'.`);
  }

  if (!Array.isArray(config.code) || config.code.length === 0) {
    throw new Error(`'code' must be a non-empty array.`);
  }

  if (!Array.isArray(config.args) || config.args.length === 0) {
    throw new Error(`'args' must be a non-empty array.`);
  }

  for (const file of config.code) {
    if (!(await fileExists(file))) {
      throw new Error(`Code file "${file}" does not exist.`);
    }
  }

  return config;
}
```

Figura 7: Exemple de control d'errors de paràmetres al client.

Després de l'obtenció dels arguments de manera exitosa, el main.mjs del client passa a executar la funció `sendTaskWithRetry(task, httpUrl)`.
La variable `task` conté el JSON de configuració parsejat anteriorment i preparat per ser accedit i modificat en javascript, mentre que `config` és una estructura o objecte javascript.

La variable `config` conté la informació que hem enviat durant l'execució de la comanda a través dels arguments, i és necessitarà per tal de realitzar la comunicació amb l'orquestrador correctament (per saber la seva direcció HTTP i WS).

```
const CONFIG = {
  HTTP_ORCH: null,
  WS_ORCH: null,
  CONFIG_PATH: null,
};
```

Figura 8: Objecte javascript de configuració al client.

La funció `sendTaskWithRetry(task, httpUrl)` es troba implementada al mòdul orchestratorAPI.mjs. En aquest fitxer tenim tota la comunicació necessària amb l'orquestrador, a banda d'un petit control d'errors per tal de poder realitzar reintents d'enviament en cas de fallar la primera vegada.

```
export async function sendTaskWithRetry(task, httpUrl) {
  while (true) {
    try {
      const res = await axios.post(`${httpUrl}/register_task`, task);
      console.log("🚀 Tasks submitted. CLIENT ID:", res.data.client_id);
      return res.data.client_id;
    } catch (err) {
      if (err.response) {
        console.error("❌ Response error:", err.response.data || err.message);
      } else if (err.request) {
        console.error("❌ No response from orchestrator.");
      } else {
        console.error("❌ Error:", err.message);
      }

      try {
        await rl.question("\n🔁 Press Enter to retry, or Ctrl+C to exit...\n");
      } catch {
        console.log("\n👋 Exiting...");
        process.exit(1);
      }
    }
  }
}
```

Figura 9: Implementació funció sendTaskWithRetry(task, httpUrl).

A la Figura 8 es veu la petició que fem a l'endpoint de l'orquestrador per registrar les tasques i la recepció del clientId que ha sigut generat per l'orquestrador. Aquesta petició es fa utilitzant la llibreria "axios", que facilitza tot el procés de realitzar peticions HTTP a APIs RESTful amb javascript. A banda gestionem el cas de que hi hagi un problema amb la informació que li estem enviant (err.response), també controlem que no sigui un problema de que la petició no ha pogut arribar, per exemple perquè l'orquestrador no està executant-se (err.request) o capturem un error de qualsevol altre tipus. En cas d'error, se'ns pregunta si volem reintentar l'enviament o sortir del procés.

Una vegada s'han enviat les tasques, el client es connecta al websocket usant el mateix clientId que ha rebut de l'execució de la funció anterior. Aquesta lògica es troba implementada a la funció `connectToWebSockettot(wsUrl, clientId, maxTasks, stopwatches, sentTime, outDir=null)`.

Els paràmetres son la direcció del servidor *websocket* de l'orquestrador, el `clientId`, el nombre de tasques definides (així quan haguéssem rebut totes les tasques podem tancar la connexió), `stopwatches` (per mesurar el temps que se triga entre l'enviament i recepció de cada tasca) i `outDir` (on guardarem els resultats de les nostres tasques per fer un anàlisi posterior). Cal mencionar que els stopwatches no s'han usat durant l'ànalisi offline dels experiments, però podrien usar-se per tal de veure la diferència entre el moment del temps en que se ha enviat una tasca, quan es rep i comparar-ho amb el moment del temps en quant es comença a executar la tasca als *workers* i quan acaba, bàsicament consultant aquesta diferència veuriem les latències de comunicació entre client, orquestrador i *workers*.

Al *websocket* es defineixen una sèrie de funcions. Entre elles la funció "`open`" que conté el codi que s'executa durant la primera connexió a l'orquestrador. També tenim "`message`" que conté el codi que s'executa al rebre un resultat de l'orquestrador d'alguna de les nostres tasques. Per últim també tenim "`close`" i "`error`".

4.2 Orquestrador

L'orquestrador és la part més complexa del sistema, ja que és la part que s'encarrega de gestionar les dependències de dades entre les fases dels jobs mapreduce, a banda d'encarregar-se de la gestió de la informació dels clients i els *workers*, assignació de tasques, recepció i enviament de resultats, control d'errors, etc.

Tot el codi el trobem al directori src/. Veurem les següents parts en ordre:

- Codi principal del main: inicialitza el servidor *http* i *websockets*, escolta peticions entrants.
- Estructures de dades usades per gestionar la informació dels clients, tasques i *workers*.
- Gestió dels missatges entrants i sortints per part dels clients i els *workers*.
- Assignació i despatx de tasques.

4.2.1 Main

Al fitxer `main.mjs` es pot veure la configuració del servidor HTTP de l'orquestrador i la manera com gestiona les peticions entrants dels clients i *workers*. L'orquestrador s'inicia amb Express.js i defineix dos endpoints principals:

1. **/register_worker**: aquest endpoint permet registrar un nou worker al sistema. Quan un worker crida aquesta ruta, l'orquestrador genera un **identificador únic (`worker_id`)** mitjançant uuidv4(), i crea un objecte amb les dades del worker, incloent el nombre màxim de processadors (`maxWorkers`), el nombre de processadors disponibles (`availableWorkers`) i un array de tasques assignades. Aquest objecte es guarda al registre de workers (`workerRegistry`). Un cop registrat, l'orquestrador retorna un missatge de confirmació amb l'ID del *worker*, per tal que aquest es pugui connectar al *websocket* i confirmar la disponibilitat.

2. **/register_task**: aquest endpoint permet registrar un client i les seves tasques. Quan un client envia la sol·licitud, l'orquestrador comprova que el cos de la petició contingui `code`, `args` i `type`. Si falten aquests camps, retorna un error 400. En cas contrari, es genera un **`clientId` únic** i es registra el client al `clientRegistry`. A continuació, per a cada tasca definida en `args`, l'orquestrador genera un **`taskId`** i afegeix la tasca al registre dels clients, juntament amb el client. Finalment, s'envien les tasques als workers disponibles mitjançant la funció `dispatchTask()`.

Durant tot aquest procés, l'orquestrador escriu missatges de log per facilitar el

seguiment del registre de clients, *workers* i tasques, mostrant informació com el nombre de tasques assignades i els IDs generats. Tot això ha sigut molt necessari durant el procés de depuració.

Un cop aixecat el servidor HTTP, també es gestiona la **conversió a *WebSockets*** amb l'event upgrade. Això permet que tant clients com workers puguin establir una connexió persistent i bidireccional amb l'orquestrador, utilitzant el mateix identificador assignat prèviament. Aquesta connexió serveix per:

- Enviar tasques des de l'orquestrador als workers.
- Rebre resultats de tasques completades pels workers.
- Garantir una comunicació en temps real sense necessitat de tornar a cridar endpoints HTTP.

```
app.post("/register_worker", (req, res) => {
  const worker_id = uuidv4();
  const { numWorkers } = req.body;

  const newWorker = {
    worker_id,
    ws: null, // Will be set upon WebSocket connection
    maxWorkers: numWorkers || 1,
    availableWorkers: numWorkers || 1,
    tasksAssignated: [],
  };

  //workers.push(newWorker);
  //sortWorkers();
  workerRegistry.addWorker(newWorker);
  console.log(
    `👍 Worker registered with ID: ${worker_id} and ${newWorker.availableWorkers} processors.`
  );

  res.json({ message: "Worker registered successfully", worker_id });
});
```

Figura 10: Implementació de l'endpoint registerWorker a l'orquestrador.

4.2.2 Estructures de dades

Les estructures de dades que s'han usat per gestionar la informació son les següents:

- `ClientRegistry`
- `TaskQueue`
- `WorkerRegistry`
- `TasksMapReduce`

Explicarem una a una cadascuna d'aquestes estructures, les decisions de disseny i la implementació. En general, s'ha tractat d'aconseguir amb cada estructura una complexitat computacional mínima, arribant a O(1) en la majoria d'operacions importants.

`ClientRegistry`

Aquesta és l'estructura que s'encarrega de guardar la informació dels clients i les seves tasques asociades. Segueix el patró de disseny singleton, per tal de només tenir una instància d'aquest registre durant tota l'execució.

Els *hashmaps* són les estructures de dades idònies quan volem conseguir temps d'accés amb complexitat O(1), d'aquesta manera quan volem accedir a una tasca concreta d'un client ho podem fer ràpidament, sense tenir que iterar per tots els elements de l'estructura, lo que suposaria un cost lineal en funció del nombre de clients i tasques, i en un entorn real el nombre de clients i tasques per client poden ser molt elevades.

Tant els clients com les tasques associades de cada client són hashmaps. Les subtasques associades a cada tasca, en cas d'aquesta ser una tasca *MapReduce*, es guarden a una llista tradicional. Després veurem perquè s'ha decidit fer-ho així i quin impacte podria tenir al rendiment.

A la figura 11 podem veure els valors que es guarden al hashmap `tasks` de cada client. Cada valor d'aquest hashmap contindrà el nombre de tasques que ha demanat per executar el client, les pendents i el *websocket*, a més conté la informació de la tasca; arguments, tipus, codi, i per últim `numMappers` i `numReducers` en cas de tasques *MapReduce*.

```
addTask(clientId, task) {
  const client = this.clients.get(clientId);
  client.numTasks++;
  client.numPendingTasks++;
  client.tasks.set(task.taskId, {
    code: task.code,
    arg: task.arg,
    type: task.type,
    state: "pending",
    assignedWorkers: new Map(),
    subTasksResults: [],
    subTasks: [],
  });
  console.log(`Added task ${task} to client ${clientId}`);
  console.log(client.tasks);
}
```

Figura 11: Afegir tasca al registre de clients de l'orquestrador.

Gestió de subtasques

Com ja hem mencionat a l'inici d'aquest apartat, les subtasques de les tasques *MapReduce* es guarden usant un array.

En un moment determinat també es va valorar la possibilitat d'afegir cada subtasca amb el corresponent sufix (mapperX, reducerX) com una tasca normal al hashmap del client, però això dificultava el disseny i la recol·lecció dels resultats parcials, pel que es va acabar decidint fer-ho com es veu a la figura, que es una manera més centralitzada i neta de fer-ho.

En aquest cas `subTasks` és un array, per tant cada vegada que vulguéssim accedir a una determinada subtasca haurem de fer un recorregut lineal, com es pot veure a la següent figura a la instrucció de *return*.

```
getClientSubTask(clientId, taskId, subTaskId) {
  const client = this.clients.get(clientId);
  if (!client) return null;
  console.log(client);

  const mainTask = client.tasks.get(taskId);
  if (!mainTask) return null;
  console.log(mainTask);
  console.log(mainTask.subTasks);
  return (
    mainTask.subTasks.find((subTask) => subTask.taskId === subTaskId) || null
  );
}
```

Figura 12: Obtenir subtasca (MapReduce) a l'orquestrador.

El fet d'usar un array amb complexitat O(n) a les cerques podria suposar un problema. Aquesta funció es crida quan un worker es desconecta i reencuem una possible subtasca d'aquest worker. També es crida al processar una tasca de la cua, per obtenir la seva informació.

Es podria millorar simplement fent que en lloc d'un array també fos un hashmap, on la clau fos la clau de la subtasca (amb el sufix) i el valor la informació de la subtasca. De totes formes s'ha seguit amb la implementació actual per una qëestio de rapidesa i facilitat durant el desenvolupament, i perquè realment el nombre de subtasques no sol ser suficientment elevat com per a suposar un problema. Normalment el nombre de mappers o reducers en un sistema real no superaria de normal els milers o com a molt desenes de milers de subtasques, això ho podem veure a la investigació de Hespe et al. 2024 [15], on es menciona com el fet d'afegir molts de nodes de computació a una tasca mapreduce suposaria un augment considerable a la probabilitat de que algun d'aquests nodes falli, i en cas de fallada una recuperació en el context de mapreduce pot suposar un increment d'overhead i disminució del rendiment important, per lo que en la pràctica, tant per una qëestió de quantitat de dades (és difícil justificar la necessitat de milions de nodes) com per la complexitat afegida en cas de fallada, el nombre de nodes es sol limitar a desenes de milers en casos extrems.

Amb aquesta explicació hauriem vist les parts més importants de ClientRegistry. No explicarem totes les funcions (getters, setters, add, remove) ja que es basen en el mateix principi, consultar els hashmaps i retornar la informació trobada.

`TaskQueue`

Aquesta estructura guarda i gestiona l'accés a la cua de tasques pendents de despatxar.
L'estructura s'implementa com una cua dinàmica doblement enllaçada i segueix un patró singleton.
Tenim dos clases principals: `TaskNode` i `TaskQueue`.

`TaskQueue` consisteix en un hashmap que guarda punters directes als nodes de les tasques, on cada node consisteix del `taskId` i `clientId`. A més, també inclou els punters *head* i *tail* de la cua, que ens permeten obtenir les tasques de manera ordenada seguint la convenció *First Come First Serve* (FCFS). També inclou la variable entera *taskCount* que guarda el nombre de tasques encuades, això ens pot ser útil en cas de necessitar saber en algun moment quantes tasques hi ha encuades, tenir aquest contador ens permet no tenir que recórrer l'estructura cada vegada.
`TaskNode` guarda a cada node les "claus" (`clientId, taskId`) que s'usaran per obtenir la vertadera tasca del `clientRegistry`, així no repetim la informació. A banda tenim els punters next i prev, per mantenir l'estructura de cua.

El fet de que sigui doblement enllaçada facilita l'eliminació de qualsevol tasca amb complexitat O(1) a canvi de guardar un punter adicional a memòria per cada node. Realment al nostre sistema actual es crida a la funció `remove()` en alguns punts per assegurar que quan s'executin totes les tasques assignades d'un client, no quedi per casualitat alguna tasca a la cua, encara que això últim no hauria de passar mai.

A banda, aquesta doble cua s'ha usat principalment perquè a una versió anterior del

sistema, s'eliminaven totes les tasques del client en cas de desconnexió sense haver-se executat encara totes les tasques, però a la implementació final aquestes tasques no s'eliminem a no ser que ja s'hagin executat totes.

Quan volem eliminar una tasca, la busquem amb l'identificador al hashmap i simplement actualitzem els punters dels nodes anteriors i posteriors.

```
/** Remove a task by taskId in O(1) */
remove(taskIds) {
  const ids = Array.isArray(taskIds) ? taskIds : [taskIds];
  let removed = false;

  for (const taskId of ids) {
    const node = this.taskMap.get(taskId);
    if (!node) continue;

    if (node.prev) node.prev.next = node.next;
    else this.head = node.next;

    if (node.next) node.next.prev = node.prev;
    else this.tail = node.prev;

    this.taskMap.delete(taskId);
    this.taskCount--;
    removed = true;
  }

  return removed;
}
```

Figura 13: Funció remove(tasks) TaskQueue a l'orquestrador.

Seguint amb la justificació de l'estructura, és completament necessari que l'estructura sigui una cua, per assegurar un tractament First Come First Serve amb les tasques, i a més que sigui dinàmica, ja que no sabem prèviament quantes tasques podem arribar a tenir encuades.

El fet de que sigui dinàmica implica un augment en la memòria consumida, a causa dels punters, però augmenta el rendiment ja que no hem de crear nous arrays i copiar el contingut cada vegada que el plenèssim com passaria a una implementació estàtica.

```
class TaskNode {
  constructor(taskId, clientId) {
    this.taskId = taskId;
    this.clientId = clientId;
    this.next = null;
    this.prev = null;
  }
}

You, 4 weeks ago | 1 author (You)
class TaskQueue {
  constructor() {
    this.head = null;
    this.tail = null;
    this.taskMap = new Map(); // taskId → TaskNode
    this.taskCount = 0;
  }
```

Figura 14: Classes TaskNode i TaskQueue a l'orquestrador.

Cada vegada que encuem una tasca, creem un node i guardem les claus. A banda actualitzem els punters de la cua, per tal d'assegurar el comportament First Come First Serve. Aquest hashmap ens permet trobar el node en O(1), també podem actualitzar els punters prev i next gràcies a tenir la cua doblement encadenada i els punters "`head`" i "`tail`" ens permeten tractar l'estructura com una cua, obtenint sempre els nodes desde el principi al desencuar.

Per desencuar una tasca, obtenim el node al que apunta `head`, si és *null* vol dir que no tenim cap tasca encuada, sinó eliminem la tasca del hashmap i simplement retornem el `clientId` i `taskId` que estaven guardats al node, actualitzant els punters `head` i `tail` en conseqüència.

```
/** Remove and return the first task from the queue */
shift() {
  if (!this.head) return null;

  const node = this.head;
  this.taskMap.delete(node.taskId);

  this.head = node.next;
  if (this.head) this.head.prev = null;
  else this.tail = null;

  this.taskCount--;

  return { taskId: node.taskId, clientId: node.clientId };
}
```

Figura 15: Obtenció i eliminació de la pròxima tasca de la cua.

`TasksMapReduce`

Aquesta estructura és la més senzilla. És únicament un hashmap. També segueix el patró singleton. Més endavant veurem com usem aquesta estructura per gestionar les tasques mapreduce, per ara cal saber que la clau sempre serà l'identificador de la tasca (`taskId`) i el valor serà una rèplica de la tasca del `clientRegistry` amb alguns camps adicionals. És important que sigui un hashmap ja que així podem trobarla en O(1) quan l'orquestrador rep el resultat d'un *worker* (tot això es veurà més endavant).

`WorkerRegistry`

El `WorkerRegistry` manté un registre de tots els *workers* disponibles, els workers totals, i els workers ordenats per nivell de disponibilitat. En un futur es podria millorar la lògica per tal d'ordenar els workers de manera descendent, però en lloc de fer-ho només en funció del nombre de processadors disponibles també fer-ho respecte a altres factors com la latència de comunicació, potència del processador, nombre de tasques ja assignades, reputació, etc. Això ja ho fan alguns projectes com el **BOINC**.

```
class WorkerRegistry {
  constructor() {
    if (WorkerRegistry.instance) return WorkerRegistry.instance;
    this.workersById = new Map(); // worker_id => Worker object
    this.availabilityMap = new Map();
    this.numWorkers = 0; // Total number of workers
    this.totalAvailableWorkers = 0; // Total available workers across all buckets
    WorkerRegistry.instance = this;
  }
```

Figura 16: Classe i constructor WorkerRegistry a l'orquestrador.

Tenim dos *hashmaps* principalment, el `availabilityMap` i el `workersById`. `workersById` manté a l'estructura la informació de cada worker, com les tasques assignades, websocket, etc. On cada worker es troba indexat pel seu identificador worker_id, de manera que aquesta és la clau del hashmap. Aques hashmap ens permet accedir a la informació de cada worker en temps O(1).

`availabilityMap` és una estructura també implementada com un hashmap, on cada entrada consisteix en una clau que indica el nombre de procesadors disponibles a cadascun dels workers, i el valor és un altre hashmap de workers indexats pel seu identificador i un valor aleatori, booleà en aquest cas, ja que no volem repetir informació. Cadascun dels workers d'aquesta entrada de l'`availabilityMap` tenen la disponibilitat de processadors indicada per la clau. Aquesta estructura és útil per tal de poder assignar tasques als workers de manera ordenada, donant-li prioritat als workers amb un nombre de processadors majors. A més ens permet controlar la quantitat de processadors disponibles de cada worker de manera eficient, en un temps O(1). Si volem disminuir la disponibilitat d'un worker, simplement el movem d'una zona de disponibilitat a una altra, eliminant-lo de la primera i afegint-lo a la segona.

És rellevant notar que `availabilityMap` també es podria haver implementat com una llista, i cada vegada que es necessités un worker per assignar-li una tasca, recórrer aquesta llista i anar recol·lectant els workers necessaris. Un problema inherent a usar una llista en aquest cas és que per mantenir-la ordenada a l'hora d'afegir un nou worker tendriem un cost O(n) o O(log n).

Quan s'afegeix un nou worker a l'estructura, s'inicialitza el seu atribut *tasksAssignated* amb un nou hashmap. Aquest hashmap és el que ens permet poder marcar tasques del worker com a completades, pendents o error en temps O(1). A més a més, s'afegeix el worker amb aquesta informació al hashmap `workersById` i a la zona de disponibilitat pertinent de `availabilityMap`, indicant la clau com el nombre de processadors proporcionats pel worker.

```
addWorker(worker) {
  // worker should have: worker_id, availableWorkers, tasksAssignated
  worker.tasksAssignated = {
    map: new Map(),
    len: 0,
  };
  this.workersById.set(worker.worker_id, worker);
  this._addToAvailability(worker.worker_id, worker.availableWorkers);
  this.numWorkers++;
  this.totalAvailableWorkers += worker.availableWorkers;
}
```

Figura 17: Afegir worker al registre de workers de l'orquestrador.

Quan es vol assignar una tasca a un worker, simplement s'ha d'obtenir la informació del worker de l'estructura `workersById`, afegir la tasca al seu atribut *tasksAssignated* i moure el worker de la zona de disponibilitat actual a una inferior, indicant que té menys processadors disponibles. Notem que aquesta operació suposa un temps O(1).
En cas de voler marcar una tasca com a completada al worker, hem de fer el justament tot el contrari. Eliminem la tasca de `tasksAssignated` i el movem a una zona de disponibilitat major.

4.2.3 Gestió de la comunicació amb clients i workers

Comunicació amb els clients

La gestió de la comunicació amb els clients es troba al fitxer clientSocketHandler.mjs.

Aquest fitxer implementa la gestió de la connexió *WebSocket* dels **clients** amb l'orquestrador i l'enviament de tasques des de l'orquestrador cap al client.

Funció `sendTaskToClient(task, ws)`

Aquesta funció s'encarrega d'enviar el resultat d'una tasca individual a un client a través del WebSocket associat. Abans d'enviar, comprova que el WebSocket estigui obert (ws.readyState === ws.OPEN); si no és així, es mostra un **avís** i no s'envia la tasca.

El missatge que s'envia al client es construeix en format JSON i conté informació clau de la tasca:

- `message_type`: tipus de missatge (per distingir els diferents fluxos).
- `type`: tipus de tasca (ex. mapreduceterasort, mapreducewordcount).
- `taskId`: identificador únic de la tasca.
- `status`: estat actual de la tasca (error, completada, etc.).
- `result`: resultats de la tasca.
- `metadata`: informació addicional sobre la tasca (temps d'execució).
- En cas de tasques amb subtasques, també s'inclouen `numMappers` i `numReducers`.

Si l'enviament té èxit, la propietat *sent* de la tasca es marca com a true (a continuació

veurem per què això és necessari). En cas d'error durant l'enviament, es registra un **error** amb informació del problema.

```
function sendTaskToClient(task, ws) {
  if (!ws || ws.readyState !== ws.OPEN) {
    console.warn(
      `WebSocket is not open for client. Cannot send task ${task.taskId}`
    );
    return;
  }

  let message;
  if (task.subTasksResults.length > 0) {
    message = {
      message_type: task.message_type,
      type: task.type,
      numMappers: task.numMappers,
      numReducers: task.numReducers,
      taskId: task.taskId,
      status: task.status,
      result: task.results,
      metadata: task.metadata,
    };
  } else {
    message = {
      message_type: task.message_type,
      type: task.type,
      taskId: task.taskId,
      status: task.status,
      result: task.results,
      metadata: task.metadata,
    };
  }

  try {
    ws.send(JSON.stringify(message));
    task.sent = true;
    console.log(`✅ Task ${task.taskId} sent to client.`);
  } catch (error) {
    console.error(`❌ Failed to send task ${task.taskId}:`, error);
  }
}
```

Figura 18: Enviament del resultat d'una tasca al client.

Funció `handleClientSocket(ws, clientId)`

Aquesta funció s'executa quan un client estableix connexió amb l'orquestrador:

1. **Connexió del *WebSocket*:** es guarda la connexió en el registre de clients `clientRegistry` utilitzant el `clientId` proporcionat. Això ens serà útil per poder iniciar comunicacions amb el client posteriorment (per enviar els resultats de les tasques).

2. **Enviament de tasques pendents:** es recuperen totes les tasques associades al client i s'envien aquelles que encara no s'han marcat com a *sent*. Cada tasca es processa amb `sendTaskToClient(task, ws)`. Aquesta part té una

importància rellevant, ja que en un sistema real, desde que el client realitza la petició a l'endpoint "`register_task`" fins que es connecta al websocket de l'orquestrador proporcionant el *clientId* pot passar una estona, estona durant la qual s'ha pogut executar alguna tasca del client que s'ha tractat d'enviar abans de que aquest estigués connectat. Per tal de no perdre aquestes tasques s'usa aquest atribut *sent* de la tasca, així quan es connecti podrà rebre tots els resultats.

```
assignTaskToWorker(worker_id, clientId, taskId) {
  const worker = this.workersById.get(worker_id);
  if (!worker) return false;
  worker.tasksAssignated.map.set(taskId, clientId);
  worker.tasksAssignated.len++;
  this.updateAvailability(worker_id, worker.availableWorkers - 1);
  return true;
}
```

Figura 19: Assignació de tasca a un worker a l'orquestrador.

3. **Gestió de la desconnexió del client:** quan el client es desconnecta s'executa la funció `ws.on("close")`, amb la següent lògica de neteja:

    i. Si totes les tasques del client s'han executat, s'eliminen de la cua `taskQueue`, en cas de que no s'hagin eliminat correctament abans (cosa que en teoria no hauria de passar), i s'elimina el client del registre `clientRegistry`.

    ii. En cas que encara hi hagi tasques pendents, actualment la connexió es tanca però no s'eliminen les tasques pendents, la qual cosa permetria reenviar-les si el client es reconnecta, però aquesta funcionalitat no es troba implementada (ho veurem a l'apartat Limitacions i següents passos), encara que es podria implementar en un futur per assegurar una millor robustesa i facilitat d'ús. D'aquesta manera, les tasques es continuen executant i el resultat es podria consultar accedint al bucket de l'orquestrador. Com ja hem mencionat, la millor decissió de disseny seria proporcionar una nova funcionalitat per tal que el client, a l'iniciar l'execució, se pugués passar un nou argument que indiqués el seu `clientId` i així poder obtenir els resultats de les seves tasques.

```
export function handleClientSocket(ws, clientId) {
  console.log(`🔌 Client connected ${clientId}`);


  clientRegistry.getClient(clientId).ws = ws;

  const tasks = clientRegistry.getClientTasks(clientId);
  if (Array.isArray(tasks) && tasks.length > 0) {
    for (const task of tasks) {
      if (task.sent === false) {
        sendTaskToClient(task, ws); // Send task
      }
    }
  }

  ws.on("close", () => {
    console.log(`❌ Client disconnected ${clientId}`);
    if (clientRegistry.allTasksExecuted(clientId)) {
      const clientTasks = clientRegistry.getClientTasks(clientId);
      const taskIds = clientTasks.map((task) => task.taskId);
      taskQueue.remove(taskIds); // remove in case tasks are still pending
      clientRegistry.removeClient(clientId);
    }
  });
}
```

Figura 20: Gestió de la comunicació amb els clients a l'orquestrador.

Comunicació amb els workers

El mòdul `workerSocketHandler.mjs` gestiona la connexió *WebSocket* dels ***workers*** amb l'orquestrador i la recepció de missatges amb els resultats de les tasques que executen.

Funció `handleWorkerSocket(ws, workerId)`

Aquesta funció s'executa quan un worker es connecta a l'orquestrador:

1. **Connexió del *worker*:**

   i. Es recupera el *worker* del registre `workerRegistry` a partir del *workerId*. Recordem que el worker s'ha registrat quan ha executat la crida al endpoint *register_worker*, ara s'està connectant amb el *workerId* que se li havia proporcionat com a resposta.

   ii. Es guarda la connexió *WebSocket* (`worker.ws = ws`) actual, per tal de poder accedir posteriorment quan necessitem enviar-li tasques.

   iii. Es crida `processTaskQueue()`. Cada vegada que un nou worker es connecta cridem a aquesta funció, ja que un nou *worker* disponible significa que podem executar alguna tasca que tinguéssim encuada.

2. **Gestió de desconnexió (`ws.on("close")`):**

   i. Quan un *worker* es desconnecta, primer es comprova si tenia tasques assignades.

   ii. Les tasques pendents o incompletes es reassignen a la cua `taskQueue` per tal que altres workers puguin executar-les.

   iii. Es diferencien les subtasques (*mappers* o *reducers*) de la tasca principal amb una expressió regular. D'aquesta manera, podem encuar tant subtasques (*mappers* o *reducers*) i també tasques completes.

   iv. Finalment, el *worker* s'elimina del registre amb la funció `workerRegistry.removeWorker(workerId)`.

   Aquest apart és molt important, ja que si no tornéssim a afegir les tasques que tenia assignades el *worker* a la cua, l'orquestrador no tendria manera de finalitzar l'execució de la tasca demanada pel client i el client es quedaria esperant per sempre.

3. **Recepció de missatges del worker (`ws.on("message")`):**

En aquesta part de la implementació trobem gran part de la lògica que tracta els resultats de les tasques i les depèndencies de dades en tasques *MapReduce*. Tota aquesta lògica està repartida i explicada durant els següents apartats.

i. Cada missatge rebut des de qualsevol worker s'analitza com a JSON. Si és invàlid, es registra un error.

ii. Es marca la tasca corresponent com a completada al worker `workerRegistry.completeTaskOnWorker(worker_id, taskId)`. Això actualitza la disponibilitat del worker per tal de poder assignar-li futures tasques.

iii. Es neteja el `taskId` eliminant possibles sufixos de subtasques per obtenir la tasca principal. Cada tasca *mapper* o *reducer* afegeix un sufix mapperX o reducerX al `taskId` generat per l'orquestrador durant el registre del client i la creació de cada identificador de tasca.

iv. Es recupera la tasca del client `clientRegistry.getClientTask(clientId, taskId)` i es comprova que no estigui ja marcada com a *done* o *error*. Això és crucial perquè

si un *mapper* o *reducer* falla, el sistema no recrea la tasca, assumint que l'error es deu a factors com codi incorrecte, bucket inexistente o permisos insuficients. Quan això passa, l'orquestrador elimina la tasca MapReduce de l'estructura de tasques i envia al client els resultats de les subtasques que s'hagin completat. Després, marca la tasca com a error, continua amb les següents tasques i elimina el client si era l'última. Si qualsevol altre *mapper* envia un resultat per a aquesta mateixa tasca, es detecta que ja està marcada com a error i s'ignora el missatge. Aquesta comprovació simple evita que una tasca fallida sigui tractada com a tasca normal del client, cosa que podria provocar la finalització prematura de la connexió abans que s'executin totes les tasques reals.

```
if (clientTask.state === "done" || clientTask.state === "error") {
  return;
}

let infoTask = mapreduceTasks.get(cleanedTaskId);

if (infoTask && msg.status === "error") {
  mapreduceTasks.delete(cleanedTaskId); // Mapreduce task failed if any mapper or reducer fails
  const metadata = {
    [msg.taskId]: [
      parseFloat(msg.initTime) || 0,
      parseFloat(msg.readTime) || 0,
      parseFloat(msg.cpuTime) || 0,
      parseFloat(msg.writeTime) || 0,
      parseFloat(msg.endTime) || 0,
      msg.workerId || 0,
    ],
  };
  clientTask.subTasksResults.push(metadata);
}
```

Figura 21: Comprovació d'error de subtasca a l'orquestrador.

4. Gestió de tasques MapReduce:

Per tal de gestionar les tasques *MapReduce*, tenim un hashmap anomenat `mapreduceTasks` que usa com a clau la `taskId` i com a valor la informació de la tasca, com podem veure a la figura següent.

```
mapreduceTasks.set(task.taskId, {
  numMappers: task.numMappers,
  numReducers: task.numReducers,
  codeReduce: reducerCode,
  type: task.type,
  resultsMappers: [],
  resultsReducers: [],
});
```

Figura 22: Creació d'una tasca MapReduce a l'orquestrador.

Aquesta tasca es crea i es guarda a l'estructura `mapreduceTasks` durant la creació dels mappers, i després s'accedirà i modificarà durant la recepció dels resultats dels mappers (`resultsMappers`) per tal d'enviar els arguments als reducers i guardar els resultats dels reducers (`resultsReducers`).

Si la tasca és una *MapReduce*, es fa seguiment del nombre de *mappers* i *reducers*. Com podem veure a la següent figura, fixem-nos que tornem a obtenir la tasca de l'estructura `mapeduceTasks`, això ho fem cada vegada que al codi anterior s'hagi pogut eliminar el job (en aquest cas en cas d'error d'una de les subtasques) o simplement no sigue una tasca mapreduce. A més, podem veure com s'obté la metadata de la tasca (temps d'execució), s'afegeix el resultat al camp `resultsMappers` i guardem la metadata al client.

```
infoTask = mapreduceTasks.get(cleanedTaskId);

if (infoTask && infoTask.numMappers > 0) {
  infoTask.numMappers--;
  const metadata = {
    [msg.taskId]: [
      parseFloat(msg.initTime) || 0,
      parseFloat(msg.readTime) || 0,
      parseFloat(msg.cpuTime) || 0,
      parseFloat(msg.writeTime) || 0,
      parseFloat(msg.endTime) || 0,
    ],
  };
  infoTask.resultsMappers.push(msg.result);
  clientTask.subTasksResults.push(metadata);
  console.log(
    `📦 Worker ${workerId} completed a mapper for task ${cleanedTaskId}. Remaining: ${infoTask.numMappers}`
  );
}
```

Figura 23: Codi per l'execució de cada mapper.

Quan tots els *mappers* s'han completat, es crea automàticament la fase de reduce, cridem a la funció `dispatchTask(task)` que s'encarrega de crear els corresponents *reducers* i assignar-los als workers disponibles, o encuar-los fins que es puguin executar. El nombre de *mappers* s'assigna a –1, per poder detectar posteriorment si estem a una fase de reduce per executar el codi de gestió de resultats dels *reducers* (també podriem tenir un camp anomenat *isReducePhase* però no és necessari). El tipus serà *reducewordcount* o *reduceterasort*, depen de quin tipus de tasca MapReduce estiguéssim executant. Si hi ha un error durant el despatx, eliminem la tasca, això requerirà que de nou es torni a obtenir la tasca de l'estructura *mapreduceTasks* abans de continuar amb la següent part del codi, així es pot saber què ha passat abans i continuar en conseqüència.

```
if (infoTask && infoTask.numMappers === 0) {
  infoTask.numMappers = -1;
  const type =
    infoTask.type === "mapwordcount" ? "reducewordcount" : "reduceterasort";
  const reduceTask = {
    code: infoTask.codeReduce,
    arg: infoTask.resultsMappers,
    taskId: cleanedTaskId,
    clientId: msg.clientId,
    type,
    numReducers: infoTask.numReducers,
    numMappers: infoTask.numMappers,
  };
  console.log(
    `🔄 Map stage completed for ${cleanedTaskId}. Starting reduce phase.`
  );
  const dispatched = dispatchTask(reduceTask);
  if (!dispatched) {
    mapreduceTasks.delete(cleanedTaskId);
    msg.status = "error";
    msg.result = `Failed to dispatch reduce task for ${cleanedTaskId}. Task removed.`;
    console.error(
      `❌ Failed to dispatch reduce task for ${cleanedTaskId}. Task removed.`
    );
  } else {
    return; // Exit early if reduce task was dispatched successfully
  }
}
```

Figura 24: Finalització de la fase Map i preparació de la fase Reduce a l'orquestrador.

Seguiment de la fase de *Reduce*. De la mateixa manera que abans anavem obtenint i guardant els resultats dels *mappers*, ara fem el mateix per als *reducers* durant la fase de *Reduce*. La diferència és que quan ja hem executat tots els *reducers* guardem els resultats a *l'array* `results` i eliminem la tasca de l'estructura *mapreduceTasks*.

```
infoTask = mapreduceTasks.get(cleanedTaskId);
let results = [];
if (infoTask && infoTask.numMappers === -1) {
  infoTask.numReducers--;
  const metadata = {
    [msg.taskId]: [
      parseFloat(msg.initTime) || 0,
      parseFloat(msg.readTime) || 0,
      parseFloat(msg.cpuTime) || 0,
      parseFloat(msg.writeTime) || 0,
      parseFloat(msg.endTime) || 0,
    ],
  };
  infoTask.resultsReducers.push(msg.result);
  clientTask.subTasksResults.push(metadata);
  if (infoTask.numReducers === 0) {
    results = infoTask.resultsReducers;
    mapreduceTasks.delete(cleanedTaskId);
    console.log(`✅ All reducers for task ${cleanedTaskId} completed.`);
  } else {
    console.log(
      `📦 Reducer completed for ${cleanedTaskId}. Remaining: ${infoTask.numReducers}`
    );
  }
}
```

Figura 25: Codi per l'execució de cada reducer.

Resultat final. Una vegada hem passat per totes aquestes fases i comprovacions poden passar dos coses. O que la tasca en cap moment hagi format part d'un job *Mapreduce*,

en aquest cas simplement no s'hauria entrat a cap d'aquests blocs de codi. L'altre cas és que sí que hagi sigut una tasca *MapReduce*, però no hagi sigut l'última, per lo que no s'enviarà res al client encara. En cas de ser una subtasca *MapReduce* i ser l'última, o ser una tasca normal, s'executarà el codi del següent apartat.

5. **Enviament de resultats al client:**

En cas de que la tasca sigui un job *MapReduce* finalitzat, retornarem la metadata que hem anat guardant a l'atribut de la tasca del client `clientTask.subtasksResults` per cada *mapper* i *reducer* executat, aquesta metadata es crea usant la funció *reduce* de javascript a l'array, que ens permet juntar tots els objectes JSON en un mateix. Aquesta mateixa metadata és la que usa el client per realitzar l'anàlisi offline.

En cas de tasca normal, enviem la metadata que ens ha enviat el worker i no enviem nombre de mappers ni reducers.

Enviem el resultat al client, establim *sent* a true i si ja se han executat totes les tasques, eliminem el client i la connexió.

En cas d'error perquè el websocket no es trobava disponible (el client encara no se ha connectat), establim *sent* a false (abans ja hem explicat perquè això és necessari).

```
  if (client?.ws) {
    client.ws.send(JSON.stringify(jsonToSend));
    console.log(`📦 Before sending: numTasks = ${client.numTasks}`);      You, 3 weeks ago • execution TIME OK …
    clientRegistry.setClientTask(msg.clientId, cleanedTaskId, {
      ...clientTask,
      ...jsonToSend,
      sent: true,
    });
    console.log(
      `📦 Sent result for task ${cleanedTaskId} to client ${msg.clientId}. Remaining tasks: ${client.numTasks}`
    );
    if (clientRegistry.allTasksExecuted(msg.clientId)) {
      const clientTasks = clientRegistry.getClientTasks(msg.clientId);
      const taskIds = clientTasks.map((task) => task.taskId);
      taskQueue.remove(taskIds); // remove in case tasks are still pending
      clientRegistry.removeClient(msg.clientId);

      console.log(`✅ All tasks for client ${msg.clientId} completed.`);
      console.log(`🗑️ Client ${msg.clientId} removed from registry.`);

      client.ws.close();
    }
  } else {
    console.error(
      `❌ Client ${msg.clientId} for task ${msg.taskId} not found or disconnected.`
    );
    clientRegistry.setClientTask(msg.clientId, cleanedTaskId, {
      ...clientTask,
      ...jsonToSend,
      sent: false,
    });
  }
}
```

Figura 26: Enviament del resultat d'una tasca al client per part de l'orquestrador.

6. **Reprocessament de la cua de tasques:**

Finalment, la recepció d'un missatge d'un *worker* implica obtenir el resultat de l'execució d'una tasca. En aquest moment, el sistema disposa d'un *worker* addicional i, per tant, es torna a invocar la funció `processTaskQueue()` amb l'objectiu d'assignar alguna de les tasques pendents al *worker* que ha quedat lliure.

Fins ara, s'ha vist que la funció `processTaskQueue()` s'ha invocat tant en el moment de registrar un nou *worker* com en rebre el resultat d'un d'ells. En canvi, no s'invoca en registrar un nou client, ja que en aquest cas es crida directament a `dispatchTask()`, i, si no és possible dur a terme el despatx de la tasca, aquesta queda automàticament encuada.

4.2.4 Assignació i gestió de tasques

En aquest apartat veurem les funcions implementades per tal de gestionar la creació i assignació de tasques als *workers*, també veurem el funcionament de la cua i com es processa.

Funció `processTaskQueue()`

El procés consisteix en un bucle que s'executa mentre hi hagi tasques pendents a la cua i existeixi algun *worker* disponible. Inicialment, s'invoca la funció `peek()` sobre la cua, que retorna la pròxima tasca sense eliminar-la. En aquest context, tal com s'explicarà més endavant, també seria possible utilitzar directament la funció `shift()`, atès que la tasca s'eliminarà de la cua tant en cas d'error (tasca inexistent) com en cas que es despatxi correctament.

Per identificar si la tasca és una subtasca, s'utilitza una expressió regular que comprova si l'identificador conté el sufix *mapper* seguit d'un dígit obligatori (operador `+`) o bé *reducer* seguit d'un dígit opcional (operador `*`).

Un cop realitzada la comprovació, la informació detallada de la tasca s'obté del `clientRegistry`. Aquesta decisió s'ha pres per evitar redundància en la cua: en lloc d'emmagatzemar tota la informació de la tasca, la cua només conté l'identificador del client i de la tasca a mode de clau. Mitjançant aquests identificadors, és possible consultar de manera eficient el `clientRegistry` i recuperar la informació necessària (arguments, codi, tipus, nombre de *mappers* i *reducers* en el cas de tasques *mapreduce*). L'eficiència d'aquesta consulta és crítica, ja que una implementació amb complexitat O(n), on *n* representa el nombre de clients o tasques, obligaria a recórrer tota l'estructura per cada element de la cua, incrementant el temps de computació de l'orquestrador i reduint el rendiment global del sistema.

Finalment, la tasca es despatxa amb la informació obtinguda o bé es cancel·la en cas que no es localitzi al registre. Aquesta situació no es produeix en la implementació actual, ja que l’eliminació d’un client del registre (i, per extensió, de totes les seves tasques) només té lloc quan ha finalitzat l’execució completa. Tanmateix, en versions anteriors del sistema, el client s’eliminava també en cas de desconnexió, encara que les seves tasques no haguessin estat completades.

Funció `dispatchTask(task)`

El funcionament de la funció `dispatchTask` es pot descriure de la manera següent.

En primer lloc, es comprova si la tasca correspon a un patró *mapreduce*. En cas afirmatiu, s'invoca la funció `dispatchMappers()`, la qual té la responsabilitat de crear els diferents *mappers* i, posteriorment, despatxar-los o bé emmagatzemar-los a la cua.

A continuació, es verifica si la tasca és una subtasca. En aquest cas, s'intenta directament el seu despatx. Aquesta comprovació es realitza mitjançant una expressió regular, tal com ja s'ha descrit en seccions anteriors.

Si la tasca correspon a la fase inicial del procés de *reduce,* creada al `workerSocketHandler` quan tots els *mappers* han finalitzat la seva execució,, s'executa la funció `dispatchReducers()`. Aquesta funció presenta un funcionament similar a `dispatchMappers()`, però amb algunes diferències específiques que justifiquen la seva implementació separada.

Finalment, es procedeix al despatx de la tasca, ja sigui una subtasca o una tasca ordinària. Per fer-ho, s'obté el primer *worker* disponible, tenint en compte que el `workerRegistry` es troba ordenat en funció del nombre de processadors disponibles per cada *worker.* En la implementació actual, cada *worker* disposa d'un únic processador i d'un procés *node* en execució; no obstant això, en futures ampliacions es podria permetre la gestió de *n* processadors i *n* processos simultanis.
Un cop seleccionat el *worker,* s'utilitza exclusivament el primer element de l'array `code` (tal com s'ha exposat anteriorment, qualsevol altre element es descarta). A continuació, es reserva el *worker* i se li envia la tasca mitjançant el *websocket*, utilitzant la funció `reserveWorkerAndSendTask()`, representada a la figura següent.

```
/**
 * Assigns a task to a worker and sends it via WebSocket.
 */
function reserveWorkerAndSendTask(worker, task) {
  worker.ws.send(JSON.stringify(task), (err) => {
    if (err) {
      console.error(
        `❌ Failed to send task to ${worker.worker_id}:`,
        err.message
      );
      throw new Error(
        `Failed to send task to worker ${worker.worker_id}. Error: ${err.message}`
      );
    } else {
      workerRegistry.assignTaskToWorker(
        worker.worker_id,
        task.clientId,
        task.taskId
      );
      clientRegistry.markTaskRunning(
        task.clientId,
        task.taskId,
        worker.worker_id
      );
      console.log(
        `✅ Sent task ${task.taskId} from client ${task.clientId} with arg ${task.arg} to worker ${worker.worker_id}`
      );
    }
  });
}
```
Figura 27: Funció per reservar i enviar tasca al worker.

Controlem qualsevol possible error (vegeu la figura anterior) i, en cas d'èxit, assignem la tasca al *worker*. Això implica guardar el *clientId* i el *taskId* associats al *worker*, per tal de poder reenquar les seves tasques en cas de desconnexió o bé mantenir un registre detallat de quines tasques executa cada *worker* en cada moment. Paral·lelament, marquem la tasca al *clientRegistry* amb l'estat "running".

Actualment, l'estat de la tasca s'utilitza sobretot per marcar-la com a "done" o "error". Com hem vist abans, això resulta útil en els *jobs* de tipus *mapreduce*, ja que permet saber si una tasca ja s'ha executat o ha acabat amb error. Tanmateix, aquest estat també seria valuós si volguéssim informar periòdicament al client del progrés de les seves tasques mitjançant *polling*.

Despatx de mappers

En primer lloc, es detecta si la tasca és de tipus *mapreduce*. En el cas concret de les tasques de tipus *wordcount*, el nombre de *reducers* es fixa sempre a 1. El sistema no ofereix suport per a un nombre major de *reducers*, ja que això requeriria implementar una lògica de partició i *shuffling* similar a la utilitzada a les tasques de tipus *terasort*, però que no està disponible en aquest cas. Aquesta limitació simplifica el processament del *wordcount* però restringeix la seva flexibilitat.

A continuació, es procedeix a la creació de la tasca *mapreduce*. Un bucle iteratiu, que es repeteix tantes vegades com el nombre de *mappers* especificat pel client, genera cada subtasca *mapper.* Cada una d'aquestes subtaskes incorpora un sufix identificador i un índex corresponent al *worker* assignat. Aquesta informació, juntament amb el nombre total de *mappers*, és utilitzada pel *worker* per dur a terme una lectura parcial de les dades d'entrada. Gràcies a aquest mecanisme, el sistema reparteix la càrrega d'entrada de manera distribuïda. L'anàlisi detallada d'aquesta interacció amb l'*input* es descriu posteriorment a l'apartat dedicat als *workers* (Worker).

Finalment, en cas que no hi hagi *workers* disponibles per executar una subtasca en el moment de la seva creació, aquesta s'incorpora a la cua de tasques pendents. Aquesta estratègia permet despatxar immediatament totes les subtaskes possibles amb els *workers* lliures i diferir l'execució de la resta fins que quedin recursos disponibles. D'aquesta manera, el sistema garanteix un repartiment eficient i dinàmic de les subtaskes, sense necessitat de disposar simultàniament de tots els *workers* requerits pel conjunt de *mappers*.

Despatx de reducers

La lògica principal d'aquesta funció és similar a la utilitzada en el despatx dels *mappers*. No obstant això, cal analitzar amb deteniment el tractament dels arguments d'entrada, ja que presenten diferències significatives respecte a la fase *Map*.

En aquest cas, la tasca que defineix la fase *Reduce* conté una matriu com a argument d'entrada, en contraposició a l'*array* emprat a la fase *Map*. Cada fila de la matriu correspon al resultat produït per un dels *mappers*. A més, cada fila està formada per "numReducers" columnes, de manera que cada *mapper* genera un resultat específic per a cadascun dels *reducers*. Aquests resultats incorporen el sufix *reducer-X*, on *X* identifica el *worker* destinatari. En conseqüència, un *reducer* disposa de tants arguments d'entrada com *numMappers*, mentre que cada *mapper* produeix tants resultats com *numReducers*. En conjunt, la matriu conté "numMappers × numReducers" elements.

La lògica del bucle intern de la funció té com a objectiu comprovar que cada element de l'entrada és efectivament un *array*, confirmant així que es tracta d'una matriu. En cas contrari, es retorna un error. Quan es disposa d'una matriu vàlida, el *reducer* recorre cadascuna de les files i selecciona l'element que li correspon. Amb els arguments seleccionats (que ara constitueixen un *array*), es crea la tasca per al *reducer*, que posteriorment es reserva i s'envia al *worker* assignat o bé s'incorpora a la cua en cas de no haver-hi recursos disponibles.

Amb aquesta descripció es conclou l’apartat tècnic de l’orquestrador, en el qual s’han analitzat els mòduls que el componen i la lògica que permet assolir un sistema funcional i eficient.

4.3 Worker

A continuació veurem els diferents mòduls que conformen el worker.

4.3.1 Main

El `main` executa les següents tasques en ordre (com es mostra a la figura següent): obté els arguments (funcions del fitxer *utils.mjs*), inicialitza l'entorn de Pyodide, i executa la funció de registre del *worker* a l'orquestrador, quedant a l'espera de noves tasques a executar.
La sintaxi emprada permet que les instruccions siguin asíncrones (no consumeixen temps actiu de CPU), però garanteix que cada tasca no s'executi fins que l'anterior hagi finalitzat.

```
(async () => {
  try {
    obtainArgs(CONFIG);
    await initPyodide();
    await registerWorker();
  } catch (e) {
    console.error("❌ Fatal error:", e.message);
    process.exit(1);
  }
})();
```
Figura 28: Codi main del worker.

4.3.2 PyodideRuntime

Aquesta part és important, ja que és on es defineix l'entorn de Pyodide, crucial per tal de poder executar correctament les tasques Python que rep el *worker*. Per tal que Pyodide funcioni correctament, és necessari haver instal·lat la llibreria Pyodide mitjançant el gestor de paquets *npm*. D'aquesta manera, els arxius corresponents es troben disponibles al directori de treball (*node_modules*).

Concretament, si ens dirigim al directori *node_modules/pyodide* després d'haver executat la comanda `npm install -y`, hi trobem un fitxer anomenat **pyodide.asm.wasm**, que conté l'intèrpret de Python compilat a WebAssembly. També hi ha el fitxer **pyodide.js**, responsable de carregar l'intèrpret a la memòria del navegador, inicialitzar-lo i proporcionar una interfície per poder interactuar amb aquest entorn des de JavaScript.

A continuació, veurem com inicialitzem l'entorn.

```
export async function initPyodide() {
  console.log("⏳ Loading Pyodide ... ");
  pyodideInstance = await loadPyodide();
  await pyodideInstance.loadPackage("pandas");
  await pyodideInstance.runPythonAsync(`
# pyedgecompute.py
import pickle, base64, json
import pandas as pd
import numpy as np


# CODE MAPREDUCE
def serialize_partition(result):
    raw = pickle.dumps(result)
    return base64.b64encode(raw).decode('utf-8')

def deserialize_partitions(b64_list):
    parts = []       You, 4 weeks ago • refactoring wo
    for b64 in b64_list:
        raw_bytes = base64.b64decode(b64)
        part = pickle.loads(raw_bytes)
        parts.append(part)
    return parts

def deserialize_input_string(bytes_string):
    string_data = bytes_string.decode('utf-8')
    return string_data
```

Figura 29: Càrrega i inicialització del runtime de Pyodide al worker.

Primerament, es carrega el fitxer *pyodide.js*, que s'encarrega d'inicialitzar l'intèrpret de Python i allotjar-lo a la memòria del navegador. A continuació, es carreguen els paquets necessaris, com ara *pandas* i *NumPy*. En cas que no estiguin prèviament disponibles, aquests es descarreguen des d'internet i s'emmagatzemen en el directori virtual `node_modules/pyodide/` en format *.whl*.

Un cop l'entorn i els paquets han estat carregats, s'executa un bloc de codi que defineix un conjunt de funcions que els clients poden utilitzar a l'hora de definir les seves tasques dins del framework. Entre aquestes funcions hi trobem les mostrades a la figura anterior, així com altres relacionades amb les operacions *terasort*.

Finalment, aquestes funcions s'injecten a l'entorn mitjançant els mòduls de Pyodide, de manera que quedin disponibles per ser importades i utilitzades per les tasques dels clients.

```
# Inject module
import types
pyedgecompute = types.ModuleType("pyedgecompute")
pyedgecompute.serialize_partition = serialize_partition
pyedgecompute.deserialize_partitions = deserialize_partitions
pyedgecompute.deserialize_input_string = deserialize_input_string
pyedgecompute.deserialize_input_terasort = deserialize_input_terasort
pyedgecompute.partition_data = partition_data
pyedgecompute.concat_partitions = concat_partitions
pyedgecompute.sort_dataframe = sort_dataframe


import sys
sys.modules["pyedgecompute"] = pyedgecompute
`);
  console.log("✅ Pyodide ready");
  return pyodideInstance;
}
```

Figura 30: Injecció dels mòduls de Python a l'entorn d'execució de Pyodide.

4.3.3 TasksExecutor

Tota la lògica que gestiona la lectura de les dades d'entrada, l'escriptura de les dades de sortida i l'execució de la tasca es gestiona a aquest mòdul, que el podem trobar al fitxer tasksExecutor.mjs.

Durant l'execució de la tasca, pot ser que el bucket d'entrada sigui en local (MinIO) o S3.

A partir d'aquesta part del codi, dependent de si "isS3" és vertader o fals, usarem les funcions definides per la llibreria MinIO o les definides per la llibreria de AWS S3. La funcionalitat és la mateixa, però canvien certes crides i la inicialització del client.

```
const isS3 = (url) => {
  if (Array.isArray(url)) {
    return url.every((u) => typeof u === "string" && u.startsWith("s3://"));
  }
  return typeof url === "string" && url.startsWith("s3://");
};
```

Figura 31: Comprovació bucket local o S3 al worker.

4.3.4 Serialització/Deserialització de resultats

En aquest subapartat del *TasksExecutor* es descriu el procés de serialització i deserialització tant de les dades d'entrada com de les de sortida. A la figura següent es mostra l'obtenció de les dades d'entrada segons el tipus de tasca: si es tracta d'un *mapper*, s'obté l'entrada parcial; si és un *reducer*, s'obté l'entrada serialitzada, corresponent al resultat dels *mappers*; i, en el cas d'una tasca normal, es crida la funció `getText*()`, que equival a cridar `getPartialObject*()` quan l'objecte parcial coincideix amb l'objecte complet. Cal destacar que `getPartialObject*()` sempre acaba cridant a `getText*()`, establint un *offset* i la quantitat de bytes a obtenir.

Un cop obtingudes les dades, aquestes són processades segons el tipus de tasca. Els *mappers* i les tasques normals sempre reben com a entrada un fitxer de text. Per evitar problemes amb caràcters especials o comes que podrien alterar l'execució del codi Python, el contingut de la variable `bytes` es codifica en *Base64* i s'introdueix al codi Python final com a descodificació de *Base64*.

Els *mappers* escriuen els seus resultats serialitzant-los amb la llibreria `pickle` de Python i codificant-los també en *Base64*. Els *reducers* reben els resultats dels *mappers* codificats en *Base64*. Per interpretar correctament els *bytes* llegits com a *string*, s'utilitza `JSON.stringify` en el cas dels *reducers*.

Un cop realitzades aquestes comprovacions, es defineix el codi Python final que s'executarà en l'entorn. En el cas específic d'una tasca *mapteraserot*, cal definir la variable `num_partitions` a Python, que s'introdueix com a entrada en una de les funcions predefinides de l'entorn Pyodide. Aquesta variable determina el nombre de particions de sortida, que depèn del nombre de *reducers* definits per a la tasca, de manera que cada *mapper* crea tantes particions com *numReducers*. Posteriorment, s'acaba generant la matriu d'arguments que l'orquestrador utilitza per crear les tasques *reducers*.

Després de totes aquestes comprovacions, es pot definir el codi final i executar-lo a l'entorn de Pyodide (veieu següent figura).

```
    let rawBytesLine;
    if (task.type === "reduceterasort" || task.type === "reducewordcount") {
      rawBytesLine = `raw_bytes = ${JSON.stringify(bytes)}`;
    } else {
      const b64 = bytes.toString("base64");
      rawBytesLine = `raw_bytes = base64.b64decode("${b64}")`;
    }

    let extraPythonVars = "";
    if (task.type === "mapterasort") {
      console.log("MAPTERASORT");
      console.log("NUM_PARTITIONS: ", task.numReducers);
      extraPythonVars = `num_partitions = ${task.numReducers}`;
    }

    const pyScript = `
${task.code}
${rawBytesLine}
${extraPythonVars}
try:
    result = task(raw_bytes)
except Exception as e:
    result = str(e)
result
    `;

    let result = await pyodide.runPythonAsync(pyScript);
```

Figura 32: Creació i execució del codi final usant Pyodide al worker.

Per últim, es guarden els resultats i es retorna el resultat a l'orquestrador amb la informació adient i control d'errors pertinent.

És important notar que el que s'acaba retornant al client no és el resultat com a tal, sinó una URL que apunta al bucket i fitxer on es troba emmagatzemat el resultat final.

4.3.5 Comunicació amb l'orquestrador

En aquest apartat es descriu la comunicació entre el *worker* i l'orquestrador, incloent tant la petició inicial de registre del *worker* com la comunicació posterior a través de *websockets*, destinada a la recepció de tasques i l'enviament de resultats a l'orquestrador.

No s'analitzarà aquesta part en profunditat, sinó que es remarcarà que el procediment és anàleg al dels clients: primer, el registre inicial es realitza mitjançant una petició a l'endpoint de l'API HTTP RESTful de l'orquestrador; si la petició té èxit, es retorna l'identificador del *worker*; posteriorment, aquest identificador s'utilitza per establir la connexió amb el servidor *websocket*.

A la figura següent es mostra com es rep cada tasca, amb control d'errors, i com s'inicia la seva execució.

```javascript
ws.on("message", async (msg) => {
  try {
    const task = JSON.parse(msg.toString());
    const { clientId, taskId, code, arg, type } = task;

    if (!clientId || !taskId || !code || !arg || !type) {
      console.error(
        "❌ Invalid task received from ORCHESTRATOR. Missing clientId, taskID, code, arg or type."
      );
      ws.send(
        JSON.stringify({
          clientId,
          taskId,
          status: "error",
          result:
            "Invalid task received from ORCHESTRATOR. Missing clientId, taskID, code, arg or type.",
        })
      );
      return;
    }
    console.log(
      `▶️ Worker ${workerId} received task ${taskId} from client ${clientId} with arg ${arg}`
    );
    await executeTask(task, ws, stopWatch); // ⟵ Execute the task received from the orchestrator
  } catch (err) {
    console.error("❌ Error parsing message from ORCHESTRATOR:", err.message);
  }
});
```

Figura 33: Recepció i execució d'una tasca al worker.

4.3.6 Interacció amb el Storage

En aquest apartat es descriu la interacció del *worker* amb l'entrada i sortida de dades, és a dir, els processos de lectura i escriptura.

Les funcions i mòduls implicats es troben definits a `awsS3Client.mjs`, `awsS3Storage.mjs`, `minioClient.mjs` i `minioStorage.mjs`. Aquests proporcionen les funcionalitats necessàries per guardar els resultats serialitzats, obtenir els *inputs*, els *inputs* parcials i els *inputs* serialitzats. No s'analitzarà en detall el funcionament intern d'aquestes funcions, ja que es limiten a interactuar amb les API corresponents i no impliquen decisions de disseny rellevants.

Es focalitza, en canvi, en el càlcul necessari per obtenir els *inputs* parcials i com els *reducers* recuperen els resultats dels *mappers*.

Per tal d'obtenir un *input* parcial, és necessari conèixer el número assignat al *worker* dins del *stage* Map (offset) i el nombre de *mappers* (`numMappers`). Si aquests paràmetres no es proporcionen, s'executa una crida normal a `getObject()`. Quan s'indiquen, es calcula l'*offset* corresponent: primer s'obté la mida total de l'objecte en bytes, es divideix pel nombre de *mappers* i, utilitzant l'offset i el quocient resultant, s'extrauen els bytes corresponents per a cada *mapper*. El codi corresponent es mostra a la figura següent.

```
const totalSize = metadata.ContentLength;
const chunkSize = Math.floor(totalSize / numMappers);
const start = offset * chunkSize;
let end = (offset + 1) * chunkSize - 1;
if (offset === numMappers - 1) end = totalSize - 1;

const range = `bytes=${start}-${end}`;
const command = new GetObjectCommand({
  Bucket: bucket,
  Key: objectName,
  Range: range,
});
const response = await client.send(command);
return streamToBuffer(response.Body);
```

Figura 34: Càlcul d'offset i obtenció d'objecte parcial per part d'un mapper.

Per garantir que els *reducers* obtinguin correctament els resultats dels *mappers*, cal tenir present el format de les dades rebudes. Com s'ha esmentat anteriorment, aquestes dades es presenten com a *strings* en *Base64* dins d'un array, amb un *string* per a cada resultat de *mapper.*

Aquest array es processa amb la funció `JSON.stringify` per establir un format interpretable per Python. D'aquesta manera, és possible injectar l'array al codi Python i convertir cada *string* en *Base64* a un objecte serialitzat amb *pickle*, que posteriorment es transforma en un format tractable pel *reducer.*

```
export async function getSerializedResults(results) {
  if (!Array.isArray(results) || results.length === 0)
    throw new Error("Invalid results array");
  const b64List = [];
  for (const result of results) {
    let partialResult = await getTextFromS3(result);
    b64List.push(partialResult.toString("utf-8"));
  }
  return b64List;
}
```

Figura 35: Obtenció dels resultats dels mappers per part d'un reducer.

A continuació es pot veure la funció que rep l'array de strings base64 i els deserialitza. Aquesta funció es troba definida al Pyodide Runtime.

```python
def deserialize_partitions(b64_list):
    parts = []
    for b64 in b64_list:
        raw_bytes = base64.b64decode(b64)
        part = pickle.loads(raw_bytes)
        parts.append(part)
    return parts
```

Figura 36: Funció deserialize_partitions definida al Pyodide Runtime.

4.4 Docker (proves locals)

Per tal de realitzar les proves locals s'ha utilitzat Docker, ja que permet crear entorns aïllats i replicables de manera senzilla, sense dependre de la configuració de la màquina local.

Amb Docker s'ha obtingut la imatge de MinIO del Docker repository i s'ha executat un contenidor en local per simular el bucket d'inputs que proporciona el client, de manera que els workers poden accedir a les dades d'entrada de manera controlada i reproduïble. Docker facilita el port forwarding, permetent exposar els ports del contenidor al sistema host i així interactuar amb els serveis que s'executen dins del contenidor sense problemes de xarxa.

La mateixa estratègia s'ha aplicat per a l'orquestrador, on s'ha utilitzat docker-compose per definir i executar simultàniament dos contenidors: un contenidor de MinIO que simula el bucket de l'orquestrador per guardar i obtenir resultats, i un altre contenidor que executa el codi JavaScript de l'orquestrador. Aquest enfocament garanteix que tots els serveis es poden iniciar, aturar i replicar amb facilitat, i proporciona un entorn aïllat que facilita la depuració i les proves sense afectar altres serveis locals.

A la figura següent es mostra el docker-compose de l'orquestrador, que inclou el contenidor de MinIO (com el client) i, a banda, el contenidor que executa el propi codi de l'orquestrador.

```yaml
services:
  ▷ Run Service
  minio:
    image: quay.io/minio/minio
    container_name: minio-orchestrator
    environment:
      MINIO_ROOT_USER: minioadmin
      MINIO_ROOT_PASSWORD: minioadmin
    command: server /data --console-address ":9001"
    ports:
      - "9002:9000"  # Access MinIO from host via localhost:9000
      - "9003:9001"  # Access MinIO Console from host via localhost:9001
    networks:
      - shared_network

  ▷ Run Service
  orchestrator:
    build: ./
    working_dir: /app
    volumes:
      - ./:/app
    ports:
      - "3000:3000"  # Access Orchestrator from host via localhost:3000
      - "9229:9229"  # Debugging port
    depends_on:
      - minio
```

Figura 37: Fitxer Docker compose de l'orquestrador.

A la figura anterior es pot observar que no només es fa port forwarding del port 3000 del contenidor, sinó també del port 9229, que s'utilitza per a la depuració de codi JavaScript amb eines com Chrome DevTools o Visual Studio Code. Aquesta configuració permet connectar-se al procés Node.js en execució dins del contenidor i inspeccionar el seu estat intern, tal com s'explica en el següent apartat.

4.5 Depuració i descobriment d'errors

La depuració ha sigut una part també important d'aquest projecte, ja que sense aquesta eina no es podrien haver descobert certs "bugs" i comportaments estranys del projecte. Cal mencionar l'ús de la depuració mitjançant el protocol *Node.js Inspector*, accessible pel port 9229, que ha permés connectar el projecte a eines com Chrome DevTools o Visual Studio Code. Això ha facilitat la inspecció de variables, l'execució pas a pas i la detecció de problemes de rendiment de manera eficient.

El flux de depuració en Node.js s'inicia habitualment mitjançant l'execució de l'aplicació amb el comandament node --inspect o bé node --inspect-brk per aturar-se a la primera línia. Aquest mode obri el port 9229, permetent que entorns de desenvolupament com Chrome DevTools o Visual Studio Code amb l'extensió adequada, es connecten al procés en execució. Un cop establerta la connexió, és possible inserir punts de ruptura (*breakpoints*), inspeccionar l'estat de variables i seguir l'execució línia a línia. Internament, aquest sistema s'enganxa al *event loop* de Node.js, la qual cosa permet pausar l'execució sense interrompre el funcionament general de l'entorn, obtenint així una visió detallada del comportament de l'aplicació en temps real.

Aquest mecanisme de depuració es basa en el *Chrome DevTools Protocol* (CDP), un protocol utilitzat pels navegadors per a la inspecció i execució controlada de codi JavaScript. Node.js implementa aquest protocol sobre una connexió WebSocket al port 9229, la qual cosa li permet "simular" el comportament d'un navegador de cara a les eines de depuració. D'aquesta manera, entorns com Chrome DevTools o Visual Studio Code poden interactuar amb l'aplicació Node.js com si fos una pàgina web, però amb accés directe al seu codi i estat intern.

A l'apartat Annexos podem veure tres problemes de rendiment importants que es van descobrir gràcies a la depuració del sistema.

5 Avaluació

Per tal de realitzar els experiments hem usat la infraestructura al núvol de AWS (IaaS) amb instàncies de EC2. La instància de l'orquestrador és del tipus t2.micro (1 vCPU, 1 GB RAM), mentre les instàncies worker són t2.medium (2 vCPU, 4 GB RAM). Per als experiments del patró wordcount s'han pogut utilitzar instàncies t2.micro als workers, però en el cas de l'experiment terasort hem hagut d'usar les t2.medium a causa d'un problema de memòria que mencionarem a l'apartat Limitacions i següents passos.

Cal mencionar que tots els experiments que veurem en aquest apartat han estat realitzats amb les últimes modificacions aplicades al codi i mencionades a l'apartat Annexos.

5.1 Definició de funcions de rendiment

Wall-clock time

$$WallClockTime = T_{final} - T_{inici}$$

on:

$$T_{inici} = min_{i \in first\ stage}(initTime_i)$$

és a dir, el temps d'inici de la primera tasca *Map* que comença a executar-se, i

$$T_{final} = max_{j \in final\ stage}(endTime_j)$$

és a dir, el temps de finalització de l'última tasca *Reduce* que acaba l'execució.

Aquesta definició reflecteix el temps total real transcorregut des de l'inici de la primera tasca de map fins a la finalització de l'última tasca de reduce, proporcionant una mesura directa del rendiment global del job. Entre mig també es té en compte la comunicació dels resultats dels workers de la fase *Map* a l'orquestrador, la preparació de la fase reduce i el despatx de les tasques reducers (gestió de les tasques per part de l'orquestrador) i per últim l'enviament de les tasques als *workers*.

Si volem eliminar la interferència d'aquesta latència intermitja que mencionavem, podem utilitzar una altra mètrica anomenada *Max-stage sum*.

Max-stage sum

$$MaxStageSum = \sum_{s \in S} (max_{t \in s} d(t))$$

és a dir, el sumatori de la duració de tasca màxima de cada *stage*

on:

$$d(t) = t.endTime - t.initTime$$

és a dir , la diferència entre *endTime* i *initTime*.

Speedup

El *speedup* es definex de la següent manera:

$$S = T_1 / T_N$$

on:

T_1 és el temps d’execució per a un únic processador o node, i
T_N és el temps d’execució amb n processadors o nodes.

Eficiència

L’eficiència es definex de la següent manera:

$$E = S / N$$

on:

S és el *speedup* aconseguit amb N processadors, i
N és el nombre de processadors.

5.2 Experiments patró MapReduce Terasort

TeraSort (Yahoo Sort Benchmark, [16]) és un dels benchmarks més reconeguts per avaluar el rendiment dels sistemes Hadoop. El seu objectiu principal és mesurar la capacitat d'un clúster per ordenar grans volums de dades de manera eficient. Aquest procés combina proves de les capes HDFS (Hadoop Distributed File System) i MapReduce, proporcionant una visió integral del rendiment del sistema.

Algorisme de Mostreig

TeraSort implementa una variant de l'algorisme de mostreig (sample sort), on es seleccionen punts de divisió (split points) a partir d'una mostra de les dades d'entrada. Aquests punts es distribueixen entre els reducers per assegurar una classificació eficient i equilibrada. La selecció adequada d'aquests punts és fonamental per optimitzar el rendiment del sistema.

Rendiment i Escalabilitat

El TeraSort ha estat utilitzat per establir rècords en la classificació de grans volums de dades. Per exemple, Yahoo! va aconseguir ordenar 1 TB de dades en només 209 segons utilitzant un clúster de 910 nodes, establint un estàndard per a la comunitat Hadoop. Aquestes proves permeten als administradors de sistemes identificar colls d'ampolla i optimitzar la configuració del clúster per a tasques de classificació intensives [17].

Aplicacions Pràctiques

TeraSort és útil per:

- Avaluar la configuració del clúster i identificar possibles millores en la distribució de tasques.
- Comparar el rendiment entre diferents implementacions de Hadoop o entorns de núvol.
- Establir bases per a la calibració d'altres benchmarks, com TestDFSIO, per obtenir una visió completa del rendiment del sistema.

En aquest cas, s'utilitzarà Terasort per tal de comprovar el rendiment del sistema distribuït utilitzant diferents configuracions de mappers, reducers i workers.

Com a fitxer d'entrada hem usat el terasort-240m, un fitxer d'aproximadament 200MB, del qual podem veure un fragment a la següent figura.

```
X,9ZnUs@_=  0000000000000000000000000030C30  5555DDDD11113333888822226666EEEE00009999999911111111
$>v|j</{I2  0000000000000000000000000030C31  88888888DDDD4444DDDDFFFF66664444BBBBBBBB777799996666
|NU'%&/I(w  0000000000000000000000000030C32  DDDDCCCCCCCCCCCC4444BBBBAAAA7777AAAABBBBFFFF6666FFFF
hZi7QLCnUW  0000000000000000000000000030C33  3333AAAABBBB6666AAAADDDD44449999EEEE66662222EEEECCCC
hDQmRs^YtE  0000000000000000000000000030C34  CCCC00002222111133330000999EEEEFFFF99991111999DDDD
E-hS!"\afo  0000000000000000000000000030C35  000055555555777AAAA8888FFFFBBBBBBBB3333DDDD55552222
u/EjhP{)76  0000000000000000000000000030C36  33332222666699999999999995555DDDDCCCC777733331111BBBB
he~LQsiNd$  0000000000000000000000000030C37  0000FFFF00003333CCCCDDDD9999AAAABBBB1111888844448888
v!`S2k*"&6  0000000000000000000000000030C38  EEEE444466663333DDDD44445555FFFF2222DDDDBBBB66669999
W&\RQc;k(6  0000000000000000000000000030C39  8888AAAAFFFFAAAACCCC666688881111888822229999444EEEE
o~+E}H1G4<  0000000000000000000000000030C3A  22222222EEEEFFFFFFFFCCCC22227777CCCC3333666600007777
g5Rn'\1wjk  0000000000000000000000000030C3B  2222EEEE4444333388884444EEEE111199996666999EEEE4444
e5&Zirq-wx  0000000000000000000000000030C3C  4444BBBBCCCC6666EEEE999966666666AAAACCCC222277775555
8nS[0T8)HJ  0000000000000000000000000030C3D  66663333BBBB00006666EEEE1111FFFFEEEEFFFF77778888AAAA
@T^Azt0{wt  0000000000000000000000000030C3E  CCCC00000000FFFFAAAA00003333AAAAFFFF777444433333333
>,?go6I;C_  0000000000000000000000000030C3F  22225555CCCCFFFF5555FFFFDDDDBBBB77777773333CCCC0000
~GTj1;oxt=  0000000000000000000000000030C40  00002222666655550000444488888CCCCCCCCDDDD2222CCCC1111
```


Figura 38: Fragment del fitxer terasort-240m.

Els següents gràfics s'han extret després de realitzar diverses execucions del benchmark Terasort amb diferents configuracions. En aquest cas les configuracions indiquen el nombre de *mappers* i *reducers* usats amb cada execució, de manera que això indica el paral·lelisme, o en quants de fragments s'ha dividit el fitxer d'entrada per tal que el processen el nombre de *mappers* i *reducers* indicats. A més, el nombre de *workers* disponibles sempre ha sigut el necessari per tal de que es pugui assignar cada subtasca a cada *worker*, assegurant així paral·lelisme real, d'aquesta manera sempre hem tingut tants de *workers* disponibles per cada execució com nombre de *mappers* i *reducers* necessaris.

La idea d'aquest experiment és demostrar com una major paral·lelització del problema aporta un millor rendiment, ja que es redueix el temps d'execució total.

Figura 39: Wall-clock time amb Terasort, augmentant nombre de workers.

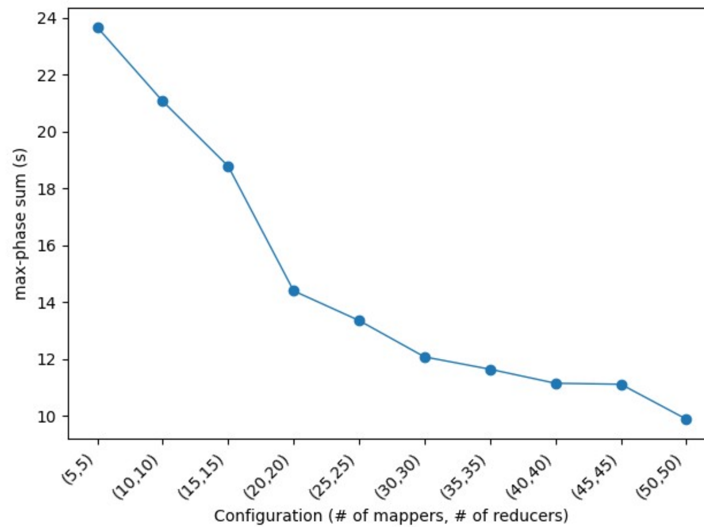

Figura 40: Max-stage time amb Terasort, augmentant nombre de workers.

Es veu a les dos anteriors figures que el sistema es comporta de manera correcta. Quan executem la tasca Terasort amb un major nivell de paral·lelització, obtenims temps d'execució menors.

Gairebé no hi ha cap diferència entre les dues gràfiques, ja que la latència és mínima al nostre sistema. No obstant això, en cas de disposar d'un orquestrador molt més lent (*overhead* d'orquestració), d'una càrrega de treball elevada durant l'execució, o si els *workers* es troben més allunyats de l'orquestrador, amb connexions més dèbils o amb fallades temporals de connexió, aquesta latència podria incrementar-se. Es pot observar la relació entre el temps del *max-stage time* i el *wall-clock time* a la gràfica següent.

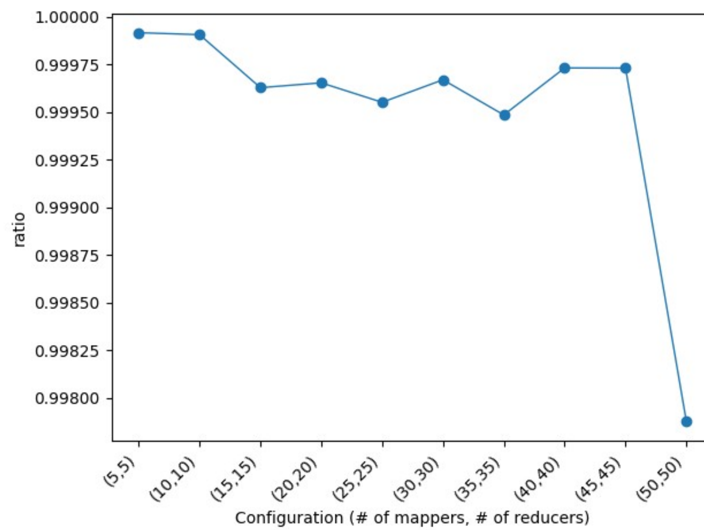

Figura 41: Max-stage time / Wall-clock time amb Terasort, augmentant el nombre de workers.

Amb aquesta última gràfica s'ha volgut representar la relació entre el *max-stage* i el *wall-clock*, ja que l'única diferència entre els dos temps és l'*overhead* d'orquestració i les latències de comunicació.

Aquest ratio al sistema és gairebé 1 per a totes les execucions (la latència és gairebé nul·la), excepte en l'últim experiment amb 20 *mappers* i 20 *reducers*, on la latència s'incrementa. Aquest augment pot deure's a qualsevol dels factors esmentats anteriorment. Al sistema, aquesta latència és gairebé insignificant, tot i que és interessant destacar aquesta diferència.

A continuació es pot visualitzar el *speedup* aconseguit amb el patró terasort. S'ha calculat respecte del *wall-clock time* amb 5 *mappers* i 5 *reducers*, encara que també es podria haver calculat respecte a un únic *mapper* i *reducer*.

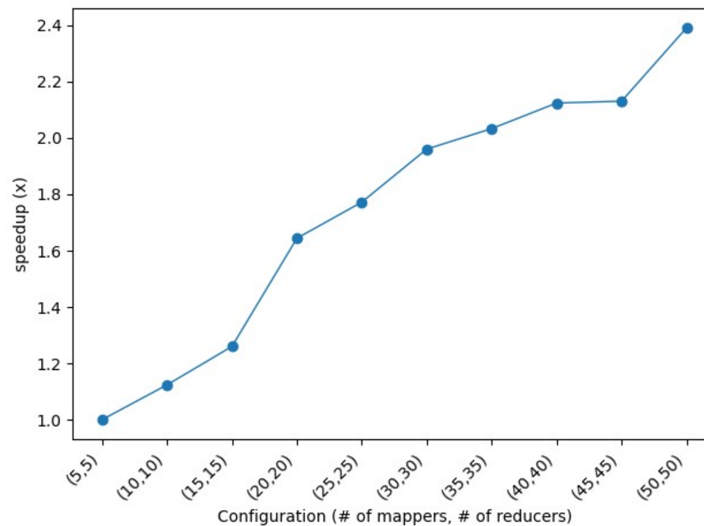

Figura 42: Speedup amb Terasort, augmentant el nombre de workers.

S'aconsegueix un *speedup* de 2.4 amb la configuració (50,50). Com es pot veure a la següent gràfica, l'eficiència aconseguida és baixa.

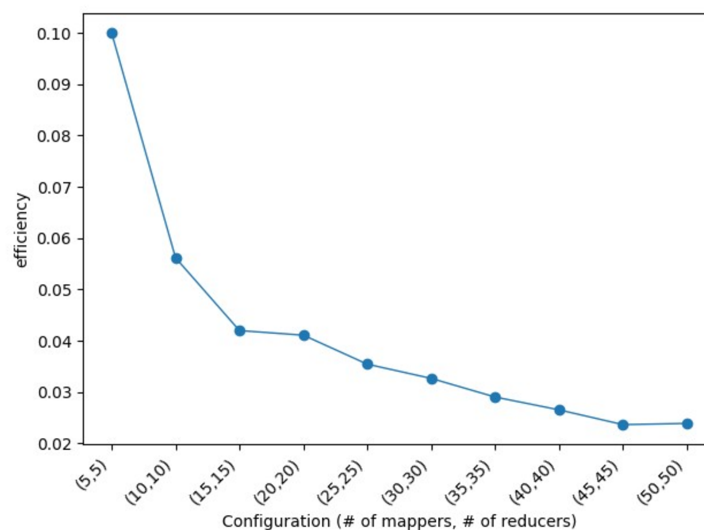

Figura 43: Eficiència amb Terasort, augmentant el nombre de workers.

A l'apartat següent (Experiments patró MapReduce Wordcount) es compararan els resultats obtinguts del Terasort amb els del Wordcount per tal d'extreure observacions interessants.

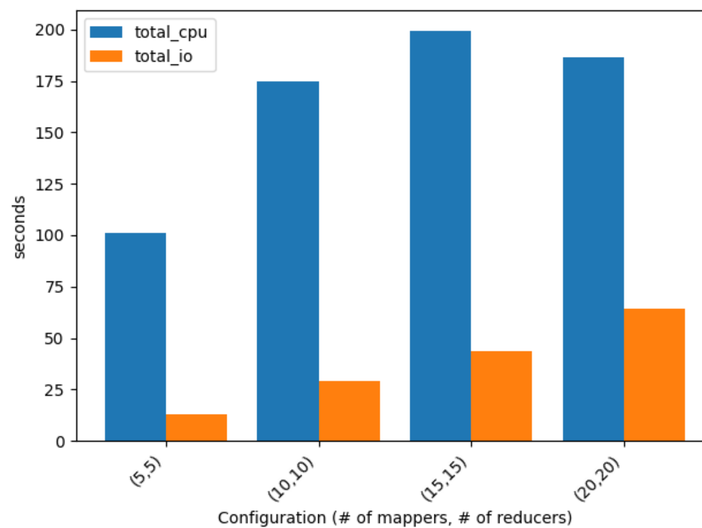

Figura 44: Comparació temps CPU total vs temps E/S total amb Terasort, augmentant nombre de workers.

Aquesta última gràfica representa el temps de CPU afegit entre totes les subtasques del job MapReduce Terasort, en comparació del temps de E/S. Es pot veure que és una tasca CPU bounded, i a més a més, un aspecte curiós sobre aquesta gràfica, és que el temps afegit de CPU i E/S hauria de mantenir-se constant, simplement a l'augmentar el nombre de nodes en paral·lel cada node trigarà menys pero en suma haurien de consumir el mateix temps de CPU, el que es veu al contrari és que es consumeix cada vegada més CPU i E/S. Això podria ser perquè es repeteixen crides a funcions i operacions que en cas d'haver sigut executades al mateix sistema no haguéssin ocurrit.

Una tècnica habitual per reduir la sobrecàrrega d'I/O és l'ús de escriptures en lot (*batch writes*). Aquesta consisteix a acumular diverses operacions d'escriptura en memòria i després enviar-les totes juntes en una única petició al sistema d'emmagatzematge. D'aquesta manera, es redueix el nombre d'accessos al disc i la latència agregada de moltes operacions petites. Aquesta tècnica és especialment útil en sistemes distribuïts o bases de dades amb gran volum d'escriptures, ja que permet optimitzar el rendiment sense perdre coherència de dades.

En aquest cas, el fet d'augmentar el nombre d'operacions de lectura i escriptura explica el possible increment en el temps afegit d'E/S. En el cas de la CPU, no s'ha trobat cap explicació raonable, pot ser és una simple variació que depen de factors com la càrrega del sistema en cada moment.

5.3 Experiments patró MapReduce Wordcount

El patró **MapReduce WordCount** és un exemple clàssic de processament distribuït que il·lustra com es poden comptabilitzar ocurrències d'elements en grans conjunts de dades.

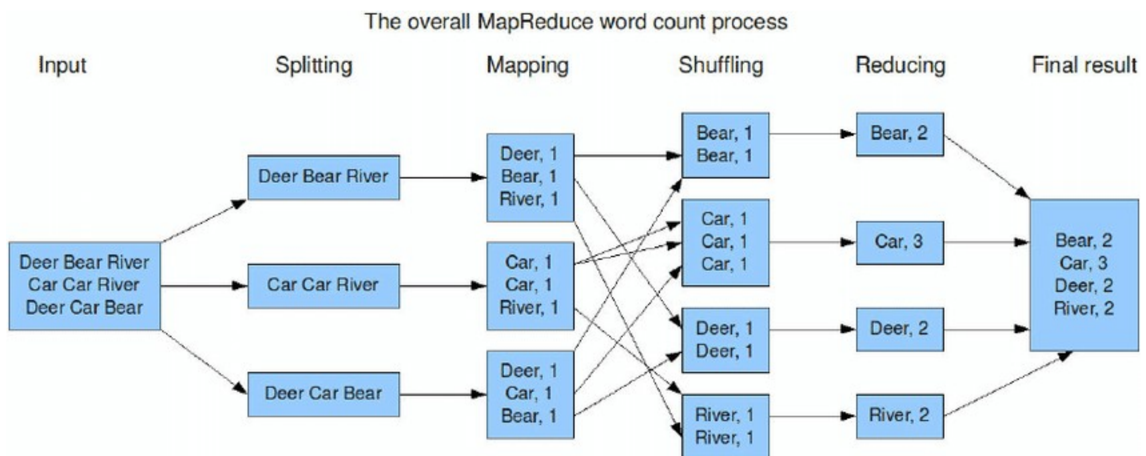

Figura 45: Diagrama d'un job MapReduce Wordcount (Pereira, Óscar, 2014 [18]).

Aquest patró consisteix en dues fases principals:

1. **Map:**

   Cada *worker* rep un fragment del dataset i aplica la funció *map* per transformar les dades d'entrada en parells clau-valor.

   En el cas de WordCount, la clau és la paraula i el valor inicial és 1. Així, cada ocurrència d'una paraula es representa com (paraula, 1).

2. **Shuffle i Sort:**

   Els parells generats per tots els *workers* es reorganitzen de manera que totes les ocurrències d'una mateixa clau es trobin juntes.

   Aquesta fase implica transferir i agrupar dades entre nodes, sovint gestionada per l'orquestrador del sistema.

3. **Reduce:**

Cada *worker* o node rep un conjunt de parells amb la mateixa clau i aplica la funció *reduce*, que combina els valors associats a la clau.

En WordCount, això significa sumar tots els 1 associats a cada paraula, obtenint així el nombre total d'aparicions de cada paraula al dataset complet.

Aquest patró és útil en entorns distribuïts, ja que permet processar grans volums de dades de manera paral·lela i eficient. WordCount serveix com a exemple per entendre conceptes clau de MapReduce, com la divisió de dades en tasques, l'orquestració de *workers*, la combinació de resultats i l'eficiència del processament paral·lel.

En aquest cas, no s'ha implementat el shuffle i sort, per tant només es disposarà d'1 reducer per a qualsevol job MapReduce Wordcount.

El fitxer d'entrada per realitzar les proves ha sigut el fitxer concatenated_gutenberg_books.txt, concatenant textos de la base de dades literària Project Gutenberg [19]. Aquest és un fitxer de text d'aproximadament 173 MB de tamany. Es pot veure un exemple de fragment de text d'aquest fitxer a la següent figura.

```
The Project Gutenberg eBook of Frankenstein; Or, The Modern Prometheus

This ebook is for the use of anyone anywhere in the United States and
most other parts of the world at no cost and with almost no restrictions
whatsoever. You may copy it, give it away or re-use it under the terms
of the Project Gutenberg License included with this ebook or online
at www.gutenberg.org. If you are not located in the United States,
you will have to check the laws of the country where you are located
before using this eBook.

Title: Frankenstein; Or, The Modern Prometheus

Author: Mary Wollstonecraft Shelley

Release date: November 23, 2012 [eBook #41445]
        Most recently updated: October 23, 2024

Language: English

Original publication: United Kingdom: Lackington, Hughes, Harding, Mavor, & Jones, 1818

Credits: Produced by Greg Weeks, Mary Meehan and the Online
     Distributed Proofreading Team at http://www.pgdp.net
     Revised by Richard Tonsing.


*** START OF THE PROJECT GUTENBERG EBOOK FRANKENSTEIN; OR, THE MODERN PROMETHEUS ***

[Transcriber's Note: This text was produced from a photo-reprint of the
1818 edition.]
```

Figura 46: Fragment de text concatenated_gutenberg_books.txt.

Les gràfiques per a l'algorisme del Wordcount s'han obtingut executant els mateixos experiments que amb el Terasort (veure apartat Experiments patró MapReduce Terasort), encara que les configuracions són diferents.

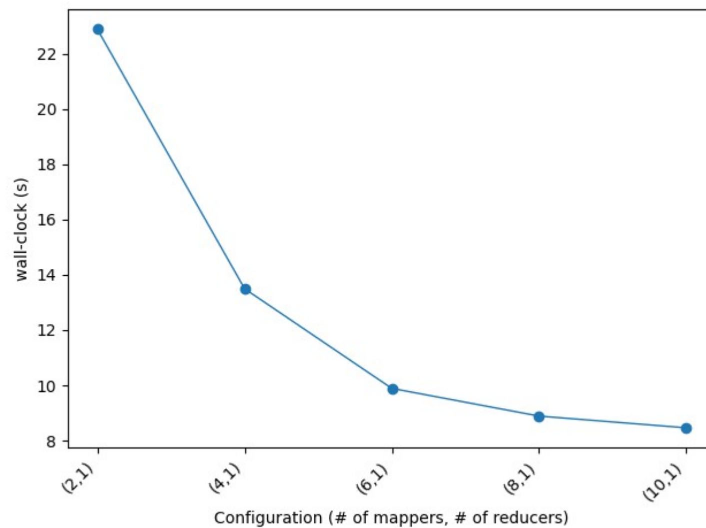

Figura 47: Wall-clock time amb Wordcount, augmentant nombre de workers.

L'anterior figura conté l'evolució del *wall-clock time* amb les diferents configuracions.

Podem veure que en el cas del wordcount el patró de disminució del temps segueix una forma molt més suau i pronunciada, en comparació al Terasort, la tendència és molt més clara, encara que veiem com satura a l'arribar a configuracions més altes, indicant que possiblement augmentar més el nombre de workers en paral·lel deixa de ser rentable, inclús pot ser contraproduent degut a la latència que introdueix el fet de gestionar més tasques per part de l'orquestrador.

Aquesta diferència entre el benchmark Terasort i el Wordcount possiblement té a veure amb la quantitat de computació que s'ha de fer en cada experiment. Més endavant, quan vegem el temps total de CPU i E/S del Wordcount es realitzarà una comparació amb el del Terasort i s'extreuran algunes observacions rellevants.

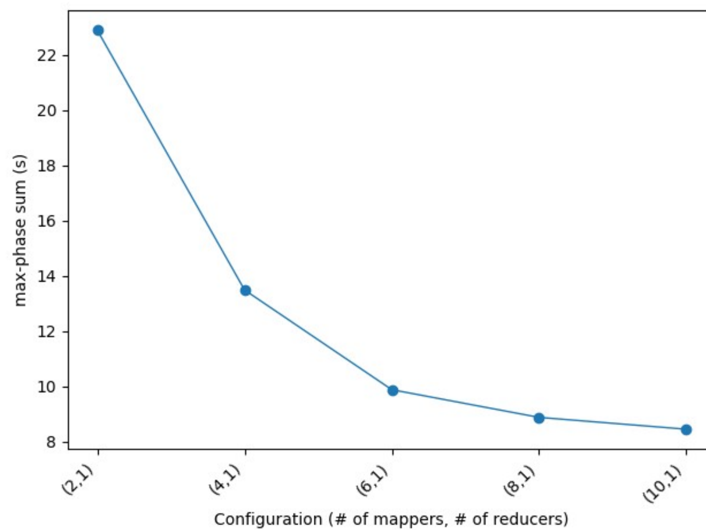

Figura 48: Max-stage time amb Wordcount, augmentant nombre de workers.

Com al cas del Terasort, la gràfica del *max-stage* segueix la mateixa forma que la del *wall-clock*, com a molt poden hi haure mínimes diferències que visualitzarem a la següent gràfica del ratio *max-stage / wall-clock*.

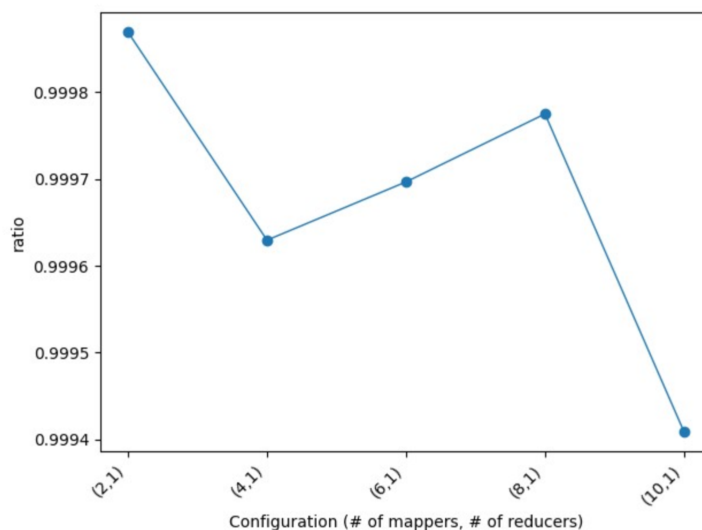

Figura 49: Max-stage time / Wall-clock time amb Wordcount, augmentant el nombre de workers.

En aquest cas el gràfic del ratio max-stage / wall-clock no segueix una forma tant clara com la del Terasort. El comportament normal seria una disminució gradual del ratio, ja que l'orquestrador hauria de gestionar més tasques, indicant majors latències a mesura que aumenten el nombre de tasques a gestionar. Encara que ja s'ha mencionat durant l'explicació del Terasort que aquestes latències es poden deure a factors més aleatoris

com la càrrega puntual del sistema, baixada del rendiment ocasional de les comunicacions de xarxa, etc.

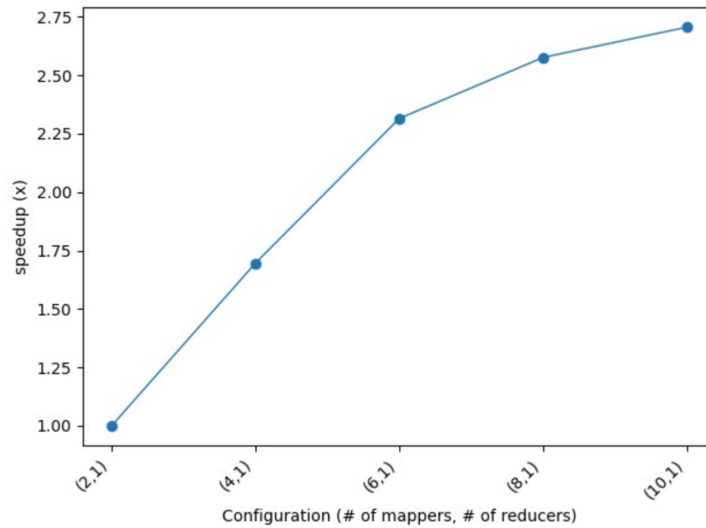

Figura 50: Speedup amb Wordcount, augmentant nombre de workers.

En el cas del Wordcount el speedup és més pronunciat, encara que com ja s'ha mencionat comença a saturar donada una configuració elevada.
En aquest cas afegir 8 workers en paral·lel més (2,1) vs (10,1), ens permet aconseguir un speedup de 2.75 aproximadament.

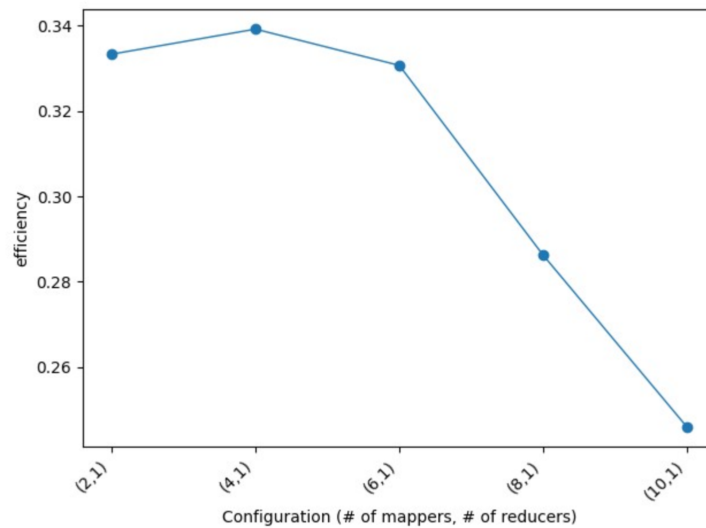

Figura 51: Eficiència amb Wordcount, augmentant nombre de workers.

L’eficiència al cas del terasort arribava a 0.02 aproximadament mentre que el wordcount arriba a una eficiència mínima de 0.26 aproximadament. En els dos casos l’eficiència disminueix, però la del terasort disminueix d’una manera molt més pronunciada.

Llavors, es troba una diferència de 0.35 al speedup entre els dos patrons, és a dir, el wordcount aconsegueix un speedup un 14% superior i a més una eficiència 13 vegades superior, ja que requereix de molts menys workers per arribar-hi.

Amb aquest anàlisi es pot concloure que el patró wordcount és un benchmark que aprofita molt millor el paral·lelisme de *MapReduce* que el patró terasort al nostre sistema. Això passa principalment perquè el nostre wordcount només implementa la part del Map i el Reduce, però no fa shuffling ni sorting, de manera que la computació que realitza cada worker és molt menor.

El terasort, en canvi, realitza shuffling i sorting, això representa el punt crític. Cada *worker* ha de rebre dades de múltiples *mappers*, deserialitzar-les i ordenar-les per clau. Aquesta operació és costosa en CPU i representa el punt d’ampolla principal.

A més, el terasort és un patró de comunicació tots a tots (n:m) i el wordcount que implementa el nostre sistema és tots a un (n:1), això també justifica el major temps d'E/S agregat que presenta el terasort respecte el wordcount.

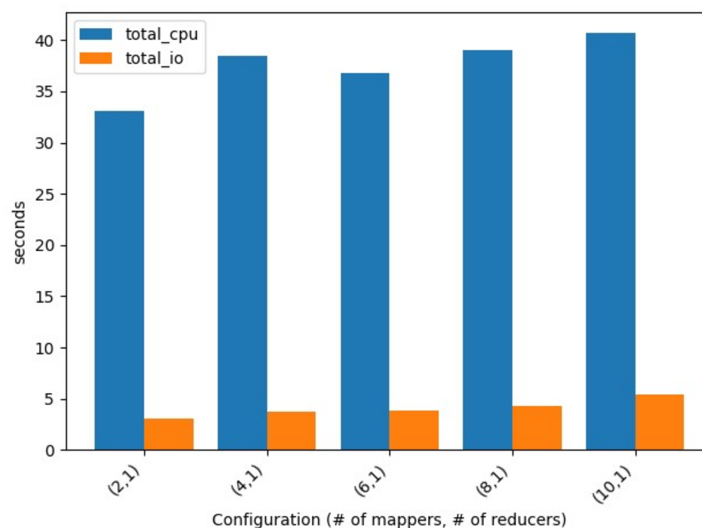

Figura 52: Comparació temps CPU total vs temps E/S total amb Wordcount, augmentant nombre de workers.

De nou es veu una comparació entre el temps afegit de CPU i el d’E/S. A diferència del Terasort, aquí sí que els temps, tot hi haure algunes petites variacions, és manté constant

al llarg de les diferents execucions. Ja s'ha mencionat que això segurament té a veure amb petites variacions a la càrrega de CPU del sistema de cada *worker* o diferències de rendiment de la xarxa a l'hora de mesurar l'E/S.

A més, es veu el que s'ha mencionat anteriorment, el temps de CPU afegit és més o menys 40, fins a 5 vegades menys que el Terasort. Pot ser per això si al Wordcount s'han afegit 8 *workers* per veure una millora substantial del rendiment, al Terasort s'han d'haver afegit 5 vegades més, és a dir 40 *workers* aproximadament, respecte la configuració inicial, per veure un *speedup* paregut. Recordem que el *speedup* observat per al Wordcount ha sigut 2.75 i per al Terasort 2.4, encara que per al Wordcount només s'han afegit 8 workers adicionals i al Terasort s'han afegit 45, i encara així s'ha obtingut un *speedup* menor, això ja s'ha vist a l'apartat anterior d'eficiència.

5.4 Gestió del paral·lelisme i rendiment

La idea amb aquests experiments ha sigut deixar el nombre de workers fixe, i repartir les subtasques dels jobs *MapReduce* entre els *workers* disponibles en cada moment. A més a més, el nombre de *reducers* també s'ha fixat a 5.

Aquest experiment té l'objectiu de comprobar que el sistema és capaç de gestionar tasques i anar-les assignant encara que no tinguéssim suficients *workers*, i encara així observar una millora en el temps d'execució total.

Cal remarcar que aquest experiment s'ha realitzat amb un fitxer de terasort més petit, el terasort-20m, per tal de disminuir el temps d'execució, però la tendència seria la mateixa amb el fitxer més gran, només que l'escala seria diferent.

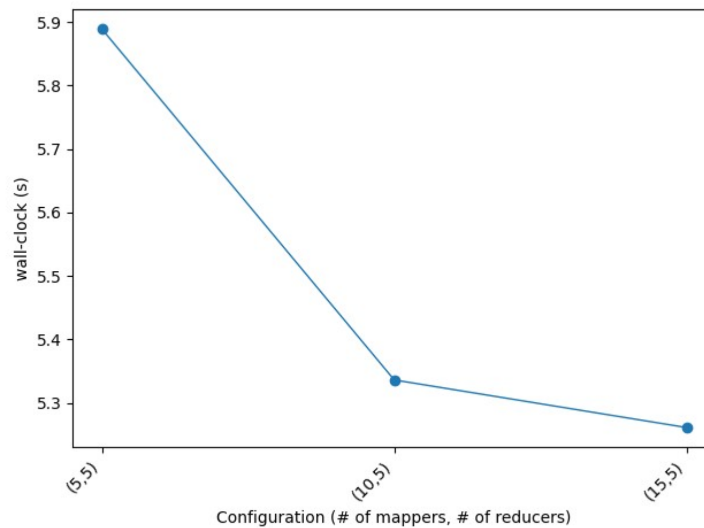

Figura 53: Wall-clock time amb Terasort, nombre de workers fixat a 5.

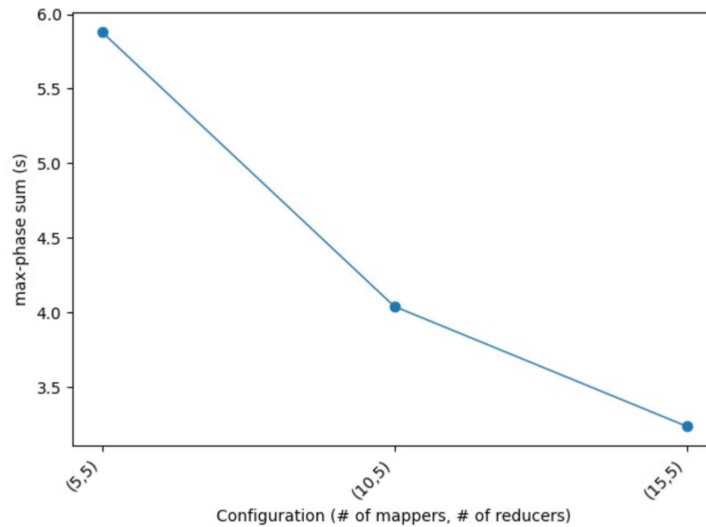

Figura 54: Max-stage time amb Terasort, nombre de workers fixat a 5.

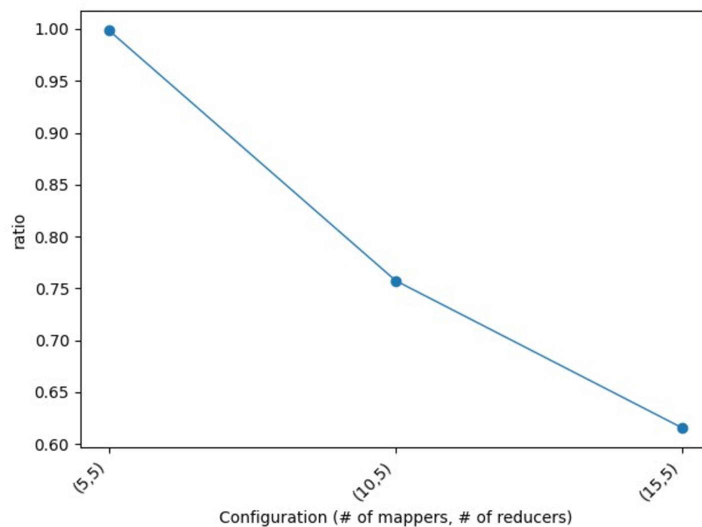

Figura 55: Max-stage time / Wall-clock time amb Terasort, nombre de workers fixat a 5.

Es pot apreciar clarament com encara tenint un nombre de workers fixe a 5, el fet d'augmentar el nombre de tasques que s'executen en paral·lel fa disminuir tant el wall-clock time com el max-stage sum. Això implica que tot i donades les esperes dels mappers per executar la seva part, el fet de crear més tasques petites incrementa el rendiment.

Tot i que la tendència és similar per a les dues mètriques, a partir de la configuració (10,5) el *max-stage time* mostra valors menors que el *wall-clock time*, tal com es veu a

les gràfiques i en la caiguda del ratio. Això passa perquè, amb més de 5 mappers i 5 workers, alguns mappers han d'esperar, i aquest temps d'espera només s'inclou al *wall-clock time*, fent-lo més gran i reduint el ratio. En canvi, amb la primera configuració els temps coincideixen (ratio = 1). Atès que ja s'havia comprovat que la latència entre workers i orquestrador en la fase de reduce és negligible, es conclou que la reducció del ratio amb 5 workers es deu principalment a l'espera dels mappers, i no a problemes de rendiment o de xarxa.

5.5 Anàlisi d'overhead pyodide

La càrrega de l'entorn d'execució Pyodide als *workers* suposa un temps de computació i possibles comunicacions de xarxa (en cas de requerir la instal·lació de la llibreria o els paquets de codi Pandas o Numpy) que hauriem de tenir en compte a l'hora d'evaluar el rendiment del nostre sistema.

Per tal de comprobar aquest *overhead* introduit per la càrrega de Pyodide, hem executat els mateixos experiments que als apartats Experiments patró MapReduce Terasort i Experiments patró MapReduce Wordcount, tot i que amb el Terasort hem arribat a configuracions més baixes ja que la tendència seria la mateixa. En aquest cas, per cada *worker,* a banda dels resultats de les subtasques, també retornen el temps que ha trigat Pyodide en carregar-se. Per representar el temps de càrrega `pyodide_load` es calcula la mitjana entre tots els temps de càrrega dels *workers*.

Per tal de representar la baixa influència del temps de càrrega inicial de **Pyodide**, que és constant, respecte al temps total d'execució de les tasques, s'ha analitzat el **ratio** `pyodide_load / wall-clock`. A mesura que s'incrementa el nombre de *mappers* i *reducers*, el *wall-clock time* disminueix, mostrant que l'impacte del temps de càrrega de Pyodide inicial és negligible en comparació amb el rendiment global del sistema.

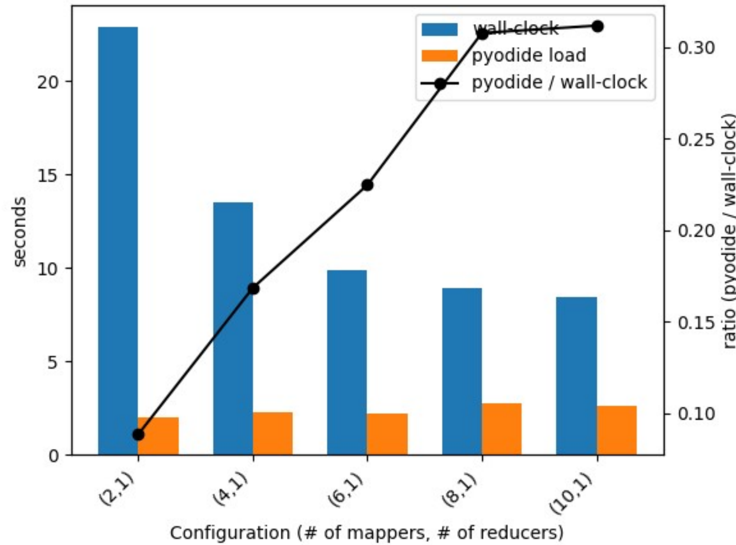

Figura 56: Gràfica de comparació de l'overhead de Pyodide amb Wordcount.

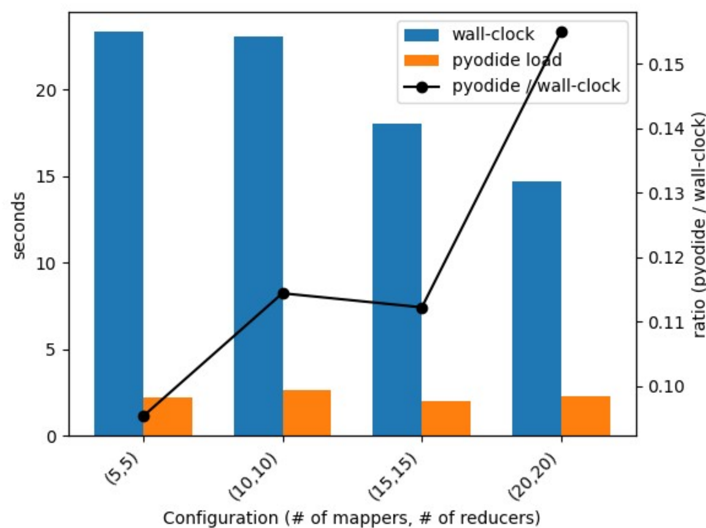

Figura 57: Gràfica de comparació de l'overhead de Pyodide amb Terasort.

Observacions:

- El temps de càrrega de Pyodide és un overhead inicial que es produeix únicament a l'inici de l'execució de cada *worker*, quan es carrega l'entorn de WebAssembly i s'inicialitza la màquina virtual de Python.
- Aquest temps és constant i es manté similar entre diferents *workers* (entre 2 i 3 segons com es pot observar a les figures anteriors), tot i que pot variar lleugerament segons la capacitat de processament single-threaded i la càrrega del sistema en aquell moment.
- En cas que el *worker* hagi de carregar o instal·lar paquets addicionals com pandas i numpy, el temps de càrrega s'incrementa aproximadament en 2 segons, ja que aquests paquets són grans i requereixen més temps per a la seva inicialització.
- Aquest overhead és un cost fix per *worker* i no depèn del nombre de tasques a executar, pel que es pot amortitzar en execucions amb múltiples tasques.
- En sistemes amb recursos heterogenis, aquesta variabilitat podria ser major, però l'overhead inicial segueix sent un factor previsible dins del disseny del sistema.

6 Limitacions i següents passos

6.1 No Multitenancy

Una de les limitacions més importants és que el sistema actualment no aïlla correctament l'execució de múltiples *jobs* al mateix temps al sistema, de manera que tasques de diferents *jobs* acaben sent assignades al mateix *worker* simultàniament, empitjorant el rendiment (apartat Detecció de problema amb execució Multitenancy).

6.2 Error accés S3

Com es menciona a l'apartat Detecció i optimització d'un coll d'ampolla a l'orquestrador, s'hauria d'investigar per què es dona l'error `getaddrinfo ENOTFOUND <bucket_name>` per tal de poder usar el sistema quan tenim menys workers disponibles que *mappers* i *reducers* a executar.

6.3 Gestió de memòria

Durant l'execució dels experiments amb el fitxer terasort-240m vam observar errors `heap out of memory` en màquines petites (t3.micro, 1 GB RAM). Això s'explica per dues causes complementàries: (1) **V8 (el motor de Node.js)** té una mida de heap per defecte limitada**,** la grandària de l'"old space" es calcula automàticament i sovint és molt inferior a la RAM física disponible, i, si cal ampliar-la, cal iniciar Node amb l'opció `--max-old-space-size=<MB>` per assignar més heap a V8; i (2) l'ús de **Pyodide / WebAssembly** i biblioteques com NumPy/Pandas implica memòries lineals (ArrayBuffers / WASM memory) i *allocations* natives que poden sumar-se a la memòria total del procés i esgotar la memòria de la VM (Màquina Virtual). Per aquests motius, una actualització a instàncies amb més RAM (p. ex. t3.medium, 4 GB) i/o ajustar `--max-old-space-size` són solucions pràctiques.

6.4 Avaluació a dispositius voluntaris reals

Els experiments que s'han realitzat i avaluat a l'apartat Avaluació s'han portat a terme usant la infraestructura al núvol d'Amazon Web Services. En un entorn real, tendriem dispositius voluntaris molt més heterogenis, amb rendiments desiguals, latències desiguals, connexions menys confiables, etc. S'hauria de probar el sistema en aquestes condicions per veure el rendiment que ofereix, encara que la tendència hauria de ser la mateixa, millora del rendiment i reducció del temps d'execució a mesura que augmentem paral·lelització.

6.5 Implementació de càrregues més complexes

Actualment el sistema només suporta DAGs MapReduce de 2 stages, s'haurien de implementar DAGs més complexes a futur per tal de tenir un framework complet i adaptat al context de computació actual, on les càrregues solen ser més complexes.

6.6 Millora de la interfície del client

Volem integrar un sistema de programació de DAGs més refinat, similar a altres frameworks de data analytics.
A més, en futures versions, seria desitjable implementar mecanismes que permetin al client reconectar-se i recuperar els resultats de les seves tasques.

6.7 Seguretat i control d'accés

L'actual identificació de clients i workers amb uuidv4 pot donar lloc a col·lisions improbables però problemàtiques. Caldria implementar un sistema de control d'accés més robust i segur per evitar pèrdues de dades o interferències entre sessions, a més de permetre l'execució de tasques només a clients autoritzats.

6.8 Gestió de workers stagglers

Seria interessant implementar detecció i gestió de *workers stagglers*, és a dir, tasques que es queden estancades executant una tasca durant un període prolongat. En aquests casos, es podria assignar la mateixa tasca a un altre worker i prendre el primer resultat que arribi. Si l'execució falla repetidament, s'hauria de retornar un error al client per evitar bloquejos i assegurar la disponibilitat del sistema.

6.9 Refactorització del codi

El codi requereix una revisió i refactorització per eliminar operacions redundants i millorar l'estructura, facilitant la mantenibilitat i predicció del comportament en situacions diverses.

La refactorització del codi és necessària per optimitzar el rendiment, eliminant possibles operacions redundants que consumeixin temps i memòria de manera ineficient. Aquesta refactorització també ajudaria a millorar la mantenibilitat i a assegurar un comportament predictable del sistema en diferents situacions, encara que actualment el codi ja presenta una modularització important, encara faltaria refinar-lo més de cara a un escenari realista.

6.10 In memory cache

Un altre aspecte rellevant és que l'emmagatzematge dels resultats intermedis es fa actualment en disc, fet que provoca que l'execució de les diferents fases s'alenteixi a causa del temps necessari per llegir i escriure aquesta informació. En un sistema de producció, s'hauria d'implementar una memòria cau en memòria (com Redis) per guardar els resultats intermedis dels *mappers*, cosa que acceleraria significativament l'execució. A més, es podria considerar també guardar els resultats dels *reducers* en memòria cau, permetent suportar l'execució de tasques iteratives, tal com fan sistemes avançats com Apache Spark. Aquest problema ja es detectava en Apache Hadoop, i Spark el va millorar gràcies a l'ús d'*RDDs* (Resilient Distributed Datasets), que permeten gestionar dades en memòria de forma eficient (Zaharia et al., 2012 [20]).

6.11 Assignació intel·ligent de tasques

Es podria millorar l'assignació de tasques. Actualment el sistema assigna les tasques als *workers* donant prioritat als que tenen més processadors disponibles. En un sistema real es poden realitzar assignacions als *workers* basant-se en més factors com la velocitat del processador, quantitat de memòria, ample de banda, latència, reputació del *worker*, etc.

6.12 Sistemes de recompensa

Crear un sistma de recompensa per incentivar la col·laboració. No tenen que ser incentius monetaris, sinó per exemple incentius socials (puntuacions, rankings) per tal d’assignar cert “prestigi” a aquells grups o individus que proporcionen més computació. Això ja ho han fet sistemes com el SETI@home o el Folding.

7 Aplicació de principis ètics i de responsabilitat social

El projecte planteja la creació d'un *framework* per a l'execució de tasques d'anàlisi de dades en paral·lel mitjançant un sistema distribuït basat en la col·laboració voluntària d'usuaris. Aquesta proposta comporta diverses implicacions socials i ètiques que cal considerar.

En primer lloc, el disseny i l'ús del sistema s'han concebut evitant qualsevol **biaix de gènere** o discriminació, ja que es tracta d'una eina tecnològica neutra que no condiciona la participació dels voluntaris per raó de sexe, edat o altres factors socials.

En segon lloc, des d'una **perspectiva mediambiental**, el sistema contribueix al desenvolupament sostenible en reutilitzar recursos de computació ja existents en lloc de fomentar la creació de nous centres de dades. Això implica una reducció del consum energètic i de les emissions associades, millorant així l'eficiència global.

Pel que fa a la **responsabilitat social**, el projecte promou la col·laboració oberta i voluntària, tot afavorint la creació d'una comunitat tecnològica en què cada usuari pot aportar valor amb els seus recursos. Aquest plantejament té un impacte positiu en termes d'accés i democratització de la capacitat computacional.

Finalment, des del **punt de vista ètic** i deontològic, el sistema s'ha dissenyat sota criteris de transparència i respecte a la privacitat dels participants, garantint que les dades i les tasques compartides es gestionin de manera responsable i d'acord amb les normes que regeixen la comunitat universitària i professional.

8 Conclusions

S’ha aconseguit desenvolupar una implementació funcional que permet l’execució distribuïda i paral·lela de tasques d’anàlisi de dades aprofitant recursos computacionals voluntaris. Tot i que el prototip presenta algunes limitacions, ha demostrat la viabilitat i factibilitat del concepte, alhora que ha servit per identificar les principals àrees de millora que caldria abordar per assolir un sistema robust i preparat per a producció.

A nivell personal, aquest treball m’ha permès comprendre en profunditat els reptes actuals associats a l’anàlisi de dades distribuïda i la importància creixent d’aquest àmbit. He après sobre les diferents solucions tecnològiques, des dels clústers tradicionals i serveis cloud fins a la computació voluntària, i la manera com es poden combinar per optimitzar recursos.

Durant el desenvolupament, m’he enfrontat a diversos reptes tècnics que m’han obligat a aprofundir en tecnologies com Node.js. He entès el seu model d’execució basat en event loop, la diferència entre microtasks i macrotasks, i per què un entorn single-threaded pot ser eficient per a tasques I/O bound. També he descobert el funcionament del motor V8 i la seva compilació Just-In-Time, aspectes clau per valorar el rendiment i les limitacions de Node.js en aplicacions de computació distribuïda. A més, la capacitat multiplataforma de Node.js ha estat un avantatge clau per al desplegament del servei.

Altres aspectes apresos inclouen la gestió de la memòria en Node.js, especialment el heap, la serialització i deserialització de dades, la comunicació entre components mitjançant HTTP i WebSockets, així com aspectes pràctics relacionats amb AWS: la preparació d’AMIs, els scripts per crear instàncies, la gestió de permisos per accedir a buckets S3, el funcionament de l’IDMS Metadata Server i el disseny i execució d’experiments, juntament amb l’anàlisi offline dels resultats mitjançant scripts en Python.

En resum, aquest projecte m’ha permès adquirir coneixements pràctics i teòrics en anàlisi de dades distribuïda, computació al núvol, computació voluntària, programació amb Node.js, computació distribuïda, AWS i metodologia d’experimentació.

9 Referències

[1] Chen, M., Mao, S., & Liu, Y. (2017). Big Data: Challenges, Opportunities and Realities. arXiv preprint arXiv:1705.04928. https://arxiv.org/abs/1705.04928

[2] Wangsom, Peerasak & Lavangnananda, K. & Bouvry, Pascal. (2017). Measuring Data Locality Ratio in Virtual MapReduce Cluster Using WorkflowSim. 10.1109/JCSSE.2017.8025944.

[3] Nowicki, M. Comparison of sort algorithms in Hadoop and PCJ. *J Big Data* **7**, 101 (2020). https://doi.org/10.1186/s40537-020-00376-9

[4] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., … & Stoica, I. (2016). *Apache Spark: A unified engine for big data processing*. Communications of the ACM, 59(11), 56–65. https://doi.org/10.1145/2934664

[5] Rocklin, M. (2015). *Dask: Parallel computation with blocked algorithms and task scheduling*. In Proceedings of the 14th Python in Science Conference (pp. 130–136). https://doi.org/10.25080/Majora-7b98e3ed-013

[6] White, T. (2012). *Hadoop: The definitive guide* (3rd ed.). O'Reilly Media.

[7] Ghukasyan, T., Altunyan, V., Bughdaryan, A., Aghajanyan, T., Smbatyan, K., Papoian, G. A., & Petrosyan, G. (2025). *Smart distributed data factory volunteer computing platform for active learning-driven molecular data acquisition.* Scientific Reports, 15, 7122. https://doi.org/10.1038/s41598-025-21234-5

[8] Lavoie, E., & Hendren, L. (2019). *Development of a new framework for high performance volunteer computing.* ResearchGate. https://www.researchgate.net/publication/387203638_Development_of_a_new_framework_for_high_performance_volunteer_computing

[9] González, J., & Pérez, A. (2021). *A collaborative citizen science platform for real-time volunteer computing and games.* arXiv. https://arxiv.org/abs/2104.12345

[10] A. Semjonov, H. Bornholdt, J. Edinger and G. R. Russo, "Wasimoff: Distributed Computation Offloading Using WebAssembly in the Browser," 2024 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops), Biarritz, France, 2024. https://ieeexplore.ieee.org/document/10503392

[11] Amazon Web Services. (2023a). *Amazon Simple Storage Service (S3) – Object storage built to store and retrieve any amount of data.* https://aws.amazon.com/s3/

[12] Amazon Web Services. (2023b). *Amazon Elastic Compute Cloud (EC2) – Scalable virtual servers in the cloud.* https://aws.amazon.com/ec2/

[13] Chorazyk, P. & Byrski, Aleksander & Piętak, Kamil & Kisiel-Dorohinicki, Marek & Turek, W.. (2017). Volunteer computing in a scalable lightweight web-based environment. Computer Assisted Methods in Engineering and Science. 24. 17-40.

[14] David P. Anderson. "BOINC: A System for Public-Resource Computing and Storage." *5th IEEE/ACM International Workshop on Grid Computing*, pp. 365–372, Nov. 8, 2004, Pittsburgh, PA. IEEE. DOI: 10.1109/GRID.2004.14

[15] Hespe, D., Hübner, L., Mercatoris, C., & Sanders, P. (2024). *Scalable Fault-Tolerant MapReduce*.

[16] Terasort benchmark YahooHadoop https://sortbenchmark.org/YahooHadoop.pdf

[17] Apache Hadoop Examples: Terasort https://hadoop.apache.org/docs/current/api/org/apache/hadoop/examples/terasort/package-summary.html

[18] Pereira, Óscar & Capitão, Micael. (2014). Mediator Framework for Inserting Data into Hadoop.

[19] Project Gutenberg https://www.gutenberg.org/

[20] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., & Stoica, I. (2012). *Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing*. Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI '12), 2-2. https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf

10 Annexos

En els següents apartats veurem com es van detectar certs problemes de rendiment al sistema i la solució que es va proporcionar.

10.1 Codi font del projecte

Codi del projecte: https://github.com/mrobledo07/PyEdgeCompute

10.2 Detecció i optimització d'un coll d'ampolla a l'orquestrador

Els experiments que s'executen en aquest apartat corresponen als que s'han executat a l'apartat Gestió del paral·lelisme i rendiment però en aquest cas (abans dels canvis) no es visualitzen millores del rendiment tant notables, a continuació veurem l'explicació.

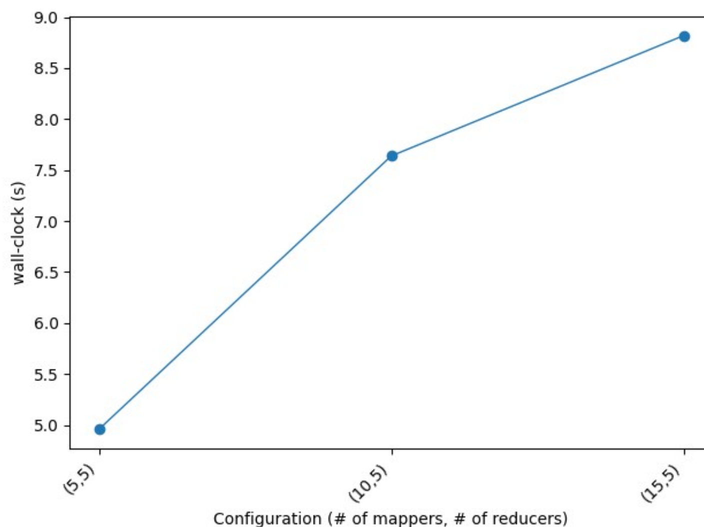

Figura 58: Wall-clock time amb Terasort, augmentant nombre de workers (annex).

Teòricament, si paral·lelitzem més una etapa, com la fase *map*, les tasques individuals són més petites i triguen menys a executar-se. No obstant això, en aquest cas hem observat que tant el *wall-clock time* com el *max stage time* augmenten significativament quan augmentem el nombre de tasques. Això possiblement tingue a veure amb una mala gestió de les tasques per part de l'orquestrador.

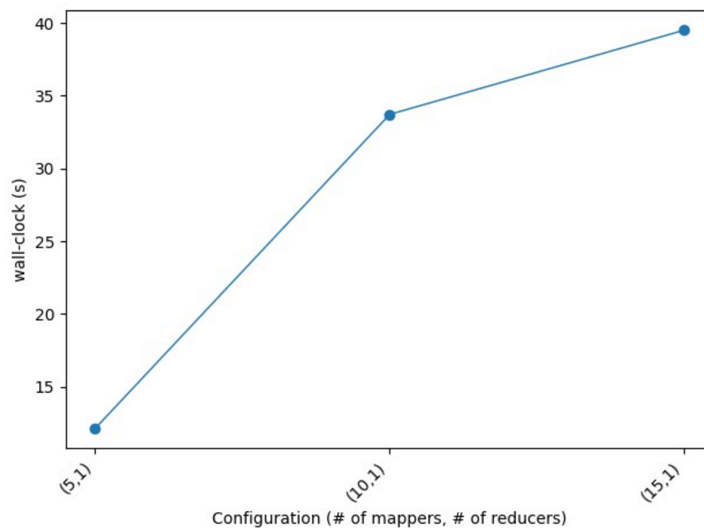

Figura 59: Wall-clock time amb Wordcount, augmentant nombre de workers (annex).

Aquesta última gràfica va en la línia del que hem mencionat abans amb el Terasort. Si deixem el nombre de *workers* fixe i augmentem el nombre de tasques en paral·lel, augmenta el *wall-clock time*.

Aquest és un problema que no té a veure amb les latències, ja que les latències mesuren la diferència en temps d'execució entre *max-stage time* i *wall-clock time*, llavors podriem tenir un cas en el que el *wall-clock time* augmenti i el max-stage disminueixi, augmentant les latències, però en aquest cas els dos augmenten.

Una possible explicació seria que tenim més tasques petites que s'han de repartir entre un nombre menor de *workers*, fent que hi hagi una sèrie de tasques que tenen que esperar a que els anteriors *mappers* finalitzin la execució, fent que el *wall-clock* time augmenti.

L'anterior explicació no té molt de sentit, ja que pot ser hi ha tasques que s'han d'esperar a les anteriors, però les anteriors s'executen més ràpid, de manera que el *wall-clock time* no hauria d'empitjorar respecte a la configuració, encara que sí que tendria sentit que empitjorés respecte a una situació on tenim més *workers* disponibles que *mappers* necessaris.

Per descartar de manera definitiva aquesta hipòtesi, si fem un plot del *max-stage* hauriem de veure la mateixa tendència que en el *max-stage* i *wall-clock* amb nombre de workers superior als *mappers* necessaris, és a dir, a la baixa, ja que cada tasca individual triga molt menys, i ja hem vist que el max-stage selecciona la màxima duració de cada *stage*.

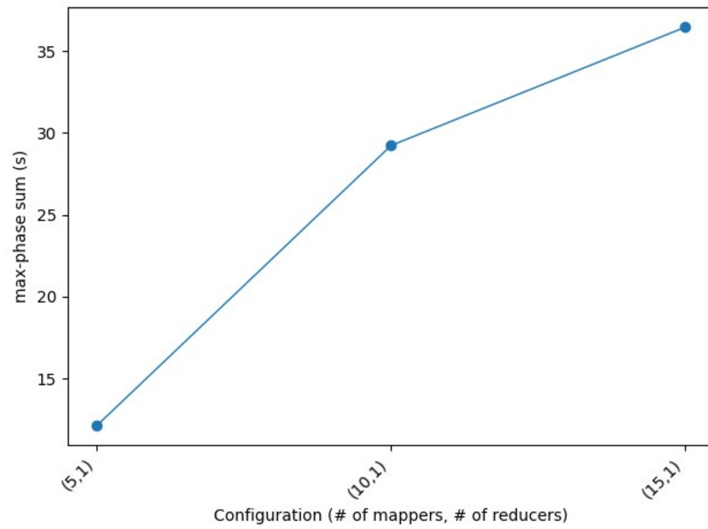

Figura 60: Max-stage time amb Wordcount, augmentant nombre de workers (annex).

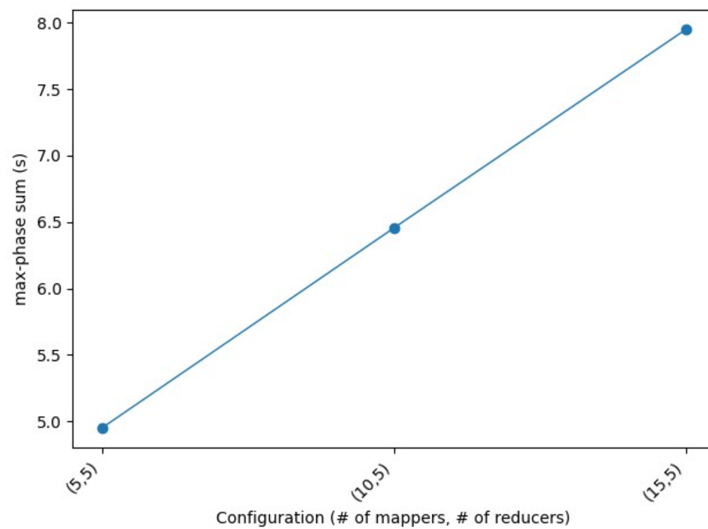

Figura 61: Max-stage time amb Terasort, augmentant nombre de workers (annex).

Veiem clarament un problema a les figures anteriors, i és que *max-stage time* també reporta temps creixents, quant haurien de ser decreixents.

Si analitzem els fitxers veiem un comportament molt estrany.


```
"metadata": {
  "fe306217-ffab-40c1-99e6-ff1f1835de90-mapper2": [
    1754492103.851,
    0.6021,
    5.8923,
    0.1015,
    1754492110.448
  ],
  "fe306217-ffab-40c1-99e6-ff1f1835de90-mapper3": [
    1754492103.852,
    0.6634,
    6.1296,
    0.072,
    1754492110.718
  ],
  "fe306217-ffab-40c1-99e6-ff1f1835de90-mapper1": [
    1754492103.85,
    0.5862,
    8.9005,
    0.2118,
    1754492113.553
  ],
  "fe306217-ffab-40c1-99e6-ff1f1835de90-mapper0": [
    1754492103.85,
    0.5919,
    9.1083,
    0.2084,
    1754492113.76
  ],
  "fe306217-ffab-40c1-99e6-ff1f1835de90-mapper4": [
    1754492103.852,
    0.5665,
    9.3206,
    0.2336,
    1754492113.974
  ],
```


Figura 62: Fitxer mapreducewordcount amb 5 mappers i 1 reducer (5 workers).

Si recordem el format que segueix cada tasca:

`id-subtasca: [ initTimeAbsolut, readTime, cpuTime, writeTime, endTimeAbsolut ]`

Aquestes són les metadades que retornava cada *worker* juntament amb el resultat.

Veiem que tots els mappers tenen el mateix initTime, té sentit ja que tenim 5 mappers i 5 workers, es pot assignar cada mapper a cada worker al mateix moment per l'orquestrador sense cap problema. A més a més si calculem endTime – initTime, és a dir, la duració de cada subtasca veiem que coincideix amb la suma de readTime, cpuTime i writeTime, cosa que també té sentit ja que entre initTime i endTime passen "readTime + cpuTime + writeTime" segons.

Ara anem al següent fitxer.

```
"metadata": {
  "13585acc-9825-43e9-a31f-7fc1d10d691d-mapper1": [
    1754492124.356,
    0.2567,
    4.0144,
    0.2013,
    1754492128.833
  ],
  "13585acc-9825-43e9-a31f-7fc1d10d691d-mapper2": [
    1754492124.357,
    0.2574,
    4.145,
    0.1716,
    1754492128.933
  ],
  "13585acc-9825-43e9-a31f-7fc1d10d691d-mapper3": [
    1754492124.358,
    0.2702,
    4.1911,
    0.196,
    1754492129.016
  ],
  "13585acc-9825-43e9-a31f-7fc1d10d691d-mapper4": [
    1754492124.357,
    0.2664,
    4.2416,
    0.2046,
    1754492129.07
  ],
  "13585acc-9825-43e9-a31f-7fc1d10d691d-mapper0": [
    1754492124.356,
    0.277,
    4.7366,
    0.2403,
    1754492129.61
  ],
```


Figura 63: Fitxer mapreducewordcount amb 10 mappers i 1 reducer (5 workers, primers 5 mappers).

Veiem que els primers 5 mappers segueixen la mateixa lògica que els del fitxer anterior. Però lo interessant ve a l'examinar els 5 següents *mappers*, que son els que se han començat a executar una vegada finalitzats els anteriors (recordem que només tenim 5 *workers*, llavors els *mappers* es reparteixen).

```
"13585acc-9825-43e9-a31f-7fc1d10d691d-mapper5": [
  1754492128.837,
  0.3391,
  1.064,
  0.0495,
  1754492141.433
],
"13585acc-9825-43e9-a31f-7fc1d10d691d-mapper6": [
  1754492128.839,
  3.6829,
  3.4718,
  3.6379,
  1754492149.138
],
"13585acc-9825-43e9-a31f-7fc1d10d691d-mapper9": [
  1754492128.847,
  0.0495,
  3.9245,
  0.1067,
  1754492149.562
],
"13585acc-9825-43e9-a31f-7fc1d10d691d-mapper7": [
  1754492128.842,
  0.0709,
  3.917,
  0.1067,
  1754492149.563
],
"13585acc-9825-43e9-a31f-7fc1d10d691d-mapper8": [
  1754492128.844,
  0.0682,
  3.6379,
  0.1067,
  1754492149.607
],
```


Figura 64: Fitxer mapreducewordcount amb 10 mappers i 1 reducer (5 workers, últims 5 mappers).

Aquí hi ha un problema interessant.

Si agafem qualsevol d'aquests mappers, per exemple el *mapper8*, calculem la duració com la diferència entre endTime i initTime obtenim uns 20.763 segons, en canvi veiem que si sumem readTime + cpuTime + writeTime aconseguim un resultat de 3.751 segons aproximadament. La diferència es abismal, hi han 20.763 - 3.751 = 17.012 segons que no estan justificats.

Seguint amb l'anàlisi, en el cas del terasort amb nombre de *workers* 5, si seleccionàvem l'opció de (10,10) o (15,15) s'obtenia un error de S3 paregut a "getaddrinfo ENOTFOUND <bucket_name>", que indica que no es pot resoldre la direcció URL proporcionada del bucket. Inclús abans d'obtenir aquest error, obteniem un altre diferent relacionat amb el IMDS d'AWS (*Instance Metadata Service*), que tenia el següent missatge: "CredentialsProviderError: Could not load credentials from any providers". Això es va solucionar amb unes modificacions al client d'AWS del worker relacionades amb obtenir una instància singleton de client AWS, per no obrir un nou client per cada tasca. Però com ja hem mencionat, arreglar aquest error ens va portar a l'anterior que s'ha mencionat.
D'aquesta manera, ja podiem veure que hi havia algun problema relacionat amb l'accés a S3.

Una de les possibles causes identificades és la saturació del client HTTP intern utilitzat pels workers per accedir a S3. Amb pocs workers i molts mappers, cada worker processa diverses tasques en sèrie llançant múltiples peticions a S3 en ràfegues molt curtes. Això pot esgotar el pool de connexions, els ports efímers o altres recursos de xarxa del procés,

provocant retards, reintents (retry/backoff) i errors transitoris com `getaddrinfo ENOTFOUND <bucket_name>`. L'efecte és un augment del temps "no comptabilitzat" entre l'inici i la fi de la tasca. En augmentar el nombre de workers, la càrrega es reparteix i la contenció desapareix, reduint aquests errors.

Per comprovar si la causa era la saturació del client S3, es van introduir modificacions a la configuració del S3Client d'AWS. Concretament, es va garantir la reutilització d'un únic client per worker (cosa que ja teniem implementada), es va ampliar el nombre màxim de connexions simultànies (maxPoolConnections) i es va limitar la concurrència de peticions S3 dins de cada worker. A més, es va configurar la política de reintents en mode adaptatiu per gestionar millor la latència i les ràfegues de peticions. Aquestes mesures pretenien reduir la contenció de recursos i minimitzar els errors i retards detectats.
Encara aplicant aquests últims canvis, el problema persistia.

Posteriorment es va pensar que l'error getaddrinfo ENOTFOUND <bucket_name> tenia a veure amb un problema de resolució DNS. Es va intentar afegir una caché amb `npm cacheable-lookup`, però tampoc va reportar cap millora. El problema de rendiment i `error getaddrinfo` seguia donant-se.

Veient que cap de les solucions anteriors funcionava, es va pensar en un possible error de implementació de l'orquestrador. En aquest cas, es va pensar que podria donar-se el cas de que un mateix worker tingués assignades múltiples tasques al mateix temps, de manera que tingués que repartir la CPU entre diferents tasques, augmentan el temps d'execució.

Tenint en compte la hipotèsi anterior, es va probar de retornar el `workerId` juntament amb cada *mapper*, per comprobar si era possible que un mateix worker executés diferents tasques al mateix moment. A continuació veiem els resultats.

```
"metadata": {
  "28efa1dd-e8bf-4f23-8d7c-8d9956a7dd14-mapper1": [
    1755271657.027,
    0.286,
    4.2281,
    0.2034,
    1755271661.745,
    "b6014f4d-c741-4ea6-b235-f1692ea43c30"
  ],
  "28efa1dd-e8bf-4f23-8d7c-8d9956a7dd14-mapper2": [
    1755271657.027,
    0.3445,
    4.3151,
    0.1503,
    1755271661.837,
    "ad0d89ff-c068-42aa-adc8-3c68f0cbf799"
  ],
  "28efa1dd-e8bf-4f23-8d7c-8d9956a7dd14-mapper3": [
    1755271657.027,
    0.2905,
    4.3542,
    0.1755,
    1755271661.848,
    "f5264828-ff70-4731-96f4-65f4837387f3"
  ],
  "28efa1dd-e8bf-4f23-8d7c-8d9956a7dd14-mapper0": [
    1755271657.026,
    0.3201,
    4.3753,
    0.1498,
    1755271661.872,
    "f7bd999d-4ec7-42c0-b78f-4c4d57a51913"
  ],
  "28efa1dd-e8bf-4f23-8d7c-8d9956a7dd14-mapper4": [
    1755271657.028,
    0.3554,
    4.3709,
    0.2012,
    1755271661.955,
    "42a0dbc0-133d-48a3-8f00-802d078ba907"
  ],
```


Figura 65: Fitxer mapreducewordcount amb 10 mappers i 1 reducer, retornant workerId (5 workers, primers 5 mappers).

Veiem a l'anterior figura que tot funciona correctament, cada worker només té 1 processador com ja se ha mencionat, de manera que se li asigna una única tasca.


```
"28efa1dd-e8bf-4f23-8d7c-8d9956a7dd14-mapper5": [
  0.2884,
  4.19,
  0.05,
  1755271674.546,
  "b6014f4d-c741-4ea6-b235-f1692ea43c30"
],
"28efa1dd-e8bf-4f23-8d7c-8d9956a7dd14-mapper7": [
  1755271661.753,
  0.0725,
  0.9246,
  0.13,
  1755271683.29,
  "b6014f4d-c741-4ea6-b235-f1692ea43c30"
],
"28efa1dd-e8bf-4f23-8d7c-8d9956a7dd14-mapper8": [
  1755271661.757,
  3.7708,
  3.4999,
  0.13,
  1755271683.311,
  "b6014f4d-c741-4ea6-b235-f1692ea43c30"
],
"28efa1dd-e8bf-4f23-8d7c-8d9956a7dd14-mapper9": [
  1755271661.757,
  0.05,
  1.0715,
  0.13,
  1755271683.338,
  "b6014f4d-c741-4ea6-b235-f1692ea43c30"
],
"28efa1dd-e8bf-4f23-8d7c-8d9956a7dd14-mapper6": [
  1755271661.752,
  3.7202,
  3.6785,
  0.13,
  1755271683.376,
  "b6014f4d-c741-4ea6-b235-f1692ea43c30"
],
"28efa1dd-e8bf-4f23-8d7c-8d9956a7dd14-reducer": [
  1755271683.378,
  0.7496,
  7.0783,
  0.2917,
  1755271691.498,
  "ad0d89ff-c068-42aa-adc8-3c68f0cbf799"
]
```


Figura 66: Fitxer mapreducewordcount amb 10 mappers i 1 reducer, retornant workerId (5 workers, últims 5 mappers).

A la figura anterior es veu clarament com el worker amb identificador "b6014f4d-c741-4ea6-b235-f1692ea43c30" executa tots els mappers restants després d'haver-se executat els 5 primers, indicant que no s'està portant a terme una bona distribució de tasques entre els workers restants, això explica la diferència entre endTime – initTime i readTime + cpuTime + writeTime, i possiblement també l'error indicant anteriorment, ja que un mateix worker realitza múltiples lectures i escriptures a S3, portant a problemes de concurrència o contenció.

A continuació ve la explicació de com es va solucionar aquest problema.

Per què passava?
El problema principal era una combinació entre:

1. **Un bucle síncron a `processTaskQueue()`** que iterava mentre availableWorkers > 0.

2. **Una actualització asíncrona de la disponibilitat**: quan s'enviava una tasca amb `ws.send()`, el decrement de la disponibilitat del *worker* no es feia immediatament, sinó dins del *callback* d'èxit de l'enviament.

Això feia que el bucle veiés el *worker* com a disponible fins que el *callback* s'executés, permetent que diverses iteracions consecutives assignessin més tasques al mateix *worker* abans que es registrés com a ocupat.

Impacte observat

- El sistema deixava de repartir bé les tasques i sobrecarregava un únic *worker.*

- Els temps totals (*wall-clock time*) de la fase de *map* augmentaven innecessàriament.

- L'eficiència del paral·lelisme baixava, ja que *workers* lliures no rebien feina mentre un altre estava saturat.

Com es va solucionar
La solució va ser decrementar la disponibilitat del worker de manera immediata i síncrona abans d'enviar la tasca.

- Això garanteix que, dins del mateix bucle, el *worker* ja aparegui com ocupat després d'una assignació, evitant sobreassignacions.

- Si l'enviament falla, es torna a incrementar la disponibilitat i es reencua la tasca.

Aquesta estratègia assegura que la distribució de tasques sigui més equitativa i que el sistema aprofiti millor tots els *workers* disponibles.

A la figura de `reserveWorkerAndSendTask()` (figura 27) ja hem vist com era la implementació errònia, a continuació veiem la implementació correcta assegurant que a la següent volta de bucle el worker aparegui com a ocupat per tal de no poder assignar-li diferents tasques al mateix temps.

```
/**
 * Assigns a task to a worker and sends it via WebSocket.
 */
function reserveWorkerAndSendTask(worker, task) {
  workerRegistry.assignTaskToWorker(
    worker.worker_id,
    task.clientId,
    task.taskId
  ); // Decrement first
  clientRegistry.markTaskRunning(task.clientId, task.taskId, worker.worker_id);
  worker.ws.send(JSON.stringify(task), (err) => {
    if (err) {
      console.error(
        `❌ Failed to send task to worker ${worker.worker_id}: ${err.message}`
      );
      workerRegistry.completeTaskOnWorker(worker.worker_id, task.taskId); // Increment back (as if completed/failed)
      taskQueue.push({ clientId: task.clientId, taskId: task.taskId }); // Re-queue
      clientRegistry.markTaskPending(task.clientId, task.taskId);
      throw new Error(
        `Failed to send task to worker ${worker.worker_id}. Error: ${err.message}`
      );
    } else {
      console.log(`✅ Sent task ${task.taskId} ...`);
    }
  });
}
```

Figura 67: Funció per reservar i enviar tasca al worker (ACTUALITZADA).

Una vegada s'han aplicat aquests canvis, el rendiment del sistema canvia dràsticament.

Tot i que la distribució de tasques després d'aquests canvis va començar a funcionar de manera correcta, seguiem obtenint l'error `getaddrinfo ENOTFOUND <bucket_name>,` i no es va poder trobar la veritable raó per la que això passava, però seria necessari solucionar-ho per poder suportar casos on el nombre de mappers i reducers és superior al nombre de *workers*.

A més, aquests canvis també pareixien indicar que permetrien que el sistema suportés *Multitenancy,* ja que cada *worker* teòricament només podria tenir assignades *n* tasques en cada moment, on *n* és el nombre de processadors disponibles, però el que s'ha vist fent experiments és que a un mateix *worker* se li poden assignar tasques de diferents *jobs MapReduce* al mateix temps, encara que no del mateix *job*. Això també seria important solucionar-ho de cara al futur (Detecció de problema amb execució Multitenancy).

Podem veure els resultats després d'aplicar aquests canvis a l'apartat Gestió del paral·lelisme i rendiment .

10.3 Solució als problemes de rendiment detectats amb configuracions elevades

Els experiments que s'executen en aquest apartat corresponen als que s'han executat a l'apartat Experiments patró MapReduce Terasort, però en aquest cas (abans dels canvis) no es visualitzen millores del rendiment tant notables, a continuació veurem l'explicació.

A continuació veiem com es comporta el sistema si executem els jobs un darrere l'altre, sense *Multitenancy.*

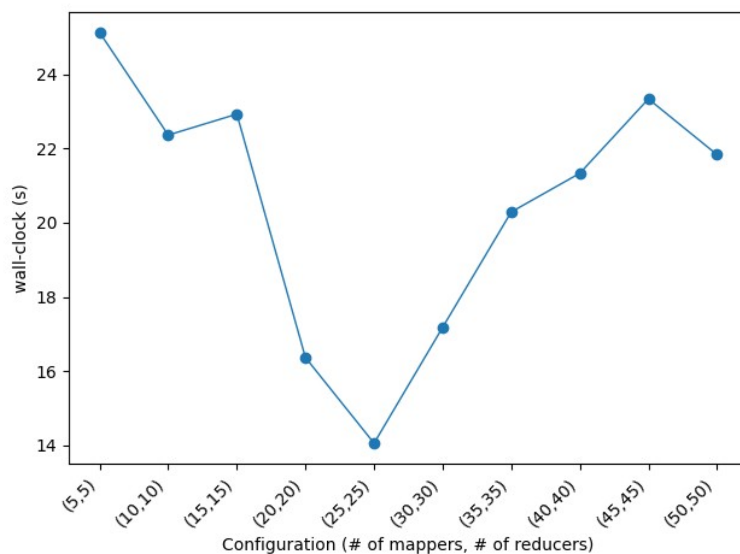

Figura 68: Wall-clock time amb Terasort (annex).

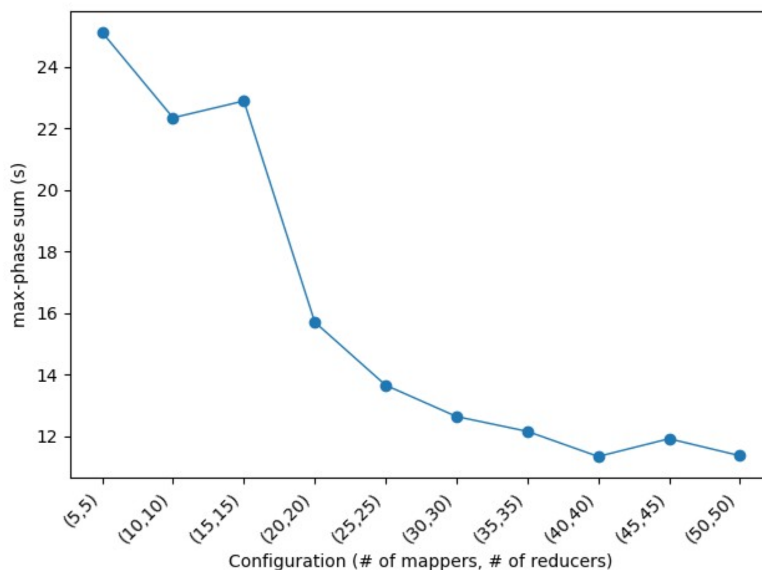

Figura 69: Max-stage time amb Terasort (annex).

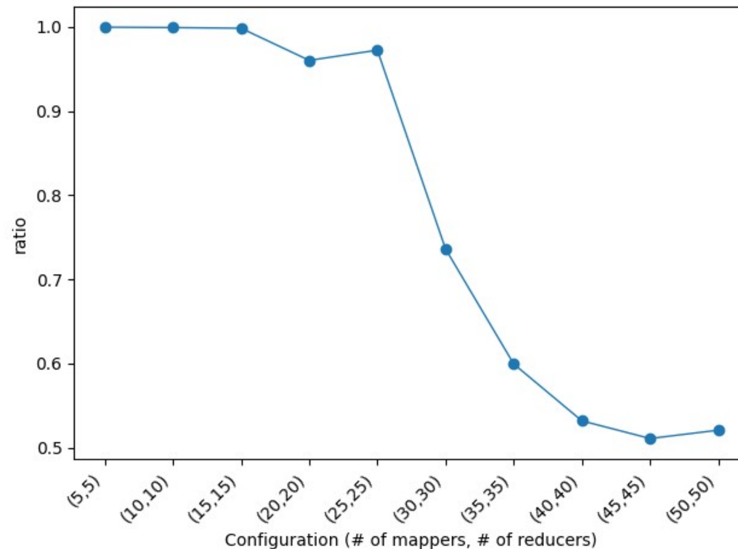

Figura 70: Max-stage time / Wall-clock time amb Terasort (annex).

Veiem que el wall-clock time comença a augmentar a partir de la configuració (25,25).
En canvi, el max-stage mostra una tendència clarament a la baixa.

Si observem la gràfica del ratio, cada vegada és més petit, indicant la presència de latències molt notòries.

Aquestes latències veiem que son baixes per a les primeres configuracions i altes per a les últimes. Això ens fa descartar que siguin possibles problemes de connexió, sinó que l'única cosa que canvia entre una configuració i la següent és el nombre de tasques que ha de despatxar i gestionar l'orquestrador.

Sabem perfectament que la implementació de l'orquestrador ja està bastant optimitzada, amb estructures de dades que presenten complexitat O(1) per a gran part de les execucions.

Mirant el codi arribem a la conclusió de que els console.log poden estar disminuint dràsticament el rendiment, ja que són crides síncrones a un dispositiu d'E/S, cosa que pot alentir bastant l'execució del codi a la CPU.

Vist això, la solució passa per eliminar totes les instruccions `console.log()` al codi de l'orquestrador, almenys les que estan involucrades en funcions que s'executen de manera recurrent.

Una vegada aplicats els canvis, el rendiment augmenta d'una manera molt notòria. Podem veure els resultats dels experiments als apartats Experiments patró MapReduce Terasort i Experiments patró MapReduce Wordcount.

10.4 Detecció de problema amb execució Multitenancy

En cas de permetre executar múltiples jobs MapReduce al mateix temps al sistema es tendria el que s'anomena un sistema *Multitenancy.*

Per raons desconegudes, que segurament tenen a veure amb la implementació de l'orquestrador, el actual sistema no soporta *Multitenancy.*

Si es consulta Detecció i optimització d'un coll d'ampolla a l'orquestrador (després d'haver aplicat els canvis), teòricament a cada *worker* se li assignen les tasques de manera síncrona, de manera que es redueix la seva disponibilitat i no hauria de ser possible assignar més tasques que processadors té disponible, però la realitat ens diu una altra cosa, com es veurà a continuació.

D'aquesta manera, en l'escenari de múltiples jobs al mateix temps, es segueix tenint el mateix problema que a l'apartat Detecció i optimització d'un coll d'ampolla a l'orquestrador abans d'aplicar els canvis, però ara no s'assignen múltiples *mappers* o *reducers* del mateix *job* a un mateix *worker*, sinó que s'assignen múltiples *mappers* o *reducers* de *jobs* diferents a un mateix *worker.*

Aquest comportament caldria solucionar-lo per aconseguir un sistema en producció, ja que sinó quan múltiples clients envien els seus jobs, o un mateix client envia múltiples jobs (amb *l'array* d'arguments del fitxer de configuració), el rendiment dels jobs es veu afectat, augmentan tant el *wall-clock* com el *max-stage*, perjudicant gravíssimament al *speedup*, llavors seria completament necessari implementar un sistema *Multitenancy* si s'hagués de portar a producció.

A continuació es veuen els resultats dels experiments.

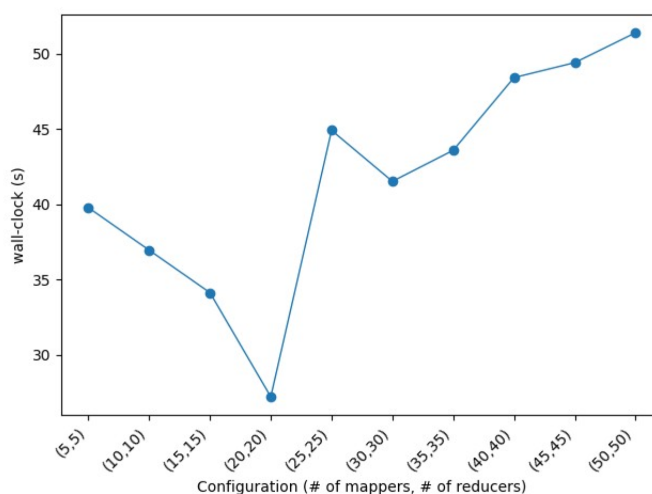

Figura 71: Wall-clock time amb Terasort, augmentant el nombre de workers i multitenancy.

Es veu com clarament, a partir de la configuració (20,20) la tendència és alcista, de manera que empitjora el rendiment com més tasques tenim executant-se en paral·lel. Això passa pel que es mencionava al principi d'aquest apartat, l'explicació és que el sistema no suporta *multitenancy.*

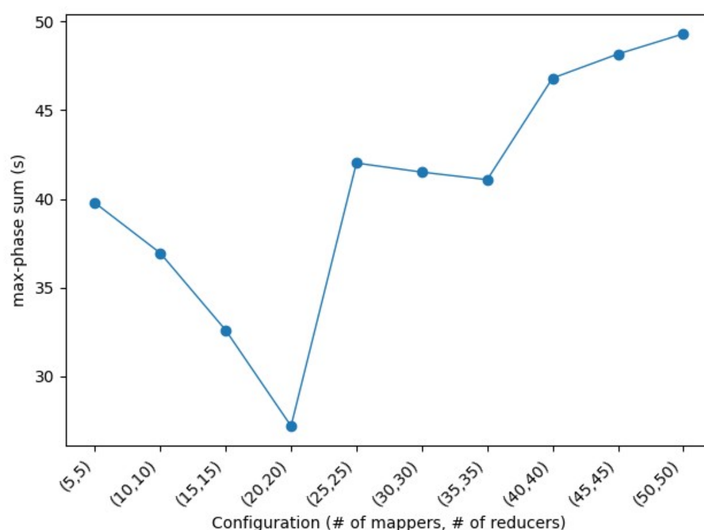

Figura 72: Max-stage time amb Terasort, augmentant el nombre de workers i multitenancy.

Es pot apreciar que en el case del *max-stage* també es segueix la mateixa tendència, llavors no es pot dir que pugui ser un possible problema de latències o falta de rendiment de l'orquestrador.
De fet a la següent figura es pot veure que la latència és mínima, encara que variable

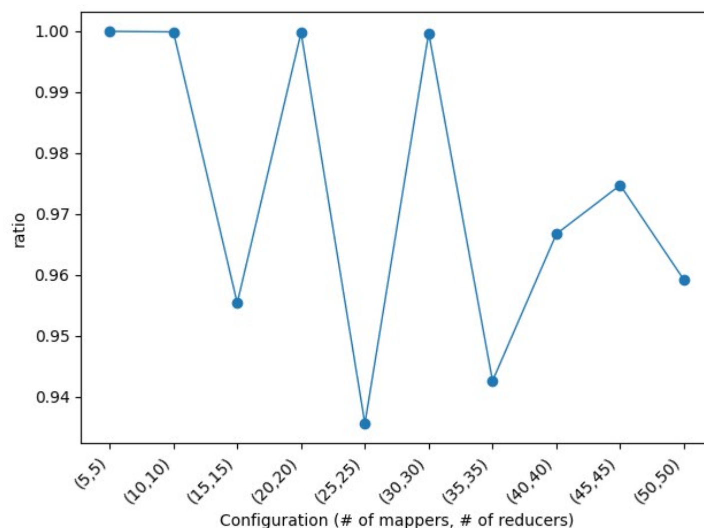

Figura 73: Max-stage time / Wall-clock time amb Terasort, augmentant el nombre de workers i multitenancy.

En resum, aquest experiment deixa clar que el sistema actualment no suporta *multitenancy*, i els temps d'execució empitjoren al tenir múltiples jobs executant-se al mateix temps.